Méthode de prototypage virtuel permettant l'évaluation précoce de la consommation énergétique dans les systèmes intégrés sur puce

Khouloud Zine El Abidine

Thèse de doctorat de l'université Pierre et Marie Curie - Paris VI

Spécialité Informatique

(École Doctorale Informatique, Télécommunication et Électronique)

Présentée par Khouloud Zine Elabidine

Pour obtenir le grade de

Docteur de l'université Pierre et Marie Curie

Méthode de prototypage virtuel permettant l'évaluation précoce de la consommation énergétique dans les systèmes intégrés sur puce

Soutenue le 16 octobre 2014, devant le jury composé de

M.	Daniel Chillet	ENSSAT	Rapporteur
Mme	Cecile Belleudy	Université de Nice	Rapporteur
M.	Olivier Romain	Université Cergy Pontoise	Examinateur
M.	Sylvain Guilley	TELECOM-ParisTech	Examinateur
M.	Habib MEHREZ	UPMC Paris VI	Examinateur
M.	Alain Greiner	UPMC Paris VI	Directeur de thèse

Résumé

Depuis quelques années, les systèmes embarqués n'ont pas cessé d'évoluer. Cette évolution a conduit à des circuits de plus en plus complexes pouvant comporter plusieurs centaines de processeurs sur une même puce.

Si la progression des techniques de fabrication des systèmes intégrés, a permis l'amélioration des performances de ces derniers en terme de temps et de capacité de traitement, elle a malheureusement amené une nouvelle contrainte de conception.

En effet, cette nouvelle génération de systèmes consomme plus d'énergie et nécessite donc la prise en compte, pendant la phase de conception, des caractéristiques énergétiques dans le but de trouver le meilleur compromis (performance / énergie).

Des études montrent qu'une estimation précoce de la consommation – i.e. au niveau comportemental – permet une meilleure diminution de l'énergie consommée par le système.

L'outil EDPE (Early Design Power Estimation), objet de cette thèse, propose en réponse à ce besoin, une procédure permettant la caractérisation énergétique précoce d'une architecture de type MPSoC (MultiProcessor System on Chip) dans la phase de prototypage virtuel en SystemC.

EDEP s'appuie sur des modèles de consommation par composant pour en déduire l'énergie dissipée par le système global lorsque le système est simulé au niveau CABA (Cycle Accurate Byte Accurate) ou encore TLM (Transaction Level Model). Les modèles proposés par EDPE, ont été intégrés dans la bibliothèque de prototypage virtuel SoClib. Ainsi, pendant la phase d'exploration architecturale, le concepteur dispose en plus des caractéristiques temporelles et spatiales de son circuit, d'une estimation précise de sa consommation énergétique.

L'élaboration de modèles de consommation pour les différents composants matériels d'un système, à l'aide d'EDPE, est simple, homogène et facilement généralisable.

Les résultats obtenus montrent la capacité d'EDPE à prédire la consommation énergétique de différentes applications logicielles déployées sur une même architecture matérielle de manière précise et rapide.

Abstract

Technological trends towards high-level integration combined with the increasing operating frequencies, made embedded systems design become more and more complex.

The increase in number of computing resources in integrated circuit (IC) led to over-constrained systems.

In fact, SoC (System on Chip) designers must reduce overall system costs, including board space, power consumption and development time.

Although many researches have developed methodologies to deal with the emerging requirements of IC design, few of these focused on the power consumption constraint. While the highest accuracy is achieved at the lowest level, estimation time increases significantly when we move down to lower levels.

Early power estimation is interesting since it allows to widely explore the architectural design space during the system level partitioning and to early adjust architectural design choices.

EDPE estimates power consumption at the system levels and especially CABA (Cycle Accurate Bit Accurate) and TLM (Transaction Level Modelling) levels.

The EDPE have been integrated into SoCLib library.

The main goal of EDPE (Early Design Power Estimation) is to compare the power consumption of different design partitioning alternatives and chooses the best trade-off power/ performance.

Experimental results show that EDPE (Early Design Power Estimation) method provides fast, yet accurate, early power estimation for MPSoCs (Multiprocessor System on Chip).

EDPE uses few parameters per hardware components and is based on homogeneous and easy characterization method.

EDPE is easily generalized to any virtual prototyping library.

Sommaire

Table des figures

Chapitre 1

Introduction

L'évolution des techniques de fabrication des systèmes embarqués (SoC), a conduit à des systèmes à forte capacité d'intégration. Parmi ces systèmes on s'intéresse à une catégorie bien précise, celle des systèmes multiprocesseurs (MPSoC) à espace d'adressage partagé. Cette nouvelle génération de circuits, fonctionne avec une fréquence très élevée et peut comporter plusieurs centaines de processeurs sur une même puce permettant ainsi la réalisation de traitements complexes. Cependant, cette évolution de performance, implique une augmentation de la consommation énergétique et donc une diminution de l'autonomie de ces systèmes. Pour remédier à ce problème, l'énergie consommée par un système embarqué constitue un paramètre important qui devrait guider la phase de prototypage virtuel. Ainsi, l'exploration architecturale doit prendre en compte trois dimensions : surface de silicium, performance temporelle et consommation énergétique. Pour pouvoir prendre en compte l'énergie consommée par le système pendant la phase de prototypage virtuel, on a besoin d'un moyen permettant l'estimation de la consommation totale du système pendant l'étape d'exploration architecturale. L'outil EDPE (Early Design Power Estimation), objet de cette thèse, propose une procédure permettant la caractérisation énergétique d'une architecture de type MPSoC (MultiProcessor System on Chip). EDEP s'appuie sur des modèles de consommation par composant pour en déduire l'énergie dissipée par le système global à partir de sa description au niveau CABA (Cycle Accurate Byte Accurate) ou encore TLM (Transaction Level Model). Les modèles proposés par EDPE, ont été intégrés dans la bibliothèque de prototypage virtuel SoClib. Ainsi, pendant la phase d'exploration architecturale, le concepteur dispose en plus des caractéristiques temporelles et spatiales de son circuit, d'une estimation précise de sa consommation énergétique.

Le chapitre 2, présente les enjeux de l'élaboration d'une telle méthode et les

questions auxquelles notre travail cherche à apporter des réponses en insistant sur le choix du modèle de consommation et la caractérisation des composants de la plate-forme considérée.

Le chapitre 3, présente d'abord l'état de l'art des architectures MPSoC, ciblées par notre méthode. Il détaille ensuite le principe de l'estimation de consommation des circuits intégrés, à différents niveaux d'abstraction. Il expose enfin, les différentes approches existantes qui tentent d'apporter une solution à l'estimation précoce de la consommation énergétique des systèmes embarqués (SoC). La plupart des solutions existantes proposant une méthode de caractérisation différente par type de composant, ce qui constitue un frein considérable à la généralisation de ces solutions.

Le chapitre 4, décrit d'abord la plate-forme ciblée par la méthode EDEP, puis expose les modèles énergétiques des différents composants de cette plate-forme et leurs intégration dans la bibliothèque de prototypage virtuel SoClib. On retrouve par la suite un descriptif de la méthode de mesure physique de la consommation énergétique du circuit. Enfin, les micro-kernels utilisés pour caractériser la plate-forme sont détaillés dans un dernier paragraphe.

Le chapitre 5, présente la méthode générique de caractérisation des composants matériels que nous proposons, qui constitue une de nos principale contribution. Cette méthode générale permet de « remonter » dans le prototype virtuel les caractéristiques énergétiques des composants qui sont disponibles dans la description physique.

Enfin, le chapitre 6, présente les expérimentations ayant trois principaux objectifs : d'abord, de valider la précision du dispositif de mesure. Ensuite d'évaluer la capacité prédictive des modèles, en comparant l'erreur entre l'énergie totale mesurée par PowerPlay et l'énergie prédite par EDPE, lors du déploiement de différentes applications logicielles sur la même plate-forme. Le dernier objectif de ces expérimentations est de calculer l'accélération de la méthode EDPE comparée à une méthode d'estimation de la consommation basée sur la simulation au niveau portes logiques.

Chapitre 2

Problématique

Sommaire

Poussés par des besoins de performances, les systèmes intégrés n'ont pas cessé d'évoluer. Durant des années, la tendance a été d'augmenter la capacité d'intégration des circuits intégrés en utilisant des techniques de fabrications plus évoluées. Aujourd'hui on peut embarquer jusqu'à un milliard de transistors sur un même circuit (cf.figure 2.1). Cette évolution a permis d'embarquer des processeurs de plus en plus puissants répondant ainsi aux besoins accrus d'augmentation de la capacité de calcul et de la vitesse de traitement. Cependant, utiliser des processeurs plus gros a un grand inconvénient : cela augmente considérablement la consommation énergétique du circuit sans améliorer proportionnellement les performances. Selon la loi de Pollack [1], la performance d'un processeur occupant une surface de taille 2.x est estimée à 1.4 fois la performance d'un processeur de taille x alors que la consommation énergétique reste proportionnelle à la surface (à fréquence égale).

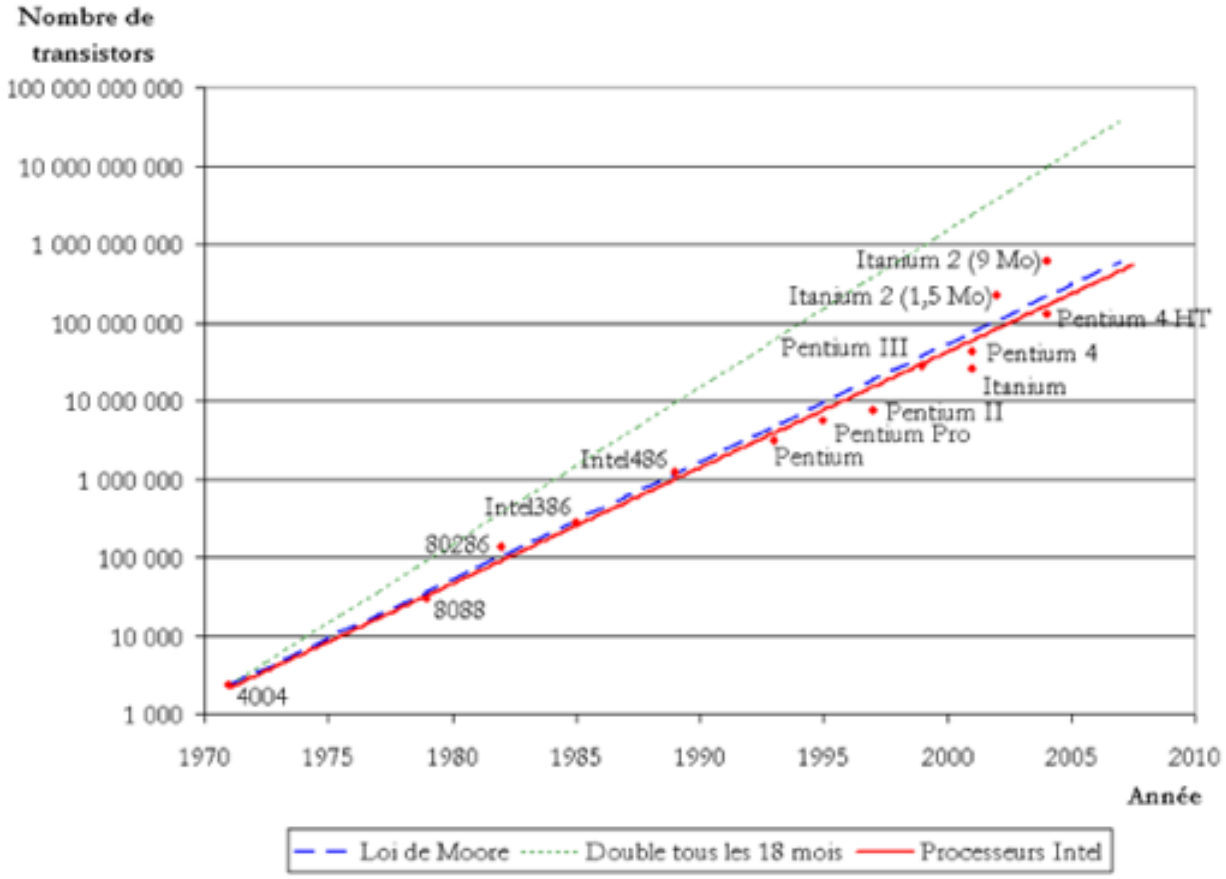

FIGURE 2.1 – Évolution de la capacité d'intégration des circuits intégrés

Partant de ces constatations, les architectures multi-core ont vu le jour. Le principe des architectures multi-core est le suivant : accroître le nombre des ressources de traitement (par exemple des processeurs ou des accélérateurs) sur une même puce et paralléliser les applications pour réduire le temps de calcul sans augmenter la fréquence de fonctionnement. Aujourd'hui il existe plusieurs types d'architectures parallèles multiprocesseurs, qui visent à améliorer le rapport performance / consommation énergétique. En effet, comparées aux architectures monoprocesseur et à performances égales, les architectures parallèles peuvent utiliser une fréquence de fonctionnement et une tension d'alimentation plus basses. Nous proposons dans cette thèse une méthode générale permettant d'évaluer d'une manière précoce la consommation énergétique d'une architecture multiprocesseurs intégrée sur puce avant fabrication, alors qu'on ne dispose que d'une description comportementale de type prototype virtuel.

2.1 Les Architectures Multiprocesseurs

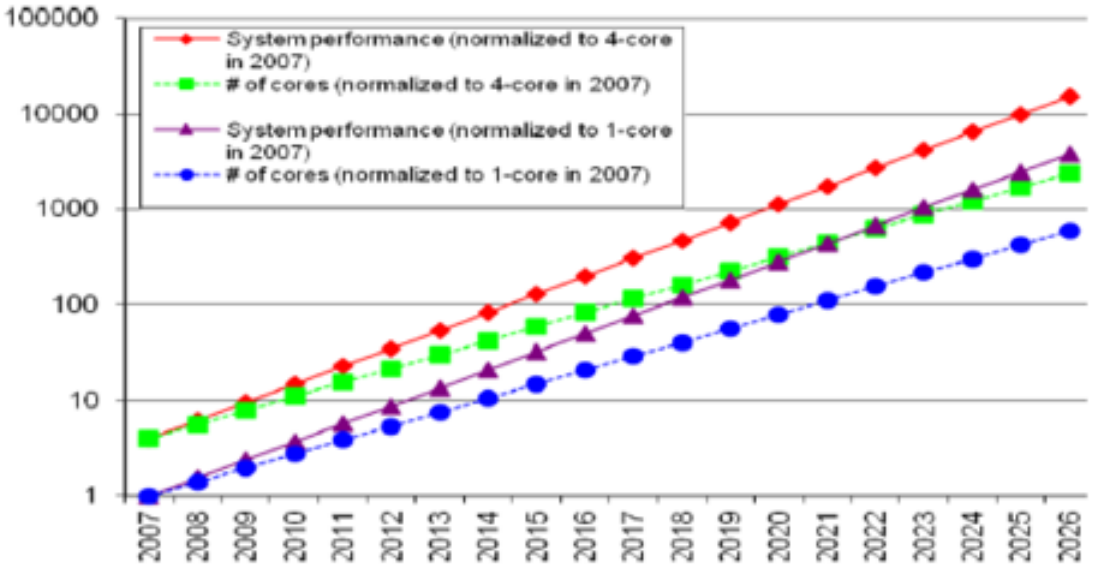

FIGURE 2.2 – Évolution du nombre de coeurs de processeurs embarqués sur une même puce (ITRS)

A titre d'exemple la figure 2.2 montre les prévisions de l'ITRS concernant l'évolution du nombre de coeurs de processeurs embarqués sur la même puce pour les architectures de type « network processor » utilisées dans les infrastructures réseaux. On peut donc s'attendre à une augmentation exponentielle du nombre de coeurs intégrés sur une seule puce dans les années à venir.

2.2 Classification des architectures multiprocesseurs

Il existe deux grandes classes d'architectures multiprocesseurs, suivant que les processeurs partagent ou non l'espace d'adressage entre les différentes tâches qui s'exécutent sur les unités de traitement.

2.2.1 Les architectures à mémoire distribuée

Dans une architecture multiprocesseurs à mémoire distribuée, chaque processeur possède son propre espace d'adressage et dispose donc d'une mémoire privée dans laquelle il est le seul à pouvoir lire et écrire. Les tâches s'exécutant sur deux processeurs distincts, communiquent entre elles par passage de message.

2.2.2 Les architectures à mémoire partagée

Dans les architectures à mémoire partagée, tous les processeurs accèdent au même espace d'adressage. La mémoire est logiquement partagée, cependant elle peut être physiquement distribuée sur la puce : il s'agit des architectures NUMA (Non Uniform Memory Access).

Dans ce cas l'architecture est souvent décomposée en sous systèmes, appelés clusters, où chaque cluster contient un petit nombre de processeurs et un contrôleur mémoire permettant d'accéder à une tranche de l'espace adressable. Dans le cadre de cette thèse, nous visons particulièrement les architectures manycore à mémoire partagée clusterisées de type NUMA pour deux principales raisons :

- Le caractère régulier de ces architectures facilite la modélisation énergétique. En effet on admet que modéliser une architecture clusterisée revient à modéliser un cluster et le réseau qui relie les différents clusters entre eux.
- La plupart des machines multiprocesseurs généralistes (ordinateurs personnels, serveurs de calculs) utilisent un modèle basé sur la mémoire partagée.

Pour comprendre comment modéliser la consommation énergétique au sein d'une architecture manycore à mémoire partagée de type NUMA, il faut d'abord comprendre les sources et l'évolution de la consommation dans de telles architectures.

2.3 Consommation énergétique dans les systèmes manycore

la consommation énergétique a toujours été un facteur important dans la conception des circuits intégrés aussi bien pour les systèmes fixes que pour les systèmes mobiles. Dans le cas des systèmes mobiles, minimiser l'énergie consommée permet d'augmenter la durée de vie des batteries et donc leur autonomie. Dans le cas des systèmes fixes, la performance du système est plus importante que son autonomie. Cependant, la performance et la consommation sont étroitement liées puisqu'augmenter la vitesse de traitement revient à augmenter la fréquence de fonctionnement et donc la puissance consommée ce qui augmente le coût du système de refroidissement.

L'énergie consommée est la somme de deux parties : une partie dynamique et une partie statique. La consommation dynamique est due à l'activité du circuit plus précisément au changement d'états des transistors tandis que la consommation statique est reliée aux courants de fuites. Les courants de fuites circulent entre la grille et

le substrat dès que le circuit est alimenté indépendamment du changement d'état du transistor. Les figure 2.3 (respectivement 2.4) montrent l'évolution des parties dynamiques et statiques de la consommation des systèmes fixes (respectivement des systèmes mobiles). On constate donc que la consommation totale ne cesse d'aug-

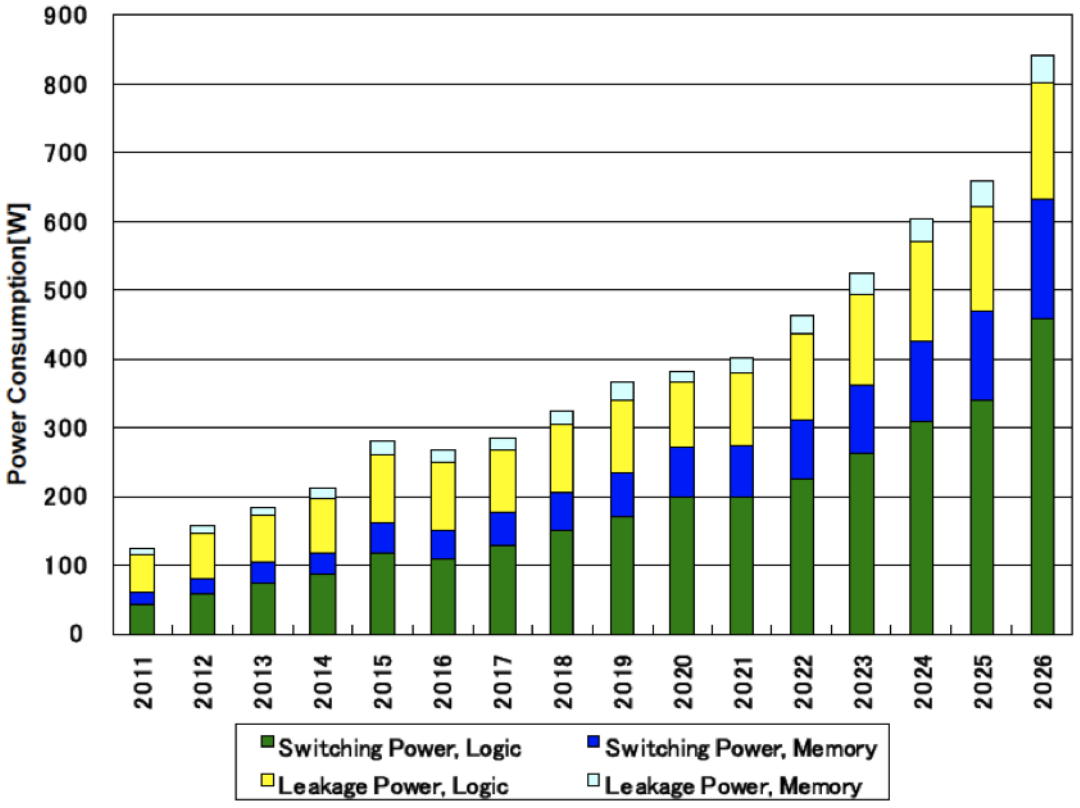

FIGURE 2.3 – Évolution de la consommation énérgétique dans les systèmes fixes (ITRS)

menter. Il est ainsi nécessaire de trouver les moyens de maitriser cette consommation et de l'évaluer.

2.4 Contrôle de la consommation énergétique

Au sein d'une puce, la puissance thermique dissipée par un transistor, lorsqu'il change d'état, se propage vers tous les transistors voisins. De ce fait, la température d'une unité ne dépend pas seulement de la puissance dissipée par cette unité, mais également de celle dissipée par les unités voisines. Quand la température de la puce augmente, ceci favorise la circulation des courants de fuites. On se retrouve ainsi dans un cercle vicieux qui lie la température aux courants de fuites. De plus le

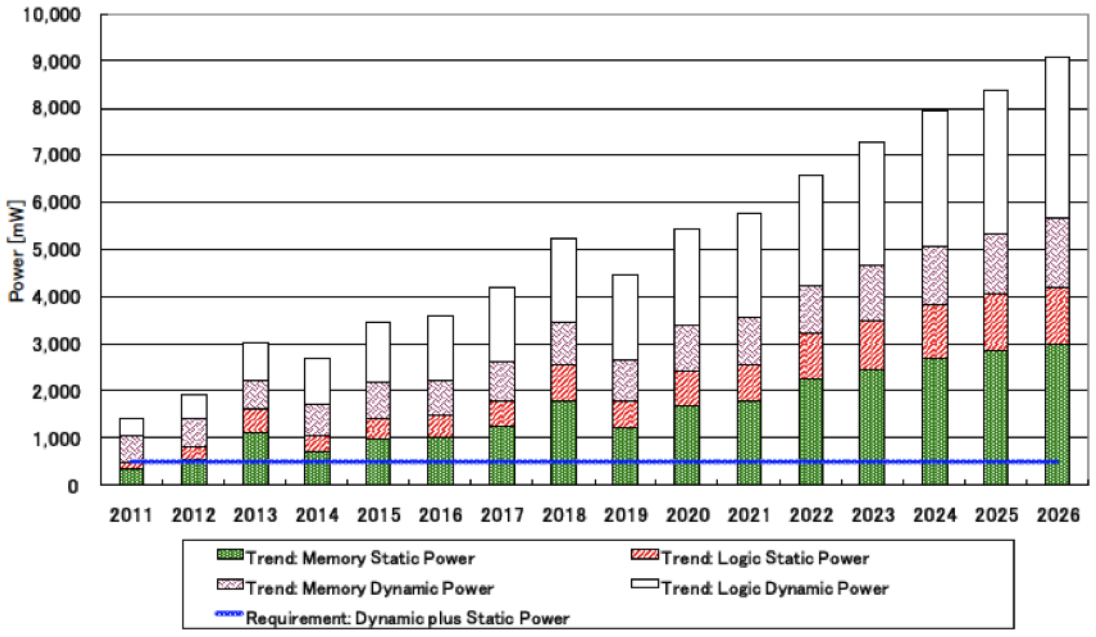

FIGURE 2.4 – Évolution de la consommation énérgétique dans les systèmes mobiles (ITRS)

phénomène de dissipation thermique ne se produit pas de la même façon au centre et sur les bords de la puce. En effet il est plus facile de dissiper la chaleur des unités qui se trouvent en périphérie. Au regard de toutes ces complications liées à la consommation des circuits intégrés, comment les architectures manycore peuvent-elles nous offrir à la fois de meilleures performances et une basse consommation ?
L'avantage des architectures manycore est qu'elles présentent généralement une régularité dans leur structure (exemple : découpage en clusters identiques). Elles offrent la possibilité de gérer le voltage et la fréquence de fonctionnement de chaque processeur ou de chaque cluster indépendamment des autres. Ainsi, il est possible d'éteindre les processeurs qui ne sont pas utilisés ou baisser leur fréquence de fonctionnement. Une telle technique s'appelle DVFS (Dynamic Voltage Frequency Scaling). Elle permet de diminuer la consommation des circuits intégrés pendant leur exécution. Une autre façon de diminuer la consommation consiste à inhiber le signal d'horloge de certains composants lorsqu'ils ne sont pas utilisés : il s'agit du «clock gating». Les deux méthodes DVFS et «clock gating» agissent sur la consommation du circuit pendant son fonctionnement «on line».

2.5 Évaluation précoce de la consommation énergétique

L'estimation de la consommation d'un système embarqué peut être réalisée à plusieurs niveaux d'abstraction. La précision de cette estimation est meilleure quand elle est réalisée sur une description du circuit proche de la réalisation physique telle que le niveau porte logique et le niveau RTL. Au niveau porte logique, la puissance dynamique est dissipée suite à un changement de valeur d'un signal c'est à dire d'un fil physique entre deux portes. Il suffit donc de détecter tous les événements de ce type pour évaluer la valeur totale de l'énergie consommée. L'estimation obtenue est très précise puisque l'on utilise une description du circuit fidèle à l'implémentation finale. Cette proximité du système final constitue à la fois un avantage et un inconvénient puisque elle permet d'avoir une bonne précision en contre partie d'un temps de simulation très long. Pour gagner en vitesse d'estimation, on peut utiliser une description plus gros grain du système tel que la description RTL (Resgister Transfer Level). Au niveau RTL le système est composé de registres. Pour modéliser la consommation à ce niveau il suffit de considérer le changement de l'état d'un registre comme source de dissipation d'énergie. Ce second type d'évènement étant plus abstrait, on perd cependant de précision. De plus, au niveau RTL, les temps de simulation et les efforts de codage restent importants. Avec l'arrivée des MPSoC, la nécessité d'une étude à des niveaux d'abstraction plus hauts de la puissance dissipée est devenue une évidence. La figure 2.5 montre que le plus tôt on intervient dans le cycle de fabrication des circuits intégrés pour minimiser la consommation, meilleurs sont les résultats. En effet, selon L'ITRS (International Technology Roadmap for Semiconductors), intervenir au niveau comportemental permet de réduire la consommation totale du système final de 40% contre 20% au niveau physique. L'étude de la consommation électrique au niveau comportemental relève du prototypage virtuel. Il existe plusieurs niveaux d'abstraction pour le prototypage virtuel suivant la précision avec laquelle on décrit les différents types de contention dans le matériel qui affectent les temps d'exécution : MISS sur les mémoires caches, bande passante limitée des bus, etc. A ce niveau les évènements significatifs du point de vue énergétique sont encore plus abstraits. Il s'agit par exemple de l'exécution d'une instruction par un processeur ou d'un MISS sur le cache processeur. Les descriptions comportementales les plus précises dites « Cycle-Accurate » modélisent précisément les caches et différents bus du système. Il est possible de simplifier encore plus la description comportementale du système considéré en se passant de la dimension temporelle. Il s'agit du niveau comportemental qui ne décrit pas les contentions.

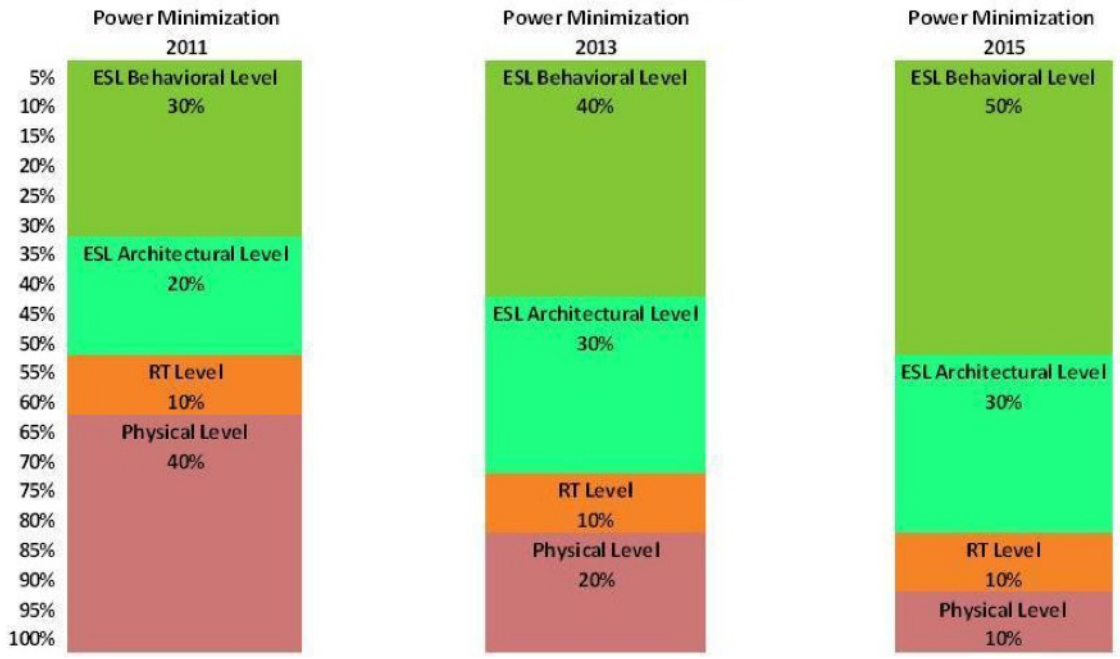

FIGURE 2.5 – Évaluation précoce de l'énergie

Pour prédire la consommation à ce niveau on peut supposer que les évènements qui engendrent la dissipation de la puissance sont à titre d'exemple les transactions de lecture et d'écriture dans la mémoire.

Dans le cadre de cette thèse, la description haut niveau du système étudié utilise la plate-forme de prototypage virtuel SoCLib présentée dans la section 2.6 qui permet une description comportementale «cycle accurate» ou proche du «cycle accurate».

2.6 SoCLib

SoCLib [2] est une plate-forme de prototypage virtuel permettant la modélisation et la simulation efficace de plate-formes multiprocesseurs à espace mémoire partagé. Le coeur de la plate-forme SoCLib est une bibliothèque de modèles de simulation pour les composants matériels (IP cores) constituant les briques de base de ces systèmes. Les modèles de simulation sont écrits en utilisant le langage SystemC. La plate-forme SoCLib fournit deux types de modèles de simulation

- Les modèles de niveau CABA (Cycle-Accurate and Bit-Accurate) [3], qui permettent une évaluation précise des performances.
- Les modèles de niveau TLM-T (Transaction Level Model with Timing) [4] qui permettent une réduction des temps de simulation au prix d'une légère perte de précision temporelle.

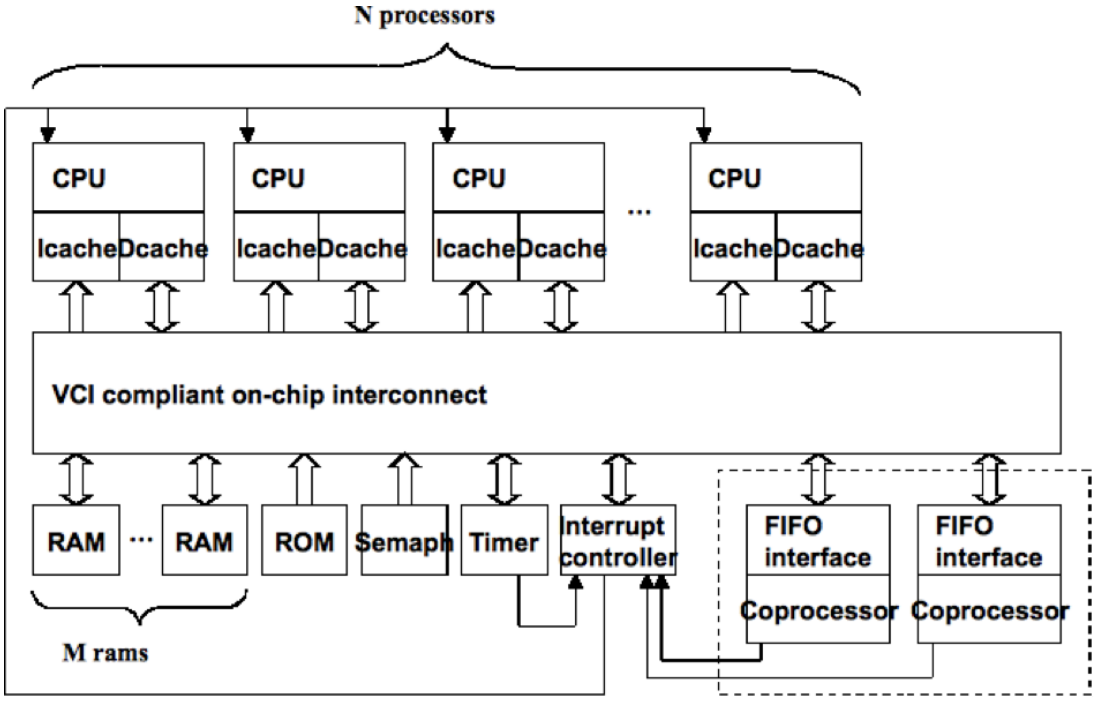

FIGURE 2.6 – Exemple d'architecture décrite avec SoCLib

En plus de cette bibliothèque, la plate-forme SoCLib fournit des outils logiciels aux concepteurs d'applications embarquées : accélérateurs de simulation, systèmes d'exploitation embarqués temps réel, outils de configuration, outils de déverminage et outils de qualification des modèles. Tous les composants matériels disponibles dans SoCLib respectent le protocole de communication VCI [5]. La plate-forme SoCLib possède deux caractéristiques qui nous intéressent particulièrement :

- La plupart des modèles sont génériques et il est donc possible d'ajuster différents paramètres matériels tels que la taille des caches, la capacité des bancs mémoires, ou bien la latence et le débit de l'infrastructure de communication. Tous ces paramètres jouent à la fois sur la performance et la consommation énergétique.
- Tous les composants matériels disponibles possèdent un modèle RTL synthétisable, ce qui permet donc la synthèse physique sur FPGA, ou sur ASIC.

La figure 2.6 montre un exemple de plate-forme multiprocesseurs qu'on peut décrire en utilisant SoCLib.

2.7 Le projet TSAR

L'architecture de référence utilisée dans cette thèse est l'architecture manycore TSAR.

TSAR [6] est une architecture à mémoire partagée dont la cohérence est assurée par le matériel. C'est une architecture NUMA (Non Uniform memory Access) qui supporte des systèmes d'exploitation généralistes de type UNIX (ou LINUX). Elle est composée d'un grand nombre de «petits» coeurs de processeurs RISC 32 bits. La figure 2.7 présente le principe général de l'architecture TSAR. En bas de cette figure on trouve d'abord les processeurs et leurs caches L1. Ils sont connectés à un réseau « L1toL2 » ayant une topologie de grille qui permet à chaque cache L1 de communiquer avec n'importe quel cache L2. Les caches L2, sont à leur tour reliés à un réseau d'interconnexion « L2toL3 » qui leur permet de communiquer avec n'importe quel cache L3 et pour finir les caches L3 sont connectés à un réseau externe qui leur permet d'accéder à la mémoire externe(voir figure 2.7). Cette architecture a été développée

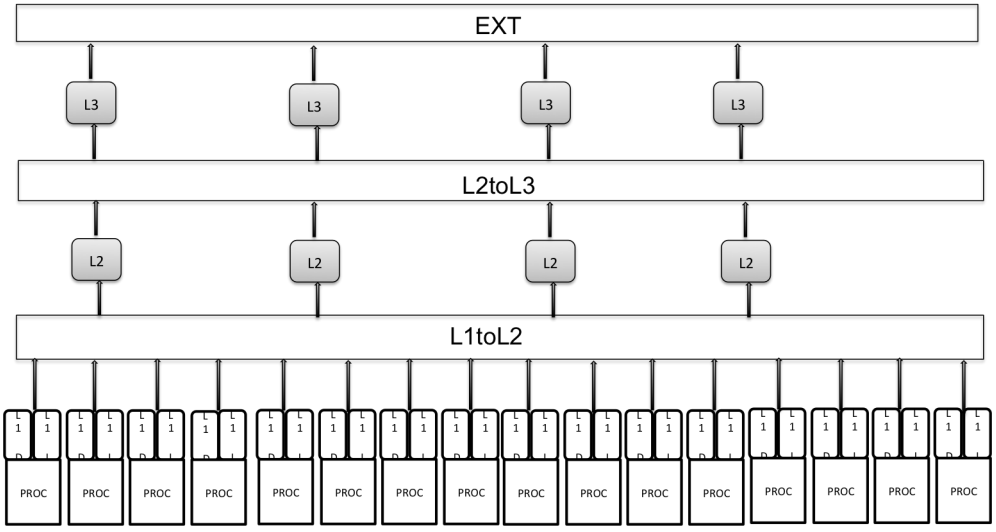

FIGURE 2.7 – L'architecture TSAR

conjointement par BULL, le LIP6 et le CEA-LETI, dans le cadre de deux projets européens CATRENE (TSAR et SHARP). En Référence à la classification présentée dans la section 2.2, il s'agit d'une architecture manycores symétrique à mémoire partagée cohérente. Le prototypage de cette architecture s'appuie uniquement sur la plate-forme SoCLib. L'objectif initial du prototypage virtuel était double :

- Le prototypage au niveau « cycle accurate » a permis la mise au point des automates matériels réalisant le protocole de cohérence mémoire entre les caches de premier et deuxième niveau.
- Il a également permis une évaluation précise des performances (nombre de cycles d'exécution), en particulier pour démontrer la scalabilité du protocole de cohérence, puisqu'on a pu exécuter les benshmarks SPLASH sur un prototype virtuel comportant 1024 coeurs.

L'exploration architecturale basée sur les modèles SystemC disponibles dans SoCLib,

permet certes d'optimiser l'architecture du système en terme de vitesse d'exécution mais ne permet pas d'évaluer la consommation énergétique. Le but de ce travail de thèse est d'introduire dans les modèles des composants de la bibliothèque SoClib la notion de consommation énergétique. Il s'agit donc d'établir des modèles de consommation pour des plate-formes décrites au niveau CABA ou TLMT en utilisant les composants de la bibliothèque SoCLib. A ce niveau d'abstraction il est possible de placer dans les modèles des différents composants matériels, des compteurs des événements susceptibles de générer une consommation d'énergie significative. Si on est capable d'associer à chaque évènement une énergie consommée, il devient possible d'évaluer la consommation énergétique d'une application particulière, en relevant les compteurs à la fin de la simulation. Il y a donc deux problèmes principaux à résoudre :

- Problème 1 : définition des évènements significatifs du point de vue énergétique au niveau d'abstraction choisi (CABA et TLMT).
- Problème 2 : caractérisation des composants matériels : cette phase consiste à déterminer, pour chaque composant matériel et pour chaque type d'évènement pouvant se produire dans ce composant, l'énergie effectivement consommée par cet événement lorsque le composant est actif.

La résolution de ces deux problèmes doit prendre en compte deux aspects : d'abord la pertinence de ces compteurs : Il faut démontrer la relation cause à effet entre le nombre d'évéments comptés et l'énergie effectivement consommée. Ensuite le niveau de précision des mesures dépend du nombre de compteurs retenus. Une fois le choix des compteurs fait, il reste à déterminer l'énergie consommée par chacun des types d'événements pour caractériser les composants. Étant donné qu'il existe pour tous les composants SoCLib un modèle RTL synthétisable sur FPGA, on a choisi d'utiliser les mesures physiques obtenues après la synthèse sur FPGA. Dans le cadre de ce travail on se propose donc de définir une méthode de caractérisation qui permet de remonter les informations de puissance disponibles au niveau RTL jusqu'au niveau CABA ou TLM-T.

2.8 Conclusion

Le travail réalisé dans cette thèse répond donc aux quatre questions suivantes :

- **Q1 : Quels sont les différents types d'évènements permettant de modéliser la consommation énergétique des composants matériels d'une architecture multiprocesseurs à espace mémoire partagée décrite au niveaux CABA ou TLM-T ?**

- **Q2 : Comment peut-on caractériser les différents composants, c'est à dire déterminer les paramètres physiques du modèle de consommation défini par la réponse à la question Q1**
- **Q3 : Quelle est la précision du modèle proposé ?**
- **Q4 : Quelles sont les caractéristiques à vérifier par des environnements autres que SoCLib pour que la méthode proposée reste applicable ?**

Chapitre 3

État de l'art

Sommaire

L'estimation de la consommation énergétique des systèmes manycores nécessite des outils capables d'évaluer les différentes sources de consommation dynamique et statique d'un système embarqué comportant des composants matériels et logiciels. Les évaluations peuvent être plus ou moins précises suivant le niveau d'abstraction dans lequel est décrit le système considéré. Plus on s'éloigne du niveau physique, moins les mesures sont précises. A contrario, l'estimation de la consommation au niveau physique est lente à effectuer comparée à une estimation réalisée à des niveaux

d'abstraction plus élevés. Cette différence est liée à la vitesse des simulateurs. À titre d'exemple dans [7], la méthode d'estimation de la consommation des SoCs au niveau transactionnel proposée est 1600 fois plus rapide qu'une estimation au niveau portes logiques. Dans le travail réalisé dans le cadre de cette thèse, on vise une méthode d'estimation de la consommation des systèmes manycores décrits aux niveaux CABA [3] et TLM [4]. Cependant, la difficulté d'une estimation précoce en SystemC de la puissance des systèmes sur puce réside dans l'absence de détails physiques à ces niveaux d'abstraction. Il faut donc un effort d'abstraction pour « remonter » dans le prototype virtuel les caractéristiques énergétiques des composants qui sont disponibles dans la description physique. Pour cette raison, il faut d'abord commencer par étudier les méthodes et les outils d'estimation de puissance qui interviennent à des niveaux d'abstraction proches du niveau physique. Ce chapitre s'organise comme suit : dans une première partie, nous nous intéressons aux méthodes d'estimation de la consommation des systèmes embarqués aux niveaux transistor, porte logique et transfert de registres (RTL). Dans une deuxième partie, nous décrivons les différentes approches d'estimation précoce de la consommation au niveau comportemental.

3.1 Les différentes approches d'estimation de la consommation des systèmes embarqués

Le processus de conception respecte généralement une approche descendante. la figure 3.1 montre les différents niveaux d'abstraction.

3.2 Estimation de la consommation bas niveau

3.2.1 Estimation de la consommation au niveau transistor

Les premiers travaux d'estimation de la consommation électrique se sont orientés vers le niveau d'abstraction contenant le plus de détails : le niveau transistor. À ce niveau d'abstraction, les modèles de consommations reposent sur les caractéristiques électriques des transistors et des capacités constituant le circuit. Il est donc indispensable dans cette approche de connaître le schéma électronique détaillé du circuit, leur tension d'alimentation et la fréquence de fonctionnement du circuit. Cette approche offre une très bonne précision en contrepartie d'un temps de simulation très important. Parmi les outils d'estimation de consommation au niveau

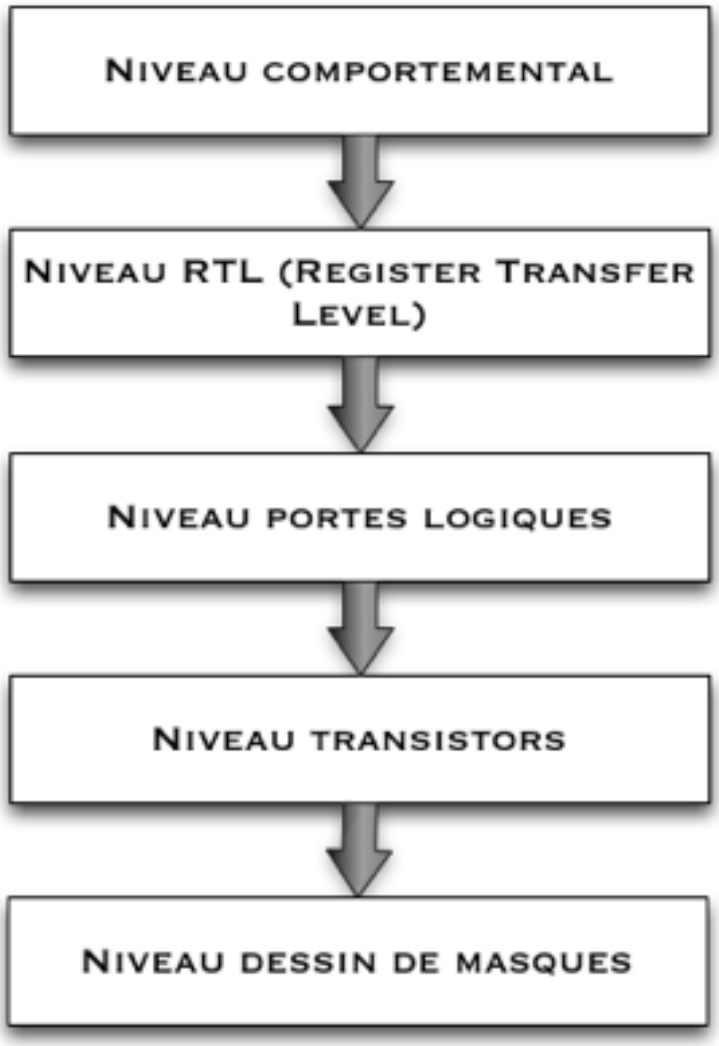

FIGURE 3.1 – Les niveaux d'abstraction d'un circuit embarqué

transistor, on retrouve SPICE [8] de Cadence, PowerMill [9] de Synopsys et Lsim Power Analyst [10] de Mentor graphics et Star-Power [11] de Avant !. Tous ces outils s'avèrent utiles quand il s'agit de concevoir des circuits simples. Néanmoins, avec l'apparition des nouvelles architectures multiprocesseur, les outils d'estimation de la consommation bas niveau se sont retrouvés confrontés à l'augmentation massive de la capacité d'intégration sur les puces.

3.2.2 Estimation de la consommation au niveau portes logiques

À ce niveau d'abstraction, le circuit est composé d'un ensemble de portes logiques. Une porte logique est le regroupement de plusieurs transistors. L'utilisation des portes logiques permet de diminuer la complexité des circuits. À ce niveau d'abstraction, la consommation du circuit est égale à la somme des consommations des portes logiques composant le circuit. La consommation d'une porte logique dépend de sa tension d'alimentation et de sa fréquence de fonctionnement. Parmi les outils académiques et industriels qui reposent sur des méthodes d'estimation niveau porte logique, il existe : PowerGate [12] (Synopsys), DesignPower [13] (Synopsys), QuickPowerr [14] (Mentor Graphics),PowerCalc [15] (Zimmerman), PowerCompiler [16](Synopsys) et Diesel [17] (Philips). Ce niveau d'abstraction permet de réduire le temps de simulation grâce à une diminution de la complexité, mais nécessite d'aller jusqu'au placement/routage pour avoir une évaluation fiable des capacités des fils d'interconnexion.

3.2.3 Estimation de la consommation au niveau RTL (Register Transfer Level)

Dans une description au niveau transfert de registre (RTL), un circuit est composé d'éléments séquentiels tels que des registres reliés entre eux par des éléments combinatoires. Une description RTL reste assez proche de la réalisation matérielle, puisqu'elle décrit explicitement les valeurs stockées dans les registres à chaque cycle. L'erreur de précision d'un modèle décrit à ce niveau d'abstraction est typiquement comprise entre 10 et 15 % par rapport à des estimations réalisées avec SPICE [8]. Parmi les outils d'estimation de la consommation au niveau RTL, on pourra citer Petrol [18] de philips, DesignPower [13] de Synopsys, QuickPower [14] de Mentor et PowerPlay [19] de Altera. L'erreur à ce niveau est beaucoup plus importante puisqu'on ne dispose pas d'informations précises sur les capacités des fils.

3.2.4 Conclusion

Pour être précises, les méthodes bas niveaux nécessitent une représentation du circuit comme un schéma faisant apparaître explicitement les fils d'interconnexion S_i entre les portes logiques (gate-level netlist) ou entre les transistors (transistor-level netlist).

Dans un cas comme dans un autre, il faut connaître précisément la capacité C_i électrique de chaque signal S_i car les évènements significatifs sont les transitions de ces signaux S_i qui doivent charger ou décharger les capacités C_i.

Du point de vue énergétique, le rôle de la simulation consiste simplement à compter le nombre de transitions de chaque signal S_i. Pour élever le niveau d'abstraction, il faut donc définir de nouveaux types d'évènements, plus macroscopiques que les transitions de signaux de la netlist, et la difficulté sera évidemment de relier ces évènements macroscopiques aux évènements microscopiques du circuit réel.

3.3 Estimation haut niveau

Certaines solutions ont choisi d'instrumenter le code en vue d'une estimation précoce de l'énergie dissipée par l'exécution de ce dernier sur une plate-forme donnée. Ces solutions sont incomplètes dans la mesure où une bonne estimation de la consommation doit considérer à la fois le logiciel et le matériel. Une autre idée consiste à étudier les caractéristiques énergétiques du circuit pendant l'étape de prototypage virtuel, ce qui permet de faire de l'exploration architecturale en se basant non seulement sur les critères de performance mais aussi de la puissance consommée. L'estimation de la consommation durant l'étape de prototypage virtuel consiste tout d'abord à déterminer les évènements dont la réalisation engendre une consommation d'une certaine quantité d'énergie. Par la suite, il s'agit d'associer une énergie à chacun de ces évènements ; on parle alors de caractérisation. Enfin la validation de ces modèles de consommation consiste à prouver leur capacité à prédire la consommation énergétique d'une plate-forme lorsque l'on change l'application logicielle. Dans cette section, on présentera quelques–unes de ces solutions, celles qui s'approchent le plus du travail réalisé dans le cadre de cette thèse. La différence entre les solutions existantes réside dans la finesse des modèles proposés ainsi que la méthode de caractérisation des différents modèles.

3.3.1 WATTCH :

WATTCH [20] est un des premiers outils académiques d'estimation de la consommation au niveau système. Le premier article décrivant cet outil a été publié en 2000. Cet outil a été réalisé au sein l'université de Princeton en collaboration avec Intel. WATTCH permet une analyse ainsi qu'une optimisation de la puissance d'un microprocesseur avec une accélération d'un facteur 1000 comparé aux outils au niveau

dessin de masque, et une erreur de moins de 10%. Cet outil fournit des modèles de consommation paramétrables pour les composants d'un microprocesseur. L'énergie totale 3.1 consommée par un système multiprocesseur est la somme des énergies consommées par des différents sous blocs du système.

$$E_T = \sum_i E_T(i) \tag{3.1}$$

où i définit le type de composant.

L'énergie 3.2 consommée par chaque sous bloc est égale au produit du nombre d'occurence d'un évenement $N(i)$ par l'énergie élémentaire consommée à chaque occurence de cet évenement $e(i)$.

$$E_T(i) = N(i,j) * e(i,j) \tag{3.2}$$

où i définit le type de composant et j définit le type d'évènements

WATTCH propose un modèle énergétique du microprocesseur composé de douze types d'évenements i qui correspondent aux accès aux douze sous blocs qui le composent, illustrés par le tableau 3.1. L'énergie $e(i)$ consommée par chaque évenement i est donnée par 3.3.

$$E = C * Vdd^2 \tag{3.3}$$

avec Vdd la tension d'alimentation et C la capacité de chargement. Vdd dépend de la technologie du circuit cible, quant à C, les auteurs affirment qu'elle est estimée en utilisant quatre méthodes différentes suivant la nature de l'unité (structure matricielle, mémoire associative, logique combinatoire et l'horloge). Cependant, ils n'expliquent pas comment sont obtenues les énergies élémentaires correspondant à chacun des douze évènements tels que l'accès au cache. Les $N(i)$ sont obtenus à partir de la simulation du microprocesseur au niveau architectural.

3.3.1.1 Conclusion

WATTCH offre un bon compromis entre la précision et la vitesse de simulation ; cependant il ne modélise pas toutes les parties qui composent le système, notamment les interconnexions. D'un autre côté la méthode de caractérisation n'est pas homogène : en effet, la détermination de l'énergie $E_T(i)$ consommée à chaque accès à un sous bloc utilise quatre différents modèles et ce suivant la nature de chaque module (ie : logique combinatoire, cache, mémoire ou horloge) ce qui la rend difficilement généralisable. D'autant plus les auteurs restent assez évasifs sur la façon avec

laquelle ils calculent l'énergie associée élémentaire, correspondant à un évènement donné, à partir de l'énergie totale du sous bloc considéré. Nous pensons donc que outre l'abscence de la modélisation des interconnexions, la caratérisation constitue également le point faible de WATTCH.

cache d'instruction
logique wakeup (wakeup logic)
logique issue selection (Issue Selection Logic)
fenêtre d'instruction (instruction window)
precdicteur de branchement
registre
TLB (translation lookaside buffer)
load - store queue
cache de données
unité de calcul entier
unité de calcul flottant
horloge globale

TABLE 3.1 – les composants matériels d'un microprocesseur selon WATTCH

3.3.2 AVALANCHE

Parmi les outils les plus connus qui reposent sur le prototypage virtuel on retrouve également l'outil AVALANCHE [21] qui a fait l'objet d'une publication en 2002 par le laboratoire NEC. AVALANCHE propose des modèles de consommation pour les composants suivants : le processeur, le cache et la mémoire.

3.3.2.1 Les modèles de consommation

Le modèle du cache

Le modèle de puissance du cache dépend de quatre types d'évenements i qui dépendent à la fois du types d'accès (read, write) et de la réponse du cache (hit, miss). L'énergie totale du cache est alors exprimée sous la forme suivante :

$$E_T(i) = \sum_j N(j) * e(j) \qquad \text{où } j \text{ est l'indice de type d'évènement} \tag{3.4}$$

Les énergies élémentaires e(i) pour les quatre types d'évènements sont respectivement :

$$e(1) = 1/2 * Vdd^2 * (Cbit_{rd} + Cword) \tag{3.5}$$

$$e(2) = 1/2 * Vdd^2 * Cbit_{wr} \tag{3.6}$$

$$e(3) = 1/2 * Vdd^2 * Cdec \tag{3.7}$$

$$e(4) = 1/2 * Vdd^2 * Cod \tag{3.8}$$

$Cbit_{rd}$ et $Cbit_{wr}$, $Cdec$, Cod et $Cword$ sont les capacités de charge ou de décharge suite à l'exécution de certaines opérations :

- $Cbit_{rd}$: lecture d'un bit.
- $Cbit_{wr}$: écriture d'un bit.
- $Cdec$: décodage.
- Cod : écriture en sortie du cache.
- $Cword$: écriture ou lecture d'un mot.

La caractérisation des quatre paramètres du modèle consiste à déterminer les capacités citées ci-dessus. Les auteurs ont seulement mentionné que cette dernière a été réalisée, dans le cas du cache, à l'aide d'un outil de synthèse bas niveau CACTI [22].

Le modèle de la mémoire

le modèle de puissance de la mémoire dépend d'un seul type d'évènement qui est l'accès à cette dernière. l'énergie totale consommée par la mémoire peut être exprimée sous la forme :

$$E_T(i) = N(i) * e(i) \tag{3.9}$$

La caractérisation de l'accès à la mémoire consiste à déterminer e. Les auteurs définissent cette énergie comme suit :

$$e = Vdd * I * t \tag{3.10}$$

avec I le courant consommé à chaque accès à la mémoire. Ce dernier est donné par 3.11.

$$I = m * Iact + m * (n-1) * Ihld + m * Idec + Iperi \tag{3.11}$$

$Iact$ correspond au courant qui circule dans les m cellules actives par accès, $Ihld$ correspond au courant nécessaire pour stocker les données dans les $m*(n-1)$ cellules non actives, $Idec$ est le courant utilisé par le décodeur et $Iperi$ le courant du circuit périphérique. La méthode d'évaluation de ces courants n'est pas précisée. Le nombre d'accès N est obtenu à partir de la simulation architecturale.

Le modèle du processeur

Le modèle énergétique du processeur définit deux types d'évènements :

$$E_{proc} = E_0 * N_0 + E_1 * N_1 \tag{3.12}$$

N_0 et N_1 sont respectivement le nombre de cycles de gel du processeur et le nombre de cycles durant lesquels le processeur est actif. N_0 et N_1 sont obtenus à partir d'un simulateur ISS. Les énergies élémentaires E_0 et E_1 sont données respectivement par 3.13 et 3.14

$$E_0 = Tw_c * V_{dd} * I_{nop} \tag{3.13}$$

$$E_1 = Tw_c * V_{dd} * I_{inst} \tag{3.14}$$

avec

1. Tw_c : le temps nécessaire pour l'exécution de l'application déployée.
2. I_{nop} : le courant consommé pendant un cycle de gel du processeur.
3. I_{inst} : le courant consommé pendant un cycle actif du processeur.

La détermination des intensités de courant I n'est pas détaillée, les auteurs font référence à des travaux réalisés par Tiwari dans [23].

3.3.2.2 Conclusion

AVALANCHE offre un bon compromis entre la précision et la vitesse d'estimation quand il s'agit d'estimer l'énergie dissipée par le processeur seul, cependant la précision est vite dégradée quand il s'agit de considérer un système plus complet. En effet, la consommation de puissance du système calculée avec AVALANCHE ne prend pas en compte la consommation des interconnexions. Les modèles des différents composants sont caractérisés de manières différentes les uns des autres ce qui rend la caractérisation difficilement généralisable.

3.3.3 PowerViP

PowerViP [7] est un outil académique conçu pour une estimation de la consommation au niveau architectural. Cet outil a été développé par Samsung Electronics

en collaboration avec l'université de Yonsei en Corée et a fait l'objet d'une publication en 2006. PowerViP a été intégré au niveau d'un simulateur au niveau TLM, ViP [24].

3.3.3.1 Les modèles de consommations

Il décompose le SoC en processeurs, mémoires, bus et des *blocs IP* correspondant à des coprocesseurs spécialisés. Il adopte une stratégie de modélisation par composant.

Le modèle du processeur

Comme pour AVALANCHE, le processeur est modélisé par deux états *busy* et *idle*.

Le modèle du cache

L'architecture du cache utilisé dans le cadre de ce travail distingue deux types d'accès : séquentiel et non séquentiel. Un prédicteur est employé pour les accès séquentiels, ce qui implique qu'un accès séquentiel fait toujours hit. L'accès non séquentiel nécessite lui la lecture de toutes les voies du répertoires et des parties données du cache. L'accès au cache peut dans ce cas être un succès ou un échec. Le transfert des données de la mémoire vers le cache est réalisé via un tampon. La copie des données du tampon vers le cache nécessite deux cycles.

Dans le cas d'un cache associatif à quatre voies on distingue six types d'évenements au lieu de quatre pour AVALANCHE :

- Lecture séquentielle : lecture d'une voie de la partie donnée du cache.
- Échec de lecture ou succès de lecture non séquentielle : lecture des 4 voies du répertoire et des 4 voies de la partie données du cache.
- Succès d'écriture : lecture des 4 voies du répertoire, une écriture dans le répertoire et une écriture de donnée.
- Échec d'écriture : lecture des 4 voies du répertoire.
- Écriture du tampon vers le cache (1er cycle) : une lecture du répertoire et 4 écritures de données.
- Suite de l'écriture du tampon vers le cache (2e cycle) : 4 écritures de données.

Le modèle du cache est donné par :

$$E_T(i) = \sum_j N(j) * e(j) \qquad \text{où } j \text{ est l'indice de type d'évènement} \tag{3.15}$$

La caractérisation des modèles du processeur et du cache

La méthode de caractérisation suppose qu'à chaque cycle, un seul type d'évènement est susceptible d'avoir lieu. En partant de cette supposition, l'énergie élémentaire correspondant à l'évènement i est alors obtenue en calculant la moyenne de la somme des énergies consommées durant tous les cycles où un évènement i a eu lieu. Les énergies consommées, à chaque cycle, par l'évènement i sont obtenues à l'aide d'un outil de synthèse niveau portes logiques qui prend en entrée la forme d'ondes des différents signaux de la plate-forme pour un *benchmark* donné. Cette méthode est applicable dans le cas du processeur mais elle est difficilement généralisable dans le cas d'autre composant tel que le cache. En effet, les auteurs n'expliquent pas comment peut ton extraire l'énergie consommée par un type d'évènement tel que la transaction de lecture.

Le modèle du bus

Le bus est modélisé par deux types d'évènements : transaction de lecture et transaction d'écriture. Chaque évènement correspond à une combinaison d'activation ou non des différents composants du bus (décodeur, routeur, arbitres). L'énergie élémentaire correspondant à une transaction de lecture ou d'écriture est la somme des énergies consommées par tous les blocs du bus qui sont sollicités durant cette opération. Les énergies des différents sous blocs sont mesurées par un outil d'estimation de consommation au niveau portes logiques. On en déduit par la suite l'énergie consommée par la transaction de lecture ou d'écriture en sommant les énergies consommées par les sous blocs solicités pendant cette transaction.

Le modèle de la mémoire

Le modèle de la mémoire utilise la technique proposée dans [25] et qui consiste à compter cinq types d'évènements : lecture, écriture, activation, pré-chargement, et mise à jour. La consommation correspondant à chacune de ces opérations est obtenue directement à partir des datasheets de la SDRAM.

Le modèle des IPs

Le modèle des IPs n'est pas explicité, cependant les auteurs précisent que les macromodèles sont élaborés au niveau RTL. L'énergie totale est également obtenue à partir de la description RTL du composant. La méthode de caractérisation des macro-évènements du modèle de l'IP n'est pas décrite.

3.3.3.2 Conclusion

L'outil PowerVip répond au problème majeur que nous cherchons à résoudre à travers ce travail de thèse qui est l'estimation précoce et précise de la consommation

des systèmes embarqués. Cependant, par soucis de précision les concepteurs de cet outil ont opté pour une méthode de caractérisation différente pour chaque composant de la plate-forme étudiée : datasheets pour la mémoire et des outils d'estimation de puissance au niveau RTL pour le reste. D'autre part PowerViP utilise une autre architecture dans l'élaboration du modèle du bus. Nous pensons que cette hétérogénéité augmente la complexité temporelle de la méthode : i.e. plus de temps pour le développement et la validation de modèles et rend difficile le développement de modèles de consommation pour d'autres composants matériels que les composants existants.

3.3.4 MCPAT

MCPAT [26] est un framework académique d'estimation de consommation énergétique des SoCs issu d'une collaboration entre l'université Notre Dame, l'université de la Californie et l'université nationale de Seoul et qui a fait l'objet d'une publication en 2009. Il offre la possibilité de réaliser la modélisation temporelle, spatiale et énergétique des architectures multicores en vue d'une exploration architecturale.

3.3.4.1 Les modèles de consommation

Il modélise les coeurs de processeur de type *in-order* et *out-of-order*, les réseaux sur puce, les caches partagés et les contrôleurs de mémoire. La méthode de modélisation MCPAT repose sur une approche à trois niveaux. Il s'agit de modéliser une architecture du type manycore d'abord au niveau architectural, ensuite au niveau circuit et enfin au niveau technologique. Les modèles de consommation au niveau architectural de chacun des composants matériels de la plate-forme ont tous un seul paramètre qui est le nombre d'accès à chacun d'eux. Quelque soit le composant, il est modélisé par :

$$E_T(i) = N(i) * e(i) \tag{3.16}$$

La caractérisation est réalisée en représentant chaque module de l'architecture manycore en sous–blocs de plusieurs types : fils, structures de tableau, logique combinatoire et arbre d'horloge. La consommation de chacun de ces blocs est ensuite déterminée par différentes techniques : soit par des outils de mesure de consommation bas niveau tels que CACTI [22], soit par des méthodes de modélisation proposées notamment par Intel [27] ou Sun [28].

Les sources de dissipation d'énergie mesurées par MCPAT sont la puissance dynamique, les courts circuits et les courants de fuites. Les courts circuits et les courants

de fuites sont calculés par des méthodes développées par Nose et al. dans le cadre de [29]. À titre d'exemple pour calculer l'énergie consommée par un accès au processeur, ce dernier est décomposé en plusieurs unités : IFU (Instruction Fetch Unit), EXU (Execution Unit) et LSU (Load and Store Unit). Ensuite chaque unité est décomposée en sous-systèmes : ALU, FPU, registres. L'énergie élémentaire est la somme des énergies consommées à chaque activation de ces sous-blocs.

3.3.4.2 Conclusion

MCPAT permet donc une estimation précise (l'erreur moyenne par composant est de 23 %) des architectures manycore en proposant des modèles qui utilisent à la fois les caractéristiques physiques fournies par le niveau circuit et les activités fournies par le simulateur niveau architectural. Cependant, l'approche de modélisation proposée par MCPAT, utilise des techniques de caractérisation différentes pour chaque composant bien que les modèles de consommation soient tous simples (i.e. à deux états). Nous pensons qu'il est important d'adopter la même démarche pour tout le système afin de faciliter la modélisation et éventuellement la modélisation de nouveaux composants.

3.3.5 HSL (Hybrid System Level Power Consumption Estimation)

HSL [30] est une méthode académique d'estimation hybride de la consommation des systèmes sur puce. Elle a été développée en collaboration entre l'université de Lille, l'université de Valenciennes et l'université de Bretagne Sud et a fait l'objet d'une publication en 2011. Les modèles de consommation proposés reposent sur la méthode FLPA (Functional Level Power Analysis).

3.3.5.1 Les modèles de consommation

Dans HSL, pour estimer la consommation d'un système multiprocesseur, il faut d'abord le décomposer en plusieurs blocs fonctionnels (mémoires, {processeur, cache} et bus). Des modèles de puissance sont ensuite développés pour chacun de ces blocs. Ces modèles ne sont pas explicités dans cet article [30]. En revanche les auteurs proposent une méthode de caractérisation intéressante que nous analysons ci-dessous.

3.3.5.2 La caractérisation

Dans HSL, la caractérisation est obtenue en réalisant plusieurs mesures physiques après synthèse sur FPGA. Pour obtenir les énergies élémentaires, différentes applications sont déployées sur l'architecture permettant en principe de simuler indépendamment chaque bloc du système. Ensuite la consommation totale du système est mesurée pour chaque application logicielle. Enfin, les énergies élémentaires du modèle sont obtenues par régression. La régression consiste à modéliser, formellement la relation entre l'énergie totale E_T et le nombre d'occurrence des différents types d'évènements N_i. De manière générale, le modèle linéaire peut s'écrire de la manière suivante :

$$E_T(i) = \sum_j N(i,j) * e(i,j) \tag{3.17}$$

où i définit le type de composant et j définit le type d'évènements.

L'équation 3.17 peut être écrite pour chacune des applications logicielles déployées sur le système considéré. Si l'on considère K le nombre d'applications logicielles, L le nombre total de types d'évènements, $N_{k,i}$ le nombre d'occurrences d'un évènement i lorsque le logiciel K est déployé, e_i l'énergie élémentaire correspondant à l'évènement i et $E_{T,i}$ l'énergie totale du système lorsque l'on exécute l'application i dessus, on obtient le système 3.18.

$$\begin{bmatrix} E_{T,1} \\ E_{T,2} \\ E_{T,3} \\ \vdots \\ E_{T,K-1} \\ E_{T,K} \end{bmatrix} = \begin{bmatrix} N_{1,1} & N_{1,2} & ... & N_{1,L} \\ N_{2,1} & N_{2,2} & ... & N_{2,L} \\ N_{3,1} & N_{3,2} & ... & N_{3,L} \\ & \ddots & \ddots & \\ N_{K-1,1} & N_{K-1,2} & ... & N_{K-1,L} \\ N_{K,1} & N_{K,2} & ... & N_{K,L} \end{bmatrix} \times \begin{bmatrix} e_1 \\ e_2 \\ e_2 \\ \vdots \\ e_{L-1} \\ e_L \end{bmatrix} \tag{3.18}$$

La caractérisation consiste maintenant à résoudre le système précédent afin de determiner les valeurs des e_i. La résolution peut se faire soit de manière exacte si $L = K$, soit en approchant les solutions si $L > K$ ce qui est généralement le cas quand il s'agit d'un système obtenu à partir de données expérimentales. Dans le cas des systèmes issus de mesures physiques il faut considérer le problème de conditionnement.

Conditionnement

Le conditionnement mesure la dépendance de la solution d'un système linéaire par rapport aux données du problème. En effet, les données obtenues à partir de

mesures expérimentales sont généralement entachées d'erreurs. Un système possédant un conditionnement bas est dit bien conditionné et un un système possédant un conditionnement élevé est dit mal conditionné. Le conditionnement de la matrice A est défini par :

$$K(A) = ||A^{-1}|| * ||A|| \tag{3.19}$$

Avec $||.||$ est la norme subordonnée de la matrice. Plus K est faible, meilleur est le conditionnement.

En pratique, pour avoir un bon conditionnement, il faut parvenir à exciter certains compteurs plus que d'autres afin de maximiser le rapport $signal/bruit$, ce qui est difficile à réaliser.

3.3.5.3 Conclusion

HSL utilise la technique de régression pour caractériser les énergies élémentaires $e(i)$. Cette technique a le mérite d'être très générale. Cependant l'efficacité de la caractérisation par régression nécessite l'écriture de vecteurs de test (c'est-à-dire différentes applications logicielles) capables d'activer un composant à la fois. Ces vecteurs de test n'ont pas été présentés par les auteurs, cependant nous pensons qu'il est très difficile de dissocier le fonctionnement de plusieurs composants, en particulier le processeur et son cache. Cette tâche devient d'autant plus difficile à réaliser quand il s'agit de plate-forme manycore.

3.4 Conclusion

L'estimation de la puissance peut être réalisée à différents stades du processus de conception d'une architecture manycore. Le travail de cette thèse s'inscrit dans le cadre d'une estimation de la consommation des architectures multiprocesseur pendant la phase de prototypage virtuel. Dans ce chapitre, nous avons présenté différentes méthodes et outils qui proposent une estimation précoce de la consommation des systèmes embarqués. Ces travaux ont été évalués de trois points de vue :

- Les modèles de consommation du point de vue de leur précision et du nombre de leurs paramètres.
- La méthode de caractérisation des modèles.
- L'homogénéité de la méthode.

Toutes les méthodes d'estimation précoce de la consommation utilisant le prototypage virtuel reposent sur le même principe : il s'agit d'utiliser le prototypage virtuel pour compter, pour chaque composant matériel, le nombre d'évènements énergétiquement significatifs qui se sont produits pendant l'exécution d'une certaine application. Il faut donc définir pour chaque composant matériel un « modèle de consommation macroscopique » qui est simplement un ensemble de « type d'évènements macroscopiques » qui pourront être comptabilisés dans le modèle de simulation du composant matériel modélisé. L'énergie dissipée par un composant i est donc :

$$E_T(i) = \sum_j N(i,j) * e(i,j) \tag{3.20}$$

où l'indice j porte sur les types d'évènements macroscopiques du composant i.

Tous les travaux présentés dans ce chapitre décrivent de façon assez claire les modèles de consommation des différents composants (c'est à dire les compteurs $N(i,j)$) mais décrivent généralement très mal la méthode de caractérisation permettant de déterminer les valeurs des paramètres $e(i,j)$. Or cette étape de caractérisation est le point clé, puisqu'il faut passer d'une description très bas niveau où un « évènement » est une transition d'un signal capacitif entre deux portes logiques à un niveau d'abstraction nettement plus élevé, où un « évènement macroscopique » est par exemple une requête d'écriture *MISS* sur un cache d'instruction. Lorsqu'elles sont détaillées, les méthodes de caractérisation présentées sont généralement spécifiques à chaque composant car le processus d'abstraction décrit ci-dessus est spécifique à chaque type de composant. Pour conclure, toutes les méthodes que nous avons présentées dans ce chapitre, décomposent le système en plusieurs composants, identifient les évènements pertinents du point de vue énergétique, puis déterminent les énergies élémentaires par composant. Cependant, toutes ces méthodes ont deux principales limitations :

1. Les modèles de consommation sont soit trop fins et reposent donc sur plusieurs compteurs, soit très gros grains et donc peu précis.
2. La méthode de caractérisation n'est pas unifiée pour tout le système.

Dans ce travail de thèse nous essayerons d'examiner de plus près ces deux axes en essayant d'apporter la réponse aux questions suivantes :

Q1 : Quelle est le meilleur compromis entre la précision d'un modèle de consommation et la facilité de sa caractérisation ?

Q2 : À quel point la modélisation énergétique lors du prototype virtuel d'un système embarqué, peut-elle être généralisée ?

Chapitre 4

EDPE : Méthode d'estimation précoce de la consommation des architectures MPSoCs

Sommaire

Dans ce chapitre nous décrivons la méthode EDPE pour Early Design Power Estimation qui permet une estimation précoce de la consommation énergétique d'une plate-forme {matériel + Logiciel} décrite au niveau CABA ou TLM.

4.1 EDPE : Principe

La méthode proposée définit un plan de route qui permet de construire le modèle énergétique d'une plate-forme matérielle MPSoC. EDPE permet une exploration architecturale qui prend en compte la consommation énergétique de la plate-forme pendant la phase de prototypage virtuel.

Le principe de la méthode est représenté sur la figure 4.1.
Notre méthode de caractérisation comporte six étapes :

1. Définir l'architecture matérielle ciblée.
2. Définir les types d'évènements pertinents d'un point de vue énergétique pour chacun des composants de la plate-forme puis rajouter des compteurs de ces évènements dans les modèles virtuels de chacun des composants de la plate-forme.
3. Définir une série d'applications logicielles caractéristiques permettant d'activer de façon différenciée les différents types d'évènements.
4. Mesurer la consommation énergétique de la plate-forme pour chacune de ces applications caractéristiques.
5. Caractériser les différents types d'évènements. En pratique il s'agit de résoudre le système obtenu en associant à chaque type d'évènement une énergie élémentaire caractéristique.

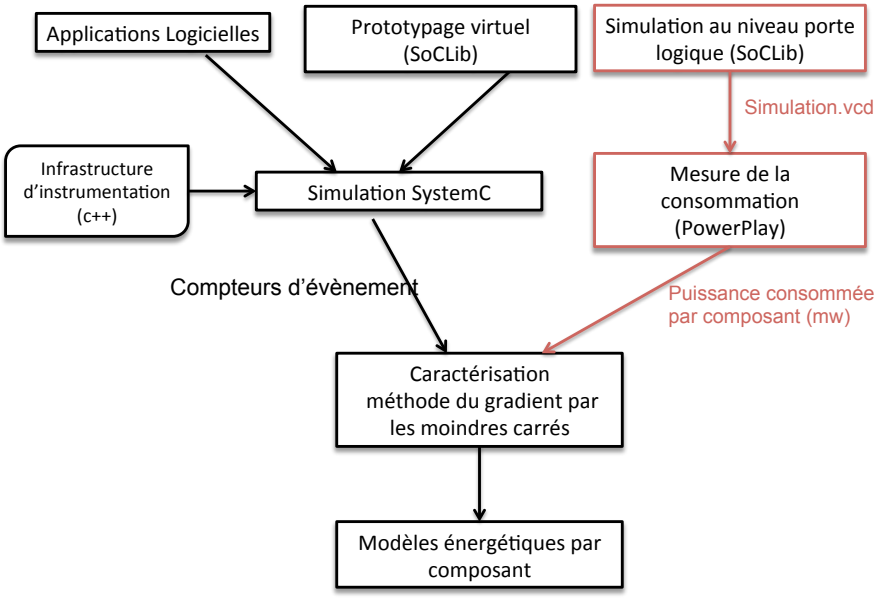

FIGURE 4.1 – Principe de la méthode EDPE

6. Valider la prédictibilité de la méthode en déployant d'autres applications logicielles sur la plate-forme et en comparant la consommation prédite et la consommation mesurée.

Les quatre premières étapes sont détaillées dans ce chapitre. Nous commençons d'abord par justifier notre choix de plate-forme, ensuite nous présentons les évènements choisis pour modéliser la consommation énergétique des composants de la plate-forme. Dans une troisième section nous décrivons les différents modes de fonctionnement utilisés pour la caractérisation. La méthode de caractérisation est présentée dans le chapitre quatre.

4.2 Plate-forme d'étude

Pour valider notre méthode nous avons opté pour une plate-forme composée des mêmes types de composants que ceux utilisés dans les travaux cités dans le chapitre 3. En effet la taille de la plate-forme et sa complexité ne sont pas importants. On se propose de modéliser la consommation énergétique d'un système composé de quatre processeurs de type RISC, chaque processeur disposant d'un cache d'instruction et d'un cache de données. La plate-forme conmprend également deux mémoires de type RAM et ROM. Les quatre processeurs ainsi que les mémoires sont reliés entre eux via un interconnect de type Ring.
La figure 4.2 représente de façon simplifiée la plate-forme utilisée pour illustrer la

méthode EDPE.

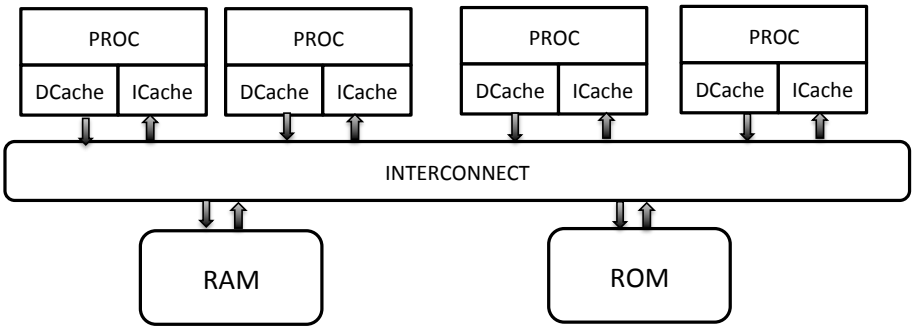

FIGURE 4.2 – La plate-forme d'étude

4.2.1 Le processeur : MIPS32

Le processeur MIPS32 est un processeur 32 bits industriel conçu dans les années 80. Son jeu d'instructions est de type RISC.
Cette architecture est suffisamment simple pour présenter les principes de base de la méthode, et suffisamment puissante pour supporter un système d'exploitation multi-tâches tel qu'UNIX, puisqu'il supporte deux modes de fonctionnement : dans le mode utilisateur, certaines zones de la mémoire et certains registres du processeur réservés au système d'exploitation sont protégés et donc inaccessibles ; dans le mode superviseur, toutes les ressources sont accessibles.
Plusieurs implantations matérielles de cette architecture ont été réalisées à l'Université Pierre et Marie Curie dans un but d'enseignement et de recherche : une version microprogrammée, simple mais peu performante (4 cycles par instruction), une version pipe-line plus performante (une instruction par cycle), mais plus complexe, une version superscalaire, encore plus performante (2 instructions par cycle) et beaucoup plus complexe. Dans le cadre de notre étude, nous avons opté pour la version pipeline qui possède le meilleur rendement énergétique (mesuré en nombre d'instruction par Watt).

4.2.2 Le cache : XCache

Le composant *Xcache* est un contrôleur de cache générique à interface VCI. Le composant VciXcacheWrapper peut encapsuler différents processeurs RISC 32 bits. Le coeur du processeur est modélisé par un ISS (Instruction Set Simulator) ce qui est un niveau d'abstraction assez éloigné du modèle RTL de référence. Le type du

proceseur instancié (MIP32, ARM, SPARCV8, PPC405, NIOS, MicroBlaze, etc.) est défini par un paramètre template du composant VciXcacheWrapper.
Le Xcache implémente deux différents caches : cache de données et cache d'instruction qui partagent la même interface VCI.

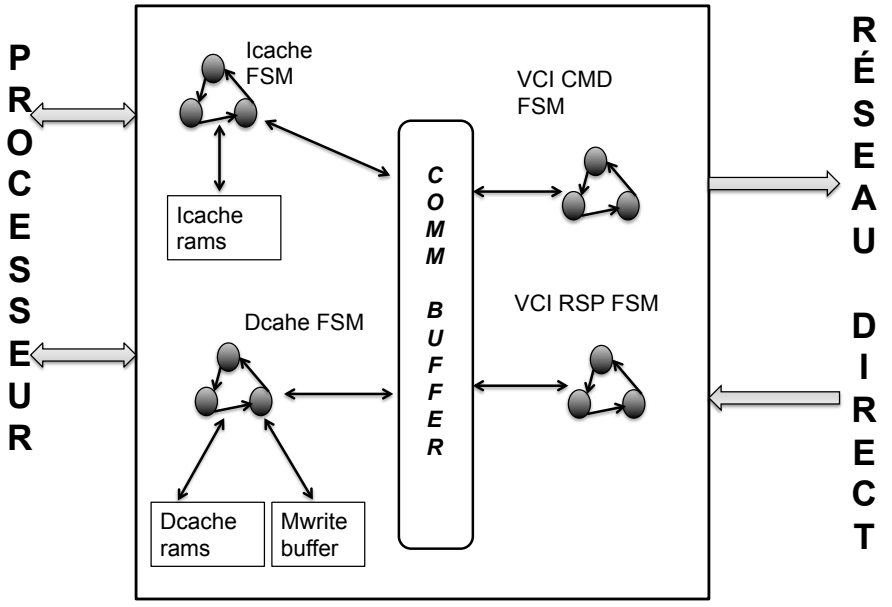

FIGURE 4.3 – Architecture du Xcache

4.2.2.1 Le cache d'instruction

- Il est accessible uniquement en lecture.
- Dans le cas d'un miss de lecture, le processeur est gelé en attendant les mises à jour du cache.

Le cache d'instruction peut émettre deux types de requêtes :

1. Une lecture rafale : dans le cas d'un miss d'une ligne de cache.
2. Une lecture d'un mot, quand ce dernier n'est pas cachable.

4.2.2.2 Le cache de données

- La politique d'écriture est une politique d'écriture immédiate ($Write{-}Through$) : c'est à dire que les données sont écrites directement en mémoire et que le cache n'est mis à jour que dans le cas d'un hit.
- Le cache de données contient un tampon d'écritures postées. Il émet une rafale d'écriture quand il y a des écritures successives dans la même ligne de cache.

- Les données cachées supportent les opérations suivantes : lecture, écriture, linked load et store conditionnel.
- Le cache de données accepte les commandes d'invalidation d'une ligne.
- Le processeur est gelé dans le cas d'un miss de lecture, d'une lecture non cachée, ou encore quand le tampon d'écriture est plein.

Le cache de données peut émettre trois types de transactions VCI [5] :

1. Une lecture rafale : dans le cas d'un miss sur une ligne de cache.
2. Une transaction d'un mot correspondant à une lecture non cachée, une lecture avec réservation ($linked - load$) ou une écriture conditionnelle ($store - conditionnel$).
3. Une écriture rafale d'une longueur variable qui ne peut excéder la longueur d'une ligne de cache.

4.2.3 L'Interconnect : Ring

Pour relier les composants entre eux on a opté pour un réseau de type anneau qui possède une interface VCI [5].
Cet anneau se comporte fonctionnellement comme un bus puisqu'il ne supporte qu'une seule transaction à la fois. La politique d'arbitrage utilisée est un jeton qui circule sur le réseau et qui est attribué aux différents initiateurs suivant l'algorithme *roundrobin*.
Lorsqu'un initiateur veut transmettre une requête, il doit d'abord s'assurer qu'il possède le jeton. Si le jeton est retenu par un autre initiateur, il doit attendre son tour. Chaque commande ou réponse émise par un intiateur ou une cible se traduit par un nombre de flit de type $DSPIN$ [31].

- 1 transaction VCI de type lecture se traduit par 2 flits dspin plus $1+n$ flit de réponse. (header + n read data).
- 1 transaction VCI de type écriture se traduit par $n+2$ flits dspin plus 1 flit de réponse.

La figure 4.4 montre l'architecture interne de l'anneau de la bibliothèque SoCLib. Il est composé de deux réseaux : un réseau de commande et un réseau de réponse.

Le réseau de commande :
Au niveau de chaque cible on retrouve une fifo dans laquelle on stocke les commandes VCI. Un automate est chargé de transformer ces commandes VCI [5] en commandes DSPIN [31]. Afin de les transmettre sur le réseau, un automate vérifie si la cible courante possède le jeton, si c'est le cas le paquet de commande peut être émis

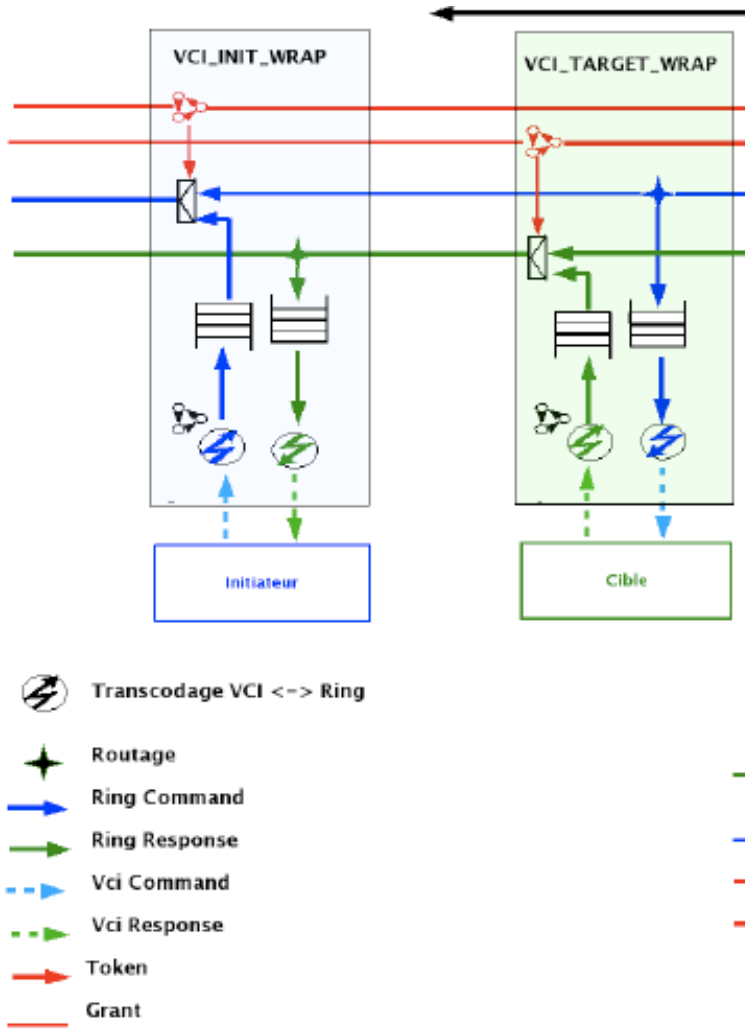

FIGURE 4.4 – Architecture du Ring

sur le réseau. Du côté de la cible, un automate procède au décodage de l'adresse, si l'adresse du destinataire correspond à celle de la cible courante, alors un autre automate se charge de transformer paquet $DSPIN$ en paquet VCI afin qu'il puisse être traité par la cible.
Le réseau des réponses est construit de façon symétrique par rapport à celui des commandes.

4.2.4 La mémoire : VCI-Simple-RAM

La Simple RAM est un contrôleur de $SRAM$ embarquée sur la puce. Il gère un ou plusieurs segments de mémoire. Chaque segment est défini par son adresse de base et sa taille. Les segments doivent être définis dans la mapping table. La RAM possède un paramètre qui permet de définir la latence des accès à la mémoire. Elle

est adressée soit en lecture soit en écriture, et les accès se font par mots de 32 bits.

- Une commande VCI de lecture d'une ligne de cache : se traduit par autant de lectures que de mots qui composent une ligne de cache.

4.3 Les modèles de consommation

Dans cette section, on définit les différents types d'évènements significatifs du point de vue énergétique pour chacun des composants décrits dans la section précédente.
Deux contraintes nous ont guidé :

1. Chaque type d'évènement doit pouvoir être clairement identifié dans les modèles de simulation SoCLib pour permettre un comptage non ambigu.
2. Pour chaque composant, nous avons cherché à minimiser le nombre de types d'évènements de façon à minimiser le nombre de paramètres et faciliter le travail de caractérisation.

4.3.1 Le modèle du processeur

Dans de nombreux travaux le processeur a été modélisé par autant d'évènements que de types d'instructions qu'il est capable de traiter. Ce modèle présente deux inconvénients majeurs qui sont d'une part, sa forte dépendance du jeu d'instruction du processeur utilisé et d'autre part, la complexité du modèle en terme de nombre d'évènements. Pour ces deux raisons, on a d'abord envisagé un modèle plus simple basé sur deux types d'évènements :

- IDLE : ce type d'évènement correspond à un processeur qui est au repos parce qu'il attend une réponse du cache d'instruction ou du cache de données.
- RUNNING : ce type d'évènement correspond à un processeur qui exécute une instruction indépendamment de son type.

On obtient Donc le modèle suivant :

$$E_{proc} = E_{IDLE} * N_{IDLE} + E_{RUNNING} * N_{RUNNING} \tag{4.1}$$

Lors des mesures de consommations réalisées pour valider le modèle du processeur, on a constaté que ce modèle n'est pas assez précis pour le cas où le processeur exécute la même instruction pendant plusieurs cycles à cause d'un blocage temporaire du pipeline, lié à une dépendance de données. En effet, il est évident que dans ce cas le

processeur consomme moins d'énergie puisque l'on garde l'instruction qui est déjà chargée. Pour remédier à ce problème on a introduit un troisième type d'évènement :

- STUTTERING : ce type d'évènement correspond à un processeur qui exécute la même instruction, qu'au cycle précédent.

Ce qui nous amène à un modèle à trois paramètres :

$$E_{proc} = E_{IDLE} * N_{IDLE} + E_{RUNNING} * N_{RUNNING} + E_{STUTTERING} * N_{STUTTERING} \quad (4.2)$$

4.3.2 Le modèle du Cache

Le cache n'est rien d'autre qu'une mémoire qui stocke des données fournies soit par le processeur soit par la mémoire. Naturellement la consommation énergétique du cache, varie lorsqu'il manipule ces données. Il existe uniquement deux façons d'accéder à ces données : lecture ou écritutre. Lorsque les données du cache restent inchangées, la consommation de ce dernier est faible puisqu'il n'y a aucune décharge ou charge de capacité. En se basant sur cette analyse nous avons fixé deux types d'évènements pour la modélisation énergétique du cache : E_{IDLE} et $E_{RUNNING}$ avec :

- E_{IDLE} correspond à un cache qui est au repos c'est-à-dire qui n'est accédé ni en lecture ni en écriture.
- $E_{RUNNING}$ correspond à un accès physique au cache soit en lecture soit en écriture d'un mot de 32 bits.

Dans un premier temps nous avions proposé un modèle énergétique du cache qui distingue les opérations d'écriture de celles de lectures. Cependant, les mesures physiques, ont montré que la différence entre l'énergie consommée par ces deux types d'accès est négligeable. On a donc retenu le modèle suivant :

$$E_{cache} = E_{IDLE} * N_{IDLE} + E_{RUNNING} * (N_{WRITE} + N_{READ}) \quad (4.3)$$

Notons que les compteurs d'évènements portent sur les accès physiques à la mémoire (par mots de 32 bits) : Lorsque le processeur envoie une requête de lecture au cache, trois cas de figures se présentent : soit l'adresse envoyée par le processeur est une adresse cachable et existe dans le cache, soit elle est cachable mais n'existe pas dans le cache, ou encore elle n'est pas cachable. Si on est dans le premier cas de figure alors on compte une lecture physique dans le cache de données. Si on est dans le deuxième cas de figure, on compte une lecture dans le cache de données, puis autant d'écritures physiques que la longueur de la ligne du cache et une lecture une fois que le cache est mis à jour.

4.3.3 Le modèle du Bus

Du point de vue électronique, le *Ring* est une nappe de fil qui transporte des données sous forme de flit $DSPIN$ c'est-à-dire un mot de 40 bits.
À chaque cycle le ring peut soit transporter des données, soit être au repos. En se basant sur cette observation, on a choisi de modéliser la consommation énergétique du ring par deux types de paramètres :

- IDLE : le ring ne transporte rien c'est-à-dire aucun flit ne circule sur le ring.
- RUNNING : Un flit circule sur le réseau.

On obtient alors le modèle suivant :

$$E_{ring} = E_{IDLE} * N_{IDLE} + E_{RUNNING} * N_{RUNNING} \tag{4.4}$$

A titre d'exemple, lorsque le cache envoie une requête de lecture vers la mémoire, cette requête se traduit par un flit de commande contenant l'adresse de lecture, et autant de flit de réponse que la longueur en mots de la ligne du cache plus un flit d'entête.

4.3.4 Le modèle des mémoires : RAM/ROM

Afin d'établir le modèle énergétique de la mémoire, on a utilisé le même raisonnement que pour le cache. Une mémoire permet de stocker des données. Elle est solicitée soit en lecture soit en écriture de mot de 32 bits. Lorsque les données de la mémoire restent inchangées et qu'aucune consultation de données n'est effectuée, on dit que cette dernière est au repos. Le modèle de la mémoire RAM est le suivant :

$$E_{RAM} = E_{IDLE} * N_{IDLE} + E_{RUNNING} * (N_{WRITE} + N_{READ}) \tag{4.5}$$

Le modèle de la ROM est le même que celui de la RAM à l'exception des commandes d'écritures qui n'existent pas dans le cas d'une mémoire de type ROM ce qui donne :

$$E_{ROM} = E_{IDLE} * N_{IDLE} + E_{RUNNING} * N_{READ} \tag{4.6}$$

4.3.5 Le modèle énergétique de la plate-forme

Le tableau suivant, récapitule les treize types d'évènement nécessaires pour caractériser une plate-forme MPSOC composée de 4 processeurs et leurs caches d'instruction et de données respectifs, d'une RAM, d'une ROM et d'un bus.

TABLE 4.1 – Récapitulatif des compteurs d'activités

Composant	Compteurs par type d'évènement
Processeur	N_{IDLE}, $N_{RUNNING}$, $N_{STUTTERING}$
Cache d'instruction	N_{IDLE}, N_{ACCESS}
Cache de données	N_{IDLE}, N_{ACCESS}
Bus	N_{IDLE}, $N_{RUNNING}$
RAM	N_{IDLE}, N_{ACCESS}
ROM	N_{IDLE}, N_{ACCESS}

4.4 Instrumentation de la plate-forme

Après avoir défini les modèles de consommation pour les différents composants de la plate-forme, il est nécessaire de mettre en place une méthode pour récolter les occurences des évènements énergétiques. Pour ce faire, nous avons ajouté dans les modèles $CABA$ et TLM de la bibliothèque SoCLib des compteurs d'évènements. Pour prélever les treize compteurs qui caractérisent énergétiquement la plate-forme représentée par la figure 4.2, on a mis en place une bibliothèque statique C++ dite $SoCView$3.0 qui permet de récolter les valeurs de ces compteurs de façon non intrusive sans modifier le comportement du matériel (cf. Annexe 1). Cette bibliothèque est composée de quatre fichiers d'entête :

- Monitor.h permet de gérer les variables de types natives telles qu'un entier codé sur 32 bits.
- Factory.h inclut Monitor.h et définit la fonction permettant de positionner un pointeur sur n'importe quel type géré par cette classe.
- Debugger.h déclare les fonctions permettant de monitorer une variable telle que : trace, get et set.
- DebuggerRunner.h permet de prélever les valeurs des variables monitorées à chaque front d'horloge.

Pour prélever les valeurs des compteurs d'évènements énergétiques, il suffit de les rajouter à la liste des signaux que l'on désire monitorer dans le fichier *Debuger.h*, puis de compiler la plate-forme considérée avec SoCLib en incluant notre bibliothèque avant de lancer la simulation. Au cours de la simulation, les valeurs des compteurs sont stockées dans un fichier. Notons qu'il est possible de modifier la fréquence de sauvegarde des valeurs des compteurs en terme de cycles d'horloges.

4.5 Mesure de la puissance consommée

4.5.1 Mesures physiques sur FPGA

La mesure de la puissance effectivement consommée par la plate-forme considérée a constitué une des difficultés majeures de ce travail de thèse. Extraire des valeurs des puissances dissipées n'est possible que pour une réalisation particulière de l'architecture. Il faut donc commencer par choisir une cible puis procéder à des mesures réelles. Nous avons choisi comme cible la synthèse sur FPGA.

La raison pour laquelle on a choisi de synthétiser notre plate-forme sur FPGA est tout d'abord le caractère reprogrammable d'une telle carte. La reprogrammabilité permet de caractériser avec la même carte différentes plate-formes matérielles. De part leur caractère flexible, un autre paramètre vient justifier notre choix : les cartes FPGA ne coûtent pas cher comparées à une réalisation ASIC.

Pour mesurer la consommation énergétique de notre plate-forme quand elle est déployée sur FPGA, on a choisi un kit Altera de la famille STRATIX III destiné à réaliser des mesures de consommation. L'avantage de ce kit est la présence de résistances en aval de chaque alimentation du coeur du FPGA. Pour obtenir la puissance dissipée par l'ensemble de la plate-forme matérielle plus le logiciel embarqué dessus, il suffit de mesurer la tension aux bornes de ces résistances. La puissance instantanée consommée est alors exprimée sous la forme :

$$P(t) = (V_{supply} * V_{mesure}(t))/R \text{ avec } R = 0,9Ohm, V_{supply} = 1,1V \tag{4.7}$$

Notre objectif est de mesurer l'énergie totale $E_{T,K}$ consommée par l'exécution d'une application logicielle K sur la plate-forme FPGA. Cette énergie est l'intégrale de la puissance instantanée $P_K(t)$ sur toute la durée de l'application K. Comme nous ne sommes pas intéressés par l'évolution de la consommation au cours du temps, la fréquence d'échantillonnage n'a pas besoin d'être très élevée et doit simplement être suffisante pour rendre négligeables les erreurs liées à l'interpolation des valeurs d'échantillonnées de la courbe $P_K(t)$. Nous avons choisi une fréquence d'échantionnage de 10000 échantillons par seconde, soit une période de 5000 cycles entre deux mesures. La tension mesurée est représentée sur 17 bits. La précision absolue sur la mesure de tension est de 0,7 micro V.

Pour conduire ces mesures, nous avons choisi un multimètre de chez national instrument : NI PXIe-1071 [32]. Il joue le rôle de deux instruments : un multimètre

numérique haute résolution et un numériseur.
La figure 4.5 montre le multimètre numérique. De gauche à droite, on retrouve d'abord les E/S de la carte qui permet le traitement des données par le multimètre et la programmation de la carte FPGA à travers le JTAG, les E/S de la carte qui permet la prise des mesures de tension et enfin l'interface d'E/S de la NI Pxie-6537 qui permet l'aquisition de 32 signaux en parallèle avec un débit de 200 MO/s.

FIGURE 4.5 – Multimètre de Mesure : NI PXI-4071

En utilisant les fonctions d'analyse de NI LabVIEW, nous avons tracé la variation de la tension au bornes de la résistance placée en aval de la tension d'alimentation du coeur du FPGA. Le module LabVIEW, permet entre autre de préciser le nombre de mesures enregistrées par seconde et la durée des mesures. En sortie il génère un fichier texte qui contient les informations suivantes :

- La valeur du signal de lancement de la mesure : start.
- La valeur du signal de la fin de la mesure : stop.
- La valeur de la tension mesurée.
- La valeur de la puissance calculée à partir de la tension.
- La vitesse d'échantillonnage.
- Le nombre de bits sur lequel sont codées les valeurs mesurées.
- Un champ date et un champ commentaire.

Pour pouvoir mesurer l'énergie consommée durant l'exécution d'un mode donné, et y associer les valeurs des compteurs d'activités correspondants, nous avons ajouté

un composant matériel « Multimètre » à la plate-forme synthétisée sur FPGA, qui au front montant de l'horloge et lorsque le reset passe à 1, envoie un signal *start* au multimètre NI PXI-4071, pour lui signaler le début des mesures puis attend 1000000 de cycles pour émettre un deuxième signal *stop* vers le multimètre NI PXI-4071 pour lui indiquer la fin des mesures.

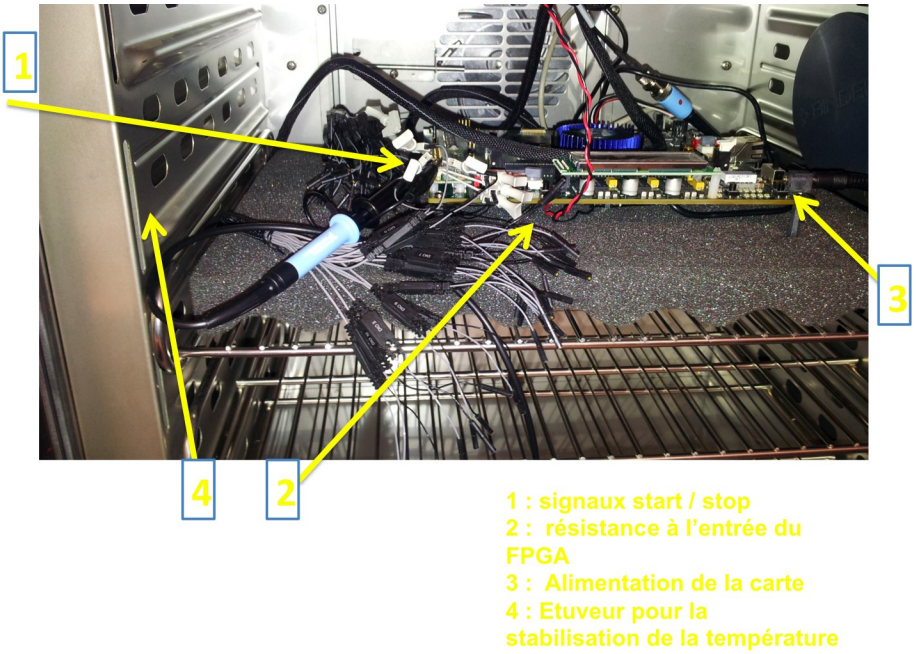

FIGURE 4.6 – Plate-forme de mesure : Kit de développement stratix III + Étuveur

Sur la figure 4.6 on voit le kit FPGA, la flèche 1 montre les pins d'E/S qui sont utilisées pour transmettre les signaux start, stop et reset, la flèche 2 montre l'emplacement de la résistance de 0.9 ohm qui se trouve à l'entrée du FPGA, la flèche 3 montre l'alimentation du kit et la flèche 4 montre l'étuveur qui a été utilisé pour stabiliser la température au cours de la mesure à 25 degrés celsius.

4.5.2 Mesure de la consommation avec PowerPlay

Pour s'assurer de la précision des mesures fournies par la plate-forme décrite ci-dessus, nous avons comparé les mesures de cette dernière avec les résultats donnés par un outil de simulation bas niveau fourni par Altera. Il s'agit de PowerPlay [19].

Le principe de fonctionnement de PowerPlay :
PowerPlay [19] donne la puissance dissipée par les différents composants de la plate-forme en mw. Afin d'obtenir des résultats précis, il faut d'abord synthétiser la plate-forme puis procéder à son placement routage sur une carte cible. Par la suite, il faut fixer la fréquence de fonctionnement souhaitée puis simuler tous les signaux de la

plate-forme au niveau portes logiques, dans le but d'obtenir une trace complête des transitions de tous les signaux. Enfin ce fichier est utilisé pour calculer la puissance dynamique de toute la plate-forme ainsi que la puissance dynamique consommée par chacun des composants.

4.5.3 Comparaison entre les deux méthodes

La méthode de mesure directe sur FPGA, présentée dans la section 4.5.1, a l'avantage d'être rapide. En effet, une fois qu'on a programmé la carte FPGA, le temps nécessaire pour mesurer la tension est égale au temps nécessaire pour l'exécution du logiciel sur la carte. Dans notre cas on utilise un FPGA avec une fréquence de 50MHz, pour exécuter des logiciels d'une durée de 1000000 de cycles. Ce qui donne un temps de mesure de 0.2s. Cependant, cette méthode est très sensible aux erreurs de mesure ainsi qu'aux approximations du modèle de consommation puisqu'elle permet de mesurer l'énergie totale consommée par toute la plate-forme.
Les résultats expérimentaux présentés dans le chapitre 6 montrent qu'il y a peu de différences entre les mesures physiques réalisées sur FPGA et les résultats fournis par PowerPlay. Pour cette raison, on a finalement opté pour la méthode présentée dans la section 4.5.2, qui utilise l'outil PowerPlay. L'avantage de cette méthode est qu'elle permet de procéder à une caractérisation par composant, puisqu'elle fournit l'énergie consommée par chacun des composants de la plate-forme. Cette méthode possède néanmoins un point faible qui est sa vitesse de mesure. En effet, pour obtenir des mesures de consommations précises avec PowerPlay, il faut tout d'abord simuler la plate-forme au niveau portes logiques pendant le temps nécessaire à l'exécution du logiciel ce qui correspond à 8 jours de calcul sur station de travail par mode de fonctionnement.

4.6 Les Modes de fonctionnement

Dans la section 4.3 nous avons défini treize types d'évènements pertinents d'un point de vue énergétique pour modéliser la consommation énergétique de la plate-forme représentée par la figure 4.2. Le nombre d'évènements de chaque type dépend non seulement des caractéristiques physiques de la plate-forme telles que la taille des caches mais aussi de l'application logicielle et de la cachabilité des données manipulées. Il est donc nécessaire de définir des *modes de fonctionnement* aussi différents que possible qui vont permettre d'activer de façon différenciée les treize types d'évènements énergétiques.

4.6.1 Mode 0 : STATIC

Ce mode nous permet de mettre en avant la consommation statique ainsi que celle de l'horloge de la plate-forme considérée. Pour ce faire on se propose de démarrer les quatre processeurs, puis de les mettre au repos en utilisant l'instruction *wait*. Dans ce mode rien ne se passe sur la carte à part l'horloge qui transite.

4.6.2 Mode 1 : DCACHE-RING-RAM

Dans ce mode de fonctionnement, nous avons cherché à augmenter le nombre de transactions de lecture de la mémoire en provoquant des miss dans le cache de données. Pour ce faire on a choisi de lire en boucle à une adresse cachée qui sera augmentée à chaque tour de la boucle de la moitié de la taille d'une ligne de cache. De cette sorte on aura pour N lectures :

- N/2 Miss.
- N/2 écritures de lignes de cache.
- (Nombre de processeurs * N * nombre de mots par ligne de cache) lectures dans la mémoire RAM.

4.6.3 Mode 2 : PROC-ICACHE

Dans ce mode nous nous proposons d'augmenter l'activité du processeur et de son cache d'instruction tout en gardant une activité quasi nulle pour le reste des composants. Pour ce faire, il suffit d'exécuter des sauts inconditionnels immédiats vers un label du code du reset de cette façon pour N cycles exécutés on s'attend à :

- N exécutions d'instruction pour le Mips et zéro cycles IDLE.
- N lectures du cache d'instruction et zéro cycles IDLE dans ce même cache.
- 0 activité dans le cache de données.
- 0 activité dans les mémoires RAM et ROM.

4.6.4 Mode 3 : RING-ROM

Ce mode de fonctionnement vise à désactiver le cache de données et d'instruction afin de solliciter la mémoire ROM et l'interconnect. Pour ce faire on garde le même code que celui du mode 2 et l'on modifie la cachabilité de la plate-forme. C'est-à-dire on rend les segments du reset, de la stack et du kernel non cachés. On peut donc prédire que l'activité du cache d'instruction sera nulle.

4.6.5 Mode 4 : ICACHE-RAM

Ce mode a pour but d'exciter la mémoire RAM ainsi que le cache d'instruction. Pour ce faire on exécute un programme formé de sauts à des étiquettes entre lesquelles on introduit l'instruction *.space*24 de telle sorte qu'à chaque passage par cette boucle on provoque 4 miss sur le cache instruction. On s'attend donc à avoir les chiffres suivants :

- Un nombre total de miss qui est égal au nombre de tours de la boucle multiplié par 4.
- Un nombre de lecture de la mémoire qui vaut le nombre total de miss multiplié par le nombre de mots par ligne de cache, le tout multiplié par le nombre de processeurs qui exécutent ce test bench en accédant à cette mémoire.
- L'activité du cache de données est quant à elle presque nulle puisque l'on n'exécute que des instructions de sauts.
- Le processeur devrait être gelé pendant un nombre de cycle égal au nombre de miss sur le cache instruction multiplié par la durée moyenne d'un miss instruction.
- Pour ce qui est du ring, le nombre total de flits qui circulent peut être calculé de la façon suivante : pour chaque miss on a une commande de lecture qui est émise vers la mémoire et elle constituée de deux flits. La réponse à ce miss est composée d'un flit d'entête plus n flits, avec n la taille d'une ligne de cache.

4.6.6 Mode 6 : DCACHE

Le but de ce mode de fonctionnement est de solliciter le cache de données en lecture. Pour ce faire, on exécute une boucle qui effectue deux *loadword* puis additionne les deux mots lus. Si l'on effectue N itérations dans la boucle, à la fin de la simultation on aura :

- 2N lectures du cache de données.
- 2N exécutions de l'instruction LW par le processeur et donc 2N stutter.
- Comme le code du main contient 9 instructions codées chacune sur 4 mots, on aura donc (5 * largeur d'une ligne de cache) lectures de la mémoire RAM. L'activité sur la mémoire devient nulle à partir d'un certain nombre de cycle.
- L'activité sur l'interconnect, le nombre de flits qui circuleront dessus vaudra : Nombre de miss instructions et données * (2 + largeur de la ligne du cache).

4.6.7 Mode 7 : RAM-READ

Le septième mode reprend le même code du mode 6, cependant le cache de données est désactivé. Dans ce cas l'activité du cache de données est reportée à la mémoire RAM. On aura également une activité plus importante sur le réseau d'interconnexion.

4.6.8 Mode 8 : DCACHE-RAM-WRITE

Le but de ce mode de fonctionnement est d'augmenter l'activité du cache de données en effectuant plusieurs opérations d'écriture sur des données cachées. Pour ce faire, on a écrit une boucle qui comporte deux instructions SW (store word) à deux adresses cachées. Ce qui devrait donner des chiffres de simulation respectant :

- Le nombre de lecture dans le cache de données est égal au nombre de SW, en effet cela correspond aux lectures avant leurs écritures.
- Comme la politique du cache est $Write - Through$, les écritures sont directement reportées à la mémoire, on aura donc autant d'écritures dans la mémoire que de SW.
- L'activité de l'interconnect sera très importante, en effet, à chaque tour dans la boucle, on effectue deux store(s) word dans la même ligne, on envoie donc 2+2 flits de commande d'écriture (header, id, 2 words) et l'on reçoit 1 flit de réponse à une écriture.

4.6.9 Mode 9 : RAM-WRITE

Dans ce mode, on reprend le même software que le mode 8, et l'on change la cachabilté des segments mémoires. En effet nous rendons le segement de données non caché. Cette manoeuvre a pour objectif d'isoler l'activité du cache de données de celle de la mémoire lorsqu'on effectue une opération d'écriture. Nous espérons alors obtenir les mêmes chiffres que dans le mode 7 à l'exception du cache de données sur lequel il n'y aura pas d'activité.

4.6.10 Mode 10 : DCACHE-RAM-READ-AND-WRITE

Dans ce mode, on cherche à souligner à la fois les écritures et les lectures dans le cache de données. Pour cela on exécute une boucle qui contient deux SW et un LW. On s'attend donc à avoir un nombre d'écritures dans le cache de données

qui est le double du nombre de lectures. Le nombre de transactions de lecture est en réalité le nombre de lectures prélevé, duquel on retranche le nombre d'écriture. En effet avant chaque écriture, une lecture de la variable considérée est réalisée. L'activité de la mémoire dépend uniquement de l'opération d'écriture. Cependant, comme on procède à un LW puis un SW sur la même adresse, puis sur une adresse de la même ligne. Les deux adresses étant espacées de 3 mots, à chaque fois que l'on passe d'un SW à un LW, le tampon d'écriture envoie une rafale d'écriture de 5 mots. On s'attend donc à un nombre d'écriture dans la mémoire qui vaut 5 fois le nombre total d'itérations de la boucle. On s'attend également à avoir autant de cycles durant lesquels le processeur reste gelé dans l'attente de la réponse du cache de données.

4.6.11 Mode 11 : PROC-STUTTER

Ce scénario cherche à mettre en avant l'évènement $STUTTER$ du processeur, tout en minimisant l'activité des autres composants. Il suffit donc d'exécuter dans le reset, une boucle comportant cinq opérations *nop*. La boucle va maintenir l'activité du proceseur et de son cache instruction et les *nop* vont augmenter le nombre de cycle *stutter* sans solliciter les autres composants.

4.7 Conclusion

Dans ce chapitre nous avons présenté le principe de la méthode EPDE (Early design Power Estimation). EDPE permet la caractérisation énergétique d'une plateforme multiprocesseurs avec un petit nombre de paramètres par composant (au plus trois compteurs dans le cas du processeur). Nous avons montré qu'il était possible de d'activer de façon différenciée ces compteurs en déployant différentes applications logicielles ou encore en modifiant la cachabilité des segments mémoires.
Pour chaque mode de fonctionnement, nous obtenons de la simulation SystemC les occurences des différents types d'évènements. Dans le chapitre suivant nous présentons la méthode permettant de calculer les treize énergies élémentaires constituant les paramètres du modèle à partir de la mesure des énergies associées aux différents modes.

Chapitre 5

Méthode de caractérisation

Sommaire

Ce chapitre décrit la méthode de caractérisation. La caractérisation consiste à déterminer l'énergie consommée par chacun des évènements élémentaires définis dans le chapitre 4. Dans le chapitre 3, on a démontré que la plupart des méthodes d'estimation de la consommation proposent des méthodes de caractérisation différentes pour chaque composant. Nous avons conclu, que ce point constitue un vrai frein à la généralisation de ces méthodes pour de nouveaux composants. Pour cette raison, nous proposons une méthode de caractérisation commune à tous les composants de la plate-forme.

5.1 Position du problème

Dans le chapitre 4 nous avons proposé treize types d'évènements qui permettent de modéliser de façon précise la consommation énergétique des différents composants de notre plate-forme d'étude. Le nombre d'occurence de ces évènements pour une

application logicielle donnée, est obtenu à partir de la simulation au niveau CABA de la plate-forme décrite avec la bibliothèque de prototypage virtuel SoCLib. Dans le chapitre 4 nous avons montré que l'énergie totale consommée par la plate-forme, peut être obtenue soit par mesure directe sur FPGA, soit en utilisant l'outil PowerPlay qui donne une estimation précise de la consommation énergétique d'une plate-forme donnée grâce à des simulations au niveau *RTL* pour différentes applications logicielles correspondant à différents mode de fonctionnement. Ce qui nous permet de mettre en équation les modèles de consommation. On obtient ainsi le système suivant :

$$\begin{bmatrix} E_{T,1} \\ E_{T,2} \\ E_{T,3} \\ \vdots \\ E_{T,K-1} \\ E_{T,K} \end{bmatrix} = \begin{bmatrix} N_{1,1} & N_{1,2} & ... & N_{1,L} \\ N_{2,1} & N_{2,2} & ... & N_{2,L} \\ N_{3,1} & N_{3,2} & ... & N_{3,L} \\ & \ddots & \ddots & \\ N_{K-1,1} & N_{K-1,2} & ... & N_{K-1,L} \\ N_{K,1} & N_{K,2} & ... & N_{K,L} \end{bmatrix} \times \begin{bmatrix} e_1 \\ e_2 \\ e_2 \\ \vdots \\ e_{L-1} \\ e_L \end{bmatrix} \tag{5.1}$$

Avec K le nombre de modes de fonctionnement, L le nombre de types d'évènements, $N_{k,i}$ le nombre d'occurrences d'un type d'évènement i pour un mode k, e_i l'énergie élémentaire correspondant à l'évènement i et $E_{T,i}$ l'énergie totale du système pour le mode i. Le problème consite donc à résoudre ce système linéaire afin de déterminer les valeurs des énergies élémentaires e_i.

5.2 Algorithme de résolution

Historiquement, on a pensé à résoudre le système 5.1 par les méthodes classiques de résolution des systèmes linéaires. Trois cas de figure se présentent :

1. Le nombre d'équations (modes) est égal au nombre d'inconnues (types d'évènements). Dans ce cas le système admet une solution unique si et seulement si la matrice N est inversible. Cependant la stabilité de la solution e dépend du conditionnement de N. On rappelle, qu'un système linéaire est bien conditionné si e satisfait également le système :

$$(N + \epsilon) * e = E \tag{5.2}$$

   où ϵ est une matrice $(K*L)$ de pertubations.

2. Le nombre d'équations est inférieur au nombre d'inconnues. Dans ce cas le système est dit sous-déterminé et admet une infinité de solutions.

3. Le nombre d'équations est supérieur au nombre d'inconnues. Le système est alors dit surdéterminé. Le système n'admet généralement pas de solution exacte.

Initialement, on s'est placé dans le premier cas de figure. Malheureusement nous avons observé que le système issu des mesures physiques est mal conditionné. On a donc décidé d'opter pour le troisième cas, en construisant un système surdéterminé, dans le but d'atténuer l'impact des erreurs de mesures sur la solution. Dans le cas d'un système surdéterminé, le but n'est pas de chercher une solution exacte, mais de trouver un vecteur e qui minimise une fonction de coût représentant l'erreur. Dans notre cas la fonction de coût est le carré de la norme euclidienne de la quantité $N * e - E$:

$$||N * e - E||^2 \tag{5.3}$$

Dans un espace à deux dimensions, et M équations, la fonction de coût prend la forme :

$$C(X, Y) = \sum_{i=1}^{M} |\alpha_i * X + \beta_i * Y + \gamma_i|^2 \tag{5.4}$$

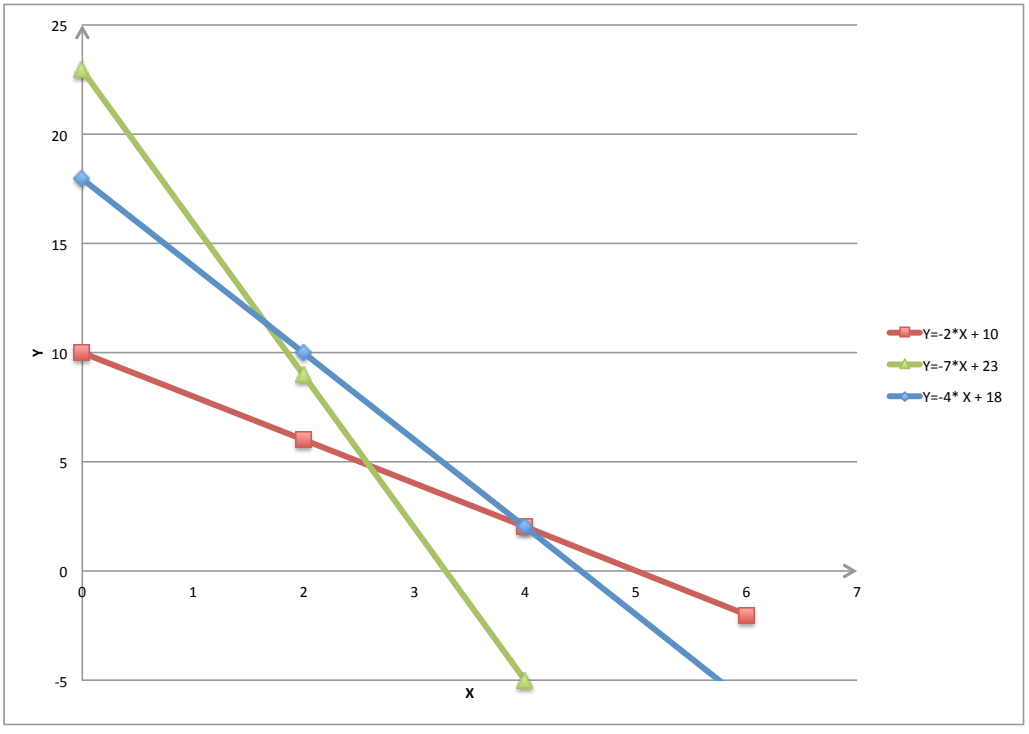

FIGURE 5.1 – Système de trois équations à deux inconnues

On remarque que l'existance de plusieurs points d'intersection est liée à la fois à l'imprécision des mesures et aux approximations du modèle de consommation lui-même. En conséquence, le minimun O est forcément dans le voisinage des Z points formés par les intersections des N droites 2 à 2.($Z = 3$ dans le cas de trois droites,

$Z = 6$ dans le cas de 4 droites, $Z = 10$ dans le cas de 5 droites, etc...). On peut démontrer que la fonction de coût 5.3 admet un minimum unique (cf. Annexe 2). Puisqu'il n'existe pas de minima locaux, on peut utiliser une technique de gradient pour déterminer les coordonnées du point O.
Le principe de notre méthode de recherche est représenté par l'organigramme 5.2 :

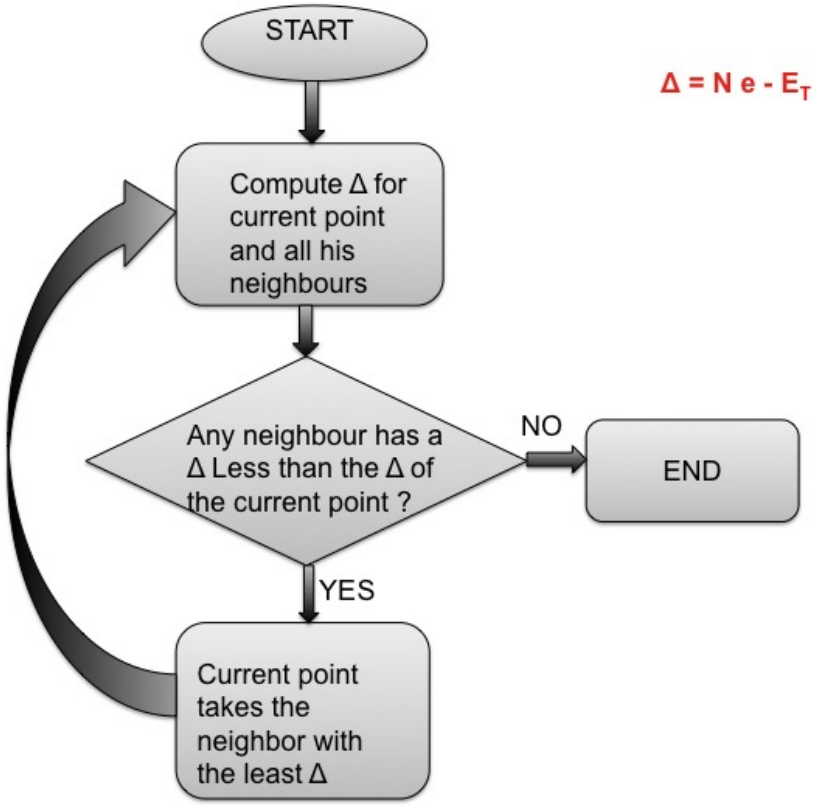

FIGURE 5.2 – Méthode de Caractérisation

5.2.1 Initialisation

Le point initial $\overset{(0)}{e}$ est un vecteur à L éléments composé des valeurs maximales que peut prendre chacune des énergies élémentaires. Pour chaque scénario K, on peut écrire :

$$E_K = N_{K,1} * e_1 + N_{K,2} * e_2 + \ldots + N_{K,L} * e_L \tag{5.5}$$

À partir de 5.5, on peut déduire l'inéquation suivante pour chaque énergie élémentaire $e_i, \forall i \in [2..L]$:

$$\overset{(0)}{e}_i \leq \frac{E_K}{N_{K,i}} \tag{5.6}$$

Cependant cette borne supérieure de e_i dépend des modes de fonctionnement K. Pour obtenir une borne supérieure absolue, il suffit de calculer le maximum des bornes supérieures par mode de fonctionnement. On obtient alors :

$$\overset{(0)}{e} = \{Max_k(\frac{E_K}{N_{K1}}), Max_k(\frac{E_K}{N_{K2}}), Max_k(\frac{E_K}{N_{K3}}), \ldots, Max_k(\frac{E_K}{N_{KL}})\} \tag{5.7}$$

Considérons le système à deux dimensions à deux inconnus :

$$\begin{bmatrix} E_1 = N_{11} * e_1 + N_{12} * e_2 \\ E_2 = N_{21} * e_1 + N_{22} * e_2 \end{bmatrix} \tag{5.8}$$

Dans ce cas, le point initial a pour coordonnées : $(E_1/N_{11}, E_2/N_{22})$.

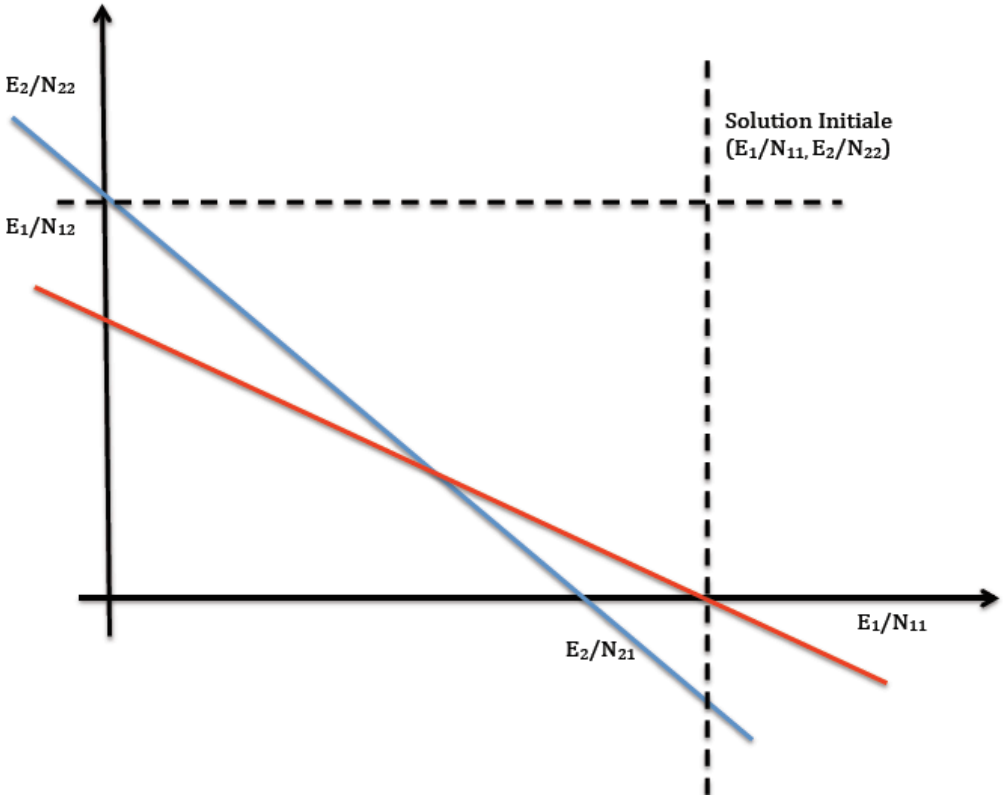

FIGURE 5.3 – Solution Initiale

5.2.2 Recherche de la solution

On parcourt l'espace des solutions à la recherche du point qui minimise la fonction de coût 5.3. Pour ce faire, on calcule les coordonnées des $2 * L$ points voisins de $\overset{(0)}{e}$ ainsi que leurs fonctions de coût. Les coordonnées des voisins d'un point e_i sont

obtenues en augmentant ou en diminuant, une par une, les différentes composantes du vecteur e_i d'un pas p_i.

Initialement p_i prend comme valeur le maximum des $\frac{Max_k(\frac{E_K}{N_{Ki}})}{2}$.
Quand la fonction de coût de tous les voisins est supérieure à celle du point courant, p_i est divisé par h avec $h \in \{2, 4, 8...\}$.

Les déplacements s'effectuent, vers le point voisin ayant la plus petite fonction de coût.

On répète les étapes 3, 4 et 5 jusqu'à obtenir une erreur inférieure à 1/100000.

5.2.3 Exemple 1

Pour mieux comprendre le fonctionnement de la méthode de caractérisation, on se propose de l'utiliser dans les cas du système de trois équations à deux inconnues suivant :

$$\begin{bmatrix} 2 * x + y = 10 \\ 7 * x + y = 23 \\ 4 * x + y = 18 \end{bmatrix} \tag{5.9}$$

En appliquant notre algorithme, nous obtenons la solution $S = \{2.52, 6.05\}$. La

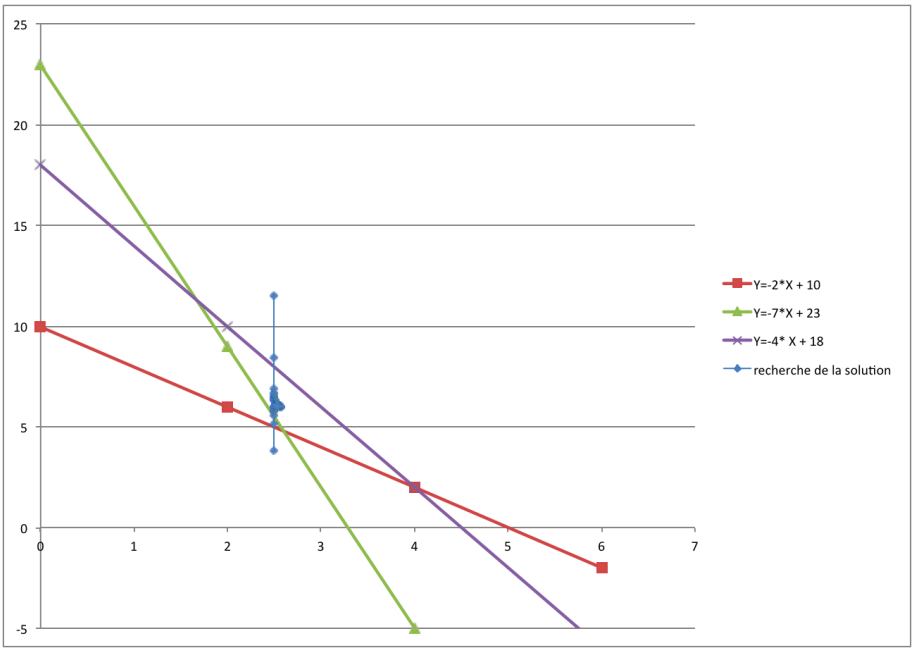

FIGURE 5.4 – Méthode de caractérisation appliquée à un système linéaire de trois équations à deux inconnues

figure 5.4 montre le parcours de l'espace des solutions à la recherche du point qui minimise la fonction de coût.

5.2.4 Exemple 2

Pour valider notre méthode de caractérisation, nous l'avons appliquée à plusieurs systèmes linéaires bien déterminés admettant une solution unique. Nous avons convergé vers la solution exacte dans 100% des cas.

5.3 Caractérisation par composant

Au final, nous obtenons un système à treize inconnues. Cependant ce système est mal conditionné et sensible aux erreurs de mesures. D'autant plus que pour résoudre ce système nous avons besoin d'au moins treize modes de fonctionnements différents. Puisque PowerPlay nous donne la consommation énergétique par composant, on a préféré résoudre un système par composant. Nous obtenons alors un système a trois inconnues pour le processeur :

$$\begin{bmatrix} E_1 \\ E_2 \\ E_3 \\ \vdots \\ E_{K-1} \\ E_K \end{bmatrix} = \begin{bmatrix} N_{1,1} & N_{1,2} & N_{1,3} \\ N_{2,1} & N_{2,2} & N_{2,3} \\ N_{3,1} & N_{3,2} & N_{3,3} \\ & \ddots & \ddots \\ N_{K-1,1} & N_{K-1,2} & N_{K-1,3} \\ N_{K,1} & N_{K,2} & N_{K,3} \end{bmatrix} \times \begin{bmatrix} e_{RUN} \\ e_{STUTTER} \\ e_{IDLE} \end{bmatrix} \tag{5.10}$$

Pour les cinq autres composants nous obtenons des systèmes à deux inconnues.

$$\begin{bmatrix} E_1 \\ E_2 \\ E_3 \\ \vdots \\ E_{K-1} \\ E_K \end{bmatrix} = \begin{bmatrix} N_{1,1} & N_{1,2} \\ N_{2,1} & N_{2,2} \\ N_{3,1} & N_{3,2} \\ \ddots & \ddots \\ N_{K-1,1} & N_{K-1,2} \\ N_{K,1} & N_{K,2} \end{bmatrix} \times \begin{bmatrix} e_{RUN} \\ e_{IDLE} \end{bmatrix} \tag{5.11}$$

5.4 Conclusion

Dans ce chapitre, nous avons présenté une méthode de caractérisation rapide, homogène (la même pour tous les composants) et facilement généralisable. Cette méthode est également peu sensible aux erreurs de mesures physiques.
Dans le chapitre suivant, nous présenterons, les résultats expérimentaux, concernant la détermination des énergies élémentaires, ainsi que les expériences qui ont été conduites pour valider la capacité prédictive de notre méthode.

Chapitre 6

Résultats expérimentaux

Sommaire

Dans ce chapitre, nous présentons les résultats expérimentaux obtenus sur la plate-forme d'expérimentation décrite dans le chapitre 4. Les questions auxquelles nous souhaitons apporter des réponses sont les suivants :

- Est-il possible d'utiliser PowerPlay comme outil de mesure ?
- Les Modes de fonctionnement définis dans la section 4.6 sont-ils assez différentiables du point de vue des activités fonctionnelles ?
- L'algorithme d'optimisation est-il auto-consistant ?
- Quelle la capacité prédictive de la méthode EDPE ?

– Quelle est l'accélération du temps d'estimation avec *EDPE* (Early Design Power Estimation) ?

6.1 Plate-forme de mesure : Corrélation temps/tension

Les énergies consommées par chaque mode de fonctionnement ont été mesurées grâce au dispositif décrit dans la section 4.5.1.

Pour vérifier la dépendance temporelle entre la consommation et le traitement éffectué par la puce, nous avons enregistré les tensions aux bornes du cœur du FPGA pour une application logicielle qui nécessite 9500 cycles d'exécution. La figure 6.1 confirme qu'il existe une corrélation forte entre l'application déployée sur la plate-forme et la consommation énergétique de cette dernière. En effet, on distingue bien le début et la fin de l'exécution respectivement à $t0 = 7,56$ ms et $t1 = 7,75$ ms.

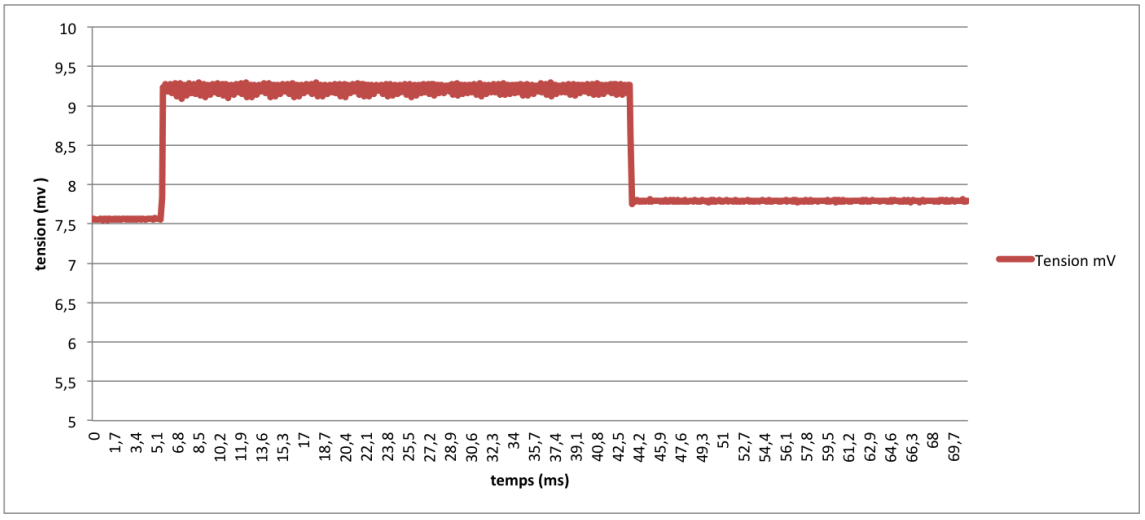

FIGURE 6.1 – Tension du FPGA en fonction du temps

6.1.1 Précision de l'outil PowerPlay

Comme précisé dans le chapitre 4, nous avons opté pour l'utilisation des mesures de consommation fournies par PowerPlay [19] car celui-ci nous permet d'obtenir une évaluation de la consommation par composant. Cette approche nous permet de limiter les erreurs sur les énergies élémentaires.

Cependant, pour valider les résultats donnés par PowerPlay [19] nous les comparons avec les chiffres de consommation obtenus par mesure physique sur FPGA.

Après avoir mesuré sur FPGA l'énergie consommée pour chaque mode de fonctionnement, on les compare aux valeurs des énergies totales fournies par PowerPlay.

Dans le tableau 6.1 on retrouve en première colonne les modes de fonctionnements, en deuxième colonne la puissance totale correspondant à chacun de ces modes de fonctionnement fournies par PowerPlay.
Les troisième, quatrième et cinquième colonnes représentent la décomposition de cette puissance en puissances statique, dynamique et puissance consommée par les entrées/sorties.
En sixième colonne on calcule l'énergie consommée à partir des puissances totales fournies par *PowerPlay* (deuxième colonne). La septième colonne contient les énergies obtenues par mesures réelles sur le Kit FPGA.
Les erreurs relatives des énergies fournies par PowerPlay par rapport aux mesures réelles sont représentées sur la huitième colonne. L'erreur relative est calculée par l'expression suivante :

$$Erreur\_Relative = \frac{(EPP - E_{\text{Mesurée}}) * 100}{E_{\text{Mesurée}}} \tag{6.1}$$

On remarque que de l'écart entre les résultats fournis par PowerPlay et les valeurs mesurées physiquement est compris entre 3% et 5%, ce qui valide notre choix d'utiliser les valeurs fournies par PowerPlay.

Mode	P Totale (mw)	P Dynamique (mw)	P Statique (mw)	P I/O (mw)	EPP (J)	EMesurée (J)	Erreur relative
Mode 0	739,93	114	583,9	41,73	0,0148	0,0143	3,37
Mode 1	905,16	279,25	584	41,73	0,0181	0,0173	4,61
Mode 2	867	241,73	583,54	41,73	0,0173	0,0166	4,23
Mode 3	971,57	344,28	585,28	41,73	0,0194	0,0186	4,29
Mode 4	787,87	163,91	582,23	41,73	0,158	0,0149	5,23
Mode 5	783,58	159,68	582,16	41,73	0,0157	0,0148	5,26
Mode 6	1011,99	384,3	585,95	41,73	0,0202	0,0195	3,62
Mode 7	777,26	153,47	582,06	41,73	0,0155	0,0148	4,72
Mode 8	892,94	267,24	583,97	41,73	0,0179	0,0170	4,67
Mode 9	888,51	263,05	583,73	41,73	0,0172	0,0164	4,96
Mode 10	910,21	284,22	584,26	41,73	0,0182	0,0143	3,48

TABLE 6.1 – Comparaison entre les énergies obtenues via PowerPlay et celles obtenues par mesure directe sur FPGA.

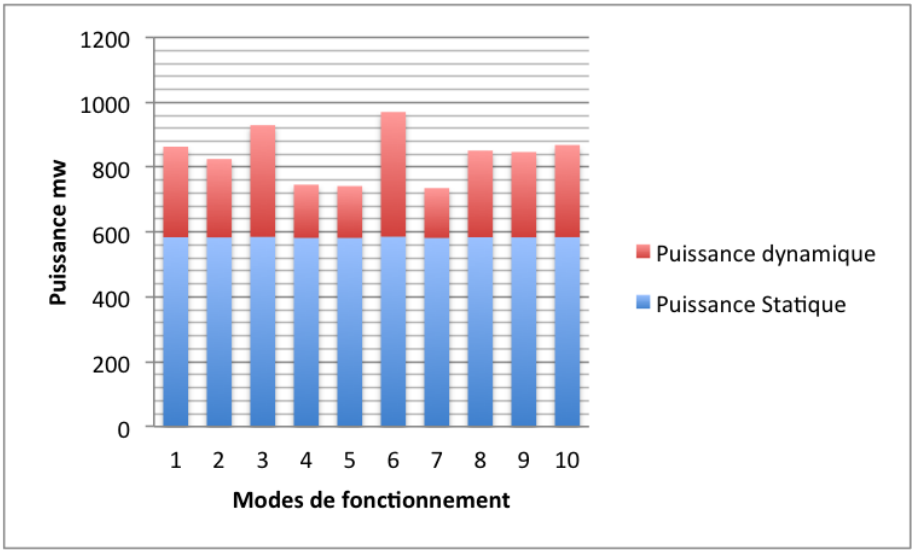

FIGURE 6.2 – Décomposition de la puissance totale en puissances dynamique et statique

6.1.2 Énergie statique Vs énergie dynamique

Ces mesures montrent également que l'énergie totale consommée par un FPGA est majoritairement une énergie statique indépendante du scénario logiciel.
Etant donné que nos modèles de consommation sont basés sur des activités fonctionnelles, seule l'énergie dynamique compte. Cette prédominance de la consommation statique constitue la difficulté principale de la caractérisation.

On constate en revanche que les modes de fonctionnement utilisés génèrent une consommation dynamique clairement différentiable.

6.2 Simulation Modelsim : Compteurs d'évènements

Les modèles de consommation énergétique présentés dans le chapitre 4, s'appuient sur l'occurrence de treize types d'évènements soigneusement définis (cf. 4.3). Pour caractériser ces évènements, nous avons utilisé les dix modes de fonctionnement décrits dans la section 4.6. Après la simulation des dix modes de fonctionnement sur une durée de 1000000 cycles, nous obtenons les résultats présentés dans le tableau 6.2. Les différentes colonnes représentent le nombre d'occurrence de chacun des treize types d'évènements divisés par 1000.

Modes	Ps	P0	P1	DC1	DC0	IC1	IC0	R1	R0	RAM1	RAM0	ROM1	ROM0
Mode 1	18	379	603	108	892	811	189	198	802	577	423	0	1000
Mode 2	540	0	460	0	1000	945	55	0	1000	0	1000	0	1000
Mode3	0	0	1000	0	1000	1000	0	0	1000	0	1000	0	1000
Mode4	26	921	53	0	1000	0	1000	289	711	0	1000	316	684
Mode 5	36	816	148	0	1000	192	808	264	736	768	232	0	1000
Mode 6	222	0	778	222	778	1000	0	0	1000	0	1000	0	1000
Mode 7	7	950	43	0	1000	0	1000	307	693	314	686	0	1000
Mode 8	125	500	375	125	875	1000	0	313	688	500	500	0	1000
Mode 9	125	667	208	0	1000	1000	0	333	667	333	667	0	1000
Mode 10	200	500	300	250	750	1000	0	300	700	600	400	0	1000

TABLE 6.2 – Simulation SystemC des dix modes de fonctionnement

IC0 = Icache IDLE
IC1 = Icache RUN
DC0 = Dcache IDLE
DC1 = Dcache RUN
R0 =RING IDLE
R1 = RING RUN
RAM1=RAM RUN
ROM0 = ROM IDLE
ROM1=ROM RUN
P0 =Proc IDLE
Ps = Proc STUTTER
P1 = Proc RUN

Le cache d'instruction est sollicité à 100% dans le 3ème mode de fonctionnement et il est complètement désactivé dans le 4ème mode. Le cache de donnée est désactivé dans les modes 2, 3, 4, 5 et 7 et travaille à plus que 20% dans les modes 6 et 10. L'anneau d'interconnexion transporte 0 flit dans les modes 2, 3 et 6, tandis qu'il est sollicité à 20% dans les autres modes de fonctionnement. La mémoire RAM est quant à elle désactivée pendant les modes 2, 3, 4 et 6 et elle effectue un traitement supérieur à 25% dans le reste des modes. La mémoire ROM est désactivée dans tous les modes sauf le 4ème. Le processeur tourne à 100% pendant le mode 3, l'évènement $STUTTER$ est mis en avant principalement dans les modes 2 et 6. L'évènement de repos du processeur est mis en avant dans les modes 4, 5 et 7 grâce notamment à des mécanismes de défauts de cache.

6.3 Modèles de consommation

6.3.1 Consommation énergétique par composant pour chaque mode de fonctionnement

Afin de déterminer les énergies élémentaires pour chaque type d'évènement, on procède tout d'abord à la mesure de l'énergie consommée par chacun des composants de la plate-forme représentée sur la figure 6.3 à savoir le processeur, les caches de données et d'instructions, le sous-système d'interconnexion et les mémoires RAM et ROM.

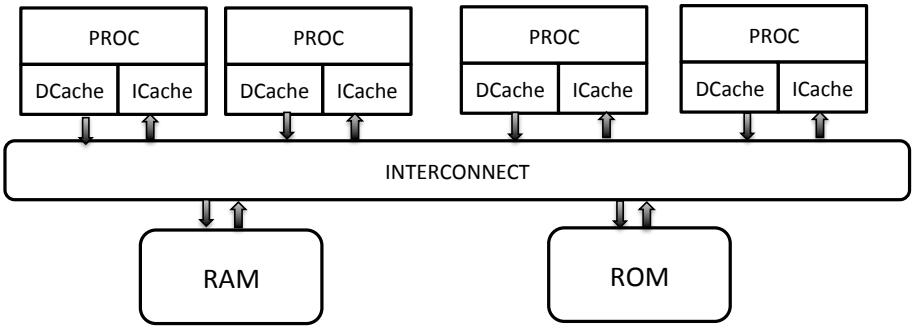

FIGURE 6.3 – La plate-forme d'étude

Le tableau 6.3 représente les puissances dynamiques fournies par PowerPlay pour chacun des composants. On remarque que les résultats fournis par Power-

Modes	Processeur (mw)	Cache d'instructions (mw)	Cache de données (mw)	RING (mw)	Mémoire RAM (mw)	Mémoire ROM (mw)
Mode 0	8,13	2,38	2,34	19,13	0,54	0,86
Mode 1	34,97	10,5	3,84	25,14	8,34	0,86
Mode 2	30,37	11,70	2,39	19,13	0,58	0,86
Mode 3	55,35	11,95	2,42	19,13	0,58	3,89
Mode 4	13,45	2,39	2,38	45,31	0,55	3,81
Mode 5	11,33	4,41	2,38	29,49	13,49	0,86
Mode 6	61,12	12,39	4,76	19,16	0,58	0,86
Mode 7	10,51	2,39	2,38	42,03	6,31	0,87
Mode 8	27,32	12,13	3,8	36,04	7,4	0,86
Mode 9	20,29	12,08	2,43	41,07	5,3	0,88
Mode 10	29,05	12,18	4,63	36,05	10,26	0,86

TABLE 6.3 – Puissance dynamique par composant - Obtenues avec PowerPlay

Play concordent avec les résultats présentés dans la section 6.2. En effet à titre d'exemple pour le cache d'instruction on a une puissance de presque 12mw quand il tourne à plein régime et de 2,39mw quand il est très peu sollicité. Pour ce qui est du cache de données l'augmentation de sa charge de 20% induit une augmentation de la puissance de l'ordre de 2mw.

La puissance de la mémoire RAM passe de 0,58mw à 13,49mw , lorsque son compteur d'évènement RUN passe de 0 à 768. Pour finir, les mesures obtenues pour le processeur sont aussi satisfaisantes. En effet lorsque le compteur d'évènements de type RUN est élevé (Mode 3 et 6) le processeur a une puissance plus importante qui est de l'ordre de 56mw. La figure 6.4, permet de constater la diversité des modes de

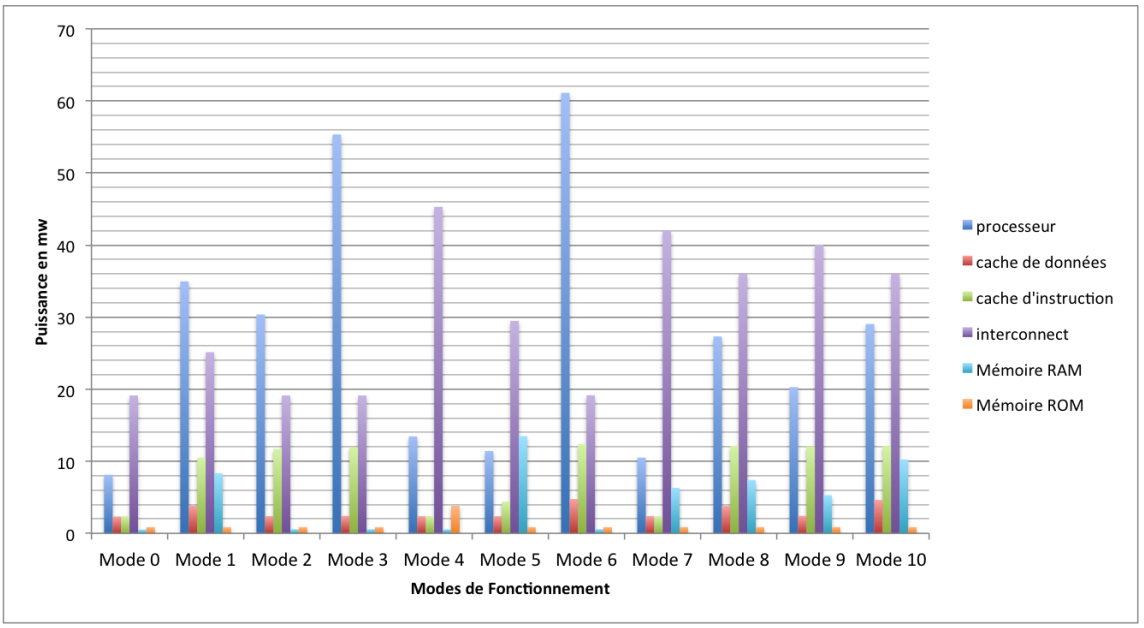

FIGURE 6.4 – Puissances dynamiques consommées par composant et par mode de fonctionnement

fonctionnement. Suivant le mode considéré, certains composants consomment plus d'énergie que d'autres.

6.3.2 Caractérisation du modèle : énergies élémentaires

Dans cette section, nous présentons les résultats obtenus suite à l'application de la méthode de caractérisation décrite dans le chapitre 5. Pour chacun des composants, nous avons appliqué l'algorithme de caractérisation à un sous-système obtenu à partir d'un côté des résultats de simulation $SystemC$ présentés dans la section 6.2 et d'un autre côté des puissances par composants et par mode de fonctionnement obtenues via PowerPlay (cf : section 6.3.1). Le tableau 6.4 liste les modes de fonctionnement utilisés pour la caractérisation de chacun des composants de la plate-forme.

La figure 6.5 montre les énergies élémentaires par composant. On vérifie bien que les énergies élémentaires correspondant à un évènement de repos ($IDLE$) sont inférieures aux énergies des évènements de type RUN. Pour ce qui est du processeur,

Composant	Modes utilisés pour la caractérisation
Processeur	1, 2, 3, 4, 6, 7 et 8
Cache de données	1, 2, 6, 8 et 10
Cache d'instructions	1, 2, 3, 4 et 5
RING	1, 2, 4 et 7
RAM	1, 2, 5 et 7
ROM	1, 2 et 4

TABLE 6.4 – Modes de fonctionnement utilisés pour la caractérisation de chaque composant

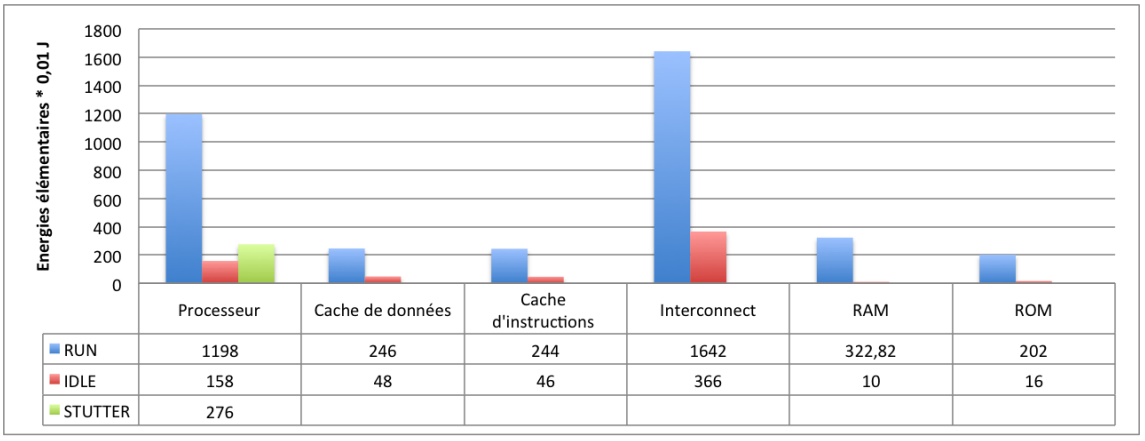

FIGURE 6.5 – Énergies élémentaires par composant

les énergies élémentaires obtenues correspondent à nos attentes, dans la mesure ou l'énergie maximale correspond à un processeur qui exécute une instruction, l'énergie minimale correspond à un processeur gelé. Quant à l'énergie consommée par le processeur lorsqu'il exécute la même instruction pendant au moins deux cycles consécutives, elle est supérieure à celle du repos et inférieure à celle de l'état RUN.

6.4 Consistance de la méthode de caractérisation

Pour prouver la consistance de la méthode de caractérisation présentée dans le chapitre 5, on peut analyser l'erreur obtenue en réinjectant les valeurs des énergies élémentaires obtenues dans le système d'équations qui a servi à la caractérisation.

Les tableaux 6.5 et 6.6 montrent les valeurs des énergies totales par composant obtenues avec l'outil PowerPlay et celles obtenues après réinjection des résultats dans le système.

Le tableau 6.7 représente l'erreur relative moyenne sur l'énergie calculée avec EDEP par composant.

$$Erreur\_Relative_{composant} = \sum_{i} E_{composant}(i) \tag{6.2}$$

Modes	Processur	Cache de données	Cache d'instruction	RING	Mémoire RAM	Mémoire ROM
Mode 1	69,94	7,68	21	50,28	16,68	1,72
Mode 2	60,74	4 ,78	23,4	38,26	1,16	1,72
Mode 3	110,7	4,84	23,9	38,36	1,16	1,72
Mode 4	26,9	4,76	4,78	90,62	1,1	7,62
Mode 5	22,86	4,76	8,82	58,98	26,98	1,72
Mode 6	120,24	9,52	24,78	38,32	1,16	1,72
Mode 7	21,02	4,78	4,78	84,06	12,62	1,75
Mode 8	54,64	7,6	24,26	72,08	14,8	1,72
Mode 9	40,58	4,86	24,16	80,14	10,6	1,76
Mode 10	58,1	9,26	24,24	72,1	20,52	1,72

TABLE 6.5 – Énergies par composant (x 0,01 J) - Obtenues avec PowerPlay

Modes	Processur	Cache de données	Cache d'instruction	RING	Mémoire RAM	Mémoire ROM
Mode 1	77,66	6,99	20,49	61,90	19,04	1,6
Mode 2	69,17	4 ,8	23,12	36,61	1	1,6
Mode 3	117,99	4,8	24,20	36,6	1	1,6
Mode 4	21,49	4,8	4,6	73,54	1	7,47
Mode 5	31,31	4,8	8,73	70,3	25	1,6
Mode 6	97,90	9,64	24,20	36,61	1	1,6
Mode 7	20,27	4,80	4,6	75,79	10,83	1,6
Mode 8	55,6	7,52	24,20	76,48	16,64	1,6
Mode 9	38,57	4,8	24,2	79,13	11,43	1,6
Mode 10	49	9,33	24,2	74,88	19,77	1,6

TABLE 6.6 – Énergies par composant (x 0,01 J) - Résultats obtenus par EDPE

Composant	Processur	Cache de données	Cache d'instruction	RING	Mémoire RAM	Mémoire ROM
Erreur Moyenne	13,47	1,66	1,62	9,53	10,96	6,68

TABLE 6.7 – Erreur relative moyenne sur les résultats fournis par EDPE

Où i est l'indice du mode de fonctionnement et $E_{composant}(i) = \frac{(EPP-EEDPE)*100}{EPP}$

On note que cette dernière est inférieure à 10% pour la plupart des composants et atteint au maximum les 14% pour le processeur.

6.5 Capacité prédictive du modèle de consommation

Afin de valider les modèles de consommation, nous avons testé sa capacité prédictive sur quatre applications, dont une application de traitement d'image et trois applications de tri. En effet pour chacune d'elles nous avons calculé la consommation totale d'abord par notre méthode ensuite en utilisant PowerPlay.

6.5.1 Applications logicielles

Toutes ces applications ont une durée de traitement avoisinant les 600.000 cycles.

6.5.1.1 Tri Fusion (Merge Sort)

Ce tri consiste à fusionner deux sous-séquences triées en une séquence triée. L'exécution de l'algorithme se déroule comme suit :

- On découpe les données à trier en deux parties plus ou moins égales.
- On trie les deux sous-parties ainsi déterminées.
- On fusionne les deux sous-parties pour retrouver les données de départ.

6.5.1.2 Tri par insertion (Insertion Sort)

Cet algorithme parcourt le tableau à trier du début à la fin. Au moment où on considère le i-ème élément, les éléments qui le précèdent sont déjà triés. L'objectif d'une étape est d'insérer le i-ème élément à sa place parmi ceux qui précèdent. Il faut pour cela trouver où l'élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion.

6.5.1.3 Tri rapide (Quick Sort)

La méthode consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite. Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

6.5.1.4 Filter

Cette application consiste à appliquer un filtre à une image de 256 pixels. Le filtre utilisé est un filtre de convolution qui se base sur une matrice de convolution de taille 3x3 . Chacun des pixels de l'image est remplacé par la somme du produit de ce pixel par les 8 pixels qui l'entourent.

6.5.2 Erreur de prédiction

Les puissances des quatre applications sont représentées sur les tableaux 6.8 et 6.9.

Applications	Proc	Icache	Dcache	RING	ROM	RAM
Quick Sort	33,21	11,57	3,37	25,19	0,94	2,09
Insertion Sort	50,18	12,32	4,51	39,33	0,94	7,81
Merge Sort	22,31	7,54	3,16	31,97	0,94	11,62
Filter	9,47	4,92	2,38	29,57	32,92	2,48

TABLE 6.8 – Puissances (mw) : résultats fournis par PowerPlay

Applications	Proc	Icache	Dcache	RING	ROM	RAM
Quick Sort	33,64	9,94	3,55	24,9	0,8	2,93
Insertion Sort	46,84	11,2	4,62	41,32	0,8	8,57
Merge Sort	18,16	7,12	3,16	33,97	0,8	9,39
Filter	13,23	6,4	2,3	29,57	32,92	2,48

TABLE 6.9 – Puissances (mw) : résultats fournies par EDPE

Applications	Erreur relative
Quick Sort	0,75
Insertion Sort	1,5
Merge Sort	0,07
Filter	6,7

TABLE 6.10 – Erreur relative de prédiction

Le tableau 6.10 montre l'erreur relative par application entre la consommation obtenue avec PowerPlay et celle prédite par notre méthode. On note que l'erreur relative ($E_{composant}(i) = \frac{(EPP-EEDPE)*100}{EPP}$) de prédiction est comprise entre 1% et 7%.

6.6 SpeedUp

Une fois qu'on a établi les modèles énergétiques d'une plate-forme matérielle donnée, la méthode EDPE permet une estimation rapide de la consommation de cette plate-forme, pour différentes applications logicielles ou différents déploiements d'une même application sur l'architecture matérielle. Le tableau 6.11 compare les temps de calcul pour réaliser une estimation de la consommation, d'abord avec une méthode d'estimation au niveau RTL telle que PowerPlay, ensuite avec $EDPE$. On remarque que l'accéleration apportée par notre méthode est de l'ordre de 10^4,

Modes	PowerPlay (Jours)	EDEP (s)	SpeedUp
1	8	9,3	9290
2	8	8,8	9818
3	8	9,4	9191
4	8	10,1	8640

TABLE 6.11 – SpeedUP : Portes Logiques (PowerPlay) Vs SystemC (EDPE)

cette différence est principalement dûe à la lenteur de la simulation au niveau portes logiques qui implique de générer un fichier composé de la trace de tous les signaux

de la plate-forme pour une estimation précise de sa consommation par les outils de bas niveaux.

6.7 Conclusion

Dans ce chapitre nous avons d'abord vérifié la forte corrélation entre l'énergie dynamique dissipée par un système embarqué et l'application exécutée par ce dernier.
Partant de cette constatation nous avons établi des modèles de consommation énergétique qui reposent sur des activités fonctionnelles.
Par la suite, nous avons proposé un algorithme d'optimisation et un ensemble de modes de fonctionnement pour caractériser ces modèles.
Pour prouver la capacité prédictive de nos modèles de consommation, nous avons procédé à la comparaison de la consommation énergétique de quatre applications distinctes prédites par EDPE aux résultats de mesures obtenus avec PowerPlay. L'écart entre ces deux mesures est inférieur à 7%.
Enfin nous avons montré que le *SpeedUp* de la méthode EDPE par rapport à la simulation RTL est de l'ordre de 10^4.

Chapitre 7

Conclusion

Nous avons présenté une méthode d'estimation précoce de la consommation énergétique (EDPE) de systèmes multiprocesseurs embarqués sur puce (MPSoC) à espace d'adressage partagé.

La méthode EDPE définit des modèles de consommation énergétiques pour les différents composants qui constituent un système MPSoC. Ces modèles ont été intégrés dans la plate-forme de prototypage virtuel SoCLib, permettant ainsi de prédire la consommation d'un circuit électronique avant même sa fabrication. Cette estimation a pour principal but de permettre une exploration architecturale qui prend en compte non seulement le coût matériel et les temps d'exécution, mais également la consommation énergétique.

Ainsi, la méthode EDPE (Early Design Power Estimation) peut être utilisée pour choisir entre différents déploiements d'une même application logicielle sur une même architecture matérielle. Elle peut aussi être utilisée pour comparer différents algorithmes logiciels déployés sur la même plate-forme matérielle, ou pour comparer la consommation énergétique de différentes plate-formes matérielles.

Nous avons d'abord vérifié la corrélation entre différentes applications logicielles exécutées sur une même plate-forme matérielle et la consommation dynamique du système. Nous avons constaté que la taille des caches et le placement des données impactent significativement l'énergie dissipée par le circuit.

Nous avons démontré – dans le cas d'une réalisation matérielle sur FPGA - qu'il était possible de modéliser d'une façon fidèle la consommation énergétique d'une plateforme MPSoC avec un petit nombre d'activités élémentaires bien choisies, et donc avec un petit nombre de paramètres par composant (au plus trois paramètres dans le cas du processeur).

Nous avons proposé une méthode de caractérisation uniforme et systématique pour tous les composants matériels de la plate-forme. Cette méthode permet de « remonter » dans les modèles comportementaux du prototype virtuel des caractéristiques uniquement disponibles dans la réalisation physique. Cette méthode repose sur la définition d'un petit nombre de scénarios logiciels activant de façon différenciée les différents compteurs d'activité du modèle EDPE, et sur la résolution par une méthode d'optimisation de type gradient, d'un système d'équations linéaires surdéterminé. Nous avons démontré que le calcul permettant de minimiser l'erreur de modélisation converge toujours vers le minimum absolu, puisqu'il n'y a pas de minima locaux. Cette méthode est facilement généralisable à d'autres plate-formes contenant d'autres composants matériels que ceux sur lesquels elle a été appliquée dans cette thèse, et elle est - par construction - peu sensible aux erreurs de mesures physiques.

Nous avons vérifié que - dans le cas d'une implémentation sur FPGA ALTERA - les évaluations de consommation énergétique fournies par l'outil PowerPlay étaient en bon accord avec les mesures physiques réalisées sur le circuit FPGA, et ont le mérite de fournir une information détaillée par composant, ce qui facilite la caractérisation énergétique des modèles.

Nous avons prouvé la capacité prédictive de la méthode EDPE pour différentes applications logicielles déployées sur une architecture matérielle comportant 4 processeurs MIPS32 et différentes mémoires interconnectées par un bus en anneau. L'erreur d'estimation entre les résultats obtenus avec la méthode EDPE et les résultats fournis par PowerPlay est comprise entre 3% et 7%.

Nous avons enfin montré que la méthode EDPE permet une accélération de quatre ordres de grandeur, comparée aux outils d'estimation de la consommation au niveau portes logiques, tels que PowerPlay.

La méthode EDEP a été implantée et évaluée dans les modèles SystemC de la plate-forme SoCLib, mais cette méthode d'estimation précoce de la consommation est facilement transposable à n'importe quel environnement de prototypage virtuel, puisque le modèle de consommation dépend entièrement du comportement fonctionnel des composants modélisés.

Annexe 1

Dans cette annexe nous présentons la bibliothèque statique *SoCView*3.0 permettant l'extraction des compteurs d'évènement de façon non intrusive sans modifier le comportement du matériel :

Listing 1 – MonitorBase.h

```
#ifndef _MONITORBASE_H
#define _MONITORBASE_H
#include <stdint.h>
#include <string>
#include <map>
class MonitorBase {
public:  inline MonitorBase() {};
    inline virtual ~MonitorBase(){};
    virtual std::string get()const = 0;
    virtual void set( const std::string &val) = 0;
    virtual  bool isArray() const = 0;
    virtual std::string getItem(size_t index)const = 0;
    virtual void setItem(size_t index, const std::string &val) = 0;
    virtual size_t arraysize()const = 0;
    virtual const char** value_list() const = 0;
};
extern std::map<std::string, MonitorBase*> ma_map;
#endif
```

Listing 2 – ArrayMonitor.h

```
#ifndef _ARRAYMONITOR_H
#define _ARRAYMONITOR_H
#include <sstream>
#include"MonitorBase.h"

template<typename T> class ArrayMonitor : public virtual MonitorBase {
public:
    inline ArrayMonitor(T a[], size_t s);
    inline virtual ~ ArrayMonitor();  inline bool isArray() const;
    inline std::string getItem(size_t index)const;
    inline void setItem(size_t index, const std::string &val);
    inline size_t arraysize()const;
```

```
13      inline std::string get()const;
14      inline void set( const std::string &val);
15      const char** value_list() const;
16  private:
17      T* _p;
18      size_t _s;
19  };
20
21  template <typename T> ArrayMonitor<T>::ArrayMonitor(T a[], size_t s)
22  {
23      _p=&a[0];
24      _s=s;
25  }
26
27  template <typename T> const char **ArrayMonitor<T>::value_list() const
28  {  return NULL; }
29
30  template <typename T> ArrayMonitor<T>::~ArrayMonitor() { }
31
32  template <typename T> bool ArrayMonitor<T>::isArray() const
33  {  return true; }
34
35  template <typename T> std::string ArrayMonitor<T>::getItem(size_t index)const {  ↩
        std::ostringstream oss;  oss << *(_p + index);  return(oss.str()); }
36
37  template<typename T> void ArrayMonitor<T>::setItem(size_t index, const std::↩
        string &val) {  std::istringstream iss( val );  iss >> *(_p + index); }
38
39  template <typename T> size_t ArrayMonitor<T>::arraysize()const
40  {  return ( _s); }
41
42  template <typename T> std::string ArrayMonitor<T>::get()const
43  { return"erreur"; }
44  template <typename T> void ArrayMonitor<T>::set( const std::string &val)  { }
45
46  #endif
```

Listing 3 – Monitor.h

```
1   #ifndef _MONITOR_H
2   #define _MONITOR_H
3   #include <sstream>
4   #include "MonitorBase.h"
5   #include "EnumValues.h"
6   template<typename T>
7   class  Monitor : public virtual MonitorBase {
8   public:
9       inline Monitor(T &a);
10      inline virtual ~ Monitor();
11      inline std::string get()const;
12      inline void set( const std::string &val);
13      inline bool isArray() const;
14      inline std::string getItem(size_t index)const;
15      inline void setItem(size_t index, const std::string &val);
16      inline size_t arraysize()const;
```

```
17      const char** value_list() const;
18
19      const enum item_type_e item_type() const
20      {
21          return _type;
22      }
23
24      const size_t item_bitsize() const
25      {
26          return _bitsize;
27      }
28
29  private:
30      T* _b;
31      enum item_type_e _type;
32      size_t _bitsize;
33
34  };
35
36  template<typename T>
37  struct sizer
38  {
39      static inline enum MonitorBase::item_type_e type()
40      {
41          return MonitorBase::MONITOR_WIRE;
42      }
43      static inline size_t bitsize()
44      {
45          return sizeof(T)*8;
46      }
47  };
48
49  template<int W>
50  struct sizer<sc_dt::sc_uint<W> >
51  {
52      static inline enum MonitorBase::item_type_e type()
53      {
54          return MonitorBase::MONITOR_WIRE;
55      }
56      static inline size_t bitsize()
57      {
58          return W;
59      }
60  };
61
62  template<>
63  struct sizer<bool>
64  {
65      static inline enum MonitorBase::item_type_e type()
66      {
67          return MonitorBase::MONITOR_WIRE;
68      }
69      static inline size_t bitsize()
70      {
71          return 1;
72      }
```

```
73  };
74
75  template <typename T>
76  Monitor<T>::Monitor(T &a)
77  {
78      _b = &a;
79      _type = sizer<T>::type();
80      _bitsize = sizer<T>::bitsize();
81  }
82
83  template <typename T>
84  Monitor<T>::~Monitor()
85  { }
86
87  template <typename T>
88  const char **Monitor<T>::value_list() const
89  {
90      if ( EnumTable<T>::count )
91          return &(EnumTable<T>::table[0]);
92      return NULL;
93  }
94
95  template <typename T>
96  std::string Monitor<T>::get()const
97  {
98      std::ostringstream oss;
99      oss << *_b;
100     return (oss.str());
101 }
102
103 template <typename T>
104 void Monitor<T>::set(const std::string &val)
105 {
106     std::istringstream iss(val);
107     iss >> *_b;
108 }
109
110 template <typename T>
111 bool Monitor<T>::isArray() const
112 {
113     return false;
114 }
115
116 template <typename T>
117 std::string Monitor<T>::getItem(size_t index)const
118 {
119     return"erreur";
120 }
121
122 template <typename T>
123 void Monitor<T>::setItem(size_t index ,const std::string &val )
124 { }
125
126 template <typename T>
127 size_t Monitor<T>::arraysize()const
128 {
```

```
129     return 0;
130 }
131
132
133 #endif
```

Listing 4 – Factory.h

```
1  #ifndef  _FACTORY_H
2  #define  _FACTORY_H
3  #include <sstream>
4  #include <stdint.h>
5  #include "Monitor.h"
6  #include "ArrayMonitor.h"
7
8  template<typename T>
9  void new_monitor(T &ref, std::string name)
10 {
11     MonitorBase *n = new Monitor<T>(ref);
12     ma_map[name] = n;
13 }
14
15 template<typename T>
16 void new_monitor(sc_core::sc_signal<T> &ref, std::string name
17                  )
18 {
19     T& val= const_cast<T&>(ref.get_data_ref());
20     new_monitor(val, name);
21 }
22
23 template<typename T>
24 void new_arraymonitor(T &ref, std::string name,size_t s);
25
26 template<typename T>
27 void new_arraymonitor(T *ref, std::string name,size_t s)
28 {
29     MonitorBase *n = new ArrayMonitor<T>(ref,s);
30     ma_map[name] = n;
31 }
```

Listing 5 – EnumValues.h

```
1  #ifndef ENUM_VALUES_H_
2  #define ENUM_VALUES_H_
3
4  template <typename T> struct EnumTable{
5    static const char* table[];
6    static const size_t count;
7  };
8
9  template<typename T> const char * EnumTable<T>::table[] = { NULL };
10 template<typename T> const size_t EnumTable<T>::count = 0;
11 #define DeclareOrderedEnum(E, ...)                                    \\
```

```
12 template <> struct EnumTable<E>{                                          \\
13 static const char* table[];                                               \\
14 static const size_t count;                                                \\
15 };                                                                         \\
16                                                                           \\
17  const char * EnumTable<E>::table[] = { __VA_ARGS__, NULL };              \\
18 const size_t EnumTable<E>::count = sizeof(EnumTable<E>::table) / sizeof(void *) ↩
       -1; \\
19  inline std::ostream & operator<<(std::ostream &o, E e)                   \\
20 {                                                                         \\
21   if (e >= 0 && (size_t)e <  EnumTable<E>::count)                        \\
22      o << EnumTable<E>::table[e];                                        \\
23   else                                                                   \\
24     o << "unspecified:" << int(e);                                       \\
25   return o;                                                              \\
26 }                                                                        \\
27 inline std::istream & operator>>(std::istream &i, E &e)                  \\
28 {                                                                        \\
29   std::string value;                                                     \\
30   i >> value;                                                            \\
31   for (size_t j = 0; j < EnumTable<E>::count; j++)                       \\
32     if (value == EnumTable<E>::table[j]) {                               \\
33       e = (E)j;                                                          \\
34       return i;                                                          \\
35     }                                                                    \\
36   return i;                                                              \\
37 }
38
39 #define DeclareValueEnum(E)                                    \\
40 inline std::ostream & operator<<(std::ostream &o, E e)         \\
41 {                                                              \\
42   o << "0x" << std::hex << int(e);                             \\
43 return o;                                                      \\
44 }                                                              \\
45 inline std::istream & operator>>(std::istream &i, E &e)        \\
46 {                                                              \\
47   int j;                                                      \\
48   i >> j;                                                     \\
49   e = (E)j;                                                   \\
50   return i;                                                   \\
51 } #endif
```

Listing 6 – Deuggersrc.h

```
1  // -*- c++ -*-
2  #ifndef _DEBUG_H
3  #define _DEBUG_H
4
5  #define SOCVIEW3
6
7  #include <systemc>
8  #include <iostream>
9  #include <fstream>
10 #include <stdio.h>
11 #include "Monitor.h"
```

```
12 #include "DebuggerRunner.h"
13 #include "MonitorBase.h"
14 #include "ArrayMonitor.h"
15 #include "Factory.h"
16 class Debugger
17 {
18 public:
19
20     Debugger();
21
22     ~Debugger();
23
24     template<typename T>
25     inline     MonitorBase* add(T &x);
26
27     template<typename T>
28     inline     MonitorBase* add(sc_core::sc_signal<T> &x);
29
30
31     template<typename T,size_t n>
32     inline     MonitorBase* add(T x [n]);
33
34     template<typename T>
35     inline     MonitorBase* add(T * ref,size_t  n);
36     // les êmmes fonctions add avc en plus un nom épass en èparamtre
37     template<typename T>
38     inline     void add(T & x,std::string name);
39
40     template<typename T>
41     inline     void add(sc_core::sc_signal<T> &x, std::string name);
42
43     template<typename T,size_t n>
44     inline     void add(T x [n],std::string name);
45
46     template<typename T>
47     inline     void add(T * ref,size_t  n,std::string name);
48
49     // fonction d èaccs!
50
51     const MonitorBase * getM( std::string name) const;
52     MonitorBase *  getD( std::string name);
53     //fonction pour tracer un signal monitore!
54     void trace_all();
55
56     void trace();
57
58     void addsig();
59
60     void updatesig();
61
62     void closetrace();
63
64 private:
65     std::ofstream objetfichier;
66     std::ofstream matrace;
67     void add(MonitorBase* n,std::string name);
```

```
68      std::string conversion(int Decimal,size_t s);
69      typedef std::map<std::string, MonitorBase*> change_me_t;
70      change_me_t this_map;
71  };
72
73  // ajout prend un elt simple en éentre
74  template<typename T>
75  MonitorBase* Debugger::add(T &x)
76  {
77      return new Monitor<T>(x);
78  }
79
80  //ajout d un sc_signal à ma mappe
81  template<typename T>
82  MonitorBase* Debugger::add(sc_core::sc_signal<T> &x)
83  {
84      T& val= const_cast<T&>(x.get_data_ref());
85      return Debugger::add(val);
86  }
87
88
89  // ajout prend un tableau dont elle ignore la taille
90  template<typename T,size_t n>
91  MonitorBase* Debugger::add(T  x[n])
92  {
93      return new ArrayMonitor<T>(x,n);
94  }
95
96  // ajout prend en éentr le tableau et sa taille
97  template<typename T>
98  MonitorBase* Debugger::add(T * ref,size_t  n)
99  {
100     return new ArrayMonitor<T>(ref,n);
101 }
102
103 // ********* les êmme fonction mais avec un nom en plus *****************//
104 template<typename T> void Debugger::add(T & x, std::string name)
105 {
106     MonitorBase * n =Debugger::add(x);
107     add( n, name);
108 }
109
110 template<typename T>
111 void Debugger::add(sc_core::sc_signal<T> &x, std::string name)
112 {
113     MonitorBase * n =Debugger::add(x);
114     add( n, name);
115 }
116
117 template<typename T,size_t n>
118 void Debugger::add(T x [n], std::string name)
119 {
120     Debugger::add( Debugger::add(x), name);
121 }
122
123 template<typename T>
```

```
124 void Debugger::add(T * ref,size_t n,std::string name)
125 {
126     Debugger::add(Debugger::add(ref,n),name);
127 }
128 #endif
```

Listing 7 – DebuggerRunner.h

```
1  #ifndef DEBUGGER_RUNNER_H_
2  #define DEBUGGER_RUNNER_H_
3
4  #include <systemc>
5
6  class Debugger;
7
8  SC_MODULE(debugger_runner)
9  {
10     Debugger &m_debugger;
11
12 protected:
13     SC_HAS_PROCESS(debugger_runner);
14
15 public:
16     sc_core::sc_in<bool> p_clk;
17
18     debugger_runner(sc_core::sc_module_name name, Debugger &d);
19
20 private:
21     void ckup();
22 };
23
24 #endif
```

Listing 8 – Debugger.cpp

```
1  #include <iostream>
2  #include "../include/Debugger.h"
3
4  Debugger::Debugger(){}
5  Debugger::~Debugger(){}
6
7  std::string Debugger::conversion (int Decimal, size_t s)
8  {
9      std::ostringstream oss;
10
11     for (int i=s-1;i>=0; i--) //pour s  bits
12     {
13
14         oss<<((Decimal >> i) & 1);
15
16     }
17     return (oss.str());
18
```

```
19
20 }
21 // fonction assurant l  ajout du monitor à la map
22 void Debugger::add(MonitorBase* n,std::string name)
23 { this_map[name]= n ;
24     std::cout << "Added " << name << " in map " << this_map[name]<< std::endl;
25 }
26
27 /*int Debugger::conversion (int Decimal, size_t s)
28  {
29
30  for (int i=s-1;i>=0; i--) //pour s  bits
31  {
32
33  cout << ((Decimal >> i) & 1)  ;
34
35  }
36  */
37
38 void Debugger::trace_all()
39 {
40
41     objetfichier.open("./monfichier.vcd", std::ios::out); //on ouvre le fichier ↩
        en ecriture
42     if (!objetfichier.bad()) //permet de tester si le fichier s est ouvert sans ↩
        probleme
43         objetfichier
44         << "$timescale 1ps $end" << std::endl
45         << "$scope module logic $end" << std::endl
46         << "$var wire 8 "<<1<<" data $end" << std::endl
47         << "$var wire 1"<<2<<" data_valid $end" << std::endl
48         << "$var wire 1 "<<3<<" en $end" << std::endl
49         << "$var wire 1 & rx_en $end" << std::endl
50         << "$var wire 1   tx_en $end" << std::endl
51         << "$var wire 1 ( empty $end" << std::endl
52         << "$var wire 1 ) underrun $end" << std::endl
53         << "$upscope $end" << std::endl
54         << "$enddefinitions $end" << std::endl
55         << "#0" << std::endl
56         << "b10000001 1" << std::endl
57         << "02" << std::endl
58         << "13" << std::endl
59         << "0&" << std::endl
60         << "1 " << std::endl
61         << "0(" << std::endl
62         << "0)" << std::endl
63         << "#2211" << std::endl
64         << "0 " << std::endl
65         << "#2296" << std::endl
66         << "b0 #" << std::endl
67         << "1$" << std::endl
68         << "#2302" << std::endl
69         << "0$" << std::endl
70         << std::endl;//*
71     objetfichier.close(); //on ferme le fichier pour liberer la émmoire
72
```

```
73      //system("cd ../../ && gtkwave testgtkwave/trace_all.vcd ");
74  }
75
76  //a appeler dans le constructeur du composant
77  void Debugger::trace()
78  {
79      matrace.open("./trace.vcd", std::ios::out); //on ouvre le fichier en ecriture
80      if (!matrace.bad()) //permet de tester si le fichier s est ouvert sans ↩
            probleme
81          matrace
82          << "$timescale"<<sc_core::sc_time_stamp()<<"end" << std::endl
83          << "$scope module logic $end" << std::endl
84          << std::endl;//*
85  }
86
87  static const char *const item_string_type[] = {
88      "wire", "",
89  };
90
91  // a appeler dans le constructeur du composant
92  void Debugger::addsig()
93  {
94      change_me_t::const_iterator i;
95      int cpt=0;
96      for ( i = this_map.begin();i!=this_map.end();i++)
97      {
98          MonitorBase &b = *(i->second);
99
100         matrace <<"$var "<< item_string_type[b.item_type()] << " "<<b.↩
                item_bitsize()<<" "<<cpt<<" "<<i->first<<" $end"<< std::endl
101         <<std::endl;
102         cpt++;
103     }
104     matrace
105     << "$upscope $end" << std::endl
106     << "$enddefinitions $end" << std::endl
107     << "#0" << std::endl
108     <<std::endl;
109 }
110 void Debugger::updatesig()
111 {
112     change_me_t::const_iterator i;
113     int cpt=0;
114     for ( i = this_map.begin();i!=this_map.end();i++)
115     {
116         matrace
117         << "#"<<sc_core::sc_time_stamp().value()<< std::endl
118         <<"b"<<Debugger::conversion(atoi( ((i->second)->get()).c_str() ),sizeof((↩
                i->second)->get()))<<" "<<cpt<<std::endl
119         <<std::endl;//*
120         cpt++;
121     }
122 }
123
124 void Debugger::closetrace()
125 {
```

```
126     matrace.close();
127     std::cout<<"taillle de 3"<<sizeof("3")<<std::endl;
128 
129 }
130 
131 const MonitorBase * Debugger::getM( std::string name)const
132 {
133     change_me_t::const_iterator i = this_map.find(name);
134     return i == this_map.end() ? 0 : i->second;
135 }
136 
137 // fonction get dynamique
138 MonitorBase * Debugger::getD( std::string name)
139 {
140     change_me_t::const_iterator i = this_map.find(name);
141     return i == this_map.end() ? 0 : i->second;
142 }
```

Listing 9 – Debugger_runner.h

```
1  #include "Debugger.h"
2  
3  void debugger_runner::ckup()
4  {
5      m_debugger.updatesig();
6  }
7  
8  debugger_runner::debugger_runner(
9                                   sc_core::sc_module_name name,
10                                  Debugger &d)
11 : sc_core::sc_module(name),
12 m_debugger(d)
13 {
14     SC_METHOD(ckup);
15     sensitive << p_clk.pos();
16 }
```

Annexe2

Dans cette annexe nous démontrons que la fonction de coût définie au chapitre 5 possède un minimum unique.
Soit la fonction de coût définie par :

$$C(X,Y) = sum_i(\alpha_i * X + \beta_i * Y + \gamma_i)^2 \tag{1}$$

$$\frac{\partial C(X,Y)}{\partial X} = \sum_i 2 * \alpha_i * (\alpha_i * X + \beta_i * Y + \gamma_i) \tag{2}$$

ce qui donne :

$$\frac{\partial C(X,Y)}{\partial X} = \sum_i (X * A_X + Y * B_X + C_X) \tag{3}$$

Avec :
$A_X = \sum_i 2 * \alpha_i^2$
$B_X = \sum_i 2 * \alpha_i * \beta_i$
$C_X = \sum_i 2 * \alpha_i * \gamma_i$

De même :

$$\frac{\partial C(X,Y)}{\partial Y} = \sum_i (X * A_Y + Y * B_Y + C_Y) \tag{4}$$

Avec :
$A_Y = \sum_i 2 * \alpha_i * \beta_i$
$B_Y = \sum_i 2 * \beta_i^2$
$C_Y = \sum_i 2 * \beta_i * \gamma_i$

La condition d'extremum est :
$\vec{\nabla}(X,Y) = \vec{(}0)$, soit $\frac{\partial C(X,Y)}{\partial X} = \frac{\partial C(X,Y)}{\partial Y} = 0$.

Il y a une seule solution puisqu'il faut résoudre le système de deux équations à deux inconnues. Les coéfficients A_X,A_Y,B_X,B_Y étant tous positifs, cet extremum

est un minimum. Cette preuve se généralise sans difficultés pour un espace à N dimensions, dans lequel il faut résoudre une système $N * N$.

Références bibliographiques

[1] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007. 3

[2] SoCLib. http://www.soclib.fr. 10

[3] CABA. http://www.soclib.fr/trac/dev/wiki/WritingRules/Caba. 10, 16

[4] Adam Rose, Stuart Swan, John Pierce, and Jean-Michel Fernandez. Transaction Level Modeling in SystemC. http://www.systemc.org, 2005. OSCI TLM Working Group. 10, 16

[5] VSI Alliance. *Virtual Component Interface Standard*, Version 2, OCB 2 2.0 edition, April 2001. 11, 36

[6] TSAR. https://www-soc.lip6.fr/trac/tsar. 12

[7] Ikhwan Lee, Hyunsuk Kim, Peng Yang, Sungjoo Yoo, Eui-Young Chung, Kyu-Myung Choi, Jeong-Taek Kong, and Soo-Kwan Eo. Powerv i p: Soc power estimation framework at transaction level. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 551–558. IEEE Press, 2006. 16, 23

[8] E Gadjeva, T Kouyoumdjiev, S Farchy, M Hristov, and A Popov. Computer modeling and simulation of electronic and electrical circuits using cadence pspice. *Sofia, Meridian*, 22, 2009. 17, 18

[9] Charlie X Huang, Bill Zhang, An-Chang Deng, and Burkhard Swirski. The design and implementation of powermill. In *Proceedings of the 1995 international symposium on Low power design*, pages 105–110. ACM, 1995. 17

[10] Sanjay Dhar. Switch-level timing simulation based on two-connected components, April 19 1994. US Patent 5,305,229. 17

[11] Star-Hspice Model Enhancements. Comparing mos models. *Star-Hspice Manual*, page 353, 2000. 17

[12] Synopsys MEDICI User's Manual. Synopsys inc. *Mountain View, CA*, 2005. 18

[13] Design Power. Synopsys power product reference v1997. 01. *Synopsys Inc., Mountain View, CA*. 18

[14] Robert Kaye. Asic power analysis using quickpower. In *Proceedings: Tenth Annual IEEE International ASIC Conference and Exhibit*, page 364. IEEE, 1997. 18

[15] R.Zimmermann. Powercalc : Power calculator for compass, 1996. 18

[16] Power Compiler User Guide Manual. Release v-2004. *Synopsys Inc., Jun*, 2004. 18

[17] Philips Electronic Design. Tools group. diesel user manual. Technical report, Technical report, Philips Research, 2001. 18

[18] Rafael Peset Llopis and Kees Goossens. The petrol approach to high-level power estimation. In *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, pages 130–132. IEEE, 1998. 18

[19] QuartusII Handbook. vol. 3, ch. 8: Powerplay power analyzer (qii53006-5.1. 0), 2005. 18, 44, 60, 61

[20] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *ACM SIGARCH Computer Architecture News*, 28(2):83–94, 2000. 19

[21] Jörg Henkel and Yanbing Li. Avalanche: an environment for design space exploration and optimization of low-power embedded systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 10(4):454–468, 2002. 21

[22] Premkishore Shivakumar and Norman P Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical report, Technical Report 2001/2, Compaq Computer Corporation, 2001. 22, 26

[23] Vivek Tiwari. *Logic and system design for low power consumption.* Princeton University, 1996. 23

[24] Soo-Kwan Eo. Vip: A practical approach to platform-based system modeling methodology. *JOUNAL OF SEMICONDUCTOR THCHNOLOGY AND SCIENCE*, 5(2):89–101, 2005. 24

[25] Jeff Janzen. Calculating memory system power for {DDR}{SDRAM}. *Designline*, 10(2), 2001. 25

[26] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009. 26

[27] Sanu K Mathew, Mark A Anders, Brad Bloechel, Trang Nguyen, Ram K Krishnamurthy, and Shekhar Borkar. A 4-ghz 300-mw 64-bit integer execution alu with dual supply voltages in 90-nm cmos. *Solid-State Circuits, IEEE Journal of*, 40(1):44–51, 2005. 26

[28] Ana Sonia Leon, Kenway W Tam, Jinuk Luke Shin, David Weisner, and Francis Schumacher. A power-efficient high-throughput 32-thread sparc processor. *Solid-State Circuits, IEEE Journal of*, 42(1):7–16, 2007. 26

[29] Koichi Nose and Takayasu Sakurai. Analysis and future trend of short-circuit power. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(9):1023–1030, 2000. 27

[30] Santhosh Kumar Rethinagiri, Rabie Ben Atitallah, Smail Niar, Eric Senn, and J-C Dekeyser. Hybrid system level power consumption estimation for fpga-based mpsoc. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 239–246. IEEE, 2011. 27

[31] I Miro Panades, Alain Greiner, and Abbas Sheibanyrad. A low cost network-on-chip with guaranteed service well suited to the gals approach. *Proc. NANONET*, 2006. 36

[32] NI PXI. Pci-5122 specification. *National Instruments Corporation*, 2006. 42