Contribution aux Aspects Dorsaux de la synthèse de systèmes monopuces. Optimisation de code pour processeurs embarqués. Analyse de la consommation dans un environnement de synthèse comportementale

Ph. Guillaume

THESE DE DOCTORAT

présentée et soutenue publiquement par

Philippe GUILLAUME

pour obtenir le grade de

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(arrêté ministériel du 30 mars 1992)

Spécialité: Microélectronique

CONTRIBUTION AUX ASPECTS DORSAUX DE LA SYNTHESE DE SYSTEMES MONOPUCES

Optimisation de code pour processeurs embarqués

Analyse de la consommation dans un environnement de synthèse comportementale

---

Date de soutenance : 11 Juin 1999

Composition du jury :

Mrs :	Yves ROBERT	Professeur, ENS Lyon	*Rapporteur*, Président
	Eric MARTIN	Professeur, Univ. Bretagne Sud	*Rapporteur*
	Pierre PAULIN	Docteur, STMicroelectronics	*Examinateur*
	Miguel SANTANA	Docteur, STMicroelectronics	*Examinateur*
	Jean FREHEL	Docteur, STMicroelectronics	*Examinateur*
	Ahmed JERRAYA	Directeur de recherche, CNRS	*Examinateur*

Thèse préparée au sein du laboratoire TIMA, sis à l'INPG, Grenoble
et du département Central R&D de STMicroelectronics, Crolles

Résumé

Les diverses branches de la conception de circuits intégrés, ont tendance aujourd'hui à se fondre en la notion de synthèse de système sur une puce ou de système monopuce. Cela est dû à l'accroissement de la densité d'intégration, couplée à l'évolution des techniques de conception assistées. Au sein du flot de synthèse de systèmes monopuces, deux tendances en particulier se détachent, qui sont l'intégration croissante de logiciel embarqué dans de tels système, et la prise en compte très tôt dans le flot du problème de la consommation. Cette thèse s'intéresse à ces deux aspects de la conception de systèmes actuels.

La première partie se focalise sur l'optimisation de programmes embarqués C. Ces travaux s'attachent principalement à optimiser à haut niveau les performances de programmes faisant un usage intensif de boucles et de tableaux, comme c'est le cas pour les applications de traitement du signal. Les optimisations étudiées et développées au cours de ces travaux, ont pour objectif de se substituer à des transformations manuelles de programmes embarqués, pratique qui reste courante de par l'incapacité de la plupart des compilateurs pour processeurs embarqués à gérer efficacement un code écrit à un niveau élevé.

La seconde partie de cette thèse se donne pour objectif de fournir une méthodologie d'estimation de la consommation dans un environnement de synthèse comportementale. C'est en effet à haut niveau d'abstraction que les stratégies de conception basse consommation ont l'impact le plus important sur la consommation du circuit final. Mais il est nécessaire pour cela de pouvoir juger de l'efficacité des stratégies basse consommation appliquées, à l'aide d'un modèle d'estimation fiable.

Mots clés : *systèmes monopuces, logiciel embarqué, optimisation de code de haut niveau, traitement du signal, estimation de la consommation, synthèse de haut niveau*

Abstract

The notion of system on a chip (SoC) synthesis tends today to group every part of integrated system design. This is mainly due to the continuous improvement of the integration technologies, and to the progress of CAD tools. Within the system synthesis flow, the integration of more and more embedded software, as well as the early handling of the consumption problem, are two points that tend to be critical in this field. This thesis deals with both these aspects of today's system design.

The first part of this thesis is dedicated to C embedded code optimizations. This work attempts to optimize programs that make an intensive use of loops and arrays, as it is the case in DSP applications. The main goal of the studied and developed optimizations is to automate manual transformations that are often currently applied, because of the inability of today's embedded core compilers to efficiently deal with high level code.

The second part of this thesis is devoted to the study and the development of a consumption estimation methodology, within a high level synthesis framework. Indeed, low-consumption strategies are more efficiently applied at a high abstraction level. It is therefore necessary to provide a way of evaluating the efficiency of the strategies applied, through a reliable consumption estimation model.

Keywords: *SoC, Embedded Software, High-Level Code Optimization, DSP, Consumption Estimation, High-Level Synthesis*

Références ISBN

ISBN 2-913329-21-7	Version brochée
ISBN 2-913329-22-5	Version électronique

Remerciements

Survient le moment où l'on se retourne et où l'on observe le chemin parcouru. Le sentier propre à cette thèse est jalonné de rencontres humaines riches qui ont su donner un éclairage ou un tour particulier aux jours passés et aux travaux réalisés. Je voudrais témoigner à l'ensemble des personnes qui m'ont guidé ou accompagné, ma sincère et profonde reconnaissance. Que ceux que j'oublie de citer ci-dessous ne voient pas en cela malice, mais simple humaine faiblesse que je leur prie de bien vouloir pardonner.

Messieurs Ahmed Jerraya, directeur de cette thèse, et Jean Frehel, qui m'ont témoigné leur confiance et grâce auxquels j'ai pu effectuer cette thèse, notamment dans le cadre d'un contrat CIFRE avec STMicroelectronics.

Monsieur Pierre Paulin qui m'a doublement gratifié de sa confiance, tout d'abord en me permettant d'intégrer son équipe, au centre de recherche et développement de STMicroelectronics, pour effectuer ces travaux de thèse, puis en faisant de moi un membre à part entière de cette équipe.

Messieurs Eric Martin et Yves Robert, pour avoir accepté d'être rapporteurs de cette thèse, et pour avoir consacré du temps à pénétrer et comprendre les travaux afférents.

Monsieur Miguel Santana, pour l'esprit rigoureux et clair avec lequel il a pu apporter son support dans l'orientation et l'exposé écrit des travaux réalisés, pour les nombreuses discussions dont il m'a gratifié, pour son soutien et ses conseils de tout ordre, et enfin pour ses talents culinaires dont je garde un souvenir ému.

Les membres passés et présents de l'équipe Embedded Systems Technology, à STMicroelectronics, et pour leur contribution Marco Cornero et son esprit d'ouverture, son enthousiasme et la qualité des discussions que nous avons eu, Benoit Boulanger pour sa contribution à ces travaux de thèse en compagnie de son comparse François Sulmont, sa bonne humeur et ses préceptes sportifs au sujet du cerveau, Clifford Liem, ses conseils et sa disponibilité, et puis Thierry, Xavier, Etienne, Eric, Pierre, Frank, Fred, Michel, Yan, Emmanuel, Christophe, Jean-Claude, et plus récemment Gilbert, Nicolas, Claire et Géraud, Thierry et Fanny. Au sein de STMicroelectronics, Francis, William, José Sanches, et bien d'autres encore...

Au TIMA et à l'INPG, Sonja, Corinne, Chantal, Isabelle, Patricia, Isabelle, Lucie, Muriel, Sylvaine, Alexandro, Nadim, Ahmed, Jean-Pierre, Hubert, Jean-François, Françoise Renzetti pour leur disponibilité et leur gentillesse. Les membres passés et présents de l'équipe System Level Synthesis. Tout d'abord les anciens, Polen, Maher, Hong Ding, Elisabeth, Kevin, Gilberto, Imed, Tarek, Richard, et puis Zoltan, Rodolphe, Philippe, Pascal,

Wonder, Fabiano. Au sein d'autres équipes, en particulier Jean-Marc, Vincent, Damien, compagnons patients et complices, Selim et d'autres là encore ...

François Naçabal pour nos débats et idées partagées, pour nos doutes communs, pour sa générosité et sa patience jamais prises en défaut, pour son soutien, pour une certaine épopée épique. Jean-Marc Daveau, pour nos discussions, pour sa générosité et son accueil, pour un certain jour à Cannes à comparer nos pognes.

Anne-Marie, pour nos joies partagées, pour son soutien dans les moments cruels, pour sa patience et son dévouement, pour sa générosité immense, pour sa confiance en moi quand la mienne faisait défaut, pour son incommensurable contribution à cette thèse.

Ma famille enfin, qui m'a toujours manifesté son soutien et sa confiance, à qui je suis redevable de tant et plus encore, et à qui je dédie avec une reconnaissance sans borne cette thèse.

Table des matières

Liste des figures

Liste des tableaux

Introduction à la thèse

"Chercher n'est pas une chose et trouver une autre, mais le gain de la recherche, c'est la recherche même"

Saint Grégoire de Nysse, *Homélies sur l'Ecclésiaste*

"Il n'est pas difficile d'avoir une idée. Le difficile, c'est de les avoir toutes"

Alain, *Propos*

"Le peu que je sais, c'est à mon ignorance que je le dois"

Sacha Guitry, *Toutes réflexions faites*

Motivations

Le domaine de la microélectronique se caractérise par une évolution technologique permanente et rapide, qui, si l'on examine en particulier l'evolution de la densité d'intégration, est sans pareille : il est aujourd'hui possible d'intégrer sur une même puce plusieurs dizaines de millions de transistors, alors qu'il y a 20 ans, cette densité se comptait en centaines de milliers de transistors.

Ce domaine a largement influencé la vie quotidienne, par les applications innombrables qui ont pu voir le jour ces dernières décennies grâce à elle : la plupart des objets familiers actuels comportent plusieurs circuits intégrés. Le marché poussant les concepteurs vers les limites toujours repoussées de la technologie concerne à l'heure actuelle les produits C^3 (Communication-Computer-Consumer) du type GSM, décodeurs, organiseurs, et tous les produits multimédia. Les applications futures de la microélectronique, compte tenu de son potentiel d'integration présent et à venir, sont virtuellement innombrables.

En parallèle avec l'évolution des technologies, la conception assistée par ordinateur de circuits intégrés a périodiquement subit des révolutions techniques, afin de permettre au concepteur de manipuler des objets toujours plus abstraits, en réponse à l'accroissement de la densité d'intégration et de la complexité des applications devant être intégrées. De nos jours, plusieurs tendances complémentaires se détachent dans la conception de circuits.

La première tendance concerne la nécessité de réutiliser des circuits déjà conçus, de façon à raccourcir et simplifier la conception. Selon M. Laurent (Matra) [97], la conception fortement sub-micronique[1] impose une réutilisation de plus de 70% pour concevoir une puce, soit une conception pure représentant seulement 30%. Cela justifie la ligne actuelle, poussant vers des standards industriels de librairies de macro-composants, comportant des cœurs de processeurs par exemple, et autres composants complexes. Cette tendance est connue sous le vocable de réutilisation de propriétés intellectuelles[2], et se traduit par l'existence de groupes de travail, qui tentent de définir des standards de conception et d'echanges orientés réutilisation[3].

Une seconde tendance peut être qualifiée de "co-everything", c'est à dire de conception conjointe généralisée. Cela va de la conception de multiples blocs complexes conjointement sur une même puce, à la conception conjointe matériel/logiciel (codesign) associée à la co-simulation, en passant par la conception conjointe analogique/numérique. Les diverses branches de la conception ont tendance à se fondre en la notion abstraite de conception de système sur une puce ou de systèmes monopuces (SoC[4]).

La prise en compte de la consommation constitue une troisième tendance, qui a fortement marqué ces dernières années. En effet, l'augmentation de la densité d'intégration, l'accroissement des fréquences de fonctionnement des circuits, et le besoin croissant d'appareils portables épargnant l'autonomie des batteries, ont remis ce problème au goût du jour. C'est ainsi que la consommation est devenue rapidement, avec la surface et la vitesse, la troisième

[1] Deep sub micron or DSM design
[2] IP reuse
[3] On peut citer OMI (Open Microprocessor Initiative) et VSI Alliance
[4] System on a Chip

dimension considérée au cours de la conception de circuits intégrés.

Une dernière tendance se manifeste par l'emploi de plus en plus massif du logiciel embarqué, lors de la conception de systèmes mono-puces. Il se justifie par un besoin de flexibilité et d'évolution rapide des produits, de façon à augmenter le temps de réponse vers un marché toujours plus exigeant et concurrenciel.

Ces travaux de thèse se positionnent dans le cadre de ces deux dernières tendances.

Contribution et organisation

Cette thèse est une thèse en CAO de circuits microélectroniques. Son objectif est de contribuer à la résolution de certains problèmes de conception assistée de circuits intégrés numériques, et plus particulièrement d'apporter une contribution au flot de synthèse de systèmes mono-puces.

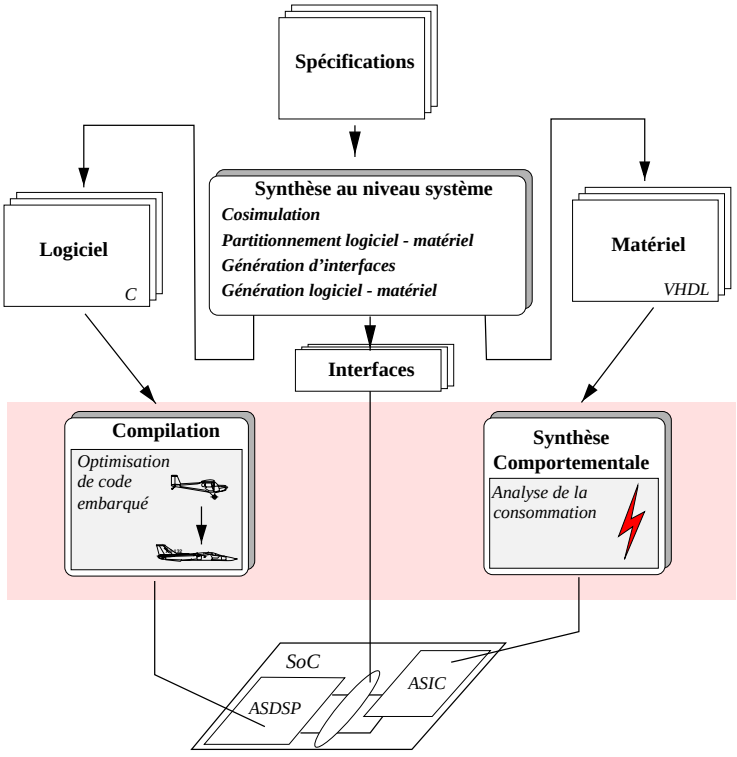


Figure 0.1: Position des travaux de thèse au sein du flot de synthèse système

La figure 0.1 situe les travaux réalisés au cours de cette thèse au sein du flot de synthèse système, qui se propose, à partir des spécifications du système, d'automatiser le partitionnement et la génération d'interfaces entre les parties matérielles et logicielles, en vue de leur intégration sur une puce. La synthèse au niveau système est une des clés de l'avenir de la conception de circuits, et suscite un intérêt généralisé, à la fois dans la recherche académique et dans l'industrie.

Au sein du flot de synthèse de systèmes monopuces, ces travaux de thèse se sont intéressés aux deux cadres de recherche que sont :

- l'intégration de logiciel embarqué dans de tels système, et la nécessité d'optimiser les programmes embarqués en conséquence ;
- la prise en compte de la consommation dès le niveau d'abstraction comportemental.

Cette thèse présente donc la particularité d'être scindée en deux parties distinctes l'une de l'autre, parties correspondant aux deux facettes des travaux réalisés.

La première partie de cette thèse, la plus conséquente, s'intéresse à l'optimisation de programmes embarqués C. Ces travaux s'attachent principalement à optimiser à haut niveau les performances de programmes faisant un usage intensif de boucles et de tableaux, comme c'est le cas pour les applications de traitement du signal. Les optimisations étudiées et développées au cours de ces travaux, ont pour objectif de se substituer à des transformation manuelles de programmes embarqués, pratique qui reste courante, de par l'incapacité de la plupart des compilateurs pour processeurs embarqués à gérer efficacement un code écrit à un niveau élevé. Le déroulement de ces travaux s'est effectué au sein d'une équipe de recherche et développement industrielle[5], et tentent de répondre à des problèmes concrets actuels.

La seconde partie de cette thèse se donne pour objectif de fournir une méthodologie d'estimation de la consommation dans un environnement de synthèse comportementale. C'est en effet à haut niveau d'abstraction, que les stratégies de conception basse consommation ont l'impact le plus important sur la consommation du circuit final. Mais il est nécessaire pour cela de pouvoir juger de l'efficacité des stratégies basse consommation appliquées. Cette seconde partie s'est déroulée au sein d'une équipe de recherche académique[6], ces travaux se voulant être les prémices de recherches dirigées vers des stratégies basse-consommation au niveau comportemental.

[5]Equipe EST (Embedded Systems Technology), Central R&D STMicroelectronics, dirigée par Pierre PAULIN

[6]Equipe SLS (System Level Synthesis), laboratoire TIMA, INPG, dirigée par Ahmed JERRAYA

Partie I

Optimisation de code pour processeurs embarqués

Chapitre 1

Introduction

1.1 Motivations

Les applications grand public liées au monde du multimédia et aux systèmes de télécommunication, représentent l'une des révolutions techniques de ces dernières années qui ont le plus bouleversé la vie quotidienne. Une surenchère incessante pousse vers toujours plus de technicité au sein du marché de l'électronique grand public. Ces avancées techniques sont le résultat de plusieurs facteurs, dont le plus déterminant d'entre eux est relatif aux technologies d'intégration toujours plus poussées, qui permettent de connecter désormais sur une même puce des composants formant un véritable système à part entière.

En parallèle aux avancées physiques, une révolution que l'on occulte souvent, concerne les outils de CAO de circuits, qui ont su évoluer sans cesse de façon à permettre au concepteur d'exploiter au mieux la puissance d'intégration à sa disposition. Ces outils tendent à considérer des niveaux d'abstraction toujours plus élevés, afin de répondre à la complexité grandissante associée à l'accroissement des possibilités de densité d'intégration.

Les industries du secteur microélectonique et applications, compte-tenu des fenêtres de marché toujours plus étroites - un produit se démode en quelques mois - doivent faire montre de réactions rapides et d'anticipations stratégiques, sur les désirs du public et les évolutions des technologies. Le fameux "time-to-market[1]" impose sa loi despotique : il faut être le premier au bon moment, c'est-à-dire dans la bonne fenêtre temporelle. Qui dit système monopuce, et là est la tendance, dit complexité importante et donc temps de conception accru, ce qui est peu compatible avec les nécessités du marché. Il faut donc ramener ce temps de conception à des intervalles raisonnables, ce qui a promu notamment l'emploi massif du logiciel et l'intégration de plus en plus répandu de cœurs de processeurs embarqués lors de la conception de tels systèmes. Les avantages se comptent essentiellement en deux points :

- Flexibilité en termes de conception : il est plus simple de corriger et modifier un programme C qu'un ASIC, notamment en cas d'interactions fortes entre le bloc considéré et le reste du système.

[1] délai de mise sur le marché d'un produit

- Flexibilité en termes d'évolution : faire évoluer un produit conçu comme un pur ASIC est une tâche très difficile. Par contre, si les parties-clés du système sont logicielles, l'évolution en est simplifiée. La simplicité d'évolution est un point-clé de la rapidité de réponse à l'évolution rapide de standards, comme les standards ISO.

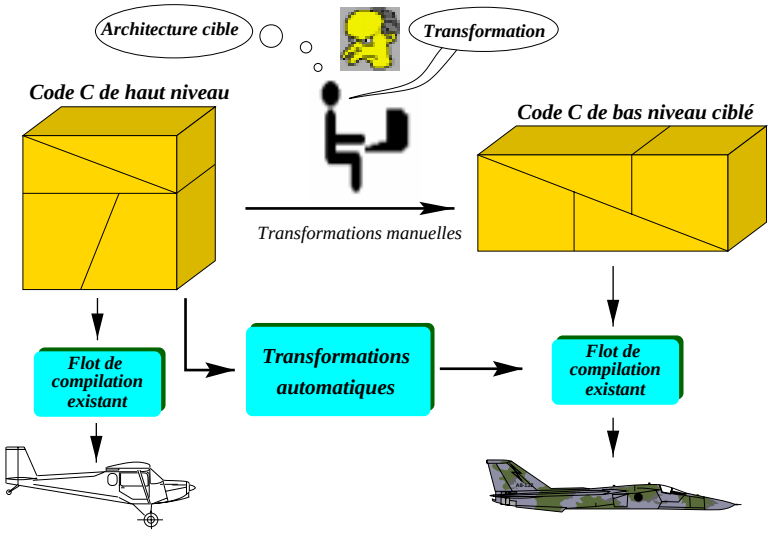


Figure 1.1: Ciblage d'un code applicatif de haut niveau vers une machine.

Compte-tenu de la complexité des nouvelles applications accompagnant les cœurs de processeurs embarqués, la programmation très bas niveau, voire assembleur, encore employée récemment, n'est plus de mise : ce type de programmation est abandonné, au profit de langages de haut niveau, tel que C. Par conséquent, ces applications ont besoin d'être compilées sur le processeur cible. L'expérience montre que la grande majorité des compilateurs accompagnant les cœurs de processeurs, sont incapables de compiler efficacement un programme écrit notamment en C, en exploitant les caractéristiques de haut niveau disponibles dans le langage. Comme il est dépeint sur la figure 1.1, une étape de réécriture manuelle de l'application s'avère nécessaire. Cette transformation manuelle, génératrice d'erreurs, augmente les performances du code généré et s'effectue en fonction des spécificités de la machine. Le code résultant, bien moins lisible, en perd dans une large mesure sa portabilité et sa maintenabilité, d'autant qu'il n'est pas rare de devoir toujours écrire directement en assembleur les parties critiques du code, comme les boucles.

1.2 Objectifs et organisation de l'exposé des travaux

Ces travaux tentent de contribuer au but ultime de la compilation : générer un code aussi efficace que peut le générer l'écriture manuelle, et cela à partir d'un programme C exploitant les caractéristiques de haut niveau du langage. Il s'agit d'automatiser les

transformations traditionnellement effectuées manuellement, car non disponibles dans les chaînes de compilation communément rencontrées, comme le dépeint la figure 1.1.

Dans le contexte industriel particulier au sein duquel ces travaux se sont déroulés, les objectifs initiaux cités ci-dessous ont sous-tendu les études et développements réalisés :

- Étudier le domaine de la compilation en général, et plus précisément les problèmes relatifs à la compilation d'applications embarquées, afin d'extraire des travaux antérieurs les transformations ayant un potentiel de gain en performance et d'application pratique les plus pertinents. L'effort s'est orienté vers les transformations plus précisément adaptées aux applications de type traitement du signal.

- Identifier et approfondir, à partir des besoins actuels et en s'appuyant sur des travaux existants, une ou plusieurs transformations de code de façon à étudier dans un contexte réel l'efficacité véritable de celles-ci.

- Développer un prototype de transformation afin d'expérimenter les problèmes et besoins conditionnant la mise en œuvre pratique de celle-ci.

Cette partie présente le résultat de ces études et développements. Le chapitre 2 donne un bref état de l'art des domaines liés au monde du logiciel embarqué aussi bien en termes de processeur, que d'application et compilation. Le chapitre 3 détaille en particulier un algorithme approfondi et développé dans l'esprit énoncé ci-dessus. Il s'agit d'une transformation consistant à remplacer les tableaux en pointeurs dans des boucles C, de façon à exploiter plus efficacement les ressources d'adressage de la machine cible. Cette transformation est orientée en tenant compte d'informations sur la machine cible. Un certain nombre de transformations identifiées comme potentiellement intéressantes au cours de ces travaux, et relatives à la transformation développée plus précisément, sont expliquées. Quelques résultats d'expérimentations à l'aide de deux processeurs cibles développés par STMicroelectronics sont dévoilés chapitre 4 avant de conclure.

Chapitre 2

Cheminement dans le monde de l'embarqué

2.1 Préambule

Intégrer une partie logicielle dans un système, suppose de gérer le trinôme "processeur – programme – compilateur". Dans le monde du logiciel embarqué, chacun des éléments de ce trinôme est indissociable des deux autres. Ce chapitre se propose de détailler les spécificités de chacune de ces entités, telles qu'elles ont pu être rencontrées au cours de cette thèse. Les points évoqués dans ce chapitre ont grandement influés sur l'orientation des travaux réalisés. En premier lieu, un processeur embarqué présente certaines caractéristiques générales que nous tenterons de percer. Les propriétés d'un programme destiné à être exécuté sur un processeur enfoui[1] feront l'objet d'une seconde partie, tandis que les particularités de la compilation de programmes embarqués seront ensuite précisées. Enfin, les interactions triangulaires entre les trois composantes du logiciel enfoui seront exposées.

2.2 Processeurs embarqués

2.2.1 Quelques points caractéristiques

Si l'on utilise une formulation naïve, un processeur embarqué ou enfoui, est un cœur de processeur, destiné à être composant d'un système plus vaste, que ce soit une carte ou plus récemment une puce unique. Par cœur de processeur, il faut entendre en règle générale, la partie centrale d'un processeur, accompagnée, suivant les vendeurs, d'un nombre plus ou moins grand de périphériques.

Les processeurs enfouis doivent répondre à des contraintes de performances fortes, qualifiées de temps réel [125, 101]. Le monde extérieur envoyant des informations en continu,

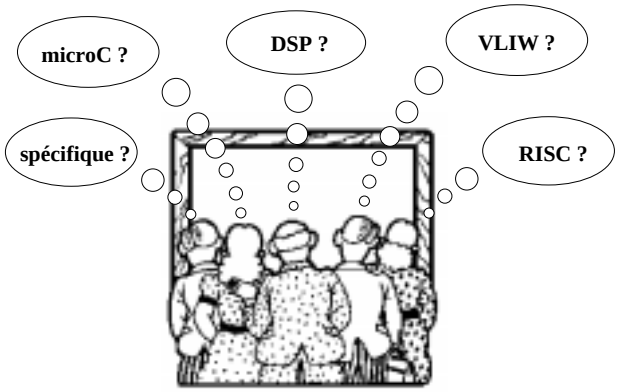
[1] Les termes *enfoui* et *embarqué* sont les synonymes les plus couramment rencontrés pour désigner un cœur de processeur intégré dans un système

il faut prévoir une forte réactivité du processeur destiné à être embarqué. Il faut notamment que les fonctions périphériques spécialisées dans le traitement des interruptions soient efficaces et répondent à la contrainte temps réel.

Le problème de la densité de code a tendance à devenir un critère majeur dans la sélection d'un processeur embarqué. Ce problème est relatif à la taille du code applicatif, et donc à la taille mémoire requise pour le stocker. Etant donné la complexité croissante des applications embarquées, la mémoire tend à occuper la majeure partie d'une puce. C'est ainsi que grâce aux possibilités actuelles d'intégration, la taille du cœur de processeur n'est plus aussi importante dans les critères de choix et/ou de conception d'un processeur embarqué. L'importance relative de la taille mémoire en est principalement la cause, et l'on préférera parfois un processeur plus gros mais fournissant une bonne densité de code, à un petit processeur donnant une densité médiocre.

2.2.2 Propriétés architecturales

Quelques critères architecturaux tendent cependant à se retrouver fréquemment dans le domaine des processeurs embarqués. Tout d'abord le style de conception RISC qui, bien que loin d'être spécifique au monde de l'embarqué, mérite un bref arrêt en raison de l'influence qu'il a pu avoir sur la conception depuis les années 80. Ses principes se retrouvent dans nombre de processeurs embarqués actuels, de façon plus ou moins pure. D'autre part, les processeurs dédiés au traitement du signal, ou DSP, représentent une très grande part des processeurs embarqués actuels : leur emploi ne cesse de croitre. Les travaux exposés dans ce document ont visé en particulier ce type de processeurs, ce qui justifie un aperçu de leurs caractéristiques. Pour les mêmes raisons, les processeurs embarqués spécifiques font l'objet d'une section.



2.2.2.1 Processeurs à jeu d'instructions réduit - RISC

Le style architectural RISC a enflammé la conception des processeurs à partir des années 80. On en retrouve certains principes dans la plupart des processeurs actuels, et notamment les processeurs embarqués, sans pour autant que ceux-ci puissent être qualifiés de RISC.

IBM fut à l'origine de nombres d'idées sous-tendant le concept RISC [47, 7, 115, 3] (mais pas de l'acronyme), et en formalisa les principes via le projet 801 en 1975. Les universités de Berkeley (RISC I, RISC II et SOAR) et Stanford (MIPS) aux alentours de 1981 ont participé à la naissance de ce concept.

Ce type de processeur présente un ensemble de caractéristiques s'opposant quasi systématiquement à tous les points que l'on retrouve dans les processeurs CISC, ou processeurs à jeu d'instructions complexe. Tout d'abord un jeu d'instructions simple, voire simpliste, de longueur fixée. La simplification de la machine permet de réduire le temps de cycle. Chaque instruction s'exécute en un cycle, par le biais de l'emploi intensif et systématique de la technique du pipeline. Les opérations se font exclusivement de registre à registre[2]; cela se justifie notamment par l'évolution plus lente de la vitesse des mémoires comparativement à celle des processeurs, ce qui nuit à la pertinence des opérations mémoire-mémoire. Un grand nombre de registres, parfois associé à la technique du fenêtrage, permet en particulier d'accélérer les appels de fonctions. Enfin, des instructions sont spécialisées dans les accès mémoire, ce qui conduit à l'emploi d'un synonyme à l'acronyme RISC : on parle d'architecture *load/store* (chargement/stockage). L'architecture elle même est plutôt de type Harvard, qui prône l'utilisation de bus séparés et de mémoires séparées pour le programme et les données, autorisant des accès simultanés à ces deux populations. Par ailleurs, la symétrie (ou orthogonalité) est de rigueur dans le jeu d'instructions : chaque instruction est sensée pouvoir opérer sur n'importe quel registre, et n'avoir ni exceptions, ni restrictions d'aucunes sortes, ni effets collatéraux. Pas d'utilisation de micro-code, qui caractérise la famille des processeurs CISC. La liste des familles de processeurs RISC est longue : les principales sont SPARC, MIPS, DEC Alpha, Motorola, IBM, Intel i960, AMD 29000...

2.2.2.2 Processeur de traitement du signal - DSP

Un DSP[3] est un processeur orienté vers le calcul intensif propre aux applications de traitement du signal. Quelques points architecturaux sont communs, voire incontournables, aux processeurs de cette famille [30, 39, 64, 65, 99, 115].

L'architecture d'un DSP se caractérise tout d'abord par la possibilité d'effectuer une opération de multiplication accumulation, plus communément désignée MAC, généralement en un seul cycle.

Les applications de traitement du signal devant traiter un flot de données continu, les architectures DSP doivent être capable de gérer un flot de données très rapide, provenant et allant vers les unités de calculs et la mémoire. C'est ainsi que l'on trouvera fréquemment des bancs de registres multi-port, ainsi que la possibilité de fournir aux unités de calculs plusieurs échantillons de données en parallèle par cycle. L'emploi d'unités de génération d'adresses, associées à plusieurs mémoires, est courant. On trouvera fréquemment par exemple deux mémoires, X et Y, ayant chacune leur unité de génération d'adresses et leur bus, et auxquelles on pourra accéder de façon concurrente. Les unités de génération d'adresses pro-

[2] le terme registre-registre est parfois utilisé pour désigner les processeurs conçus dans la philosophie RISC

[3] Digital Signal Processor (signifie aussi Digital Signal Processing)

posent des modes d'adressage spécialisés. Le plus commun est le mode d'adressage indirect avec post-incrémentation. L'efficacité des processeurs DSP est souvent très dépendante d'un minimum de parallélisme inhérent au jeu d'instructions (ILP[4] [63, 113, 81, 116]), comme le chargement parallèle de données mémoires, ou l'exécution simultanée d'une addition et d'une multiplication.

Les algorithmes de traitement du signal s'articulent, dans leur grande majorité, autour de boucles. La structure de ces boucles est souvent simple, et ses limites inférieure et supérieure connues statiquement. C'est la raison pour laquelle un mécanisme de boucle matérielle[5], autrement appelé mécanisme de boucle sans frais[6], est souvent rencontré dans les architectures DSP. Grâce à ce type de mécanisme, capables dans certains cas de gérer plusieurs boucles imbriquées, pas d'instructions additionnelles de test de fin de boucle, ou d'incrémentation du compteur de la boucle: cela est géré en dur dans le processeur lui-même. Outre ce mécanisme de contrôle des boucles, certains DSPs fournissent d'autres caractéristiques de contrôle destinées à améliorer les performance globales de la machine, comme des mécanismes de changement de contexte rapides, ou d'interruption à bas coût.

Les DSPs doivent pouvoir manipuler des problèmes avec une précision étendue, ce qui nécessite l'emploi d'accumulateurs spéciaux et de registres plus larges que la taille admise, pour éviter les dépassements et les problèmes d'arrondis. D'autres caractéristiques architecturales se rencontrent, comme les possibilités d'adressage modulo ou circulaire des données, ou bit-inversé utile pour un algorithme comme la transformée de Fourier rapide (FFT).

Le domaine d'application a littéralement explosé au cours des dix dernières années. On peut citer la reconnaissance, la synthèse, l'encodage et le décodage de la parole, les modems, l'encodage audio hi-fi, la compression d'images et tout ce qui concerne le traitement du son. Les DSPs trouvent une application dès qu'il s'agit d'interpréter et de modifier de façon complexe les données de l'environnement. De nombreux vendeurs proposent aujourd'hui des solutions DSP. Citons le D950 de ST, TI et ses Cxx, ARM et le Piccolo qui s'apparente à un coprocesseur DSP pour le cœur ARM7, Analog Devices, Motorola ... [66, 65, 14]

2.2.2.3 Zoom sur les processeurs embarqués spécifiques - ASIPs

Les ASIPs[7], ou processeurs embarqués spécifiques, possèdent un ensemble de propriétés architecturales qui présentent de grands avantages dans le cadre de systèmes embarqués, comparé à l'utilisation de processeur plus standards [25, 26, 81].

Un ASIP est le plus souvent indissociable du groupe d'application pour lesquelles il a été conçu, dans ses propriétés intrinsèques. La raison de l'existence des ASIPs et de leur intérêt est très liée aux raisons justifiant l'emploi de circuits intégrés spécifiques ou ASICs[8] : les processeurs à usage général savent tout faire avec de bonnes performances, mais cette

[4]Instruction Level Parallelism
[5]hardware looping mecanism
[6]zero-overhead looping mecanism
[7]Application Specific Instruction-set Processors
[8]Application Specific Integrated Circuits

puissance a un coût, en termes de surface et de consommation, coût parfois incompatible avec les contraintes d'intégration d'une application ou de parties de celle-ci. Il devient alors judicieux d'employer des composants fournissant la puissance de calcul requise, sans plus, mais avec un coût bien moindre. L'avantage d'un ASIP face à un ASIC, réside alors dans sa programmabilité, qui se traduit en flexibilité face à ce dernier, dont la fonctionnalité est fixée une fois pour toutes. Il est d'autre part plus facile de faire évoluer une application dans le temps, et de fournir des versions améliorées du système global, sans pour autant reconcevoir le processeur.

De façon générale, un ASIP possède un jeu d'instructions réduit, comprenant un ensemble d'instructions standards destinées aux opérations arithmétiques, au contrôle ou à la mémoire; ce jeu d'instruction est choisi dans l'objectif d'être utile à l'application associée. Le jeu d'instructions d'un ASIP peut comporter en outre un ensemble d'instructions spécialisées, dans l'objectif d'exécuter certaines fonctions typiques du groupe d'applications ciblé de la façon la plus efficace. Ces fonctionnalités spécifiques sont le plus souvent choisies de façon à réduire au minimum le temps d'exécution des parties critiques de l'application, et notamment des boucles. Un pipeline simple dépassant rarement une profondeur de 2 ou 3, et des fréquences de fonctionnement relativement basses (40 - 100 MHz) comparées aux architectures à usage général, caractérisent par ailleurs les ASIPs. Ils peuvent être dédiés au contrôle ou au calcul intensif, et présenter un certain degré d'ILP.

Les ASIPs orientés vers le calcul intensif et le traitement du signal sont plus communément nommés ASSP (Application Specific Signal Processor), ou ASDSP (Application Specific Digital Signal Processor) [39, 25].

Une des difficultés, typique de ces processeurs, est qu'ils sont souvent conçus d'une façon inamicale pour la compilation. Cela est souvent lié à une structure de registres hétérogène, c'est à dire une architecture dont les registres sont distribués et spécialisés, connectés à des unités fonctionnelles spécifiques. Cette hétérogénéité architecturale a pour objectif de diminuer la taille du mot instruction, ce qui se traduit le plus souvent par un jeu d'instructions qualifié de non-orthogonal. Un jeu d'instructions non orthogonal, ou encodé, aura différents formats d'encodage permettant de tenir compte des spécificités architecturales hétérogènes. Dans le cas de fortes contraintes d'encodage, le parallélisme offert par le processeur est plus restreint. En outre, un petit nombre de registres disponibles, la présence d'opérations très spécifiques à l'application ou au domaine d'application ciblé, la structure mémoire parfois singulière, complexifient d'autant la tâche des compilateurs.

2.3 Programmes embarqués

2.3.1 Spécificités

Un programme embarqué, est un programme destiné à être exécuté sur un processeur embarqué, c'est-à-dire un cœur de processeur, lui-même composant d'un système plus vaste.

Comparé à un programme destiné à être exécuté sur une station de travail, un programme embarqué présente aujourd'hui certaines propriétés importantes à souligner, et

qui influencent notamment leur compilation. Nous distinguons essentiellement les caractéristiques des programmes embarqués de type DSP, sur lesquels ces travaux de thèse se sont plus particulièrement penchés, par opposition aux programmes embarqués de type réactif, fortements orientés contrôle :

- La taille d'un tel programme est souvent réduite à quelques centaines, voire quelques milliers de lignes de code C, ce qui résulte en une taille mémoire entre 4ko et 16ko. Cela signifie que la totalité du programme est le plus souvent maîtrisable par le compilateur, d'autant qu'il est souvent mono-module, c'est à dire qu'il se compose d'une seule partie regroupant la totalité du code. Le compilateur a alors la possibilité de lui appliquer des transformations globales efficaces.

- Un autre point, est l'absence d'allocation dynamique de la mémoire (pas de malloc ...), couplé à l'emploi exclusif de librairies statiques, ce qui simplifie la tâche d'édition de lien du programme. Par ailleurs, la gestion des entrées/sorties est souvent simplifiée dans ce type de programme, limitée au flot de données reçu et délivré.

- Dans les applications de type traitement du signal, un programme embarqué est très souvent exécuté de façon cyclique, au rythme du flux de données qui lui est fourni. Au sein de cette boucle globale de traitement, certaines parties critiques comptent des opérations parfois complexes, exécutées très fréquemment[9].

- L'exécution d'une application de traitement du signal est souvent régie par des contraintes temporelles fortes : le traitement du flot de données reçu doit s'effectuer en un temps limité. L'objectif est alors d'obtenir des performances suffisantes pour ne pas sortir de la fenètre temporelle autorisée. Plus les performances du code machine seront élevées, plus le temps libre restant permettra d'enrichir l'application avec des fonctions supplémentaires, qui pourront faire la différence sur un marché très concurrentiel.

2.3.2 Style d'écriture

Historiquement, de larges portions d'applications embarquées étaient écrites en assembleur, notamment dans leur parties critiques. Aujourd'hui, les langages de haut niveau, comme C et dans une moindre mesure C++, tendent à s'imposer comme langages privilégiés d'écriture. Cela s'explique par l'augmentation de la complexité des applications, par le besoin de portabilité de celles-ci, par l'avènement de technologies de compilation plus efficaces, et par l'arrivée sur le marché d'architectures plus amicales pour les compilateurs, mais beaucoup moins amicales pour une écriture assembleur. Si l'on prend l'exemple de l'architecture C6x de TI, qui comporte 8 unités fonctionnelles parallèles, on peut concevoir

[9] Ces opérations se voient souvent synthétisées sous forme matérielle pure, et adjointes au processeur en tant que co-processeurs, à moins qu'elles ne soient présentes dans le premier en tant qu'instructions spécialisées.

la difficulté d'écrire un programme en assembleur pour celle-ci, en exploitant de façon optimale et manuellement l'efficacité de ces unités.

En pratique, l'expérience met en lumière un trait caractéristique du monde embarqué, qui est la sensibilité des performances d'une application relativement à son écriture. Cette sensibilité, déjà présente pour les applications logicielles à usage général, est ici exacerbée. Elle est exploitée le plus souvent à des fins d'optimisation manuelle, et tend à démontrer le travail restant à accomplir pour fournir aux concepteurs d'applications une technologie de compilation leur permettant de n'évoluer qu'à haut niveau, et de s'abstraire des exigences matérielles.

On distingue ainsi quatre niveaux d'écriture du C [72, 92], qui est le langage exclusivement considéré au cours de ces travaux de thèse comme langage d'écriture d'applications à destination d'un processeur embarqué.

1. Le premier niveau est le niveau comportemental, respectant la norme ANSI C, faisant usage de constructions de haut niveau comme tableaux et structures, et manipulant des variables classiques et les opérations disponibles dans le langage. C'est le niveau d'expression algorithmique par excellence, donnant le style de code le plus lisible et dont la portabilité est totale. L'efficacité du code exécuté sur le processeur cible est en général assez mauvaise, voire médiocre, tant en taille qu'en vitesse.

2. Le second niveau, ou niveau intermédiaire, voit les références à des structures de données de haut niveau remplacées par l'utilisation de pointeurs. Les variables peuvent être allouées à des classes de registres ou à des bancs de registres, par l'intermédiaire de directives standards ou de raccourcis d'écriture. Des fonctions prédéfinies font leur apparition, permettant d'améliorer les performances. Le code reste compilable et exécutable sur une machine hôte, donc portable, à condition que les directives permettant de cibler le code initial vers la machine cible, ainsi que les fonctions prédéfinies, soient bien conçues. Mais sa lisibilité est amoindrie par rapport au précédent niveau, et sa portabilité plus faible de part l'orientation sensible de l'écriture vers l'architecture destination.

3. Le troisième niveau permet l'assignation manuelle de variables, données ou pointeurs, à des registres précis de l'architecture cible à l'aide, là encore, de directives spécifiques. La portabilité baisse encore d'un degré, de même que la lisibilité. Le code peut rester compilable et exécutable sur une machine hôte, avec les mêmes contraintes que le niveau précédent.

4. Le dernier niveau d'écriture considère un mélange de C bas niveau agrémenté de directives d'assignation, et d'assembleur. Lisibilité proche de celle de l'assembleur, c'est-à-dire noyade de l'algorithmique dans des opérations élémentaires, portabilité nulle, mais performances très améliorées.

L'évolution manuelle d'un code comportemental C vers les niveaux d'écriture plus bas se fait donc au prix de la lisibilité et de la portabilité, mais avec à la clé une compilation facilitée et des performances bien meilleures. Cette distinction de niveaux d'écriture, couplée

à la performance du code ainsi réécrit, est fondamentale dans le cadre de ces travaux. Cette particularité a en effet été exploitée à des fins d'expérimentations et de prototypage, afin d'automatiser certaines transformation traditionnellement effectuées manuellement et de tester leur efficacité en générant du code de bas niveau, en l'occurence de niveau 2 et 3, à partir d'un code de haut niveau.

[65] relate une expérience intéressante, consistant à fournir à divers vendeurs de DSPs et compilateurs associés, des fonctions typiques du traitement du signal, écrites en C, afin de tester l'efficacité du code obtenu après compilation. L'expérience consiste donc en deux volets : compiler le code sans le modifier[10], et compiler le code modifié manuellement de façon à obtenir de meilleures performances, sans changer évidemment la fonctionnalité. Une simulation des codes non-modifiés et modifiés est employée afin de fournir les performances en vitesse d'exécution. Les vendeurs participant à l'expérience sont parmi les acteurs principaux du marché : DSP Group et le cœur OakDSPCore, Hitachi SH-DSP, Motorola DSP5600x et DSP563xx, TI C54x et C6x, ... Les résultats sont édifiants : l'amélioration en termes de nombre de cycles d'exécution va du simple au triple, entre le code original et celui modifié manuellement.

2.4 Compilation de programmes embarqués

La conception d'un compilateur tend à devenir primordiale dans le domaine des systèmes embarqués. Le besoin est grand de générer rapidement des programmes efficaces, ce qui reste difficile lorsque des optimisations manuelles doivent être appliquées. Cette section se consacre aux aspects compilation pour programmes embarqués.

2.4.1 Généralités

De manière générale, la compilation est l'art de traduire un langage dit source, en un langage dit cible. Nous nous intéresserons ici à la compilation d'un langage évolué (C) vers un langage machine (assembleur). La compilation se déroule en plusieurs étapes, destinées à analyser le code source et vérifier sa cohérence, le traduire dans une structure intermédiaire, le transformer, pour finalement générer un code exécutable sur le processeur cible. C'est l'étape de transformation qui retiendra plus particulièrement notre attention ici [1, 89]. Quelques mots cependant sur les principales phases d'un compilateur très général, dépeint sur la figure 2.1.

La partie frontale d'un compilateur consiste essentiellement à analyser le programme source et à le traduire en une structure intermédiaire (RI) propre à être manipulée[1, 62]. L'analyse se découpe en analyse lexicale[11], qui extrait les mots clés du langage source, et analyse syntaxique[12] qui vérifie la conformité du programme à un certain nombre de

[10] ou très légèrement (Ex: short → int)

[11] Un analyseur lexical très connu est *lex*, et sa version plus récente *flex*

[12] Par exemple *yacc*, ou *bison* qui en est dérivé

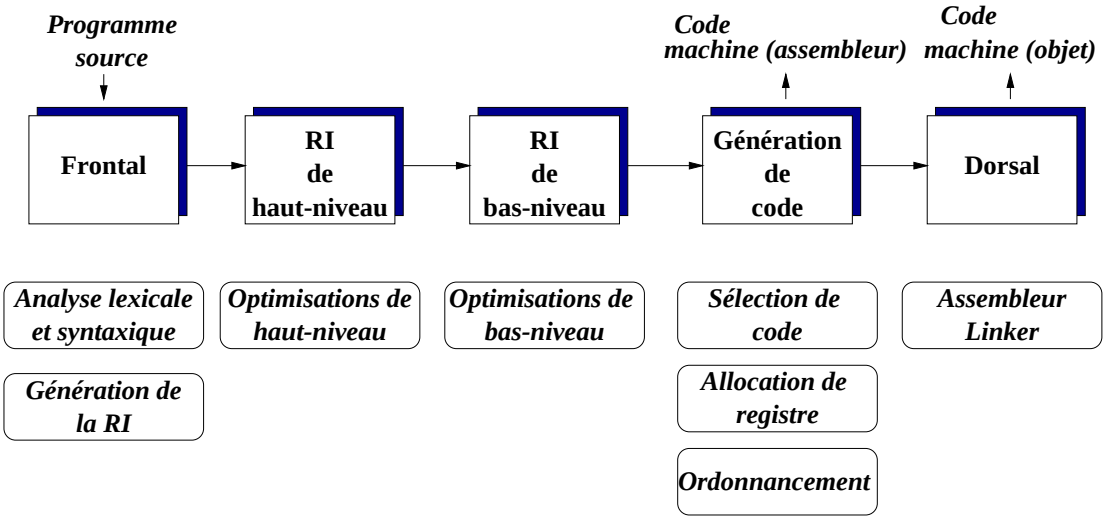


Figure 2.1: Etapes générales de compilation

règles grammaticales propres, là encore, au langage source. Cette étape frontale aide à la vérification de la cohérence (syntaxique et sémantique) du programme source.

La représentation intermédiaire résultant de l'étape frontale peut être de haut niveau, c'est à dire très proche du programme source dans sa structure, préservant notamment les structures de contrôle complexes comme les boucles, et les opérations complexes. Elle peut être aussi de bas niveau, c'est à dire proche d'un niveau de langage compréhensible par le processeur cible : les structures de contrôle se réduisent à des branchement à des labels, les opérations complexes initiales sont éclatées en micro-opérations plus facilement transposables aux opérations disponibles dans la machine cible. Sur chacune de ces représentations intermédiaires, certaines optimisations s'appliquerons avec plus ou moins de succès, tandis que d'autres s'appliqueront sur les deux. Le fait est que chaque représentation a ses avantages. A haut niveau, certaines transformations sont plus aisées, notamment celles s'appliquant sur les boucles et nécessitant un connaissance approfondie de leur propriétés (c'est le cas de la transformation dépeinte dans le détail plus bas). A bas niveau, le rapprochement de la structure avec la machine cible autorise l'application de transformations plus ciblées. Certains compilateurs basculent sur une représentation de bas niveau après application d'optimisations sur la structure de haut niveau, tandis que d'autres se focalisent directement sur une représentation de bas niveau.

L'étape de génération de code, traduit la structure intermédiaire en code. Ici, est distinguée la génération de code assembleur, de la génération de code objet gérée par une partie dorsale. Au cours de l'étape de génération de code, on peut distinguer trois grandes phases que sont la sélection de code, l'allocation de registres et l'ordonnancement. Ces trois phases interagissent fortement entres elles, et leur qualité conditionne très souvent la qualité du compilateur global, si bien qu'elles feront le plus souvent l'objet d'intenses efforts. La sélection de code est l'étape de transposition entre les opérations de la structure intermédiaire, et les opérations de la machine cible. L'allocation (et l'assignation qui

est en général implicite dans ce terme) de registres est une étape essentielle. Elle décide quelles seront les variables du programme placées dans des registres, et dans quels registres les placer. Une technique très répandue est l'allocation de registre par coloriage de graphe [38, 18, 49]. Le plus ardue au cours de cette étape, consiste à gérer efficacement le problème du spilling, c'est à dire de la libération de valeurs stockées dans des registres afin de rendre disponibles ces registres pour des calculs intermédiaires. Le spilling sous-entend une sauvegarde mémoire des valeurs stockées, suivie plus tard d'accés mémoire pour restaurer ces valeurs, ce qui peut s'avérer coûteux. Enfin, l'ordonnancement consiste à ordonner les instructions générées et/ou les opérations effectuées, de façon à optimiser l'exécution du programme. Cette étape permet notamment de remplir plus efficacement les étages de pipeline de façon à éviter les bulles, ou étages vides, dans celui-ci. Cette étape permet par ailleurs d'ordonner les opérations afin de maximiser le taux de parallélisme au niveau instruction (ILP), disponible par exemple dans les processeurs de type VLIW. Compte tenu de leur importance, ces trois étapes, bien que considérées un peu à part généralement, peuvent être vues comme des optimisations de code au même titre ou presque que les transformations dont traite la prochaine section.

2.4.1.1 Optimisations de code

L'optimisation de code consiste à modifier celui-ci, *sans altérer le résultat de son exécution,* afin de le rendre plus efficace. Cette efficacité peut correspondre à plusieurs cibles :

- *taille* : l'objectif est de minimiser la taille mémoire requise par le programme une fois compilé, c'est-à-dire sous sa forme binaire, afin notamment de réduire la surface de la mémoire requise dans le cas de programmes embarqués. On peut par exemple chercher à supprimer le code mort (code jamais exécuté). D'autres procédés sont moins triviaux, comme maximiser l'utilisation du parallélisme disponible dans le jeu d'instructions.

- *vitesse d'exécution* : il s'agit d'accélérer l'exécution du programme sur la machine cible, par exemple en évitant de recalculer plusieurs fois une même expression, en évitant des accès mémoires trop nombreux. Les cibles privilégiée de telles optimisations sont les boucles du programme.

- *puissance ou énergie consommée* : générer un code de telle façon que son exécution conduise à une consommation réduite du processeur hôte, est un sujet de recherche relativement récent. Certaines techniques ordonnent les instructions, en s'appuyant sur le constat que certaines combinaisons d'instructions consomment moins que d'autres. Il est fréquent de constater une corrélation entre les transformations visant à augmenter la vitesse d'exécution, et celle visant à diminuer la consommation, les premières permettant d'exécuter une tâche avec moins d'instructions, donc moins d'efforts.

Certaines optimisations sont dites globales, c'est-à-dire appliquées à une fonction ou procédure dans sa totalité; d'autres sont dites locales, et appliquées à de simples blocs de base[13]. La plupart des optimisations dites locales ont leur contrepartie globale. Certaines optimisations, s'appliquant à un niveau inter-procédural, peuvent même être qualifiées de super-globales. La plupart des optimisations de code classiques, s'appliquent avec succés dans le cadre de la compilation de programmes embarqués. Au sein de la légion de transformations existantes, quelques unes s'affirment comme incontournables [1, 6, 31, 44, 63]: en voici quelques unes.

Elimination de code Relative à la suppression de parties d'un programme identifiées comme inutiles, cette transformation se décline en élimination de variables mortes[14] (par exemple quand une variable définie n'est jamais utilisée), élimination de code inutile (qui consiste à éliminer toute opération dont le résultat n'est jamais employé), et élimination de code inaccessible ou mort (comme la suppression de boucles ne réalisant aucunes itérations, ou de branches conditionnelles dont la condition initiale est identifiée comme toujours fausse).

Elimination (ou propagation) de sous-expressions communes Lorsque deux (ou plus) expressions partagent une sous-expression identique, il est possible de ne calculer cette sous-expression qu'une fois, de mémoriser le résultat, et de remplacer toutes les occurences du calcul par le résultat calculé.

Propagation de constante - calcul de constante Le calcul de constante[15] s'applique, lorsqu'une expression contient une opération dont les opérandes sont des constantes. Il s'agit donc d'un pré-calcul au moment de la compilation, évitant de perdre du temps au moment de l'exécution. La propagation de constante consiste à remplacer une variable à laquelle est assignée une constante par sa valeur, à chacune de ses occurrences.

Réduction de force Appliquée dans le cas général, cette transformation[16] se propose de remplacer des opérations coûteuses pour la machine cible, en opération moins coûteuses. Typiquement, le produit $2 \times x$ sera remplacé par $x \ll 1$, de même $x \times 2^c$ par $x \ll c$ etc ...

Expansion en ligne des appels de procédures Etant donné le coût d'un appel de fonction, lié notamment à la sauvegarde du contexte, il est parfois préférable de remplacer un appel de procédure par une copie de son corps. Cela suppose la mise à jour des variables

[13] Un bloc de base est un ensemble d'opérations s'exécutant séquentiellement, sans aucune coupure due à des expressions de contrôle comme saut, branchement conditionnelle et appel de procédure. Un exemple d'une telle représentation pourra être trouvé annexe A.2 page 181.

[14] dead variable elimination, useless code elimination, unreachable (dead) code elimination

[15] constant propagation et constant folding ou constant computation

[16] strength reduction

intervenant dans le corps de la procédure, et un éventuel renommage des variables créant des conflits.

Optimisations s'appliquant sur les boucles Ces optimisations possèdent une aura particulière dans le sens où, les boucles étant par définition des séquences de code exécutées de façon répétitive, leur poids dans le temps d'exécution global du programme est important. Par conséquent, l'impact d'une optimisation de boucle est d'autant plus grand.

Déplacement de code invariant On distingue:

- Déplacement d'expressions invariantes[17]: Toute expression dans la boucle, dont le calcul est indépendant des variables modifiées au cours des itérations de celle-ci, peut être exécutée avant la boucle. C'est l'objectif de cette optimisation[18], qui consiste donc à déplacer les portions de code répondant à la propriété d'invariance dans ce qui est nommé le prologue de la boucle. Cette transformation évite le re-calcul à chaque itération des portions d'expression, voire des expressions invariantes.

- Déplacement de code conditionnel invariant[19]: Lorsque une expression conditionnelle est indépendante des variables de boucle, il est alors possible de déplacer la condition correspondante dans le prologue. Mais cela suppose la création d'autant de boucles que de branches issues de cette condition. La figure 2.2 représente de façon plus explicite ce phénomène.

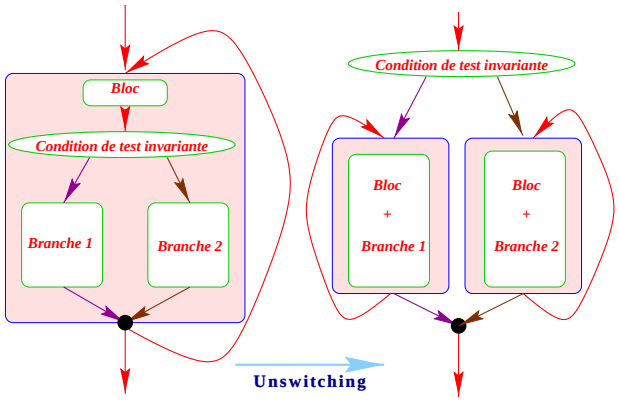


Figure 2.2: Déplacement de code conditionnel dans le prologue d'une boucle et conséquences

[17] loop invariant expression motion
[18] Loop invariant code motion
[19] loop unswitching

Elimination de variables d'induction Dans certains cas, une variable d'induction, c'est à dire une variable de boucle évoluant d'un pas constant tout au long des itérations, peut être éliminée. C'est le cas par exemple lorsque deux variables d'induction évoluent en parallèle, et que l'une des deux n'est employée que pour le test de sortie de boucle. Alors, l'une peut remplacer l'autre moyennant quelques ajustements. La transformation de tableaux en pointeurs, dans le cas où les variables d'induction impliquées ne servent qu'à induire la séquence d'adresse, peut conduire aussi à l'élimination de ces variables.

Réduction de force appliquée aux variables d'induction Cette transformation est étudiée plus en détail section 3.3.3.1 page 55. Elle consiste essentiellement à remplacer les multiplications mettant en jeu des variables d'induction par des additions, moins coûteuses.

Transformation de références tableaux en références pointeurs Cette transformation fait l'objet du chapitre 3.3.4. Elle est souvent abordée comme une extension de la réduction de force appliquée aux variables d'inductions, mais mérite que l'on s'y attarde un tantinet.

Migration de variables globales Cette optimisation se propose de stocker des variables globales très utilisées dans une boucle, dans des registres, cela avant exécution de la boucle, et pour la durée de celle-ci. L'avantage est de remplacer ainsi tous les chargements et écritures mémoire, en accès à des registres, ce qui s'avère beaucoup plus efficace.

Fusion de boucles La fusion de boucles[20] remplace deux boucles par une seule. Pour cela, les calculs dans une boucle ne peuvent dépendre des calculs dans la précédente, et les limites d'exécution doivent être les mêmes.

Déroulage de boucle C'est une transformation majeure consistant à dupliquer un certain nombre de fois le corps d'une boucle. Le gain résulte alors d'un nombre réduit d'opérations liées à l'exécution de la boucle, ainsi que de possibilités de parallélisation accrues.

Pipeline logiciel C'est aussi une transformation majeure, dont le nom provient de sa similitude avec la technique de pipeline matériel [67, 112, 113]. Son objectif est de paralléliser les opérations d'une boucle en éliminant les dépendances pouvant exister entre elles dans le corps de boucle initial. Les dépendances entre opérations successives du corps de boucle au cours d'une itération, sont éliminées en exécutant ces opérations simultanément mais *pour des itérations différentes* de la boucle. Il s'agit donc d'un ré-ordonnancement effectué sur plusieurs itérations d'une boucle, équivalent à dérouler la boucle plusieurs fois, réordonner pour un maximum de parallélisme, et reformer la boucle, avec ajout de nécessaires épilogues et prologues afin d'assurer le continuum sémantique.

[20] loop fusion ou loop jamming

2.4.1.2 Notions complémentaires

Un problème souvent évoqué dans le domaine de la compilation est le problème du couplage de phases[21]. Ce problème est relatif aux interactions entre elles des transformations optimisantes, ainsi qu'à l'ordre de leur application. Bien que plus communément évoqué dans le cas des interactions entre les étapes d'ordonnancement et d'allocation de ressources, ce problème se pose pour tout type de transformation.

Certaines transformations améliorent à la fois la taille du code généré et ses performances: c'est le cas de l'élimination de code qu'il est intéressant d'appliquer régulièrement, par exemple après propagation de constantes, qui peut mettre en évidence des conditions toujours vraies (ou fausses). L'élimination de sous-expressions communes est destinée à augmenter les performances temporelles, mais peut avoir l'effet inverse, dans la mesure où elle nécessite la création de temporaires pouvant avoir des durées de vie importantes, augmentant par la même la pression sur les registres disponibles. Pour peu que ce nombre soit restreint, cela peut causer des accès mémoire supplémentaires et coûteux afin de libérer des registres pour les calculs[22]. La migration de variables globales peut générer le même type d'inconvénients. D'autres optimisations, destinées elles aussi à augmenter la vitesse d'exécution du programme, peuvent avoir un impact parfois difficilement acceptable sur la taille du code. On retrouve dans celles-ci l'expansion de code en ligne, le déroulage de boucle et le déplacement de conditions invariantes dans les boucles.

L'ordre d'application des transformations tient un rôle conséquent sur l'efficacité globale. L'élimination de code doit être appliquée régulièrement, à cause d'opportunités offertes par de nombreuses transformations. La propagation de constante et son homologue, le calcul de constante, doivent être appliqués de concert et de façon itérative, l'une offrant des opportunités à l'autre et inversement. Il est bon d'autre part de propager les constantes avant les transformations s'appliquant sur les boucles, pouvant ainsi par exemple faciliter le déroulage en explicitant les limites de la boucle. Dans [89] est donnée une description de ce que devrait être la séquence de transformations d'un compilateur optimisant classique et efficace.

2.4.2 Compilation reciblable

La mise au point d'un compilateur est une étape compliquée, nécessitant ressources et temps. La compilation reciblable ambitionne de réduire et la complexité, et le temps de conception, en "automatisant" cette dernière. Deux approches se distinguent [67] : la première emploie un générateur de compilateurs qui, à partir d'une description textuelle de la machine cible, produit un compilateur sous forme d'un nouveau programme. La seconde, considère un environnement de compilation fixé, qui génère du code pour un processeur donné en se basant sur une description textuelle de celui-ci; c'est l'approche de re-ciblage automatique. Flexcc [72, 92, 69], une chaîne de compilation reciblable, basée sur la technique de re-ciblage à base de règles, est représentée sur la figure 2.3.

[21] phase coupling and ordering problem
[22] problème du "spilling"

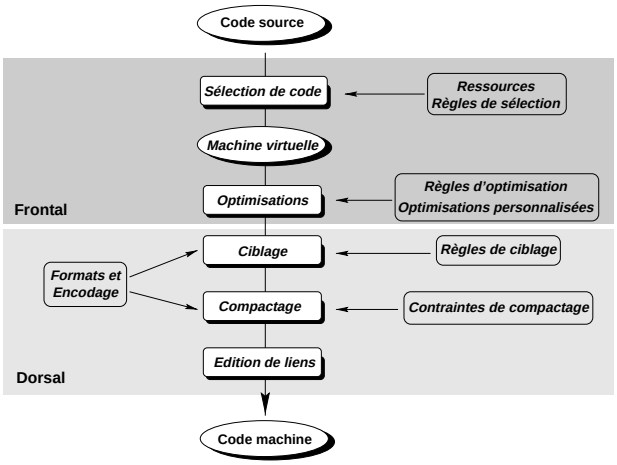


Figure 2.3: Enchaînement des phases de compilation dans la chaîne reciblable Flexcc.

Plus précisément, la compilation avec Flexcc est divisée en quatre phases principales :

1. La sélection de code virtuel est faite pour une machine virtuelle, entièrement spécifiée par le développeur. Elle peut même se confondre avec la machine cible. Le code généré est strictement séquentiel, repoussant l'étape de compactage (exploitation du parallélisme) plus tard dans le flot. Elle se base sur deux types d'information : la description des ressources (principalement les bancs de registres, les formats d'opérandes, les modes d'adressages) et un ensemble de règles de sélection.

2. L'optimisation, autrement appelée *peephole*, consiste à remplacer une suite d'instructions par une autre, selon des règles de substitution définies par le développeur. Les règles acceptent une syntaxe simple à base d'expressions régulières.

3. La traduction vers l'assembleur cible consiste à remplacer les instructions virtuelles en micro-opérations réelles, selon des règles d'équivalence données par le développeur. Nous obtenons alors une suite séquentielle de micro-opérations qui peuvent être compactées, si l'architecture le permet.

4. Le compactage de code consiste à produire un code binaire selon les instructions effectivement implémentées dans l'architecture, en respectant les contraintes d'encodage et de parallélisation. Là encore, un ensemble de règles dictées par le développeur guide le compactage.

L'intérêt majeur de cette approche basée sur un ensemble de règles, est d'être extrêmement flexible. La flexibilité des langages de sélection de code et de compaction, permet d'envisager l'exploration de plusieurs solutions architecturales en un temps réduit.

La compilation reciblable est particulièrement intéressante dans le cadre des processeurs embarqués dédiés. De tel processeurs sont souvent initialement conçus pour un ensemble très restreint d'applications. Mais il n'est pas moins fréquent que leur durée de vie s'allonge,

au prix d'extensions et de reciblages vers d'autres types d'applications. Dans ce contexte, l'emploi de la compilation reciblable est extrèmement bénéfique, puisque le compilateur, fonctionnant à partir d'une description du processeur, peut facilement évoluer en parallèle avec le processeur.

2.4.3 Traits singuliers de la compilation de programmes embarqués

Le monde embarqué présente un ensemble de simplifications et de défis pour la compilation plus classique. Simplifications car les applications embarquées sont de nos jours, dans leur grande majorité, des programmes beaucoup plus facilement caractérisables comparés, par exemple, à des programmes destinés à être exécutés sur station de travail. Cela permet à un compilateur, de pouvoir appréhender la totalité d'une application embarquée en une seule fois, ce qui ouvre la porte à des optimisations globales agressives.

Dans l'ensemble des défis, outre les problèmes posés par les ressources parfois limitées des processeurs embarqués, le compromis taille-vitesse d'exécution prend une dimension toute particulière. En effet, la taille du code destiné à être embarqué coûte de plus en plus cher. Par rapport à la compilation classique qui concentre ses efforts sur les performances, la compilation pour processeur embarqué doit générer le code le plus mince possible, avec des performances temporelles les plus grandes, de façon à répondre aux contraintes temps réel qui souvent vont de pair avec le monde embarqué. C'est ainsi que les transformations telles que expansion en ligne de procédures et déroulage de boucles, qui améliorent les performances aux dépens de la taille de code, doivent être appliquées avec circonspection.

Un compilateur pour programme embarqué doit être capable de compiler rapidement lors de la conception de l'application. Il rejoint en cela la compilation classique. Cependant, un programme embarqué étant destiné à resider *ad vitam eternam* sur une puce, le compilateur doit être capable de générer un programme final très optimisé. Au cours de cette phase, le temps de compilation n'est plus le problème majeur, ce qui autorise l'application d'optimisations efficaces, mais longues, pour peu que les performances finales le justifient.

Un problème majeur est relatif au temps de développement d'un compilateur, surtout dans le cas de la réalisation d'un processeur spécifique. Dans le cas d'un processeur à usage général, dont la conception est très longue mais est compensée par une durée de vie très longue, il est envisageable de fournir en un temps important un compilateur associé très performant, et d'y consacrer les ressources requises. Les impératifs sont différents dans le cas de la mise au point d'un processeur dédié à une application. Compte-tenu des contraintes de mise sur le marché, la mise au point du compilateur associé est très souvent déterminante dans ce cadre, notamment pour valider très vite la conception du processeur. Avoir alors à sa disposition une chaîne de compilation reciblable, permet d'obtenir très rapidement un compilateur, et autorise une conception conjointe *processeur–compilateur* convergeant plus rapidement vers une solution fonctionnelle.

Enfin, alors que pendant longtemps, l'accent était mis sur les performances et la densité de code avec dans l'idée une programmation en assembleur pour les parties critiques, la tendance est aujourd'hui à faire en sorte, lors de la conception d'un processeur et notamment d'un DSP, de faciliter le travail du compilateur. Une branche extrême du domaine

propose même de mettre d'abord au point le compilateur, puis de construire une architecture de façon à ce qu'elle s'adapte à ses caractéristiques [99]. Cela confirme l'accroissement de l'importance de la position que prend aujourd'hui le compilateur et ses performances, face à la conception du processeur même.

2.4.4 Auguste trinité Architecture-Compilateur-Application

Il existe une interdépendance triangulaire forte entre architecture cible, compilateur associé et application, interdépendance qui tend à devenir la pierre angulaire des processeurs de la prochaine génération [98]. Cette section tente d'analyser les degrés existant dans cette interdépendance dans le cas d'applications embarquées. Dans le monde embarqué, on peut distinguer trois cas de figures lors de l'intégration d'un cœur de processeur sur une puce.

1. Le choix se porte sur un cœur existant accompagné de son compilateur. Le seul degré de liberté à la disposition des concepteurs, pour optimiser l'intégration du cœur et de l'application sur la puce, est de concevoir ou reconcevoir cette dernière. L'écriture de l'application dépend des capacités du compilateur et des possibilités d'écriture qu'il supporte : il est ainsi fréquent de pouvoir employer des raccourcis d'écriture orientant le compilateur et lui facilitant la tâche. L'application est écrite d'autre part en fonction des contraintes architecturales imposées par la machine cible, et modifiée ultérieurement en fonction des performances obtenues, dans un processus itératif. On peut distinguer deux grandes boucles d'itération : la boucle application → compilateur → processeur → application, qui, par les retours de performance et de taille, influence les modifications à apporter à l'application, et la boucle compilateur → architecture → compilateur, qui peut jouer sur l'application itérative des différentes transformations disponibles au sein du compilateur, de façon à trouver la meilleure combinaison possible. Ce cas de figure est très fréquent actuellement. Il suppose un environnement logiciel accompagnant le processeur, comprenant entre autre un simulateur de jeu d'instructions ou ISS[23]précis au niveau cycle, afin d'obtenir des performances temporelles réelles.

2. Le cœur de processeur est fourni et fixé : c'est par exemple un cœur de processeur spécifique, conçu en fonction des besoins d'une application. Il faut donc développer le compilateur conjointement à l'écriture ou à la modification de l'application, ou des applications, devant être exécutées sur le processeur. Il y a là de nombreux bénéfices à l'utilisation d'un environnement de compilation reciblable, qui permet d'obtenir rapidement un premier jet de compilateur. Le compilateur est alors réglé finement de façon à répondre le mieux possible aux particularités de l'application. En retour, celle-ci est réécrite de façon à faciliter la compilation et l'exécution sur le processeur cible. Là aussi, un ISS est utile.

3. Conception conjointe compilateur - processeur à la lumière des besoins requis par l'application. Considérant une application et ses besoins spécifiques, l'opportunité est ici maximale de fournir un couple application - processeur des plus efficace.

D'autre part, la conception conjointe du compilateur et du cœur de processeur est tenue comme un point clé du succès des prochaines architectures [99]. Cela suppose une étroite relation entre les équipes de conception logicielle et matérielle, voire l'émergence d'un nouveau corps de métier, maîtrisant les caractéristiques matérielles déterminantes pour obtenir une architecture à la fois efficace et amicale pour la compilation, ainsi que les aspects logiciels propres à exploiter l'architecture du mieux possible. La figure 2.4 donne un aperçu des interactions dans ce cas entre les trois entités. En traits pleins, sont indiquées les influences sur la conception d'une entité (processeur, compilateur et application) sur une autre. En traits pointillés, est suggéré le flot d'utilisation des trois entités. Le compilateur et l'application influencent la conception du processeur, qui est un ASIP, de façon à ce que celui-ci soit une cible plus simple pour la compilation, et possède des propriétés optimisant l'exécution de l'application. Des interactions fortes interviennent à la fois au cours de la conception de chacune des trois entités, mais aussi au cours de l'utilisation de celle-ci.
Un flot typique mettant en jeu ces deux aspects peut être le suivant : un premier jet d'architecture est conçu à la lumière des besoins de l'application [1 et 2], permettant le développement rapide d'un premier compilateur, qui se base par ailleurs sur le type d'application [3, 4 et 5]. L'application peut être alors modifiée [6 et 6'] de façon à être compilée et exécutée plus efficacement [7 et 8]. Bien que le compilateur puisse dans un premier temps fournir une estimation grossière en terme de performance temporelle, le développement conjoint au processeur d'un ISS rajoute à la complexité de la tâche, mais s'avère très souvent nécessaire, surtout pour les applications embarquées associées à des contraintes temps réel. Ce premier jet de résultat conduit à une seconde version du processeur, orientée par les besoins du compilateur [9] et par les performances obtenues. Et ainsi de suite.

Le troisième point est d'ores et déjà une solution viable et pratiquée, et qui ne peut que prendre de l'ampleur, si l'on considère comme un signe l'émergence de sociétés proposant un environnement de développement propre à fournir en un temps très court, non seulement le compilateur, mais aussi le simulateur de jeu d'instructions, le débogueur et ... le processeur !

Target[24] propose Chess/Checkers, un environnement de re-ciblage pour ASDSP. Basé sur un modèle unique de description de processeur, utilisant le langage nML, cet environnement génère un compilateur (Chess), et un simulateur de jeu d'instructions (Checkers). Une sortie en langage de description de matériel ou HDL (VHDL ou Verilog) est prévue, toujours à partir du seul nML, description qui peut alimenter un outil de synthèse logique. Une des forces de cet environnement, réside dans une description unique, strictement architecturale, du processeur cible, permettant à un non-expert en compilation, de générer les deux piliers du l'environnement logiciel du monde embarqué que sont compilateur et simulateur.

[23]Instruction Set Simulator ou ISS
[24]Target Compiler Technologies NV, Leuven, Belgique

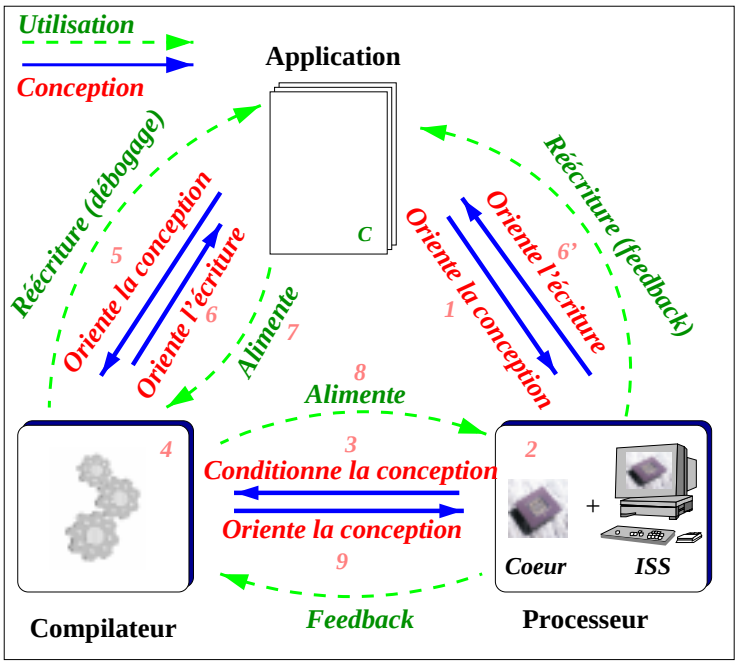


Figure 2.4: Interaction entre processeur, compilateur et application en terme de conception et de flot d'utilisation.

Tensilica[25] propose une démarche différente. L'environnement fourni se base sur un cœur de processeur, Xtensa, conçu pour être petit, efficace, et contenant un ensemble d'instructions élémentaires et de caractéristiques génériques. L'utilisateur a la possibilité de modifier ce cœur de processeur en lui ajoutant de la fonctionnalité par l'intermédiaire de nouvelles instructions, de façon à générer un processeur spécifique à une application. La série d'outils de développement logiciel générée, en même temps que le processeur, se base sur la suite de développement GNU[26]. Un simple fichier de configuration suffit pour décrire l'ensemble des nouvelles fonctionnalités voulues par l'utilisateur, et pour générer le nouveau cœur et sa série d'outils, comprenant compilateur, assembleur, linker, débogueur, simulateur, profileur. Le processeur est généré non seulement sous une forme synthétisable, mais accompagné des scripts de synthèse requis pour obtenir un layout efficace. Le cœur de processeur est orienté plutôt contrôle, dans la même lignée que la famille ARM, mais contient la possibilité de rajouter des caractéristiques purement DSP.

2.5 En résumé

Ce chapitre s'est intéressé aux trois aspects du logiciel embarqué que sont le processeur, le programme et le compilateur. Un très grand nombre de processeurs destinés à être embarqués sont disponibles sur le marché. Une branche particulière est celle des processeurs

[25] Tensilica, Inc.
[26] Gnu is Not Unix

spécifiques, très intéressants de part leur taille et leur efficacité, mais souvent un défi pour la conception des compilateurs associés. Une solution viable, semble être de concevoir en parallèle chacune des trois entités, afin de converger rapidement vers une solution pertinente. Dans ce cadre, l'expérience montre que la conception du compilateur conditionne à la fois la conception du processeur, mais aussi sa facilité d'intégration dans un système, ainsi que son emploi. Cela justifie l'utilisation de chaines de compilation reciblables, flexibles, et la mise au point de techniques d'optimisations de code pointues, se substituant aux optimisations manuelles encores pratiquées. Les études et développements menés au cours de cette thèse, l'ont été dans le cadre du monde logiciel embarqué, qui a conditionné les orientations adoptées. Ces travaux sont exposés au cours des prochains chapitres.

Chapitre 3

Optimisations de programmes embarqués

3.1 Préambule

Ce chapitre s'intéresse aux optimisations de code permettant d'automatiser les étapes d'écriture, ou de réécriture manuelle de programmes destinés à être embarqués, afin d'améliorer leur performances. Il expose en particulier une optimisation qui a absorbé l'essentiel des travaux réalisés, et qui permet une meilleure exploitation des ressources d'adressage de la machine cible. Elle consiste en une transformation de références tableaux en pointeurs, dans des boucles d'un type fréquemment rencontré dans les applications de traitement du signal. L'algorithme de transformation est exposé, à la suite d'un état de l'art des travaux existant dans la littérature sur ce point précis. Une description des ressources d'adressage de la machine cible est employée, afin d'orienter la transformation. La façon de décrire ces ressources, ainsi que l'exploitation de cette information architecturale est détaillée. Enfin, un certain nombre d'optimisations, en relation directe ou indirecte avec la transformation principale, sont décrites.

3.2 Approche de transformation

L'approche de transformation choisie au cours de ces travaux est dépeinte sur la figure 3.1. Cette approche considère une représentation intermédiaire proche du code source, en s'appuyant sur des informations architecturales, et privilégie les optimisations s'appliquant sur les boucles. Le code résultant est regénéré dans le langage initial, C.

3.2.1 Transformer au niveau source à la lumière des spécificités de la machine cible

Un code décrit sous une forme intermédiaire de bas niveau [10], présente l'avantage d'être proche de la machine cible et de pouvoir agir plus efficacement pour celle-ci, mais présente

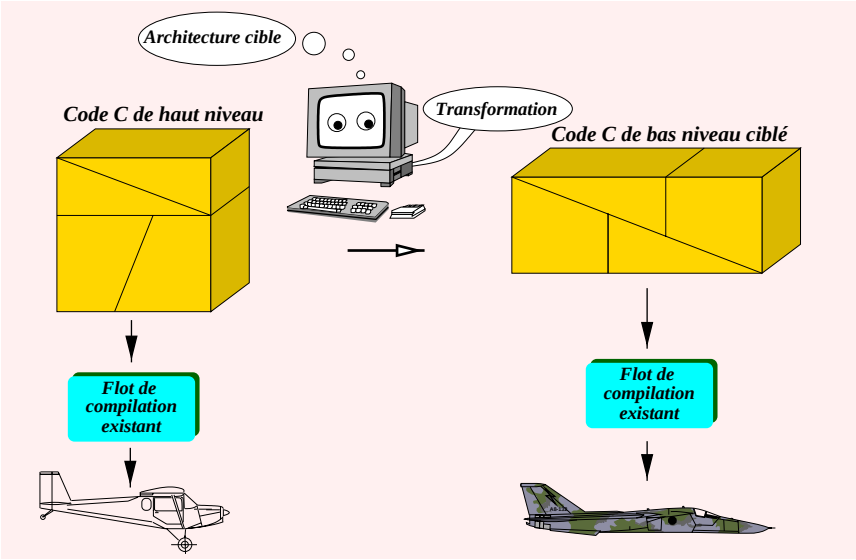


Figure 3.1: Approche transformationnelle adoptée au cours de ces travaux

l'inconvénient majeur de la décomposition des déclarations initiales en micro-opérations. Les boucles sont éclatées en branchements élémentaires, les expressions complexes sont décomposées. Dans le cas de certaines transformations, il y a grand intérêt à préserver ces informations de haut niveau. C'est pourquoi ces travaux choisissent de mettre en œuvre des transformations à partir d'une représentation de haut niveau, préservant toutes les informations du code original.

Afin de faire face à l'éloignement de la machine cible qui découle de ce choix, l'alternative consiste à fournir toutes les informations sur la machine cible par le biais d'une description des caractéristiques utiles de celles-ci pour la transformation. Un autre profit concernant ces informations supplémentaires sur la machine, est relatif à la possibilité de re-ciblage qu'elle implique, donc d'exploration d'architecture, puisqu'en changeant la description, la transformation produit un résultat à priori différent. A vrai dire, les approches bas niveau et haut niveau, loin d'être antagonistes, sont plutôt complémentaires. Il y a des transformations qui sont plus puissantes effectuées à haut niveau, comme le prouvent les travaux antérieurs et ceux exposés plus bas. Mais il reste nécessaire d'exécuter d'autres transformations à bas niveau, ne serait-ce parce que la vision de la machine est plus détaillée, ou que le passage du niveau programme vers le niveau machine crée des opérations invisibles au premier.

3.2.2 Les boucles comme cibles privilégiées

Les parties du code s'exécutant de façon répétitive, autrement dit les boucles, représentent une cible de choix pour la plupart des compilateurs optimisants. Ce sont en effet des parties du code d'un programme qui représentent une fraction de sa taille, mais dont le poids à l'exécution est grand. Elles justifient la fameuse règle affirmant que beaucoup

d'applications passent 80% de leur temps à exécuter 10% du code programme. La loi d'Amdahl [47] confirme alors, entre autres, l'intérêt de consacrer des efforts à accélérer les petites portions de code que sont les boucles, l'accélération résultante totale pour le programme étant importante.

Parmi la masse des travaux consacrés au traitement des boucles, peu d'entres eux développent la transformation de tableaux en pointeurs afin d'exploiter plus efficacement les ressources d'adressage disponibles. Celle-ci peut à vrai dire s'extrapoler d'une optimisation relativement courante qu'est la réduction de force. Les travaux engagés au cours de cette thèse, de même que les travaux précédents, tendent à prouver qu'il s'agit pourtant là d'une transformation majeure, qui mérite un peu plus d'efforts qu'une simple réduction de force.

3.2.3 Transformer de source à source

Enfin, dans un souci principal d'expérimentation, le C a été choisi comme langage cible à l'issue des transformations effectuées sur le code source. Le C permet d'exprimer des concepts de haut niveau, mais autorise une écriture de très bas niveau. Régénérer un programme dans le langage source C présente beaucoup d'avantages, d'un point de vue prototypage et expérimentations :

- *Vérification aisée* : Bien que moins lisible que le code initial, la vérification de la cohérence de la transformation reste bien plus facile avec un code C qu'avec un code assembleur. Par ailleurs, il est aisé de vérifier la cohérence lors de l'exécution du programme, avant et après transformation, en compilant les deux instances du même code sur une machine hôte, et en comparant rapidement les traces d'exécutions.

- *Re-ciblage facilité* : Un code C adapté à une machine cible est bien plus simple à générer que du code assembleur pour cette machine. Cela aurait été possible en intégrant les transformations comme module de la partie frontale d'un compilateur, ce qui n'a pas été envisagé dans le contexte de ces travaux. L'approche de transformation peut cibler toute machine, le ciblage d'une machine se faisant via le fichier de spécification, donnant les propriétés du processeur cible utiles à la transformation. De plus, la possibilité d'introduire des directives de bas niveau, et de permettre notamment d'assigner manuellement des variables à des registres, propriété intéressante des compilateurs appréhendés au cours de ces expérimentations, à été exploitée afin de cibler plus finement le code généré.

3.3 Transformation tableaux - pointeurs dans les boucles

3.3.1 Principe du problème

L'emploi de tableaux est une caractéristique de haut niveau d'un langage de programmation. Dans les algorithmes DSPs, il est plus que courant de trouver des éléments accédés

de façon itérative, et arrangés sous forme de vecteurs ou de matrices. Traduits en un langage de programmation de haut-niveau, des tableaux n-dimensionnels se substituent aux vecteurs et matrices, tandis que le parcours itératif se traduit par l'emploi de boucles. Une illustration simple de ce type d'écriture est celui de la figure 3.2.

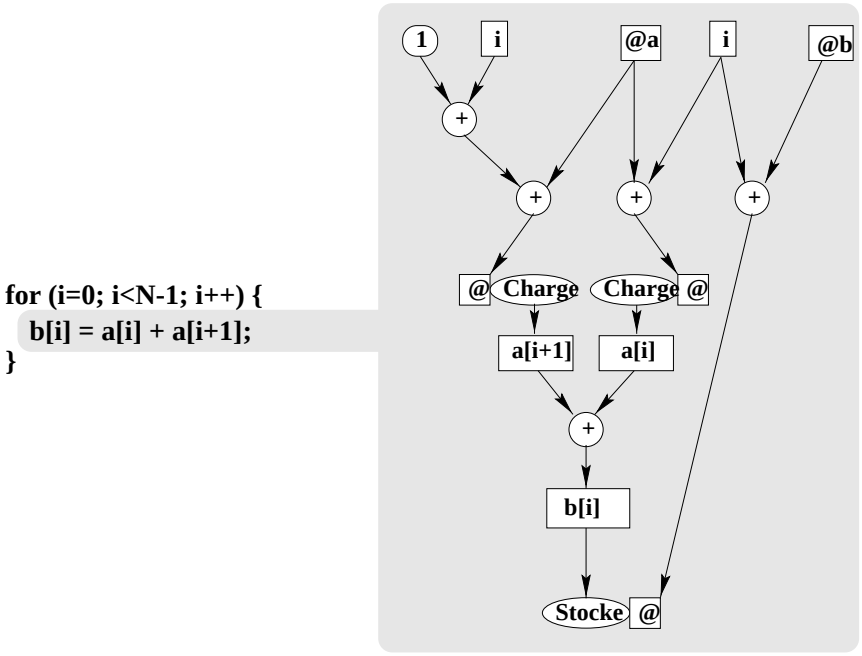


Figure 3.2: Boucle simple avec tableau - Graphe de flot de données du corps de boucle

D'un point de vue performance, l'écriture sous la forme de tableaux produit souvent de pauvres résultats. La raison en est le calcul préalable de l'indice, puis de l'adresse, avant de pouvoir accéder à l'élément mémoire référencé par ce dernier. Si l'on fait le compte de l'ensemble des calculs requis pour l'expression corps de boucle de la figure 3.2, en imaginant une machine virtuelle fournissant les opérations élémentaires requises, on obtient grossièrement une addition (i+1), trois calculs d'adresse, deux chargements, une addition et pour finir un stockage de la nouvelle valeur. Huit opérations élémentaires doivent donc être exécutées, opérations présentant entre-elles de fortes dépendances, ce qui limite l'exploitation du parallélisme inhérent à la machine. A ceci, peuvent venir s'ajouter les multiplications des indices des tableaux par la taille de ses éléments (par exemple 4 pour des entiers stockés sur 4 octets), opérations non représentées sur cette figure.

L'idée de remplacer les références tableaux par des pointeurs rejoint l'un des principes d'élimination des variables d'induction, sur lequel nous reviendrons en détail plus loin : remplacer l'utilisation d'une variable pour induire une séquence de valeurs, par le calcul direct de la séquence de valeurs. Au lieu d'employer un tableau, et le calcul d'indice et d'adresse coûteux correspondant, un pointeur sur ce tableau va successivement pointer sur les valeurs requises. Le code de la figure 3.3 illustre l'application de ce procédé sur l'exemple précédent. Outre la réduction du nombre d'opérations, qui ne se monte plus qu'à 7 comparé

```
ptra1 = a;
ptra2 = &a[1];
ptrb = b;
for (i=0; i<N-1; i++) {
  *ptrb = *ptra1 + *ptra2;
  ptra1++;
  ptra2++;
  ptrb++;
}
```

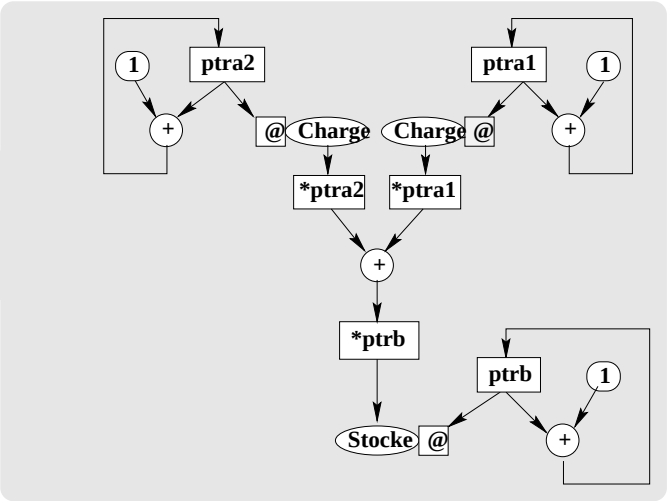


Figure 3.3: Boucle simple avec pointeurs - Graphe de flot de données du corps de boucle

à la boucle précédente, on peut noter la présence d'opérations d'auto-incrémentation. Il est fréquent dans nombre d'architectures modernes, ainsi que souligné au cours du chapitre précédent, que cette incrémentation puisse s'effectuer en parallèle avec un chargement ou un stockage mémoire. En exploitant un mode d'adressage auto-incrémenté, le calcul se réduit à 4 opérations : cette nouvelle écriture génère donc un code bien plus efficace que le précédent. D'autre part, dernier point important, la variable i n'est plus employée que pour les opérations de boucle (test et compteur). Si un mécanisme de boucle matérielle[1] est disponible sur la machine cible, i n'est plus utile et peut être éliminée totalement.

3.3.2 Quelques définitions

Avant d'aborder le cœur du problème - l'automatisation de la transformation précédemment illustrée - un certain nombre de termes méritent d'être précisés plus avant. La plupart de ces termes, employés dans cette section et plus loin, sont relativement communs [1].

Constantes ou invariant de boucle On appelle ainsi toute variable dans une boucle qui n'est jamais modifiée dans celle-ci. Une expression composée exclusivement d'invariants de boucle est fort logiquement appelée ***expression invariante*** ou ***expression constante***.

Variable d'induction de base Se dit d'une variable de boucle i exclusivement modifiée par des déclarations de la forme $i = i \pm k$, k étant le pas d'incrémentation (ou de décrémentation) de la variable d'induction de base. k est une constante de boucle. Couramment, une et une seule déclaration de ce type se trouve dans une boucle; c'est

[1] Mécanisme de prise en charge directe de l'exécution d'une boucle par le processeur, avec un coût faible par rapport à l'utilisation d'instructions de branchement et de comparaison

le cas pour la boucle précédente. Dans un cas plus général, il faudra tenir compte de plusieurs modifications possibles de i. C'est ainsi que l'on pourra associer à chaque variable d'induction de base un ensemble de pas, chaque pas correspondant à une assignation donnée dans le code. On associera de même à chaque variable d'induction de base, plus commodément nommée BIV[2]par la suite, sa valeur d'initialisation.

Variable d'induction C'est une variable de boucle j qui, à chaque fois qu'elle est modifiée, voit sa valeur augmentée ou diminuée d'une constante, ou plus généralement d'une constante de boucle. Cette définition inclut celle des variables d'induction de base. Une variable d'induction ou IV[3]simple, est de la forme $j = a \times i + b$, a et b étant des constantes de boucles. A noter que l'on peut avoir $a = \frac{1}{c}$ ou encore $a = -c$ avec c constante de boucle. Si i est une BIV, j sera dite *de la famille de* i. Si i est elle-même une variable d'induction de la famille de x, alors j sera de la famille de x. L'expression à droite de la déclaration ci-dessus est appelée ***expression d'induction*** ou IE[4]. Toute déclaration de type $j = IE$ définira donc une IV j. La définition suivante étend le concept d'IE et donc d'IV tel que défini dans [1].

Expression d'induction C'est une expression fonction affine de variables d'induction, de base ou non, obéissant à la formulation $\sum A_i \times I_i + B$, avec $\forall i$ A_i et B invariants ou expressions invariantes, et I_i variables d'induction. Par exemple, la déclaration $k = i - 2 \times j + 3$, avec i BIV et j IV, déterminera l'IV k.
Dans [1], chaque IV j défini par l'IE $a \times i + b$ se décline sous forme du triplet *(i, a, b)*, avec a noté ***échelle*** et b ***déplacement*** [10], i étant une BIV. Ici, nous utiliserons la définition étendue suivante : à chaque IE est associé un ensemble $P = \{\forall i,\ (A_i, I_i)\ |\ A_i \in Ctes,\ I_i \in IVs\}$, c'est-à-dire un ensemble de paires (A_i, I_i)avec A_i constante de boucle et I_i IV ou BIV; une constante de boucle B sera d'autre part associée à P, soit $IE = [P, B]$. Etant donné la définition des IVs ci-dessus, on aura $j = IE = [P, B]$.

Expression d'induction indexée Il s'agit d'une IE ayant la propriété d'exister en tant qu'indice d'une référence tableau. L'ensemble des expressions d'induction indexées d'une boucle ne comprend donc aucune des IEs définissant une IV. Par la suite, le terme IIE[5]sera plus communément employé.

[2]Basic Induction Variable
[3]Induction Variable
[4]Induction Expression
[5]Indexed Induction Expression

3.3.3 Travaux antérieurs

3.3.3.1 La voie du Dragon et consœurs

Une technique très connue visant à simplifier les calculs d'indices tableaux dans les boucles, est exposée dans [1] sous le nom d'élimination des variables d'induction, ou plus précisément de "réduction de force"[6] appliquée aux variables d'induction. S'appuyant sur une représentation intermédiaire de bas niveau, cette méthodologie se propose d'éliminer les multiplications impliquant des variables d'induction, notamment les multiplications des indices par une constante dépendant de la taille des éléments du tableau, par des additions. Cette méthodologie ne s'intéresse qu'aux expressions simples du type $x = a \times i$ ou $x = a \times i + b$ c'est-à-dire aux variables d'induction simples, telle que l'on peut les trouver dans une représentation de bas niveau.

Cette méthodologie est aisément transposable à une transformation de plus haut niveau telle qu'elle nous intéresse ici, c'est-à-dire transformation de tableaux en pointeurs, où les calculs d'adresses restent moins explicites. Fischer [31] décrit une telle transposition. Pour faire une synthèse de ces deux techniques, nous utiliserons la notion de triplet définissant une variable d'induction telle qu'introduite dans [1] : à chaque IV j est donc associé un triplet *(i, c, d)*, où i est une BIV et c et d deux constantes de boucles telles que j est donné par $j = c \times i + d$. Nous définirons une IE de la même manière. A noter que ces définitions sont restrictives comparées à celles données dans la section 3.3.2.

Dans [31], la réduction de force est étendue à toutes les IEs, et non plus seulement à celles définissant une IV, de telle sorte que chaque IE, définie par son triplet *(i, c, d)*, peut être remplacée par un temporaire *tmp* initialisé à la valeur $i_0 \times c + d$ dans le prologue de la boucle. Immédiatement après chaque affectation de la variable d'induction i sous la forme $i = i \pm k$ comme précisé dans [1], et non pas seulement à la fin de chaque itération de boucle, comme indiqué dans [31] qui se restreint aux BIVs les plus simples, on rajoute l'expression $tmp = tmp \pm c \times k$.

Pour reprendre l'exemple de la figure 3.2, si l'on réécrit le code sous la forme équivalente suivante :

a)

```
for (i = 0; i < N-1; i++)
   *(b+i)= *(a+i)+ *(a+i+1);
```

b)

```
tmp1 = b;
tmp2 = a;
tmp3 = a+1;
for (i = 0;  i < N-1;  i++) {
   *(tmp1)= *(tmp2) + *(tmp3);
   tmp1++;
   tmp2++;
   tmp3++;
}
```

On peut distinguer trois IEs, dont les triplets respectifs sont *(i, 1, b)*, *(i, 1, a)* et *(i, 1, a+1)*. En appliquant les étapes précédemment décrites, on obtient une solution similaire à celle de la figure 3.3, qui est le code b) ci-dessus. Cette méthodologie synthétique est

[6] induction variable strength reduction

essentiellement indépendante de la machine cible. Elle est efficace dans les cas de tableaux indicés à l'aide d'expressions simples.

3.3.3.2 La voie associative

Il peut s'avérer intéressant d'étudier les relations pouvant exister entre IEs, de façon à regrouper les références tableaux présentant une évolution inter-itération parallèle de leurs indices. Si l'on reprend l'exemple de la figure 3.2 et sa transformation représentée sur la figure 3.3, il est manifeste que la distance entre les indices des tableaux *a[i]* et *a[i+1]* vaut toujours 1 au cours des itérations de la boucle. Autrement dit, les pointeurs *ptra1* et *ptra2* présentent une évolution de leur valeur strictement parallèle. On peut alors envisager de n'utiliser qu'un seul et même pointeur au lieu de deux, ce qui conduit aux deux possibles écritures suivantes :

```
ptrb = b;
ptra = a;
for (i = 0;  i < N-1;  i++) {
   *ptrb = *ptra + *(ptra+1);
   ptrb++;
   ptra++;
}
```

ou

```
ptrb = b;
ptra = a;
for (i = 0;  i < N-1;  i++) {
   tmp=*ptra;
   ptra++;
   *ptrb = tmp + *ptra;
   ptrb++;
}
```

La première écriture ne modifie pas la valeur de la variable pointeur au moment de son utilisation, mais lui rajoute la distance requise pour obtenir la bonne adresse; la valeur de la variable pointeur n'est modifiée que pour préparer l'itération suivante. La seconde écriture, par contre, modifie la valeur intrinsèque de la variable pointeur lorsque cela est nécessaire, ce qui justifie la création d'un temporaire. Chaque écriture ne nécessite que deux pointeurs.

On peut encore aller plus loin si l'on considère que le parcours du tableau b est lui aussi parallèle aux deux premiers. Dans ce cas, un seul pointeur est nécessaire mais il faut calculer la distance entre a et b qui sont les adresses de départ des tableaux. Si l'on prend l'adresse de départ du tableau a comme référence, et si l'on met la distance entre a et b dans un temporaire, on obtient :

```
ptra = a;
dist = b-a;
for (i = 0;  i < N-1;  i++) {

*(ptra+dist) = *ptra + *(ptra+1);
   ptrb++;
   ptra++;
}
```

ou

```
ptra = a;
dist1 = b-a;
dist2 = dist1 - 1;
for (i = 0;  i < N-1;  i++) {
   tmp1=*ptra;
   ptra++;
   tmp2=*ptra;
   ptra+=dist2;
   *ptra = tmp1 + tmp2;
   ptra-=dist2;
}
```

A noter qu'en C, soustraire deux adresses de départ de tableau comme dans l'exemple ci-dessus est interdit[7]. Pour être exact, cette écriture est tolérée par la plupart des compilateurs C, mais aux risques et périls du programmeur car le résultat reste indéterminé,

[7] deux pointeurs peuvent être soustraits, s'ils pointent tous deux sur des positions différentes du *même* tableau [55]

dépendant essentiellement de la machine cible. Deux conditions doivent être satisfaites :

- La première, qui est la plus forte, découle du placement en mémoire des tableaux a et b. S'ils sont stockés dans le même espace mémoire, alors les règles d'alignement des données en mémoire garantissent une distance cohérente entre les deux. Par contre, si plusieurs mémoires coexistent, et s'il n'existe aucun moyen de s'assurer du placement en mémoire de a et b, alors l'écriture ci-dessus est impossible.

- La seconde est relative à la distance entre les tableaux a et b. Le compilateur utilisera vraisemblablement un registre d'index, si la machine en dispose, pour stocker la valeur de *dist*. Dans le cas idéal, les deux tableaux occupent en mémoire des positions consécutives : *dist* est alors estimable par avance. Dans le cas général, il est impossible de connaître à l'avance la valeur de *dist*, et cette valeur peut-être telle qu'elle dépasse les capacités des registres d'index. On peut donc être amené à se poser la question de la faisabilité d'une telle écriture (comparaison taille mémoire - capacité des registres d'index par exemple).

Benitez [10] propose une approche de type associative, enrichissant la méthodologie classique de réduction de force et introduisant des considérations machine. Elle s'applique sur une représentation intermédiaire de bas niveau,[8] de façon à pouvoir transposer plus facilement le code aux possibilités de la machine, notamment en terme d'adressage. Les indices tableaux considérés restent très simples, de la forme $c \times i + d$.

La génération d'adresse liée au problème de références tableaux est aussi traitée par Leupers [64], qui s'inspire largement de Araujo [5]. Cette approche décrit un algorithme permettant de transposer les calculs d'adresses liées aux références tableaux, aux possibilités de la machine cible, proposant une maximalisation de l'utilisation des possibilités d'auto-in/décrémentation que l'on trouve dans les DSPs classiques. Pour cela, les références tableaux ayant entre elles une distance inférieure ou égale à 1 sont regroupées.

Ici aussi, de nombreuses restrictions limitent l'accès à des IEs relativement simples (de la forme i+c, i BIV et c constante de boucle) et l'application de l'algorithme à des boucles ayant un nombre d'itérations fixé et une seule BIV. Ce dernier point est lié au langage d'entrée employé, qui est le DFL (Data Flow Language), lui même dérivé du Silage, langage de spécification pour les algorithmes DSPs développé à Berkeley.

Dans [4] sont développées plus précisément les idées présentées dans [5]. Les références tableaux sont allouées à des registres d'adresses virtuels de façon à en minimiser le nombre. Est employé pour cela un graphe des distances entre index dont les nœuds sont les références; un arc existe entre deux références si la distance entre les deux fait partie des possibilités d'in/décrémentation de la machine. Les exemples simples présentés considèrent des indices très simples, comparables à ceux gérés par [64], et ne donnent aucune indication sur la façon de gérer des indices plus complexes. D'autre part, ces exemples considèrent un ensemble de références d'un même tableau et le partage de ces références en terme de registres d'adresse. Le cas de plusieurs références à des tableaux différents n'est pas évoqué.

[8] low-level intermediate language ou LIL

La voie associative dynamique

Une approche originale à la question présente est décrite par C. Liem [71, 70, 72], dont les travaux se sont déroulés au sein de l'équipe ayant supporté les travaux sous-tendant cette thèse. La transformation de références tableaux en pointeurs se fait au niveau source, c'est-à-dire qu'elle génère du C à partir d'un source C initial. La transformation s'appuie par ailleurs sur une description, là encore de haut niveau, des possibilités d'adressage de la machine cible.

Outre cet aspect transformationnel source-source, l'originalité de l'optimiseur tient dans la façon dont le code est analysé pour permettre la transformation. La détermination des BIVs, IVs et IEs d'une boucle nécessite une analyse fine du flot de données à l'intérieur du corps de boucle. La méthode décrite dans [72] s'abstrait de cette difficulté en générant une image dynamique du source C à transformer. Cela est réalisé en simulant le source, qui doit donc être exécutable, et en traçant les indices de tous les tableaux. Cette trace permet de déterminer les accès tableaux dont les indices évoluent de façon linéaire tout au long des itérations de la boucle. Cette analyse est dénommée analyse de stabilité. Les tableaux stables sont directement transformés en pointeurs, avec ajout de l'incrément associé correspondant à la valeur d'incrémentation de l'indice du tableau, entre chaque itération de la boucle.

Vient ensuite une phase de combinaison des pointeurs et d'assignation de ceux-ci aux registres d'adresses, qui se base sur les informations données de façon annexe sur les ressources d'adressage de la machine cible. L'objectif premier est de réduire le nombre de pointeurs utilisés de telle sorte que ce nombre soit inférieur ou égal au nombre de registres d'adresses disponibles. Un ensemble de règles simple de combinaison est appliqué pour cela. Une fois cet objectif atteint, il s'agit d'assigner à chaque pointeur un registre physique de la machine[9], et de transposer les opérations d'adressage aux opérations disponibles, ce qui inclut la prise en compte de l'assignation des valeurs de déplacement à d'éventuels registres d'index ou constantes câblées. Les informations de profilages issues de l'exécution du programme aident à cette tâche. Cette approche a été appliquée aux seuls tableaux à une dimension.

Un des intérêts de la méthodologie dynamique, est qu'elle se joue de la complexité des IIEs : qu'une IIE soit simple, ou expression affine complexe fonction des IVs, la trace donne les informations nécessaires à la transformation c'est-à-dire la valeur initiale et le pas d'incrémentation. C'est un atout majeur, dans la mesure où la détermination des IEs du corps de boucle peut nécessiter un lourd travail d'analyse. Mais cette méthodologie a ses limites. Si l'on imagine par exemple le cas d'une IIE contenant des constantes de boucles définies avant la boucle en fonction de conditions d'entrée, cette méthodologie ne peut alors plus s'appliquer sans danger : la trace n'est plus déterministe puisque l'évolution du tableau aux cours des itérations varie. Son grand intérêt simplificateur est donc tempéré par cette dépendance aux données, propre à toute optimisation basée sur une exécution du programme.

La transformation détaillée au cours des sections suivantes reprend ces travaux en

[9] Cela n'a de sens que si une assignation manuelle est autorisée par le compilateur dorsal.

adoptant une approche strictement statique d'analyse du code original. De cette façon, les incertitudes et contraintes liées à l'exécution du code s'effacent. Ces travaux sont d'autre part étendus, en traitant les tableaux multi-dimensionnels, et en fournissant un moyen d'explorer un plus grand nombre de solutions de transformation, notamment en considérant les possibilités de modes d'adressage pré-calculés.

3.3.4 Approche de transformation

3.3.4.1 Flot de transformation

Le flot de transformation est détaillé sur la figure 3.4.

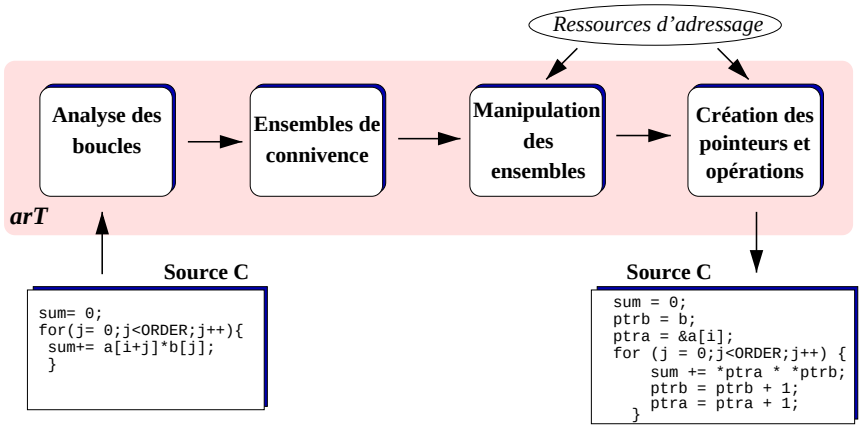


Figure 3.4: Flot de transformation de tableaux en pointeurs dans un programme embarqué C.

L'approche choisie pour la transformation de tableaux en pointeurs, considère une ou plusieurs boucles imbriquées C contenant des références tableaux, et génère un code C de même fonctionnalité, mais contenant des références pointeurs. Dans tous les cas, une seule boucle est considérée à la fois. La méthodologie développée étend la complexité des IEs communément traitées à celle définies dans la section 3.3.2. Elle est statique, par opposition à l'approche dynamique exposée dans la section précédente. Compte-tenu de la propension des boucles de type DSP à contenir des références tableaux, cet algorithme s'applique fructueusement à ce type de boucles. Deux points préalables :

- Les ensembles de connivence sont les briques de base de la transformation. Les déterminer impose l'application d'une série d'analyse de flot de données sur le code C original. Ces ensembles peuvent être ensuite manipulés avant la génération des pointeurs.

- La genèse des pointeurs peut prendre deux directions, correspondant aux deux façons les plus communément admises pour écrire et incrémenter un pointeur. Dans la première, l'adresse permettant d'accéder à l'élément en mémoire est pré-calculée.

Dans la seconde, la valeur du pointeur est modifiée après accés à la bonne adresse. Les deux approches requièrent un traitement différent : l'introduction de post-opérations sur les pointeurs suppose une gestion plus complexe du statut du pointeur, en prenant en considération l'ensemble des in/décrémentations qu'il subit.

Les sections suivantes détaillent dans un premier temps chacune des étapes de la transformation directe, c'est à dire sans exploiter les informations sur la machine cible. L'orientation de la transformation par les ressources d'adressage, qui se manifeste au cours de la manipulation des ensembles de connivence ainsi que lors de la génération des pointeurs, sera traitée à part.

3.3.4.2 Analyse préalable des boucles

En préambule à la transformation elle-même, il est nécessaire d'analyser le code constituant le corps de boucle (cf. annexe A). Cette analyse a pour but principal de déterminer les IIEs, c'est-à-dire les IEs ayant comme propriété d'être des indices de tableaux. L'analyse des boucles et des tableaux (et leurs indices) qu'elles peuvent contenir, est une pratique que l'on retrouve dans les compilateurs parallélisants, dont l'objectif est de produire un code exécutable pour machines parallèles à partir d'un code séquentiel, en paraléllisant notamment l'exécution d'expressions faisant appel à des tableaux [6, 28, 16]. Ici, l'objectif est autre : il s'agit de déterminer les indices de tableaux, fonctions affines simples ou complexes de variables d'induction, dans le but de transformer les tableaux correspondant en références pointeurs.

Tout type de boucle est considéré : certaines approches sont restrictives et se cantonnent à traiter les boucles "bien formées" que sont les boucles *for* ayant un prototype de type fortran[10]. Avant de rentrer dans le détail de chacune des analyses requises, la figure 3.5 donne le graphe de dépendance entre celles-ci afin de parvenir aux informations exploitables par la transformation.

Excepté pour la détermination de l'ensemble des IIEs, établir les ensembles de BIVs, IVs et IEs correspond à la définition de certaines propriétés, qui sont la transposition en terme de fonctions des définitions données dans la section 3.3.2.

Variables d'induction de base

La première étape consiste donc à déterminer les BIVs existant dans le corps de boucle traité (cf. section 3.3.2 page 53). Il peut y en avoir plus d'une, chaque BIV pouvant être multi-incrémentée, c'est-à-dire pouvant voir son contenu affecté à plusieurs reprises dans le corps de boucle. A chaque BIV, seront donc associées les propriétés ou valeurs illustrées sur la figure 3.6.a.

La valeur initiale, si l'on se place dans le cas général d'une boucle *while*, sera assignée avant le corps de boucle. Il est possible que cette valeur soit indéterminée, c'est-à-dire

[10] `for (i = val_init;  i<val_max;  i+=inc)`

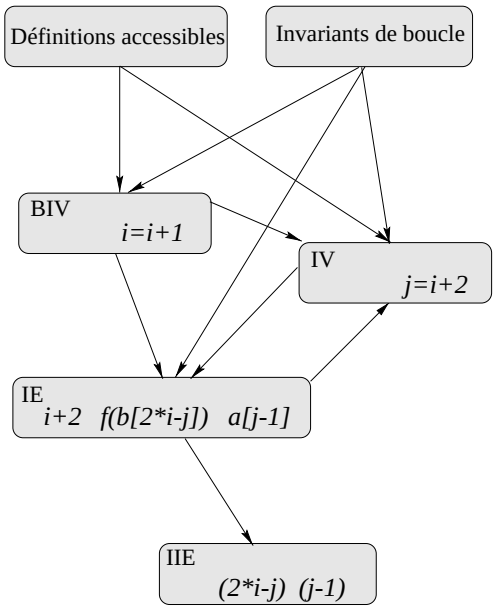


Figure 3.5: Dépendances entre analyses

définie plusieurs fois dans des chemins d'exécution différents avant le corps de boucle. Cela nécessite donc l'exécution d'une première passe d'analyse des définitions accessibles.

Si la BIV est multi-incrémentée, chaque doublet (pas, position) est stocké dans une liste. Le pas d'incrémentation n'est pas forcément une constante, ce qui justifie la nécessité de déterminer au préalable les constantes de boucle.

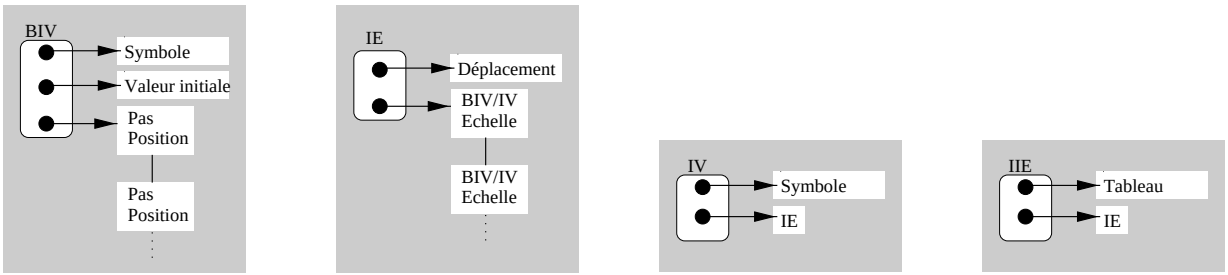


Figure 3.6: Cibles de l'analyse

Une difficulté dans ces propriétés est la résolution du prédicat : "l'affectation est de la forme : $i = i \pm \textit{constante de boucle}$". Pour pouvoir répondre à cette question, il faut parcourir les expressions du corps de boucles à la recherche de motifs donnés. Ce problème, encore relativement simple pour le calcul des variables d'induction de base, devient crucial et représente en réalité toute la difficulté du calcul des autres propriétés, dont la description suit.

Expressions d'induction et variables d'induction

La détermination des IEs et celle des IVs interagissent, une IV étant une IE associée à un symbole comme illustré sur la figure 3.6. En théorie, l'ensemble des IVs comprend logiquement l'ensemble des BIVs. Pour des raisons de clarté et de commodité, nous considérerons ici deux ensembles distincts. La grande difficulté de l'analyse dans le cas présent réside dans le postulat, pour une IE, "est une fonction affine des BIVs/IVs". Cela requiert un parcours systématique des expressions afin de rechercher des motifs du type $C \times Var$, séparés par des opérations + ou - avec C constante de boucle, et Var BIV ou IV. C'est ici que le choix de conserver une représentation intermédiaire proche du code source prend tout son sens, car cette recherche, bien que complexe, en est grandement facilitée. Par exemple, le moteur d'analyse est capable de retourner que l'expression $i - 2 \times j + 3$ est une IE ce qui suppose que i et j sont, soit des BIVS, soit des IVs.

Les IEs, au même titre que les BIVs, possèdent une valeur initiale et un pas d'incrémentation. Mais ces valeurs doivent être issues d'un calcul, et non de l'analyse du code. Le pas d'une IE représente la valeur totale dont se verra incrémentée l'IE à chaque exécution de la boucle. On peut représenter une IE comme un arbre de façon à mieux appréhender ce calcul, ainsi qu'illustré sur la figure 3.7. Schématiquement, les feuilles de l'arbre sont les BIVs, les nœuds étant des IVs. Un calcul, de pas ou de valeur initiale d'une IE, est un parcours en profondeur d'abord de cet arbre. Dans l'exemple de la figure 3.7, le pas de IE1 est la somme des pas partiels de l'expression correspondante, c'est-à-dire la somme des produits des cœfficients de chaque paire par le pas de la BIV ou de l'IV associée. Pour la paire *(j, -2)*, il faut d'abord calculer le pas de l'IE définissant *l'IV j* (IE2) : il vaut 2. Le pas de IE1 est donc $1 + (-2) * 2 = -3$. Le procédé est similaire pour la valeur initiale de IE1, pour le calcul de laquelle il faut connaître la valeur initiale de chaque paire. Elle est nulle pour la paire *(i,1)*, et vaut -2 pour la paire *(j, -2)*. Ajouté au déplacement de IE1, cela donne $0 + (-2) + 3 = 1$.

Dans le cas où i est multi-incrémentée, par exemple incrémentée deux fois de la valeur 1, sa valeur totale d'incrémentation pour la boucle est 2. C'est cette dernière valeur qui doit être utilisée pour le pas de l'IE.

En ce qui concerne l'ensemble des IIEs, dont la détermination est le but final de la manœuvre, c'est donc le sous-ensemble des IEs indices d'un tableau. La détermination de l'ensemble des IIEs s'effectue donc en même temps que celle de l'ensemble des IEs.

Bilan

Finalement, on obtient un ensemble d'expressions, indices de tableaux, qui ont la particularité de voir leur valeur évoluer de façon constante au cours de l'exécution de la boucle. Cette évolution est garantie par construction, et représente une condition fondamentale pour la méthodologie développée. La figure illustre le résultat d'une telle analyse sur un corps de boucle relativement complexe, comportant deux BIVs, une IV, et un grand nombre d'IIEs.

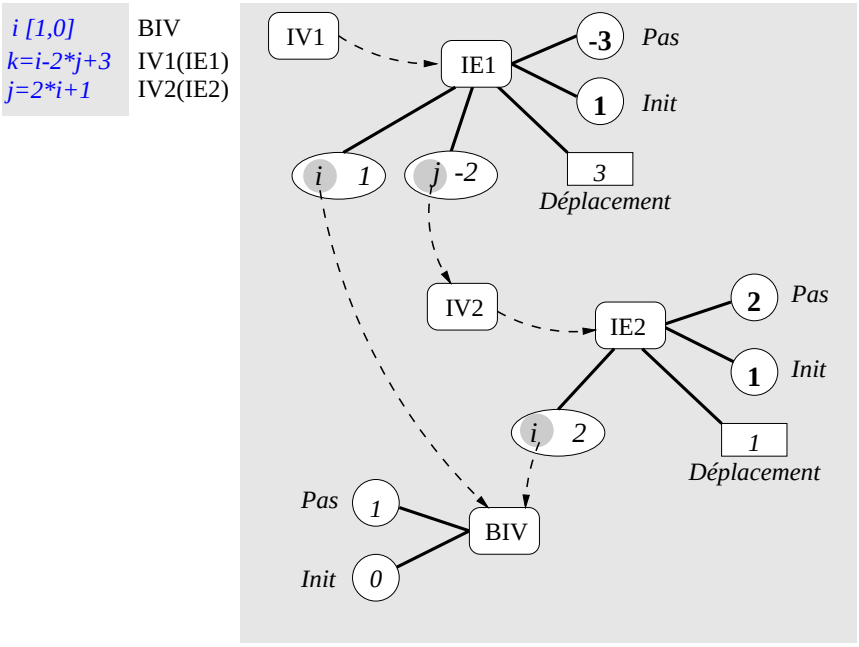


Figure 3.7: Arbre de dépendance d'une expression d'induction

3.3.4.3 Extraction des ensembles de connivence

A partir de l'ensemble des IIEs de la boucle traitée, il s'agit d'obtenir les sous-ensembles d'IIEs, de telle façon que toutes les références au tableau correspondant à chaque sous-ensemble, puissent être représentées par un même pointeur. A l'issue de cette étape de partitionnement, on obtient ce qui sera nommé par la suite les ensembles de connivence de la boucle.

Deux IIEs iie1 et iie2 appartiennent au même ensemble de connivence, si et seulement si :

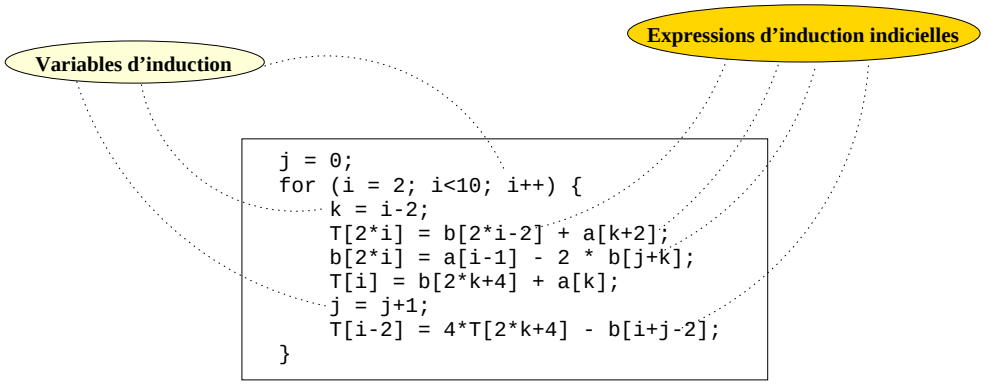


Figure 3.8: Exemple de l'analyse d'une boucle

1. $BIV(iie1) = BIV(iie2)$ où l'on note *BIV(iie)* le sous-ensemble des BIVs dont dépend *iie*. Autrement dit, si *iie1* et *iie2* sont de la même famille.

2. Les pas partiels sont identiques pour *iie1* et *iie2*. Par exemple les IIEs $i+2\times j+2$ et $i+2*(j+3)$ correspondent à cette condition (pas partiels respectifs 1 et 2 associés à i et j) avec i et j deux BIVs de pas 1. Tandis que les expressions $i+2\times j+2$ et $2\times i+j+3$ ne correspondent pas (pas partiels respectifs 1 et 2, et 2 et 1) bien que le pas global de ces deux IIEs soit le même (3).

3. Deux cas de figures interviennent enfin :

   (a) *iie1* et *iie2* sont indices du même tableau : alors elles appartiennent au même ensemble de connivence

   (b) *iie1* et *ii2* sont indices des tableaux a et b respectivement :

      i. ces deux tableaux sont dans le même espace-mémoire et le calcul de la distance entre les deux tableaux est possible (cf. 3.3.3.2 p. 56) : les deux IIEs sont alors placées dans le même ensemble de connivence.

      ii. ces deux tableaux ne sont pas dans le même espace mémoire et/ou le calcul de la distance entre ces tableaux est impossible : les deux IIEs ne sont pas placées dans le même ensemble de connivence.

A noter que, par la suite, seules les conditions 1, 2 et 3.a) seront considérées pour la construction des ECs. La prise en compte de la condition 3.b) suppose de connaître le placement mémoire des tableaux, ou de l'imposer par une étape préalable de placement.

Dans chaque ensemble de connivence, toutes les références tableaux correspondant aux IIEs qu'il contient, peuvent être référencées par le même pointeur pour les deux raisons suivantes :

- chaque expression indicielle évolue de façon constante au cours de l'exécution de la boucle : cela est certifié par l'analyse.

- toutes les expressions indicielles d'un même ensemble de connivence évoluent parallèlement, c'est-à-dire avec le même écart à chaque itération de la boucle.

La figure 3.9 illustre le procédé d'extraction des ensembles de connivence, en se basant sur la même boucle que précédemment. 5 ensembles sont extraits : 2 associés au tableau b, 2 au tableau T et 1 au tableau a. La variable d'induction k est de la famille de i, ce qui explique par exemple l'appartenance des IIEs k, *k+2* et *i-1* au même ensemble.

Avant la transformation elle-même, est construit le graphe des distances entre les IIEs constituant chaque EC. Les nœuds de ce graphe sont les IIEs elles-même. C'est un graphe orienté. Chacun des nœuds est relié à tous les autres sauf à lui même. Un arc entre deux nœuds est pondéré par la distance entre les deux IIEs correspondantes, c'est-à-dire la constante à ajouter à l'IIE source pour obtenir l'IIE destination. De tels graphes sont représentés sur la figure 3.9.

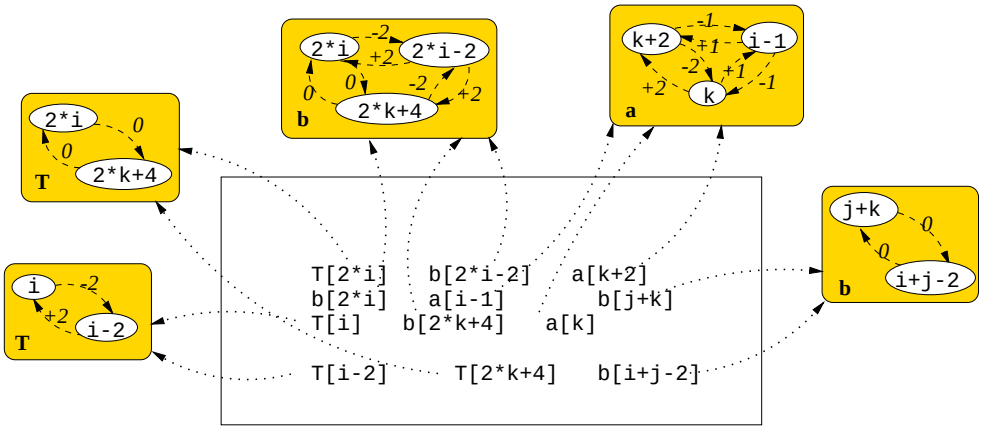


Figure 3.9: Exemple d'extraction des ensembles de connivence

Un ensemble de connivence sera nommé plus commodément EC au cours de ce document.

3.3.4.4 Transformation avec pré-calcul des pointeurs

Cette transformation ne modifie pas les variables pointeurs entre les références tableaux remplacées. Cette valeur est modifiée seulement pour préparer la prochaine itération de la boucle.

Chacun des ECs précédemment déterminés est traité l'un après l'autre. Chacun d'eux ne nécessite la création que d'un seul pointeur : les références tableaux correspondant aux IIEs de l'EC courant, pourront toutes être remplacées par une référence à ce pointeur, moyennant ou non une in/décrémentation.

Les premières étapes consistent à tout d'abord créer le nouveau pointeur correspondant à l'EC courant, puis à l'initialiser. Cette initialisation se fait à l'adresse d'un élément du tableau correspondant, premier de la séquence à être appelé. Cet élément va dépendre de l'IIE de base choisie, qui peut être n'importe laquelle à priori, bien que ce choix puisse être influencé par les caractéristiques de la machine cible, et par le mode d'adressage ciblé comme nous le verrons plus loin. L'IIE de base est l'IIE de référence d'un EC donné au cours de la transformation. Ainsi, si l'expression de base est *2.i+3*, indice du tableau *a,* à une position donnée du code source, et que i est initialisé à 1, alors le pointeur sera initialisé à *&a[5]*.

Ensuite, parcourant les EIIs de l'ensemble, est calculée pour chaque expression rencontrée, sa distance avec l'expression de base. Par exemple, si l'expression de base est *i+3* et que l'expression courante est *i+5*, la différence ou distance entre les deux est 2.

Trois cas de figure peuvent alors survenir, orientés par la valeur de cette distance :

1. La distance est 0 (EII de base ou autres). On remplace le tableau et son expression indicielle par l'expression équivalente **ptr*.

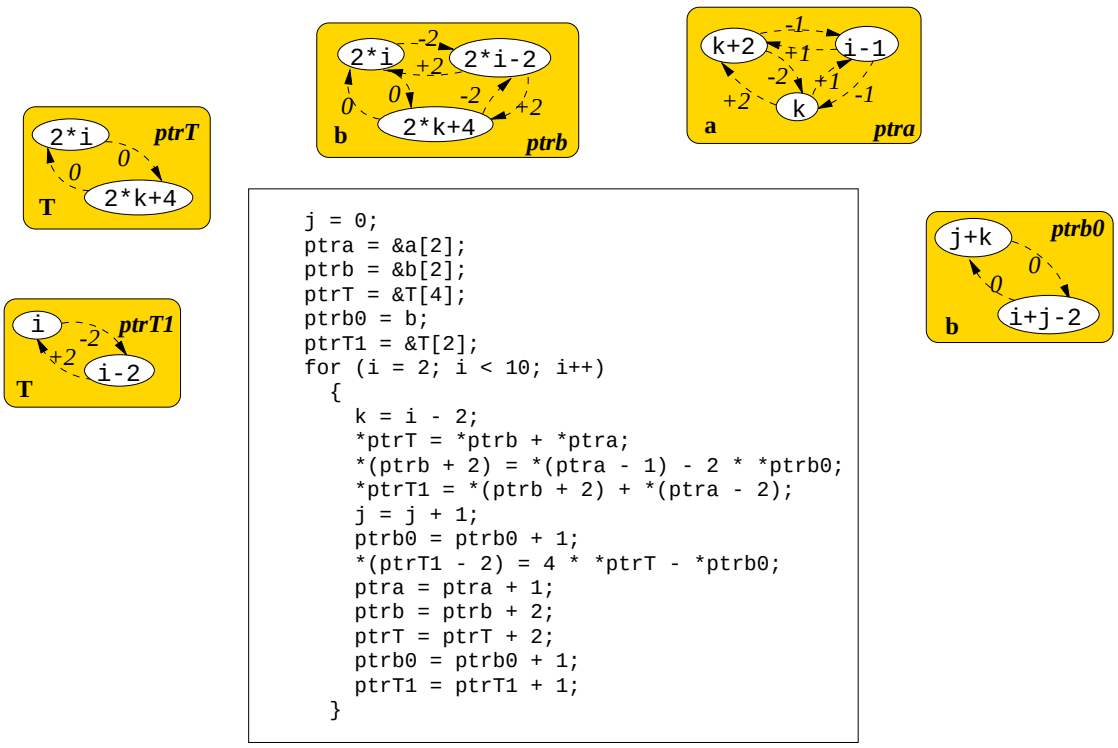


Figure 3.10: Exemple de transformation avec pré-calcul des pointeurs

2. C'est une constante entière : on remplace le tableau et son expression indicielle par l'expression équivalente **(ptr+cte)*.

3. C'est une constante de boucle (voir définition des variables d'induction) : aussi introduit-on le calcul de la distance dans le prologue de la boucle, calcul stocké dans une variable temporaire *dist*. Puis, le tableau et son expression indicielle sont remplacés par l'expression équivalente **(ptr+dist)*.

Il ne reste plus qu'à rajouter l'in/décrémentation du pointeur. La valeur considérée va dépendre de l'expression de base utilisée (échelle) et du pas d'incrémentation de la variable d'induction de base. Ainsi, si l'expression de base est *2.i+3* et que *i* évolue avec un pas de 2, alors l'incrémentation du pointeur aura la forme *ptr+=4*. La position de l'insertion dans le code de cette incrémentation est dépendante des variables d'induction de base que l'on pourra y rencontrer. Une loi générale est que cette insertion doit se situer de façon *immédiatement adjacente* à l'incrémentation de la variable d'induction de base concernée. Dans le cas de références tableaux dont l'indice est une fonction affine de plusieurs BIVs, le pointeur correspondant sera incrémenté autant de fois qu'il y a de BIVs présentes dans l'indice, avec un pas égal au pas partiel correspondant dans l'indice. Cette règle vaut de même dans le cas de BIV multi-incrémentées : le pointeur sera lui-même multi-incrémenté. Dans le cas particulier d'une boucle *for* sans variable d'induction supplémentaire que l'index, cette incrémentation se positionnera en fin de boucle.

La figure 3.10 dépeint le résultat de la transformation pour l'exemple déjà employé. Les pointeurs créés pour chaque ensemble sont indiqués dans leurs ensembles respectifs. On notera l'incrémentation double de *ptrb0*, dont l'ensemble contient des IIEs multi-BIVs, dont l'une de façon adjacente à l'incrémentation de *j*. Les IIEs de référence sont respectivement *2*i-2*, *k+2* et *i* pour les ECs dont les pointeurs sont *ptrb*, *ptra* et *ptrT1*. Pour les deux derniers ECs, cela n'a pas d'importance puisque la distance entre leurs IIEs est nulle.

Remarque Dans cet exemple, on peut remarquer l'évolution parallèle des BIVs *i* et *j*. En effet, on a la relation *j=i-2* avant *j=j+1*, et la relation *j=i-1* après, cela pour chacunes des itérations de la boucle. Dans ces conditions, on peut remplacer l'utilisation de *j* par *i-2* ou *i-1* suivant la position dans le code. Outre l'intérêt qu'il peut y avoir à supprimer une variable d'induction dans la boucle, l'avantage est ici la réduction des EC de 5 à 4 (fusion des ECs de *ptrb* et *ptrb0*).

3.3.4.5 Transformation avec post-modification des pointeurs

Cette transformation modifie la valeur des variables pointeurs entre les références tableaux remplacées. Elle est appelée ainsi, car force l'exploitation de mode d'adressage post-modifiés.

Les étapes préliminaires sont identiques : choix de l'IIE de base, création et initialisation du pointeur. Le remplacement des références tableaux par la référence au pointeur courant diffère. En effet, il est nécessaire de garder à tout instant la valeur dont a été incrémenté le pointeur au cours des précédentes étapes, de façon à faire pointer ce dernier sur la bonne position. C'est ainsi que, pour une référence tableau donnée, il faut préalablement incrémenter le pointeur de la valeur de la distance entre l'IIE de référence et l'IIE courante, diminuée de l'état d'incrémentation du pointeur. Cette incrémentation prend place après remplacement de la référence tableau précédemment traitée.

La dernière étape consiste à rétablir l'incrémentation du pointeur, de façon à préparer l'incrémentation suivante. L'état d'incrémentation du pointeur est encore utile pour cette étape. Pour le cas simple d'une boucle for bien formée, où la BIV s'incrémente en fin de boucle, il faut incrémenter le pointeur en tenant compte des incrémentations déjà subies, et du pas global de l'IIE de référence. Il faudra donc incrémenter le pointeur de la valeur *pas_global - état*. Dans le cas multi-incrémenté, ou dans le cas où un EC dépend d'une BIV in/décrémenté au milieu du corps de la boucle (et non seulement à la fin), on se retrouve dans la même situation, puisque le pointeur correspondant est incrémenté de façon adjacente à l'in/décrémentation des BIVs correspondantes, et que ces incrémentations sont prises en compte par le compteur d'incrémentations *état*.

En appliquant cette transformation au même exemple que précédemment, on obtient le code représenté sur la figure 3.11.

On observe une baisse de lisibilité du code avant et après transformation : cela est un argument en faveur de l'automatisation de celle-ci. Manifestement, un certain nombre d'opérations supplémentaires sont rajoutées, par rapport au code produit par la transformation avec pré-calcul des pointeurs. Il s'agit des modifications de la valeur des pointeurs entre les références tableaux qu'ils sont sensés représenter, et des incrémentations

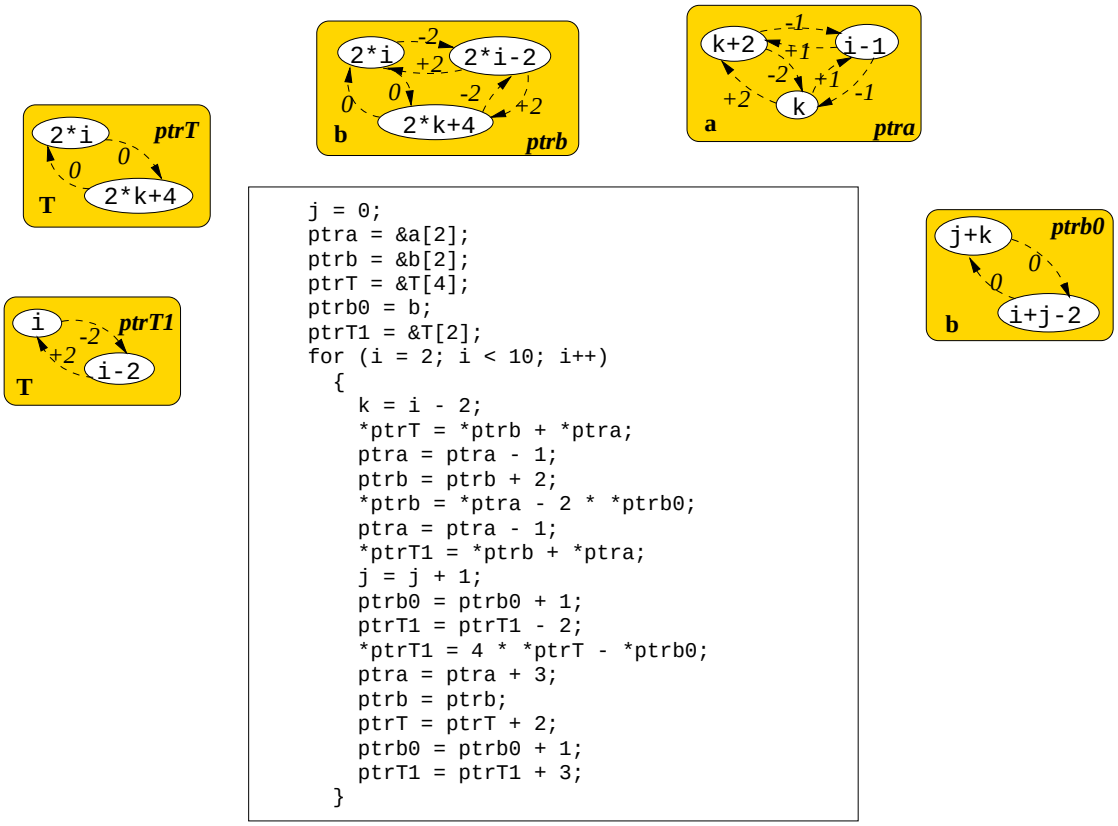


Figure 3.11: Exemple de transformation avec post-modification des pointeurs

ou rétablissement globaux à la boucle. Noter par exemple le rétablissement de la valeur de *ptra* en fin de boucle ($-1-1+3=+1$). *ptrb0* remplace les références tableaux dont les indices sont fonction affine des deux BIVs i et j. Ces deux IIEs s'incrémentent de 2 à chaque itération de la boucle, ce qui explique l'incrémentation finale du pointeur, en plus de l'incrémentation adjacente à celle de j.

D'autre part, le cas étant le même pour la transformation avec pointeurs pré-calculés, il est évident ici que certaines opérations ne sont plus utiles et peuvent être tout bonnement éliminées. Les variables j et k sont à présent inutiles, de même que les opérations concernées. La variable de boucle i reste utile pour le fonctionnement de la boucle. Comme il est aisé de calculer que la boucle s'exécute 8 fois, on pourrait exploiter ici l'existence dans le processeur cible d'un mécanisme de boucle matérielle.

Présence de code conditionnel dans la boucle

Un problème se pose dans le cas de la post-modification, et de l'introduction d'expression d'incrémentation de pointeur entre les différentes références, dans le cas de code conditionnel dans la boucle, comme illustré par l'exemple de code 3.12.a.

Ainsi que mis en évidence en sortie de code conditionnel, sur le code 3.12.b, la référence tableau de base choisie étant *a[i]* avec i BIV de pas 1 : comment savoir s'il faut ou non

a)

```
... a[i+1] ...
if (cond) {

... a[i+1] ...
}
else {

... a[i+2] ...
}
... a[i+1] ...
```

b)

```
ptr++;
... *ptr...
if (cond) {

... *ptr ...
}
else {
   ptr++;

...*ptr...
}
ptr--;
? ? ? ?
... *ptr...
```

c)

```
ptr++;
... *ptr...
if (cond) {

... *ptr ...
}
else {
   ptr++;

...*ptr...
   ptr--;
←
}
... *ptr...
```

Figure 3.12: Cas de la présence de code conditionnel dans le corps de boucle

décrémenter *ptr* ? Répondre à cette question étant à priori impossible, deux solutions s'offrent. La première consiste à rajouter du code conditionnel pour l'incrémentation, ce qui est clairement lourdissime, donc à exclure; la deuxième rajoute aussi du code, mais seulement afin d'effacer les problèmes en aval, ce qui correspond au code 3.12.c.

Ainsi, si à chaque rajout d'incrémentation dans une branche, celle-ci est suivie, après l'expression tableau, d'une décrémentation de même valeur et inversement, l'ambiguïté précédente est levée. Cela reste un inconvénient, et rajoute une pierre de taille conséquente en faveur d'un mode d'écriture pré-calculé dans un tel cas.

IIEs d'un même ensemble de connivence, issues de la même expression-source

Soit l'exemple de code 3.13.a. A l'issue de l'analyse, on distingue deux ensembles de connivence : $\{i;\ i+1\}$ et $\{i\}$ dont les tableaux respectifs sont a et b. Les deux références tableaux du premier ensemble appartiennent à la même expression source. Or, une distance de 1 existe entre ces deux IIEs. Si l'on veut utiliser le même pointeur pour les deux références tableaux, il faut pouvoir incrémenter celui-ci de 1 entre les deux références.

a)

```
for (i = 0;  i < N-1;  i++) {
   b[i] = a[i] + a[i+1];
}
```

b)

```
ptrb = b;
ptra = a;
for (i = 0;  i < N-1;  i++) {
  int tmp;
  tmp = *ptra;
  ptra++;
  *ptrb = tmp + *ptra;
  ptrb++;
}
```

Figure 3.13: IIEs d'un même ensemble de connivence appartenant à la même expression-source

La solution retenue consiste à déclarer un temporaire, qui stockera la valeur de la première référence, tandis que le pointeur sera incrémenté pour accéder à la valeur de la

seconde. Cela revient à forcer l'ordonnancement des opérations, et donne l'écriture 3.13.b. L'autre solution consiste à couper en deux l'ensemble de connivence, et à considérer non plus deux, mais trois ensembles mono-élément, soit trois pointeurs.

3.3.4.6 Prise en compte des boucles imbriquées

C'est une extension logique et générale de l'algorithme simple précédent. Dans le cas de boucles imbriquées, quel que soit leur type (*for*, *while*, *do-while*), le traitement de base est appliqué à chaque boucle imbriquée en partant de la boucle la plus interne, pour finir par la boucle la plus externe. Il y a donc indépendance dans le traitement de chaque boucle par rapport à ses congénères. La seule contrainte nécessaire est un message indiquant à une boucle traitée que sa boucle interne à été préalablement transformée, ce qui provoque la reconstruction de la représentation intermédiaire (GFC cf. annexe A.3) avant application des analyses sur le corps de boucle. Cette indépendance ne veut cependant pas dire que les boucles sont invisibles les unes par rapport aux autres. Une boucle verra toutes les boucles qui lui sont internes, mais *dans son propre contexte*. Cela signifie qu'une IIE présente dans le corps d'une des boucles internes sera détectée, et les relations entre IIEs présentes dans différents corps de boucles imbriquées pourront ainsi être exploitées. Il n'existe aucune restriction quant à la profondeur d'imbrication.

Application pas à pas Soit les boucles imbriquées suivantes :

```
r = 0;
for (i = 0;  i < 60;  i += 2) {
   for (j = 40;  j > 0;  j--) {
      r = a[i + j]×b[j] + r;
      b[i] = b[j] + r;
   }
}
```

Comme indiqué ci-dessus, on applique tout d'abord la transformation (avec pointeurs indexés ou post-incrémentés, en l'occurence, le résultat est le même) à la boucle la plus interne. On obtient le code 3.14.a.

A noter que le tableau *b[i]* n'est pas pris en compte car indépendant de *j*.

Pa est initialisé à l'adresse de *a[i+3]*, *i* étant une constante de boucle pour la boucle interne. Puis, on applique la même transformation à la boucle externe, ce qui donne le code 3.14.b. Les références tableaux créées au cours de la précédente passe sont à leur tour transformées, au même titre que les références non traitées dans le corps de la boucle interne, telle *b[i]*. Quatre pointeurs sont donc requis.

Remarque: Une autre écriture est possible, usant de 3 pointeurs seulement :

```
int * pa, * pb, * pb1;
r = 0;
pa = &a[40];  /* Init */
pb1 = b;  /* Init */
```

a)

```
int * pa, * pb;
r = 0;
for (i = 0;  i < 60;  i += 2) {
   pb = &b[40];  /* Init */
   pa = &a[i+40];  /* Init */
   for (j = 40;  j > 0;  j--) {
      r = *pa×*pb + r;
      b[i] = *pb + r;
      pa--;
      pb--;
   }
}
```

b)

```
int * pa, * pa1, * pb, * pb1;
r = 0;
pa1 = &a[40];  /* Init */
pb1 = b;  /* Init */
for (i = 0;  i < 60;  i += 2) {
   pb = &b[40];  /* Init */
   pa = pa1;  /* Init */
   for (j = 40;  j > 0;  j--) {
      r = *pa×*pb + r;
      *pb1 = *pb + r;
      pa--;
      pb--;
   }
   pb1+=2;
   pa1+=2;
}
```

Figure 3.14: Application pas à pas de la transformation appliquée à deux boucles imbriquées

```
for (i = 0;  i < 60;  i += 2) {
   pb = &b[40];  /* Init */
   for (j = 40;  j > 0;  j--) {
      r = *pa×*pb + r;
      *pb1 = *pb + r;
      pa--;
      pb--;
   }
   pb1+=2;
   pa+=42;
}
```

Dans l'objectif d'automatiser l'obtention d'un tel résultat, il est nécessaire de faire communiquer les deux passes, de savoir que *a[i+j]* dépend d'une BIV de la boucle externe, et surtout de transmettre l'information selon laquelle pa est diminué de 40 à chaque application de la boucle interne, ce qui implique de rajouter à la valeur d'incrémentation de *pa* dans la boucle externe (c'est-à-dire 2), la valeur 4. La méthodologie illustrée plus haut se place dans le cas général et donne un résultat valable, bien que non optimum en terme de nombre de registres dans ce cas précis. Les résultats montrent cependant que la différence dans les performances obtenues, entre le code résultant de la transformation ici exposée, et l'écriture améliorée illustrée précédemment, n'est pas forcément en faveur de cette dernière. Ce problème se pose de la même manière pour les tableaux multi-dimensionnels qui se trouvent en général dans des corps de boucles imbriquées. Une rubrique spécifique se consacre à ce point dans le chapitre relatant les résultats d'expérimentations (cf. section 4.3.3).

3.3.4.7 Tableaux multi-dimensionnels

Soit le code classique d'une multiplication de matrice, tiré de [30]:

```
int a[10][10];
int b[10][10];
int c[10][10];
```

```
void main () {
  int i, j, k;
  for (i = 0;  i<10;  i++) {
     for(j = 0;  j < 10;  j++) {
        c[i][j] = 0;
        for(k = 0;  k < 10;  k++) {
           c[i][j] += a[i][k] * b[k][j];
        }
     }
  }
}
```

Selon [30], bien que ce type d'écriture du C soit correct et compréhensible car très lisible, il n'est pas efficace. Une solution manuelle utilisant des pointeurs est alors donnée par la même source, et correspond au code C suivant :

```
int a[10][10];
int b[10][10];
int c[10][10];
void main () {
  int i, j, k;
  int * ptra, * ptrb, * ptrc;
  for (i = 0;  i<10;  i++) {
     ptrc = &c[i][0];
     ptrb = &b[0][0];
     for(j = 0;  j < 10;  j++) {
        ptra = &a[i][0];
        *ptrc = (*ptra++) * (*ptrb++);
        for(k = 1;  k < 10;  k++) {
           *ptrc += (*ptra++) * b[k][j];
        }
        ptrc++;
     }
   }
}
```

Cette solution ne requiert que 3 pointeurs. Elle est d'autre part particulièrement élégante dans le sens où elle évite l'initialisation des éléments de la matrice c à 0. Une solution automatique, donnée plus bas, sans atteindre le degré d'élégance du code ci-dessus, est néanmoins plus efficace dans le sens où elle produit un code "tout pointeur", et ce malgré l'absence de l'optimisation consistant à éviter l'initialisation à 0 des éléments de c, difficile à systématiser ...

Dans le cas standard du placement linéaire des tableaux en mémoire, si l'on considère un tableau de dimensions $N_1 \times ... \times N_n$, dont la référence endosse la forme :

$$\mathrm{A}[f_1]...[f_n]$$

la fonction globale de parcours des éléments du tableau à partir de l'adresse du premier élément est

$$\sum_{i=1}^{n} f_i \times p_i$$

où

$$p_i = \left\{ \begin{array}{cc} \prod_{k=i+1}^{n} N_k, & pour\ 1 \leq i < n \\ 1, & pour\ i = n \end{array} \right\}$$

Pour un tableau bi-dimensionnel a, de dimensions $N \times M$, l'élément à accéder *a[i][j]* se trouve donc par convention à l'adresse $a + M \times i + j$. L'expression $M \times i + j$ est une sorte de *super expression d'induction* ou *super-IE*. La solution adoptée passe par une représentation un peu plus complexe des IIEs, telle que représentée sur la figure 3.15.

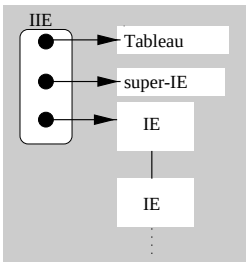


Figure 3.15: Une IIE plus riche

Le champ super-IE contient donc la fonction globale de parcours des éléments du tableau correspondant. La liste d'IEs contient la liste des indices de chaque dimension du tableau. En appliquant les techniques identiques à celles employées pour les IEs classiques, l'analyse dira si cette fonction est une fonction affine des variables d'induction courantes, et donnera son pas global. Les conditions requises vérifiées, cette expression construite à partir des différentes IIEs de chaque dimension du tableau, sera alors traitée comme une IIE normale pour ce qui est de la construction des ensembles de connivence. L'initialisation du tableau se fait en gardant la structure de la référence tableau, ce qui signifie qu'il faut garder la connaissance des IIEs de chaque indice $f_1 ... f_n$. Il faudra notamment calculer les valeurs $f_{1_{init}} ... f_{n_{init}}$. Le reste de la transformation s'exécute de façon classique. Cette façon de procéder, permet de garder les mêmes schémas de transformation. De plus, elle englobe le cas simple donné en exemple en début de section, ainsi que les cas moins simples d'indices tableaux fonctions de plusieurs BIVs. Pour revenir à l'exemple initial, l'application de la présente technique donne :

```
int a[10][10];
int b[10][10];
int c[10][10];
```

```
void main () {
  int i, j, k;

int * ptra, * ptra2, * ptrb, * ptrb2, * ptrc, * ptrc2;
  ptrc2 = &c[0][0];
  ptra2 = &a[0][0];
  for (i = 0;  i<10;  i++) {
     ptrb2 = &b[0][0];
     ptrc = ptrc2;
     for(j = 0;  j < 10;  j++) {
         ptra = ptra2;
         ptrb = ptrb2;
         *ptrc = 0;
         for(k = 0;  k < 10;  k++) {
            *ptrc += (*ptra) * (*ptrb);
            ptra++;
            ptrb += 10;
         }
         ptrb2++;
         ptrc++;
     }
     ptrc2++;
     ptra2++;
   }
}
```

Cette solution automatique requiert 6 pointeurs, soit 3 de plus que la solution manuelle. Néanmoins, il sera montré plus bas que cette solution est plus efficace (cf. section 4.3.3). A noter cette fois le remplacement de la référence à la matrice b par un pointeur, par rapport à la solution manuelle donnée par [30] : c'est là la raison de la plus grande efficacité du code "tout-pointeur".

3.3.5 Informations sur la machine cible

3.3.5.1 Modes d'adressage

Selon [47], un mode d'adressage définit une façon d'accéder aux données, que celles-ci soient dans des registres, en mémoire ou disponibles dans l'instruction même. Un mode d'adressage définit donc un moyen pour une instruction donnée d'accéder à ses opérandes. En l'occurence, nous nous intéresserons aux diverses façons d'accéder à une donnée en mémoire, via des instructions spécialisées (*load* et *store*). Les modes d'adressage portent un autre nom : "légion". Etant donné le type d'architecture considéré, DSPs destinés à être embarqués dans un système plus vaste, souvent dédiée à une application (ASDSP), certains modes d'adressage seront ignorés au profit des plus usités.

La figure 3.16 donne un aperçu des modes d'adressage-mémoire les plus communs [47, 77]. Le premier est le plus simple et correspond à un adressage direct où l'adresse est fournie directement. Il est de peu d'intérêt dans le présent contexte. Les modes d'adressage pour lesquels l'adresse est calculée de façon indirecte, à partir de registres, sont plus pertinents ici. Nous considérerons au cours de ce document que l'architecture dispose d'une unité de calcul d'adresses (réelle ou virtuelle), capable de fournir la bonne adresse mémoire selon le mode d'adressage. Les registres utilisés pour stocker les adresses pourront être des registres spécialisés ou généraux suivant le processeur.

Deux grandes classes sont propres à être distinguées : les modes d'adressage pré-calculés ou non-modifiants et post-calculés ou modifiants. La première classe comporte tous les modes d'adressage nécessitant un calcul de l'adresse immédiatement avant son utilisation, c'est-à-dire ici les modes d'adressage basés ou relatifs, et indexés. On peut concevoir des variations de ces modes d'adressage, comme le mode basé indexé. La seconde classe comporte les modes d'adressage utilisant l'adresse présente dans le registre de base, avant de modifier son contenu, c'est-à-dire ici les modes auto-in/décrémenté, post-in/décrémenté par constante ou par index. Le mode d'adressage modulo est à part, bien que la version présentée ici soit du type pré-calculée. On peut concevoir néanmoins une version post-modifiée de ce mode très particulier mais employé d'adressage. Les modes post-modifiés sont intéressants par le parallélisme qu'ils offrent, puisque l'accés à la mémoire se fait en même temps que la modification du registre d'adresse. Par contre, les modes d'adressage pré-calculés imposent le calcul de l'adresse avant d'accéder à la mémoire, ce qui se traduit en général par une pénalité en termes de cycles supplémentaires. Il existe cependant des machines dans lesquelles le cycle d'exécution est assez long pour autoriser des modes pré-calculés sans cycles de pénalité.

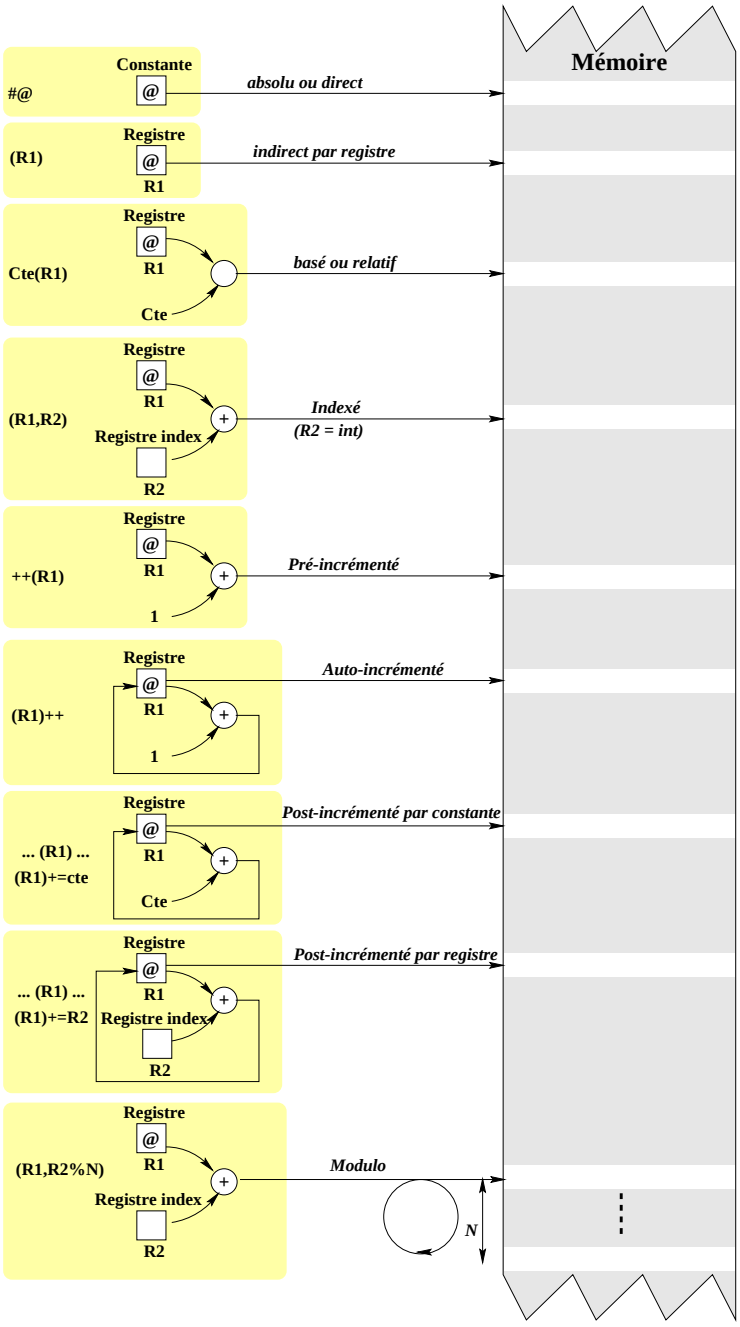


Figure 3.16: Quelques modes d'adressage parmi les plus communs

3.3.5.2 Exemples concrets

Sont décrits ci-après les AGUs[11] (ou ACUs[12]) de deux processeurs ciblés au cours des expérimentations effectuées durant ces travaux. Ils sont décrits plus en détail au chapitre 4. L'intérêt de cette section est de cerner les caractéristiques d'adressage de ces processeurs, de façon à pouvoir définir une manière de décrire ces caractéristiques afin de les exploiter par la suite.

Le Processeur Audio Numerique Sapphire Le Sapphire est un processeur spécifique (ASIP) mis au point par la division DPG de ST[13] et visant des applications de traitement du signal audio. C'est un petit ASDSP de 24 bits qui permet d'obtenir des taux de parallélisme intéressants. Il possède notamment deux unités de génération d'adresses (ACU), l'une associée à l'espace mémoire X, l'autre à l'espace mémoire Y. Chacune de ces ACUs comporte deux registres d'adresse. Pas de registre d'index dédiés: un registre-donnée est employé pour cela. Le tableau ci-dessous résume les modes d'adressages mémoire permis par ces deux ACUs. *Axi* et *Ayi* désignent les registres d'adresses des deux ACUs respectivement. *Mx* et *My* sont les registres de modulo (contenant la valeur du modulo). *Dx* et *Dy* désignent les bancs de registres-données dédiés aux deux espaces mémoire.

MODE	**Opérations correspondantes**
Direct	#immédiat(12b)
Indirect par registre	(Axi), (Ayi)
Auto-incrémenté	(Axi)++, (Ayi)++
Post-incrémenté modulo	(Axi)++%Mx, (Ayi)++%My
Post-modifié par registre	(Axi)+=Dx, (Ayi)+=Dy
Indirect Basé ou relatif	#imm(Axi), #imm(Ayi)
Indirect Indexé	(Axi,Dx), (Ayi,Dx)

Quelques précisions : pour le mode basé, seul un tout petit immédiat est disponible en guise de déplacement (4 bits, soit une valeur max de 15 pour le déplacement), ce qui peut s'avérer une information primordiale dans le cadre qui nous intéresse. Ce mode d'adressage pré-calculé ne provoque pas de pénalité de cycle. Au niveau du parallélisme instruction, certains modes sont plus intéressants que d'autres, comme les modes auto-incrémentés. Globalement, aucun cycle de pénalité ne vient allourdir ces modes d'adressages, qui affectent seulement, et de façon plus ou moins appuyée, le parallélisme autorisé.

Processeur MMDSP Ce processeur, décrit dans [72], est un ASDSP ou processeur DSP spécifique, et possédant une AGU efficace comprenant registres d'adresse, d'index et de modulo ainsi que des possibilités d'in/décrémentation par des valeurs constantes, contenues dans des registres d'index ou provenant de l'instruction même. L'AGU ne supporte que des

[11] Address Generation Units
[12] Address Calculation Units
[13] STMicroelectronics

modes d'adressages post-modifiés. Elle comprend 6 registres d'adresses, dont trois restreints (ne supportant pas le mode modulo), trois registres d'index et trois couples de registres modulo min et max. Le parallélisme n'est pas affecté par les opérations de l'AGU; aucun cycle de pénalité.

MODE	**Opérations correspondantes**
Post-incrémenté	(Axi)+=1, (Axi)+=2,(Axxi)+=1, (Axxi)+=2
Post-décrémenté	(Axi)-=1, (Axi)-=2,(Axxi)-=1, (Axxi)-=2
Post-incrémenté par constante	(Axi)+=const,Axx)+=const
Post-modifié par registre	(Axi)+=Ixj, (Axi-=Ixj),(Axxi)+=Ixj, (Axxi-=Ixj)
Post-modifié par registre, Modulo	mod(Axi, Ixj, Mxj, mxj)

Le mode post-incrémenté par constante utilise le champ immédiat du jeu d'instructions pour la valeur de la constante, sur 16 bits. Une caractéristique de cette AGU est l'association exclusive deux-à-deux entre registres d'adresses et registres d'index. En clair, le registre *Ax1* ne peut être ajouté qu'au contenu du registre d'index *Ix1*, etc ... De même pour les registres restreints Axxi.

3.3.5.3 Description des ressources d'adressage de la machine cible

Deux parties peuvent être distinguées dans la description adoptée : la façon de décrire ses ressources, et la représentation de l'AGU de façon à exploiter au mieux les informations décrites. Ce dernier point sera plus compréhensible par l'exposé de quelques exemples.

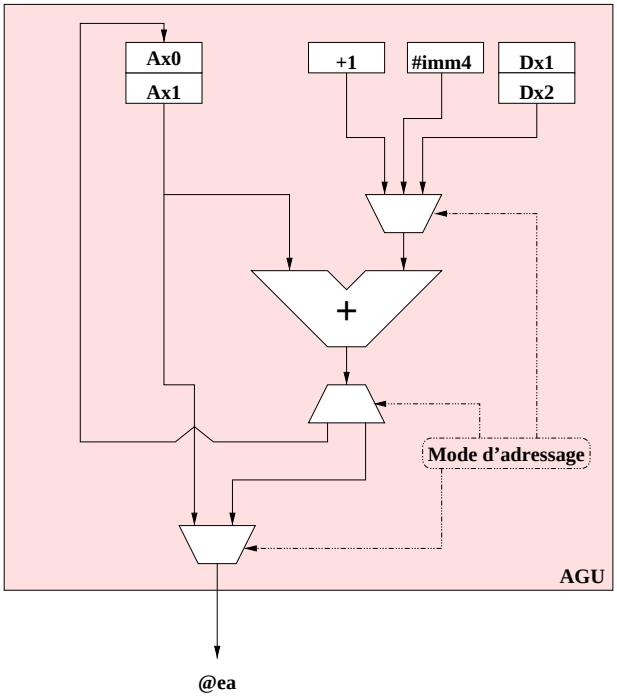


Figure 3.17: Représentation schématique de l'AGU du Sapphire

La figure 3.17 représente l'une des deux AGUs du processeur Sapphire. Le mode modulo est délibérément absent car non considéré au cours de ces travaux. *@ea* représente l'adresse effective générée par l'AGU. En imaginant avoir à sa disposition une telle représentation, un certain nombre de questions demeurent obscures. Par exemple¸ comment en extrapoler que *#imm* n'est disponible que pour le mode basé simple, ou que la constante *+1* n'est disponible que pour le mode post-incrémenté ?

Une solution consiste à se baser sur une description simple des ressources d'adressage, à partir de laquelle est extrapolée pour chaque registre d'adresse une mini-AGU. C'est la solution adoptée par [71, 70]. Les ressources communes à chaque registres, comme le partage de registres d'index, y sont factorisées. Ce style descriptif simple à été repris et étendu au cours de ces travaux:

1. En étendant la factorisation des ressources aux éventuelles constantes et autres immédiats disponibles

2. En ajoutant une notion d'échelle de valeur, notamment pour les registres d'index et les immédiats, c'est-à-dire en tenant compte de leur taille afin de pouvoir décider si une valeur peut être contenue dans un registre d'index, ou dans le champ immédiat du mot-instruction.

3. En ajoutant la prise en compte de modes d'adressages pré-calculés comme l'adressage indirect basé et indexé

4. En ajoutant un moyen de distinguer différents espaces mémoires

5. En ajoutant un moyen d'associer un coût par opération d'adressage, coût pouvant traduire une éventuelle pénalité de cycle, ou une influence sur le parallélisme inhérent au jeu d'instructions.

Les fichiers de description des ressources d'adressage du MMDSP et du Sapphire peuvent être consultés en annexe B à titre d'exemple. A partir d'un fichier de spécification sont construits deux types d'informations :

1. information structurelle liée à l'interconnexion entre ressources (registres) et opérateurs.

2. information de mode d'adressage.

Un mode d'adressage est ici décrit comme un registre d'adresse, une opération, une destination et une unité de déplacement, qui peut être un registre d'index, une constante ou un immédiat. Les avantages d'une telle description sont tout d'abord de pouvoir aisément distinguer la classe du mode d'adressage, c'est-à-dire si c'est un mode d'adressage non-modifiant ou modifiant. Dans le premier cas, la destination est une mémoire, alors que dans le second cas, la destination est un registre d'adresse (le même registre). Le second avantage est de pouvoir accéder facilement à l'ensemble des unités, comprenant valeurs immédiates et constantes, qui vont pouvoir être associées aux registres d'adresse. En effet,

chaque architecture présente un certain niveau d'orthogonalité. L'idéal est d'avoir affaire à une architecture dans laquelle tous les registres d'adresses peuvent utiliser tous les types de déplacements disponibles. Ce n'est pas toujours le cas, comme le prouve l'exemple du MMDSP, où chaque registre d'adresse est associé à un unique registre d'index. Dans le but de prendre en compte les spécificités architecturales, les modes d'adressage sont factorisés s'ils concernent le même registre, ont la même destination (mémoire ou registre) et la même opération (+, -). Le coût associé à chaque opération d'adressage est une contrainte supplémentaire dans cette factorisation, si ce coût est spécifié.

La représentation interne d'une AGU du Sapphire sera donc celle représentée sur la figure 3.18. Etant donné que les registres d'adresse possèdent les même propriétés d'adressage, ils sont regroupés en un banc qui factorise ces propriétés. On retrouve les deux grandes classes déjà mentionnées.

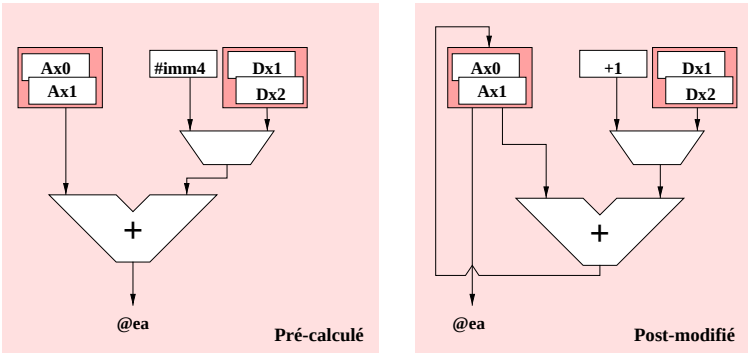


Figure 3.18: Représentation interne de l'AGU du Sapphire en fonction des modes d'adressage.

3.3.6 Transformations orientées par l'architecture

Transformer à haut niveau d'abstraction a ceci d'inconvénient que l'on est loin de la machine cible. Ce manque d'information machine peut être la cause de la sous-performance d'une transformation à priori efficace, simplement parce que celle-ci imposera l'utilisation de ressources machines absentes ou inefficaces au cours des étapes de compilation de plus bas niveau. Donner à une optimisation au niveau source des informations sur la machine telles que celles indiquées dans la section précédente, peut pallier ce manque. L'objectif n'est pas de se substituer aux étapes dorsales telles que l'allocation ou l'ordonnancement des opérations mais au contraire de faciliter leur application. C'est le cas de la transformation qui nous intéresse ici, c'est-à-dire une transformation visant à faciliter l'exploitation des ressources d'adressage de la machine. A l'aide des informations sur les ressources d'adressage, la transformation directe exposée dans la section 3.3.4 pourra s'appliquer de façon plus orientée et plus judicieuse. Les quelques sections suivantes exposeront les possibles moyens d'actions retenus à la lumière des spécifications du processeur cible. Ceux-ci n'ont pas tous été implémentés au cours de ces travaux, mais présentent suffisamment

d'intérêt pour avoir fait l'objet d'une étude en vue d'une implémentation future. C'est la raison pour laquelle ils sont détaillés ici.

3.3.6.1 Manipulation des ensembles de connivence

Les ensembles de connivence regroupent les références tableaux pouvant être remplacées par une seule et même référence pointeur. A chaque ensemble de connivence va donc correspondre un pointeur dans le code source issu de la transformation. C'est le cas simple. Celui-ci peut-être compromi ou amélioré par la connaissance des possibilités en termes d'adressage disponibles dans la machine cible. Tout d'abord le nombre de registres d'adresse disponibles est une condition forte de la faisabilité de la transformation. A la lumière des opérations d'adressage possibles, deux cibles privilégiées sont retenues pour orienter la transformation. Tout d'abord, le choix de l'élément de référence dans chaque ensemble primordial peut influer sur la qualité de la transformation. Ce choix est pertinent dans le cas unique de l'écriture avec pré-calcul des pointeurs. Un certain nombre d'alternatives s'offrent pour celui-ci, et sont précisées dans cette section. Ensuite, le partitionnement des ensembles de connivence, conditionné par le degré de liberté existant en termes de nombre de registres d'adresse, est un moyen intéressant d'améliorer la transformation pour un processeur ciblé. Diverses possibilités font l'objet d'un exposé détaillé.

Nombre de registres d'adresse disponibles et faisabilité

Il est possible que le nombre d'ECs calculés par analyse soit supérieur au nombre de registres disponibles. Autrement dit, que le nombre minimum de pointeurs nécessaires à la transformation soit supérieur au nombre de pointeurs possibles autorisés par les ressources de la machine. Dans ce cas, plusieurs solutions sont possibles, naturellement très liées aux propriétés inhérentes à la machine d'une part, et au compilateur, et plus particulièrement à l'allocateur de registres, d'autre part.

Une condition à la faisabilité de la transformation réside dans la possibilité pour la machine de vider les registres d'adresse, afin d'en libérer suffisamment lorsque le besoin s'en fait sentir. Cette option de vidage de registre, plus connue sous le nom de "spilling", est présente sur la plupart des architectures. Elle consiste à transférer en mémoire le contenu d'un registre, contenu qui sera restauré lorsque celui-ci interviendra dans les calculs. En cas d'absence, la transformation n'est pas possible, et il est fort possible que le programme initial ne tourne pas sur la machine cible.

Si le spilling est disponible, la faisabilité de la transformation dépendra de l'algorithme d'allocation des registres d'adresse, et surtout de la façon dont le vidage mémoire des registres est géré. Cela est difficilement contrôlable au niveau d'abstraction considéré ici. Une règle importante à ce sujet, est d'éviter de pratiquer le vidage de registre dans les boucles en général, et surtout dans les boucles internes en cas de boucles imbriquées. En effet, vider le contenu d'un registre en mémoire, puis restaurer ce contenu impose le rajout d'opération d'accès mémoire coûteuses. Dans ce dernier cas, puisque la transformation traite en priorité les boucles internes, l'optimisation force l'assignation des pointeurs à des registres précis dans ces boucles, et laisse le vidage pour les boucles externes. Il est aussi

possible de forcer l'optimisation à n'utiliser que le nombre autorisé de pointeurs, en laissant les tableaux non transformés tels quels.

Impact du choix de l'élément de référence de chaque ensemble de connivence

L'élément de référence d'un EC est, rappelons-le, l'IIE à partir de laquelle le pointeur sera initialisé, et à partir de laquelle les déplacements entre références du pointeur seront calculés. En pratique, l'impact du choix de cet élément ne sera utile que dans le cas de modes d'adressage non-modifiants.

Favoriser des déplacements positifs Pour une simple raison de choix d'écriture, ou de disponibilité de mode d'adressage indexé avec déplacement strictement positif, on peut être amené à désirer n'avoir une écriture finale des pointeurs que sous forme indexée avec déplacements positifs. Par exemple, si un EC est composé des IIEs dont les expressions sont les suivantes: i+1, i, i+2, i-1, i. Choisir la première IIE conduit aux déplacements respectifs suivants: 0, -1, +1, -2. Le choix de la dernière s'impose comme celui donnant des déplacements positifs: +2, +1, +3, 0, +1. Un simple parcours du graphe des distances est suffisant pour aboutir au résultat : l'IIE choisie a tous ses arcs sortants pondérés positivement.

Favoriser les déplacements les plus petits Il est fréquent de trouver des machines proposant des modes d'adressages avec constante directement câblées (1, 2, ...). Il s'agit en général de petites constantes, intervenant souvent dans les calculs d'adresses. On peut trouver par ailleurs des modes d'adressage usant de petits offsets immédiats (cas du Sapphire). Ces modes présentent souvent un coût (en termes de cycles de pénalité ou de parallélisme) inférieur à d'autres modes. Favoriser des déplacement d'une taille relativement faible peut ainsi s'avérer intéressant.

Dans l'optique de minimiser la taille des déplacements, le choix de l'IIE de référence dans les ECs peut jouer un rôle. L'heuristique simple implémentée consiste à choisir comme IIE de référence, l'IIE donnant la distance moyenne avec les autres IIE de l'ensemble la plus petite possible en valeur absolue (la somme des distances en valeur absolue fonctionne de même). Ainsi, si l'on prend la séquence d'expressions indicielles précédente, on ne peut éviter l'utilisation de la constante 2 ou -2. En excluant les IIEs correspondant aux expressions *i-1* et *i+2* (la distance de 3 est alors inévitable), chacune des autres IIEs conduit à l'utilisation des valeurs 1 et 2 en valeurs absolues comme valeurs de distance. Néanmoins, le choix de i comme IIE de référence (l'une ou l'autre des IIEs ayant i comme expression fait l'affaire) apparait comme le plus judicieux : il correspond à la moyenne des distances en valeur absolue la plus faible (1 contre 1.25 pour *i+1*). Cette méthode donne des résultats satisfaisants dans la plupart des cas.

Rechercher le plus petit nombre possible de déplacements différents pour l'ensemble des pointeurs créés. Minimiser le nombre de déplacements différents est plus difficile, et demande un travail plus global sur l'ensemble des ECs. Ce critère n'a

de réel intérêt que si le nombre de registres d'index est très limité et si, d'autre part, l'architecture est suffisamment orthogonale pour que les registres d'adresses puissent utiliser n'importe lequel des registres d'index. Dans le cas contraire, les contraintes d'association entre registres adresses et index sont trop fortes.

Il s'agit donc de choisir des valeurs de déplacement, de telle façon que celles-ci soient utilisées très souvent au sein des ECs. On peut combiner les critères ici de façon à, par exemple, favoriser les déplacements correspondants aux constantes disponibles "en dur" dans l'architecture. A priori, l'algorithme choisi est itératif. Le critère d'arrêt peut-être un nombre d'itérations maximum, ou lié à l'obtention du nombre maximum de déplacements requis, ce dernier étant le critère adopté par défaut. Le critère indiquant qu'une liste d'IIEs de base est meilleure qu'une autre, est lié à l'obtention d'un plus petit nombre de déplacements différents. La recherche exhaustive de la meilleure solution est à vrai dire envisageable. En effet, le nombre de solutions possibles est égal au produit des cardinaux de tous les ensembles de connivence de la boucle traitée. Compte-tenu du petit nombre d'ECs en général, et du petit nombre d'éléments en moyenne dans chaque ensemble, la solution exhaustive est réaliste. Le problème se corse lorsque des déplacements non constants interviennent entre les IIEs. Une solution est donnée plus bas pour éradiquer ce problème, et forcer l'emploi de déplacements tous constants.

Partitionnement des ensembles

Dans l'hypothèse où le nombre de registres d'adresse disponibles pour la transformation est supérieur au strict minimum, il est possible de profiter de ce degré de liberté pour essayer d'améliorer les performances du programme. Sont explicitées ici deux techniques agissant par le biais d'un partitionnement des ensembles de connivence.

Forcer l'utilisation de déplacements constants dans les opérations d'in/décrémentation Il est possible que, dans un EC donné, la distance entre deux ou plusieurs IIEs soit une constante de boucle. Dans un tel cas, au cours de la transformation, un temporaire sera créé et placé dans le prologue (cf. sections 3.3.4.4 et 3.3.4.5). Cela peut s'avérer être un effet indésirable, particulièrement si la boucle traitée est une boucle interne, auquel cas une opération est rajoutée dans la boucle supérieure. Pour éviter cela, la technique suivante est appliquée :

1. Déterminer un cycle constant CC dans le graphe des distances. Un cycle constant est un sous-graphe du graphe des distances, tel que tous les nœuds aient entre eux une distance constante pure. Pour déterminer un tel sous graphe CC :

   (a) $CC = \{\emptyset\}$

   (b) Partir d'un nœud quelconque N_0: $CC = \{N_0\}$

   (c) Choisir le nœud suivant N_i tel que $dist(N_0, N_i) = constante\,pure$ et $N_i \notin CC$. Tant que c'est possible, itérer. Fin lorsque la seule possibilité est $N_i \in CC$

2. Créer un nouvel EC composé des IIEs correspondant aux nœuds de CC. Ces nœuds sont éliminés de l'EC initial.

3. Recommencer avec les nœuds restants de l'EC initial, tant que le degré de liberté existe, c'est-à-dire que le nombre de registres disponibles est inférieur au nombre d'ECs.

En guise d'exemple, la figure 3.19 représente un EC dont certaines des distances inter-IIE ne sont pas constantes. C est une constante de boucle. La partie a) de la figure met en évidence un premier partitionnement après détermination d'un cycle constant dans le graphe des distances. Le graphe des distances de l'EC, résultant de ce partitionnement, est lui-même un cycle constant.

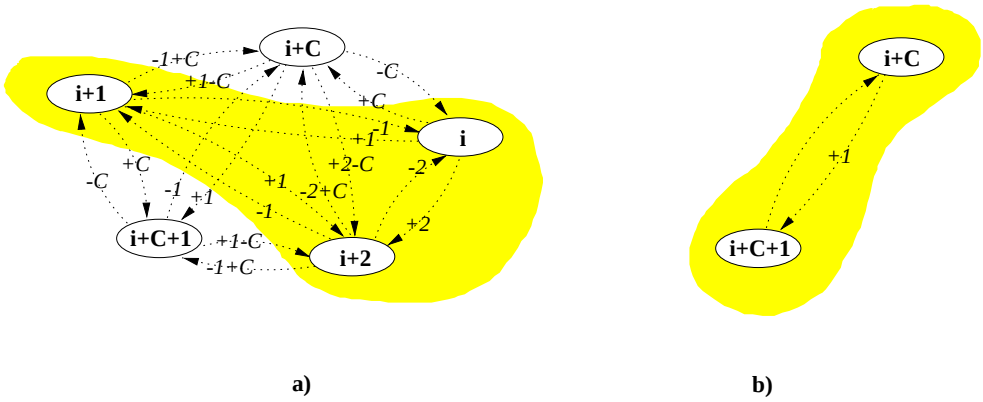


Figure 3.19: Partitionnement d'ensembles de connivence forçant l'utilisation de déplacements constants.

Cette procédure de partitionnement assure donc l'utilisation de déplacements purement constants, lors des opérations sur les pointeurs créés au cours de la transformation.

Favoriser l'utilisation d'opérations d'adressage à coût minimum En reprenant et étendant les travaux exposés dans [64], ainsi que [5], il est possible de favoriser l'emploi d'opérations d'adressage les moins coûteuses possible dans le cas des modes d'adressage post-modifiés, cela par le biais d'un partitionnement intelligent des ensembles de connivence. Dans cette notion de coût sont entendus à la fois le temps d'exécution et le parallélisme au niveau instruction.

L'heuristique détaillé dans [64] considère une boucle simple et bien formée, ou un ensemble de boucles de ce type imbriquées, chaque boucle n'ayant qu'une variable de boucle. Dans cette boucle, des références tableaux contenant des expressions d'induction indicielles simples se succèdent. L'objectif de l'algorithme est d'exploiter les capacités de la machine afin de maximiser l'emploi des opérations d'auto-in/décrémentation, considérées comme ayant un coût zéro. Seules des opérations d'adressage post-modifiées sont considérées. Les registres d'adresses sont alloués au cours de la transformation.

Il est possible d'envisager l'application de cet algorithme dans le présent environnement de transformation, en étendant celui-ci de façon à favoriser toutes les opérations à bas coût, mais sans pour autant aller forcément jusqu'à l'allocation des pointeurs, bien que cela reste une éventualité.

Partant d'un EC, de ses IIEs et du graphe des distances correspondant, il s'agit dans un premier temps de construire ce qui est appelé dans [64] l'ODG (Overall Distance Graph) ou graphe global des distances. C'est un graphe orienté acyclique reliant entre elles les IIEs. Un arc existe entre deux IIEs si elles peuvent partager un registre avec un coût minimum, compte-tenu des critères de coût retenus[14]. Nous lui préférerons par la suite la dénomination de graphe de partage minimum ou GPM. Pour construire ce graphe, la notion d'ordre d'exécution doit être introduite, puisque des modes d'adressage modifiants sont visés. En effet, un arc existera entre deux IIEs i et j si et seulement l'IIE i est exécutée plus tôt dans le code que l'IIE j. Cela se justifie par la signification que l'on souhaite voir prendre à chaque arc : un arc signifie en clair que l'adresse de j peut être générée par l'adresse de i à un coût minimum. Si dans le code initial, plusieurs références à un même tableau sont présentes dans une même expression, le besoin d'un ordre d'exécution impose de casser cette expression de façon à imposer un ordonnancement préalablement à l'application du partitionnement.

Afin d'optimiser le partitionnement, la solution de [64] tient compte des effets inter-itérations. Ainsi, pour construire le GPM, peut-on rajouter au graphe initial des IIEs virtuelles représentant l'état de chaque IIE de l'ensemble au cours de l'itération suivante. Cela est trivial dans le cas de boucles simples et bien formées, telles que manipulées par l'heuristique initiale, puisqu'il suffit de rajouter à chaque IIE son pas. Dans le cas présent, puisque les IIEs peuvent dépendre de plusieurs BIVs, et que chaque BIV peut être multi incrémentée, le problème se densifie. Pour simplifier, le rajout de l'effet inter-itérations est ici aussi limité aux ensembles comportant des IIEs mono-BIV et mono-incrémentées.

En guise d'exemple, considérons la séquence d'accès suivante aux éléments d'un tableau dans une boucle:

```
i=0;
while (i<99) {
  ...a[i+1]...I
  ...a[i]...  II
  ...a[i+2]...III
  ...a[i+3]...IV
  ...a[i]...  V
  i++;
}
```

La figure 3.20 montre le graphe des distances, et deux GPMs qui en sont issus, ainsi que leur partitionnement. Les IIEs grisées sont les IIEs virtuelles mentionnées plus haut, et correspondent à la valeur de l'IIE de même numéro au cours de l'itération suivante.

Le partitionnement se fait de la façon suivante :

[14]Dans l'algorithme initial, seuls sont retenus les modes d'adressage ayant un coût nul que sont les modes d'adressage auto-in/décrémentés.

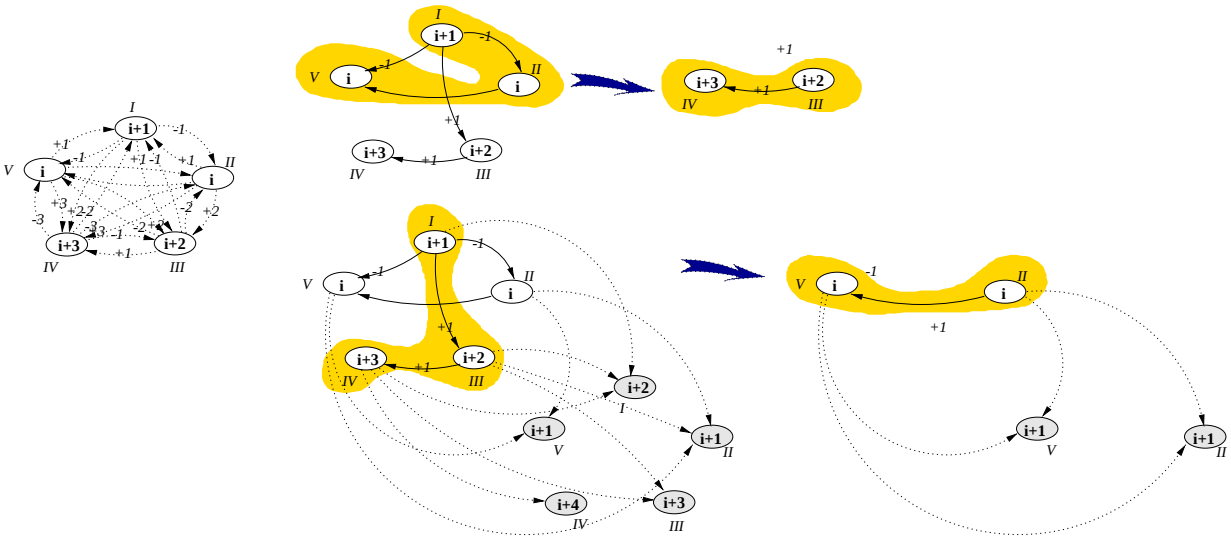


Figure 3.20: Exemple de partitionnement d'un ensemble de connivence favorisant l'emploi de modes d'adressage à bas coût

1. Calcul du plus long chemin dans le GPM. Plusieurs choix sont possibles : ils sont équivalents.

2. Formation d'un nouvel EC avec les éléments du chemin, et élimination de ces éléments du GPM avec les arcs qui en sont issus.

3. Si un nouveau partitionnement est possible, retour à 1.

Si l'on étudie le premier graphe de la figure 3.20, construit sans prendre en compte les effets inter-itération, un premier calcul donne deux chemins de longueur identique : $I \rightarrow II \rightarrow V$ et $I \rightarrow III \rightarrow IV$. Si l'on choisi le premier, les trois IIEs sont donc retirées du graphe, et le calcul recommence pour aboutir à la solution indiquée. En ce qui concerne le graphe augmenté des éléments virtuels, la recherche du plus long chemin possède la contrainte suivante : le dernier élément du chemin doit être la valeur du premier élément au cours de l'itération suivante[15]. Le seul choix possible est alors celui souligné, sachant que l'élément virtuel I participe à la mise en œuvre du partitionnement sans en faire partie lui-même. Le second calcul du plus long chemin est trivial. Au final, les deux partitionnements donnent respectivement (de gauche à droite) les résultats suivants:

[15]Cette contrainte est logique, puisque il faut préparer l'adresse de la première occurence du pointeur au cours de la prochaine itération. Cela ne peut se faire qu'à partir de la dernière occurence du pointeur au cours de l'itération courante.

```
i=0;
ptra1=&a[1];
ptra2=&a[2];
while (i<99) {
  ...*ptra1... I
  ptra1--;
  ...*ptra1... II
  ...*ptra2... III
/*IV et Prochaine itération*/
  ptra2++;
  ...*ptra2... IV
  ...*ptra1... V
  i++;
/*Prochaine itération*/
  ptra1+=2;
}
```

et

```
i=0;
ptra1=&a[1];
ptra2=&a[0];
while (i<99) {
  ...*ptra1...I
  ptra1++;
  ...*ptra2...II
  ...*ptra1...III
  ptra1++;
  ...*ptra1...IV
  ...*ptra2...V
  i++;
/*Prochaine itération*/
  ptra1--;
  ptra2++;
}
```

On pourra noter la présence d'une opération coûteuse en fin de première boucle. En revanche, la seconde boucle ne possède que des opérations d'adressage à coût nul : l'intérêt d'inclure les effets inter-itérations est ici manifeste. L'expérience montre qu'íl est tempéré si l'on considère d'autre modes que le mode auto-incrémenté comme modes bas-coûts.

Dans le cas de modes d'adressage non-modifiants, et toujours dans le but de favoriser les modes d'adressages à coût minimum, le GPM est toujours utile, mais ne traduit plus des contraintes d'ordonnancement des IIEs. Il s'agit dans ce cas d'un sous-graphe du graphe des distances, dont tous les arcs de poids supérieurs au poids autorisé sont éliminées. Si l'on prend à nouveau le même exemple illustratif, la figure 3.21 représente le procédé de partitionnement appliqué.

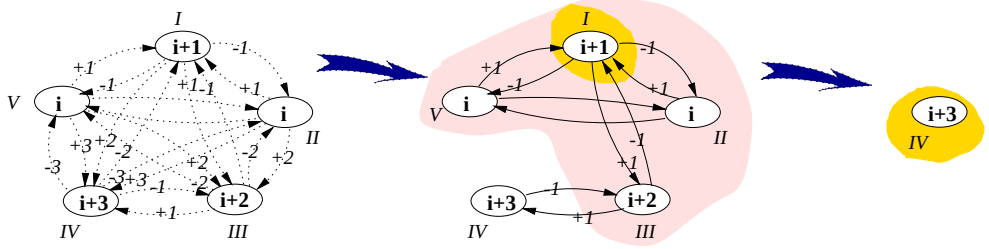


Figure 3.21: Graphe des distances et Graphe de Partage Minimum dans le cas de modes d'adressage non modifiants.

En considérant les seuls modes auto-in/décrémentés comme générateurs des coûts les plus bas, est construit le GPM comme illustré sur la figure 3.21, à partir du graphe des distances. Le partitionnement s'effectue alors comme suit :

1. Choisir le nœud du GPM ayant le plus grand nombre d'arcs sortants : ce nœud ainsi que tous les nœuds accessibles à partir du nœud choisi constituent un nouvel ensemble de connivence, dont l'IIE de référence est l'IIE choisie.

2. Eliminer ces nœuds du GPM ainsi que les arcs correspondants (entrants et sortants).

3. Reprendre au point 1 sauf si le nombre d'éléments restants est inférieur ou égal à un.

Pour l'exemple présent, I a 3 arcs sortants : il est donc choisi et le GPM résiduel ne se compose que de l'élément IV. Le code résultant est le suivant :

```
i=0;
ptra1=&a[1];
ptra2=&a[3];
while (i<99) {
  ...*ptra1...    I
  ...*(ptra1-1)...II
  ...*(ptra1+1)...III
  ...*ptra2...    IV
  ...*(ptra1-1)...V
  i++;
/*Prochaine itération*/
  ptra1++;
  ptra2++;
}
```

Un dernier détail : les procédures de partitionnement décrites dans cette partie s'appliquent dans la mesure où le nombre de registres disponibles est supérieur au nombre d'ECs. Ce degré de liberté dans la génération de nouveaux ensembles doit être, bien entendu, mis à jour à chaque partitionnement nouveau. Il constitue un critère d'arrêt absolu.

3.3.6.2 Assignation des ressources d'adressage

Comme déjà précisé plus haut, il ne s'agit pas ici de se suppléer aux étapes de compilation dorsales, ou seulement dans une faible mesure, en gardant à l'esprit la modestie nécessaire face à l'ampleur d'une telle entreprise. Les objectifs sont de tester la faisabilité relative de la transformation, et de transformer le code-source en fonction des ressources disponibles. Le problème principal rencontré au cours de l'assignation des ressources, réside dans l'absence éventuelle d'orthogonalité entre celles-ci. En effet, il est parfois des contraintes architecturales autorisant par exemple tel registre d'adresse à utiliser telle constante ou tel registre d'index comme déplacement, mais pas tel ou tel autre.

L'approche choisie pour assigner les ressources d'adressage aux opérations au cours de la transformation se fait à la volée, c'est à dire au cours de la transformation et de la création des pointeurs. Afin d'améliorer cette étape d'assignation, interviennent au préalable deux étapes d'ordonnancement qui vont agir sur les ensembles de connivence et sur la représentation des ressources machine.

1. Les ensembles de connivence sont classés par ordre décroissant de cardinaux, de façon à traiter en priorité les ensembles comportant le plus d'éléments

2. Les registres d'adresse sont classés par ordre décroissant de degré de liberté en terme de déplacement. C'est-à-dire que pour répondre à l'hétérogénéïté des ressources, les registres permettant le plus grand nombre de déplacement différents (par constante, immédiat ou registre d'index), seront alloués en priorité.

C'est ainsi qu'en combinant les avantages de ces deux ordonnancements préliminaires, les ECs les plus touffus sont associés aux registres les plus aptes. L'assignation se fait en parallèle avec la transformation. A chaque EC traité, un registre d'adresse lui est assigné à la volée, puis à chaque déplacement une unité physique de déplacement est assignée, et en priorité les constantes câblées puis les immédiats et enfin les registres d'index. Ces priorités entre unités sont évidemment modulables en fonction du coût associé à chaque opération d'adressage.

Assigner un déplacement à une constante disponible est trivial. Assigner un immédiat n'est pas non plus d'une rare complexité, si ce n'est que l'intervalle de valeurs possibles doit être pris en considération. Dans le cas des registres d'index (RI), une liste de paires *(valeur, RI)* est mise à jour chaque fois qu'une valeur est assignée à un registre d'index. C'est ainsi qu'avant de rechercher un registre d'index disponible afin de lui associer la valeur courante, un parcours de cette liste est effectué, afin de maximiser la réutilisation de valeurs déjà stockées préalablement. Il faut bien sûr s'assurer que le registre courant puisse utiliser cette valeur, dans la mesure où cette liste de paires est commune à tous les ensembles de connivence, donc unique pour chaque corps de boucle rencontré.

3.4 Transformations connexes

Transformer les références tableaux en références pointeurs, outre son intérêt propre, ouvre la porte à de nombreuses autres optimisations, cela directement et indirectement. L'influence directe se manifeste tout d'abord dans les possibilités d'élimination de code, consécutives à la transformation. Certaines variables d'induction deviennent inutiles, accompagnées de leurs opérations d'in/décrémentation. Certaines transformations peuvent améliorer les performances de la transformation de tableaux en pointeurs, mais présentent elles-mêmes un intérêt qui leur est propre : c'est le cas de l'exploitation du parallélisme entre BIVs, de la fusion de tableaux ou encore du placement en mémoire des tableaux. L'influence indirecte est liée à la somme de connaissances et d'informations sur les secrets d'une boucle que distille la transformation de tableaux en pointeurs. Cette connaissance peut-être exploitée par l'utilisation plus judicieuse de ressources machines comme les boucles matérielles, ou l'application plus aisée du déroulage de boucle. Certaines des transformations citées sont décrites au cours de cette section.

3.4.1 Élimination des variables d'induction inutiles

A l'issue de la transformation qui sous-tend ces travaux de thèse, le code généré comporte, outre les nouveaux pointeurs, toutes les opérations ayant trait aux BIVs et IVs qui se trouvaient dans le code initial. Certaines de ces opérations (voire toutes) sont désormais inutiles. Un moyen de s'en assurer est d'appliquer sur le nouveau corps de boucle une passe de nettoyage, qui consiste à s'assurer que les BIVs et IVs définies sont utilisées. Si ce n'est pas le cas, tout ce qui se réfère à celles-ci dans la boucle peut être éliminer, c'est-à-dire principalement les opérations d'in/décrémentation.

Cette passe se base sur l'analyse flot de donnée consistant à déterminer à partir d'une définition de variable, quelles sont ses utilisations accessibles. Il s'agit du pendant de l'analyse des définitions accessibles (exposée en détail dans l'annexe A. Grâce à cette analyse, il sera possible de dire si une BIV ou IV est, oui ou non, potentiellement utilisée, et d'ainsi statuer sur son sort.

Les BIVs utilisées en tant que compteur de boucle ne peuvent être supprimées au cours de cette étape. L'exploitation d'un mécanisme de boucle matérielle présent dans le processeur cible pourra y remédier.

3.4.2 Détection du parallélisme entre variables d'induction de base

Le cas des variables d'induction parallèles avait été évoqué dans la section 3.3.4.4p.67. L'exemple utilisé comportait deux variables d'induction de base i et j, évoluant tout au long des itérations de la boucle de façon concurrente. Dans un tel cas, les étapes d'optimisation suivantes peuvent être appliquées :

1. Extraction des BIVs de la boucle, ainsi que des IVs et IEs.
2. Si deux BIVs ou plus ont le même pas P :
   (a) Choisir la BIV B à conserver.
   (b) Pour chaque BIV Bj à remplacer :
      i. Remplacer chaque occurrence de Bj par $B + B_{init} - Bj_{init}$ dans les IEs situées avant in/décrémentation de Bj.
      ii. Remplacer chaque occurrence de Bj par $B + B_{init} - Bj_{init} + P$ dans les IEs situées après in/décrémentation de Bj.
      iii. Supprimer l'in/décrémentation de Bj dans la boucle.

Outre l'intérêt qu'il peut y avoir à supprimer une variable d'induction dans la boucle, ainsi que son opération d'in/décrémentation, il y a un avantage particulier à effectuer cette optimisation juste avant la transformation de tableaux en pointeurs. En effet, supprimer une BIV implique la possible suppression d'un ou plusieurs ensembles de connivence dont les éléments, autrefois de la famille de l'ex BIV, se trouvent intégrés à d'autres ensembles. Au niveau machine, la pression sur les registres d'adresse s'en trouve donc moins forte.

3.4.3 Exploitation des boucles matérielles

Le mécanisme de boucle matérielle[16] permet d'exécuter une boucle de façon très efficace. Cet artifice matériel exécute une ou plusieurs instructions un nombre fini de fois, comme le ferait une boucle, à ceci près que le compteur de boucle est implicite, de même que le

[16]Hardware do-loop ou zero-overhead looping mecanism

test de sortie et les branchements associés, tous gérés par la machine directement. De façon pratique, une boucle matérielle est souvent implémentée à l'aide de trois registres dédiés : LB, LE et LC pour Loop Begin, Loop End et Loop Counter. Quand une boucle est exécutée, ces registres contiennent respectivement l'adresse du début de la séquence d'instruction qui doit être répétée, l'adresse de la fin de cette séquence et le compteur de la boucle. Un exemple d'un tel mécanisme se trouve dans le processeur D950 de STMicroelectronics, et de façon générale dans tout DSP digne de ce nom, y compris les cœurs de DSP spécifiques ou ASDSP. On peut y distinguer trois types de boucles matérielles : le plus simple consiste à répéter un nombre i constant de fois une instruction (cf. 3.22.a. Le second permet de boucler un nombre i constant de fois sur un ensemble d'instructions (cf. 3.22.b.

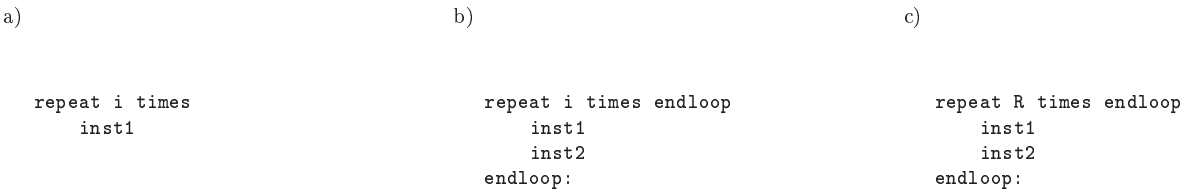


Figure 3.22: Types de mécanismes de boucles matérielles

Le troisième permet de répéter un nombre R de fois une séquence d'instructions, avec R registre (cf. 3.22.c). L'instruction assembleur `repeat` gère :

- l'incrémentation du pointer de banc de boucle (3 niveaux d'imbrication sont autorisés);
- le chargement de l'adresse de départ de la séquence dans LS;
- le chargement du nombre d'itérations dans LC;
- le chargement de l'adresse de fin de la séquence dans LE.

Compiler une boucle logicielle automatiquement sur une boucle matérielle n'est pas trivial, dans la mesure où la boucle logicielle doit satisfaire à plusieurs critères dépendant de la machine cible, et du compilateur de celle-ci. On peut citer par exemple :

- **Etre une boucle dont le nombre d'itérations est connu au moment de la compilation**[17] : c'est le cas lorsque le processeur n'autorise qu'une opération du type "repeat(n)", où n est une constante dont la valeur est connue au moment de la compilation, comme c'est le cas pour le Sapphire par exemple. Cependant, certains processeurs autorisent une boucle matérielle dont le nombre d'itérations est contenu dans un registre, ce qui autorise les boucles logicielles dont le nombre d'itérations n'est connu qu'au moment de l'exécution (execution time counting loops) à être transposées sur des boucles matérielles. C'est le cas pour le D950 et pour le MMDSP.

- **Ne pas contenir de ruptures de séquences** : cette condition ne tient plus si le jeu d'instructions autorise la modification dynamique du contenu des registres LE et LC. Le D950 contient des mécanismes spéciaux permettant de prendre en compte les ruptures de séquence de type *break* et *continue*. Quand un *break* est rencontré par exemple, la profondeur d'imbrication est simplement décrémentée tandis que le compteur de programme est positionné à LE+1 et LC à 1, ce qui signifie que la dernière itération à été effectuée. Dans le cas d'un *continue*, le compteur de programme est positionné à LE et une nouvelle itération commence.

- **Avoir un nombre limité ou nul de boucles imbriquées** : un processeur peut imposer un nombre limité d'imbrications, chaque niveau d'imbrication correspondant à une série de registres LB, LE et LC, avec un registre supplémentaire en guise de pointeur de niveau. Le D950 autorise un degré 3 d'imbrication, avec 3 bancs de registres dédiés. Il en va de même pour le MMDSP, alors que le Sapphire ne supporte pas d'imbrications. Il est cependant possible que, dans le cas d'un nombre d'imbrications supérieur au nombre autorisé, une sauvegarde du contexte de la boucle externe soit possible, avec restauration à la clé lorsque nécessaire.

Pour transformer un code source C de façon à imposer dès ce niveau l'utilisation de boucles matérielles, il faut avant tout que le compilateur dorsal accepte une écriture spéciale lui indiquant comment agir, comme le montre l'exemple 3.23.

```
ptrb = b;                                          →        ptrb = b;
ptra = a;                                                   ptra = a;
for (i = 0;  i < 99;  i++) {                                loop(100) {
   *ptrb = *ptra + *(ptra+1);                                  *ptrb = *ptra + *(ptra+1);
   ptrb++;                                                     ptrb++;
   ptra++;                                                     ptra++;
}                                                           }
```

Figure 3.23: Ecriture source permettant l'exploitation de boucles matérielles

Il faut ensuite déterminer les boucles candidates, et notamment le nombre d'itérations de ces boucles, qui peut être un nombre constant ou non. Pour une boucle for bien formée, c'est-à-dire de la forme de l'exemple 3.23, c'est relativement trivial. Cela le devient moins pour des boucles du type *while*() ou *do-while*(), pour lesquelles il faut d'abord trouver la BIV de comptage de la boucle, avant de pouvoir déterminer combien de fois elle s'exécute, si c'est seulement possible dans des limites de faisabilité algorithmiquement acceptables.

Les étapes suivantes de transformation peuvent donc être extraites des quelques points précédents :

1. Analyse et extraction des propriétés des boucles, et principalement des BIVs.

2. Identification des boucles candidates à l'exécution sur un mécanisme de boucle matérielle, compte-tenu des possibilités de la machine cible. Cela impose d'utiliser les informations de flot de données d'utilisation accessible comme dans le cas de l'élimination de

[17] compile time counting loops

variables d'inductions. En effet, cette transformation revient, entre autres, à éliminer la variable d'induction de comptage d'itérations. Si cette variable est utilisée dans un autre but au sein de la boucle, il faut préserver son initialisation (insérée dans le prologue de la boucle), et son incrémentation (insérée en fin de corps de boucle) dans le cas d'une boucle for.

3. Transformation du code source compte-tenu des possibilités d'écriture autorisées.

Une implémentation préliminaire de cette transformation, combinée avec la transformation de tableaux en pointeurs, donne des résultats particulièrement intéressants en termes de performance. Ces résultats sont indiqués dans le chapitre suivant.

3.4.4 Fusion de tableaux

Au cours de la section 3.3.3.2, ainsi qu'en abordant la construction des ensembles de connivence, a été posé le problème de l'existence potentielle d'IIEs évoluant en parallèle au cours de l'exécution de la boucle, mais dépendant de tableaux différents, donc n'appartenant pas à priori aux mêmes ECs, sauf dans le cas où la distance entre les tableaux pouvait être calculée sans danger.

Une autre solution, la fusion de tableaux, est propre à résoudre ce problème. Soit la boucle suivante :

```
int a[100];
int b[100];
int i;
for(i=0;  i<98; i++) {
  b[i] = a[i] + a[i+1];
}
```

Deux ensembles de connivence seront extraits de l'analyse préliminaire. Supposons maintenant que l'on déclare le tableau F, tel que F soit la résultante de la fusion, c'est-à-dire de la concaténation, des tableaux a et b. Alors la précédente boucle peut se réécrire :

```
int F[200];
int i;
for(i=0;  i<N-1; i++) {
  F[i+100] = F[i] + F[i+1];
}
```

Après transformation, un seul pointeur est à présent nécessaire, sans pour cela avoir à se poser la question de la faisabilité du calcul des distances entre les tableaux, une fois le code compilé sur la machine cible. Le résultat est une diminution de la pression sur les registres d'adresse dans la boucle.

Il y a des contraintes, ainsi que divers désavantages à cette transformation. Une contrainte forte est d'avoir accès au domaine de visibilité des tableaux à concaténer. En clair, étant donné que ces tableaux sont supprimés après transformation, il faut remplacer toute

référence existante à ceux-ci par une référence au nouveau tableau, et cela dans le programme entier dans le cas où ces tableaux sont globaux. Il faut alors s'assurer que le programme dans son ensemble est accessible. Par bonheur, dans le domaine des applications embarquées, il est courant de compiler le programme dans son ensemble, et non modulairement. Mais cela demeure une contrainte bloquante. Une alternative est la création locale du nouveau tableau, tout en maintenant l'existence globale des tableaux fusionnés localement. Sachant que la mémoire n'est pas gratuite, il y a là un compromis à trouver. Un désavantage à cette transformation résulte du placement en mémoire possible des tableaux initiaux, dans le cas d'une machine possédant plusieurs mémoire et des possibilités d'accès parallèles à celles-ci. On peut imaginer que le placement des tableaux dans des mémoires différentes, favorise en divers point du programme le parallélisme à l'exécution, ce que la fusion des tableaux annihile.

3.4.5 Transformation de séquences de tableaux en pointeurs

Dans certains programmes applicatifs, et notamment dans le cas de l'écriture de filtres simples comportant un petit nombre de cœfficients, il est parfois plus coûteux d'employer une boucle que de dérouler intégralement les opérations. On pourra par exemple trouver une écriture du style :

```
W = X - w[0]*coeff[0] - w[1]*coeff[1];
Y = W + w[0]*coeff[2] + w[1]*coeff[3];
w[0] = w[1];
w[1] = W;
```

Transformer ces références à des tableaux avec indices constants, en références pointeurs peut s'avérer efficace; en voici un exemple :

```
ptrw = w;
ptrcoeff = coeff;
tmp1=(*ptrw)*(*ptrcoeff);
ptrw++;
ptrcoeff++;
W = X - tmp1 - (*ptrw)*(*ptrcoeff);
ptrw--;
ptrcoeff++;
tmp1 = (*ptrw)*(*ptrcoeff);
ptrw++;
ptrcoeff++;
Y = W + tmp1 + (*ptrw)*(*ptrcoeff);
tmp1 = (*ptrw);
ptrw--;
(*ptrw) = tmp1;
ptrw++;
(*ptrw) = W;
```

Cette écriture est, certes, bien moins lisible, mais met en évidence plus de parallélisme en raison de l'exécution parallèle des incrémentations de pointeurs, des accès-mémoire et opérations arithmétiques. Un procédé très similaire à celui employé pour transformer les tableaux dont les indices sont des expressions affines de variables d'induction, peut permettre d'automatiser une telle transformation. La même notion de distance entre références-tableaux intervient, avec les mêmes modalités d'exploitation.

3.5 En résumé

Ce chapitre s'est essentiellement focalisé sur la description d'une transformation de tableaux en pointeurs dans des boucles de type traitement du signal. Cette transformation, vecteur principal des efforts de ces travaux de thèse, permet de mieux exploiter les ressources d'adressage disponibles au sein de l'architecture cible. Elle s'appuie sur une description annexe de ces ressources, et permet d'explorer l'espace de solution d'une façon plus approfondie, si l'on considère les travaux antérieurs sur ce sujet. En outre, cette transformation ouvre la porte à d'autres optimisations, dont l'application conjointe peut être d'une grande valeur ajoutée. L'exploitation des boucles matérielles présente, en particulier, un très fort potentiel. Le chapitre suivant présente le résultat d'expérimentations, effectuées afin de mesurer l'impact réel de ces transformations sur l'efficacité du code généré.

Chapitre 4

Résultats d'expérimentations

4.1 Préambule

Ce chapitre se propose de faire le bilan quantitatif des expérimentations menées dans le domaine de la compilation, au cours de ces travaux. Le contexte de réalisation du prototype de transformation, *arT*, ainsi que des expérimentations, est tout d'abord décrit. Suivent les exposés de résultats obtenus sur un certain nombre de programmes, essentiellement composés d'une ou plusieurs boucles. Les résultats obtenus sont ensuite comparés avec ceux associés au prototype précédemment réalisé au sein de l'équipe ayant supporté ces travaux, *ArrSyn*. Finalement, sont présentés les performances obtenues par l'application conjointe de arT et de l'exploitation de boucles matérielles, ciblant l'un des processeurs.

4.2 Contexte de développement et d'expérimentation

4.2.1 Prototype de transformation

Le prototype d'optimisation de code matérialisant les travaux réalisés au cours de cette thèse, baptisé *arT*, a été développé comme un programme indépendant, représentant environ 11000 lignes de code, et écrit en C++. Il s'applique comme un module frontal à un compilateur existant, accompagnant un processeur embarqué. Ce prototype se compose essentiellement de trois modules : le premier est dédié à la gestion des commandes et contient le parser de la description architecturale, le second prend en charge l'analyse des boucles du programmes, et s'occupe de récolter et construire les données utiles, enfin, le troisième orchestre les étapes de transformation et le parcours des fichiers, procédures et boucles du programme à transformer.

Les commandes acceptées par l'outil permettent de diriger la transformation, et autorisent un réel parcours de l'espace des solutions. Ces commandes comprennent le forçage de la classe d'adressage utilisée, la gestion partielle de la manipulation des ensembles de connivence, et l'indication du fichier de spécification.

SUIF[1], développé à l'université de Stanford, est la représentation intermédiaire (RI) choisie [40], sur laquelle l'analyse et la transformation du programme source s'appliquent. SUIF est plus précisément un environnement de développement de compilateurs, dédié à la recherche et à l'expérimentation de nouvelles optimisations. Il comporte un certain nombre de modules, dont un module frontal, générant la représentation intermédiaire de haut niveau, préservant de façon quasi complète la structure et les informations de haut niveau du code source. Il comporte par ailleurs un module dorsal, permettant de regénérer un programme C après transformation. Cet environnement propose essentiellement une librairie, autorisant la manipulation de la représentation intermédiaire de façon plus ou moins transparente.

4.2.2 Architectures cibles

Deux processeurs développés à STMicroelectronics, accompagnés de leur chaîne de compilation respective, sont les supports des expérimentations relatées ici. Il s'agit de processeurs DSP spécifiques, ou ASDSP, déjà partiellement décrits en termes de ressources d'adressage dans la section relative aux informations machines.

MMDSP Ce processeur est décrit dans de plus amples détails dans [72]; c'est un processeur conçu au sein de ST. C'est un ASDSP, conçu pour le traitement audio haute fidélité, incluant la décompression et le décodage MPEG2, Dolby-AC3 et Dolby Pro-logic. Ce processeur est disponible sous forme de cœur intégrable, utilisé dans des applications telles que DVD, PC multimédia, décodeurs satellite. L'architecture est celle d'un petit VLIW, Harvard, registre-registre, comportant une unité arithmétique et logique (ALU) comprenant le nécessaire MAC, et une unité de calcul d'adresses (ACU), très riche. Deux mémoires-données : une ROM principalement réservée aux cœfficients constants, utiles aux filtres de traitement du signal, et une RAM dédiée aux données intermédiaires de calcul. Trois bancs de registres sont spécialisés dans la gestion matérielle des boucles de l'application, avec une possibilité d'imbrication de 3 boucles. Le parallélisme au niveau instruction se manifeste entre opération arithmétique, calcul d'adresse et chargement de valeurs en mémoire.

Sapphire Le Sapphire est un ASDSP, mis au point par l'équipe EPT de STMicroelectronics. Ce processeur, conçu historiquement pour être embarqué dans des puces dédiées aux applications audio, a su évoluer et trouver d'autres applications autour de la téléphonie et du stockage de données. La raison de sa pluridisciplinarité revient à sa conception, très simple, mais point trop spécialisée. Son architecture ressemble à celle d'un DSP classique, permettant de réaliser un MAC, et comportant deux mémoires données. A chacune des mémoires sont associés un petit banc de registres données (2 registres), et une unité de calcul d'adresse comportant un petit banc de registres d'adresse (2 registres). Un parallélisme au niveau instruction, entre unités de calcul d'adresse, multiplieur et ALU, permet d'obtenir

[1]Stanford University Intermediate Format

des performances appréciables. La partie contrôle gère un mécanisme de boucle matérielle simple et de branchement avec délai, accroissant l'efficacité de l'architecture. Les unités de calcul d'adresses sont suffisamment étoffées pour supporter les modes d'adressage directs, indirect indexés, indirect post-modifiés avec constante et registre d'index, ainsi que modulo.

4.2.3 Protocole d'expérimentation

La figure 4.1 résume simplement le protocole employé afin d'obtenir les résultats présentés ci-après. Le code source C original est tout d'abord transformé, puis compilé ce qui permet d'obtenir à la fois la taille du code généré, en terme de nombre d'instructions, le taux de parallélisme, et enfin les performances ou temps d'exécution du code en terme de nombre de cycles. Pour cette dernière mesure, des simulateurs faisant partie de l'environnement de développement associé à chacun des processeurs sont employés. Le simulateur du MMDSP est précis au niveau cycle d'horloge, tandis que celui du Sapphire est précis au niveau instruction, ce qui conduit en l'occurrence à une équivalence de précision au niveau cycle, compte tenu du pipeline simple de ce processeur.

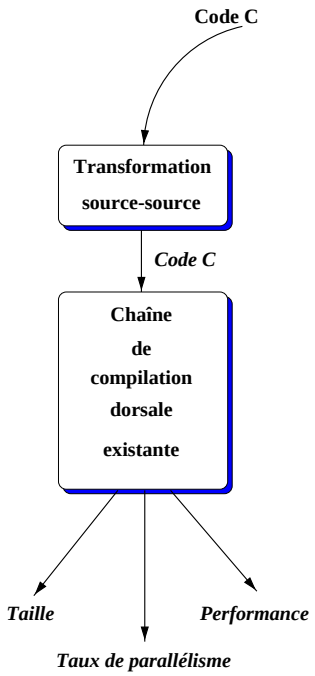


Figure 4.1: Protocole d'expérimentation

Le taux de parallélisme est déterminé en comptant simplement le nombre d'instructions présentant du parallélisme, et en le comparant au nombre d'instructions total.

4.2.4 Programmes de test

Le jeu de programmes de test élaboré en vue d'une expérimentation comporte de petits programmes, composés essentiellement de boucles. Certains de ces programmes sont des filtres ou des algorithmes que l'on retrouve fréquemment en traitement du signal : on pourra noter une multiplication de vecteurs (*vectmpy*), de matrices (*matrixmpy*), un produit de convolution (*convolution*), divers programmes de types DSP, nombre d'entre eux issus de [65] (*mac, latsynth, fir, iir, firnl, vocoder, median, interp1, interp2, addnoise*). D'autres programmes présentent certains aspects spécifiques, afin de mettre en évidence le traitement de cas extrèmes. C'est le cas des programmes *exp** et *nest**, ces derniers mettant en jeu des boucles imbriquées parfois complexes.

4.3 Application de la transformation tableaux-pointeurs

4.3.1 Tableaux de résultats

Le prototype mettant en œuvre la transformation de tableaux, a été appliqué sur un ensemble de programmes essentiellement constitués de boucles. Les résultats issus de ces expérimentations, distingués en fonction du processeur cible, sont exposés ci-après sous forme de tableaux.

4.3.1.1 Processeur MMDSP - compilateur mmdspcc

Exceptées quelques applications, pour lesquelles aucune amélioration n'est manifeste, la table 4.1 montre un gain en taille de code parfois conséquent (25%) après application de la transformation automatique de tableaux en pointeurs. Le gain le plus évident reste en termes de performances. L'application *nest3* donne les meilleurs résultats avec 63% de gain en performance. Cette application contient deux boucles imbriquées et une expression complexe mettant en jeu des tableaux bi-dimensionnels. D'autres applications comme *matrixmpy* et *nest2*, contenant elles aussi des boucles imbriquées, présentent un gain élevé. La plupart de ces résultats sont obtenus en transformant le source original dans un mode exploitant les ressources d'adressage post-modifiées de l'architecture. Dans certains cas, par exemple *exp1*, les résultats sont meilleurs en forçant la transformation à assigner un pointeur à chacune des références tableaux, ce qui est autorisé par le prototype implémenté.

4.3.1.2 Processeur Sapphire - compilateur sapphirecc

Le Sapphire est un ASDSP dont les ressources sont peu nombreuses et dont les possibilités de parallélisme sont bien moins élevées que pour le MMDSP. Les résultats en taille de code notamment s'en ressentent, puisque l'on peut constater une augmentation de la taille du code après transformation dans de nombreux cas. Sans atteindre les pourcentages élevés constatés dans le cas du MMDSP, les gains en vitesse d'exécution restent bons. Dans deux cas, *nest5* et *vectmpy*, le gain est très faible voire légèrement négatif, et la transformation est

Applications	Taille du code original	*arT*	Gain	Nombre de cycles original	*arT*	Gain
simple_loop	11	10	9.1%	42	29	31%
median	49	47	4.1%	358	291	18.7%
interp1	30	20	33.3%	423	203	52%
interp2	20	17	15%	218	176	19.3%
addnoise	30	27	10%	613	494	19.4%
exp0	19	17	10.5%	1212	913	24.7%
exp1	19	17	10.5%	100	71	29%
exp2	57	43	24.6%	508	316	37.8%
exp3	73	63	13.7%	473	336	29%
nest1	23	19	17.4%	582	383	34.2%
nest2	24	18	25%	5288	2488	52.9%
nest3	34	34	0%	1639	607	**63%**
nest4	24	23	4.2%	16928	11019	34.9%
nest5	34	29	14.7%	13327	9249	30.6%
convolution	27	25	7.41%	60214	45265	24.8%
vectmpy	22	20	9.1%	1815	1366	24.7%
matrixmpy	39	30	23.1%	17949	8571	52.3%
mac	26	26	0%	1968	1520	22.8%
latsynth	40	36	10%	1711	1315	22.4%
iir	44	44	0%	1322	1077	18.5%
fir	30	26	13.3%	32815	22816	30.5%
firnl	48	38	20.8%	19866	12566	36.8%
vocoder	47	41	12.8%	331	270	18.4%
	Moyenne		12.5%	Moyenne		31.6%

Tableau 4.1: Comparaison de la taille de code et du nombre de cycles d'exécution avant et après application de la transformation tableaux-pointeurs pour le processeur MMDSP

contre-performante puisque la taille du code augmente. Les faibles ressources du processeur sont là encore en cause.

Contrairement au cas du MMDSP, pour lequel seuls des modes d'adressage post-modifiés sont disponibles, un certain nombre d'applications donnent pour le Sapphire de meilleurs résultats en transformant le code de façon à exploiter le mode basé à bas coût disponible dans ce processeur. C'est le cas pour les applications *nest5*, *latsynth* et *exp1*. Dans la majeure partie des autres cas, les transformations favorisant l'emploi des deux classes d'adressage (modifiante et non-modifiante) conduisent aux mêmes résultats. Quelques exceptions cependant, comme les applications *nest2*, *interp1* ou *addnoise*, qui donnent les meilleurs résultats avec un mode post-modifié.

A noter par ailleurs que certaines applications ne compilent pas après transformation,

Applications	Taille du code original	*arT*	Gain	Nombre de cycles original	*arT*	Gain
simple_loop	17	13	23.5%	97	69	28.9%
median	74	74	0%	755	606	19.7%
interp1	60	49	18.3%	972	616	36.6%
addnoise	40	33	17.5%	1067	868	18.7%
exp0	31	31	0%	2522	2225	11.8%
exp1	31	17	45.2%	236	117	50.4%
exp2	129	127	1.5%	1196	1064	11%
exp3	160	153	4.4%	1183	1059	10.5%
nest1	35	39	-11.4%	1214	813	33%
nest2	46	37	19.6%	11116	5559	50%
nest3	79	87	-10.1%	4389	2199	49.9%
nest4	58	71	-22.4%	46096	41690	9.6%
nest5	70	83	-18.6%	30893	30497	1.3%
convolution	57	65	-14%	150923	117377	22.2%
vectmpy	33	37	-12.1%	4072	4076	-0.1%
matrixmpy	84	94	-11.90%	52413	36007	31.3%
mac	60	56	6.7%	5738	4840	15.7%
latsynth	93	100	-7.5%	4323	3840	11.2%
iir	120	99	17.5%	5134	3643	29%
fir	57	62	-8.8%	71123	53929	24.2%
firnl	120	115	4.2%	58173	43079	26%
	Moyenne		2%	Moyenne		23.4%

Tableau 4.2: Comparaison de la taille de code et du nombre de cycles d'exécution avant et après application de la transformation tableaux-pointeurs pour le processeur Sapphire

à cause du petit nombre de registres d'adresses disponibles et de l'impossibilité dans certain cas de pratiquer une sauvegarde mémoire pour libérer des registres[2]. Dans ce cas, l'alternative consiste à ne transformer le code que partiellement, c'est-à-dire à ne transformer que certaines références tableaux jusqu'à ce que le nombre de pointeurs utilisés égale le nombre de registres disponibles, transformation partielle qu'autorise le prototype réalisé. C'est le cas des applications *exp2* et *exp3* par exemple, qui présentent malgré tout, une vitesse d'exécution supérieure de respectivement 11% et 10.5% après transformation, avec un gain faible en taille de code.

[2]on parle plus couramment de la technique du spilling

4.3.2 Parallélisme au niveau instruction

Une raison des gains à la fois en taille et surtout en performance, outre le nombre d'opérations réduit dans les corps de boucle après transformation, tient dans une exploitation plus poussée par le compilateur du parallélisme au niveau instruction, après transformation. Le processeur MMDSP offrant beaucoup plus d'opportunités de parallélisme que le Sapphire, c'est surtout pour le premier que l'effet du gain en termes d'ILP se fait sentir. Cela est illustré sur la table 4.3 .

	ILP du code		
Applications	original	transformé	Gain
simple_loop	33.3%	66.7%	×2
median	24%	28.6%	×1.2
interp1	34.8%	66.7%	×1.9
interp2	26.7%	30%	×1.1
addnoise	22.2%	40%	×1.8
exp0	0%	25%	×∞...
exp1	30%	42.9%	×1.4
exp2	16.7%	53.9%	×3.2
exp3	17.2%	34.8%	×2
nest1	14.3%	50%	×3.5
nest2	12.5%	33.3%	×2.7
nest3	21.4%	47%	×2.2
nest4	41.2%	33%	×0.8
nest5	21.4%	42.9%	×2
convolution	36.8%	53.3%	×1.5
vectmpy	27.3%	37.5%	×1.4
matrixmpy	34.5%	55.6%	×1.6
mac	25%	33.3%	×1.3
latsynth	25%	50%	×2
iir	40%	66.7%	×1.7
fir	27.8%	42.9%	×1.5
firnl	41.7%	47.8%	×1.2
vocoder	43.5%	20%	×0.5
Moyenne			×1.75

Tableau 4.3: Taux de parallélisme au niveau instruction après transformation pour le processeur MMDSP.

On constate donc pour le MMDSP un gain en parallélisme niveau instruction frisant le plus souvent les 50%. Quelques applications voient néanmoins leur ILP baisser après transformation comme *vocoder*, ou *nest4*.

4.3.3 Cas particulier des boucles imbriquées

Cette section reprend le point soulevé section 3.3.4.6 page 70, et se posant de façon similaire section 3.3.4.7 page 71. En résumé, l'heuristique appliquée sur des boucles imbriquées nécessite l'emploi d'un nombre supérieur de pointeurs, comparé à une écriture manuelle. Cela provient du traitement de chaque boucle imbriquée, en partant de la boucle la plus interne, d'une façon indépendante (ou presque) de l'existence de boucles externes. Le problème est solvable algorithmiquement, principalement en communicant des informations à la boucle immédiatement externe à la boucle transformée. Néanmoins, les quelques résultats suivants montrent que le gain de ce raffinement supplémentaire est léger, voire nul ou négatif suivant l'application.

Sur le tableau 4.2, sont groupées un certain nombre de boucles imbriquées extraites de l'ensemble d'applications, support des expérimentations. De façon plus précise, et afin d'avoir un support de comparaison, l'application *matrixmpy* est celle utilisé à des fins d'exemple section 3.3.4.7. L'application *nest4* est l'exemple de la section 3.3.4.6.

	taille			Performances		
Applications	obtenue	"idéale"	gain	obtenue	"idéale"	gain
nest2	18	18	0%	2489	2489	0%
nest4	23	23	0%	11019	11289	-2.5%
nest5	29	30	-3.5%	9249	8529	7.8%
convolution	25	27	-8%	45265	60214	-33%
matrixmpy	30	25	20%	8571	8649	-0.9%

Figure 4.2: Comparaison entre la transformation manuelle et la transformation automatique dans le cas de boucles imbriquées pour le processeur MMDSP

Sur ce tableau, seul *nest5* présente une meilleure performance de l'écriture manuelle, tandis que *matrixmpy* transformée manuellement et usant de 3 pointeurs au lieu de 6, présente une taille de code généré moindre, mais aucun gain en performance. Une précision sur *matrixmpy* : ce programme est extrait de [30] qui donne une écriture manuelle rappelée section 3.3.4.7. Cette écriture donne une performance en nombre de cycles égale à 11768, ce qui correspond à une contre-performance de -37 % par rapport aux performances obtenues avec *arT*.

4.3.4 Comparaison avec les travaux antérieurs

Les travaux précédemment effectués au sein de l'équipe, détaillés section 3.3.3.2 p. 58, s'appuient sur une exécution de l'application à transformer permettant de s'abstraire de l'analyse de code approfondie employée dans la méthodologie exposée ici. Le prototype de transformation dynamique est dénommé *ArrSyn*. Le tableau comparatif 4.3 met en évidence les résultats obtenus après application des deux méthodologies. Le même protocole est employé pour estimer les résultats issus de *ArrSyn*, que ceux employés plus haut pour *arT*. Le gain comparatif entre les résultats après application de *arT*, par rapport à ceux

obtenus après application de *ArrSyn* est indiqué. Un gain positif indique une amélioration des résultats issus de *arT* par rapport à *ArrSyn,* et inversement pour un gain négatif.

	taille			gain	Performances			gain
	initiale	ArrSyn	*arT*	(arT/ArrSyn)	initiale	ArrSyn	*arT*	(arT/ArrSyn)
simple_loop	11	10	10	0%	42	29	29	0%
median	47	45	47	-4.44%	358	279	291	-4.3%
interp1	30	20	20	0%	423	203	203	0%
interp2	20	18	17	5.6%	218	162	176	-8.6%
addnoise	30	32	27	15.6%	613	557	494	11.3%
exp1	19	17	17	0%	100	71	71	0%
nest1	23	19	19	0%	582	383	383	0%
nest4	24	25	23	8%	16928	12220	11019	9.8%
nest5	34	35	29	17.4%	13327	8415	9249	-9.9%
vectmpy	22	22	20	9.1%	1815	1516	1366	9.9%
mac	26	26	26	0%	1968	1670	1520	8.9%
iir	44	50	44	12%	1322	986	1077	-9.2%
fir	30	28	26	7.1%	32815	22867	22816	0.2%
firnl	48	72	38	47.2%	19866	13237	12566	5.1%

Figure 4.3: Comparaison entre *arT* et *ArrSyn* sur quelques exemples

Le processeur MMDSP est ici le processeur cible. Tous les exemples précédents ne sont pas inclus, car *ArrSyn* en refuse certains, notamment ceux contenant des tableaux multi-dimensionnels. Pour les exemples pouvant être comparés, les résultats entre les deux approches sont proches, légèrement en faveur de *arT*. L'avantage de l'approche dynamique de *ArrSyn*, est que la valeur exacte de l'intervalle de valeurs dans lequel évolue l'indice d'un tableau lui est connue. Cela permet notamment, de transformer les corps de boucle contenant du code conditionnel de façon plus efficace, par exemple en réservant un pointeur aux références tableaux conditionnelles, pointeur qui évolue dans l'exact intervalle de valeurs correspondant. Cela reste dangereux puisque cette exécution conditionnelle peut dépendre de conditions externes à la boucle, elles-mêmes dépendantes de conditions non déterministes données en entrée du programme.

Le prototype *arT* permet donc d'obtenir des résultats similaires à ceux obtenus par les travaux antérieurs, s'appuyant sur une analyse dynamique. Il traite par ailleurs des cas, comme les tableaux multidimensionnels, non-traités par l'approche précédente. L'avantage principal de l'approche statique par rapport à l'approche dynamique, est d'être plus conservative, dans le sens où son indépendance vis-à-vis de stimulis et de leur taux de couverture, permet de s'assurer d'obtenir un programme ayant la même fonctionnalité après transformation. Cette approche évite par ailleurs la mise au point de séquences de stimulis donnant une observabilité totale des chemins d'exécution du programme.

4.3.5 Application conjuguée de *arT* et de l'exploitation des boucles matérielles

Applications	Nombre de cycles original	Nombre de cycles *arT* + BM	Gain
simple_loop	42	21	50%
median	358	152	57.5%
interp1	423	113	73.3%
interp2	218	115	47.3%
addnoise	613	318	48.1%
exp0	1212	715	41%
exp1	100	48	52%
exp2	508	214	57.9%
exp3	473	244	48.4%
nest1	582	234	59.8%
nest2	5288	852	**83.9%**
nest3	1639	425	74%
nest4	16928	7333	56.7%
nest5	13327	8372	37.2%
convolution	60214	25270	58%
vectmpy	1815	918	49.4%
matrixmpy	17949	5553	69%
mac	1968	922	53.2%
latsynth	1711	1019	40.4%
iir	1322	879	33.5%
fir	32815	12722	61.2%
firnl	19866	9172	53.8%
vocoder	331	238	28.1%
Moyenne			53.6%

Tableau 4.4: Application conjointe de *arT* et de l'emploi de boucles matérielles (BM).

Une implémentation préliminaire de l'exploitation au niveau source des possibilités de boucles matérielles présentes dans l'architecture cible (cf. section 3.4.3 p.90), a été appliquée ici, conjointement avec *arT*. Ce dernier, éliminant l'utilisation de variables d'inductions, facilite cette optimisation.

Les résultats obtenus sont intéressants, avec une augmentation de performance moyenne de 53%, et quelques pointes importantes pour les applications *nest2*, *matrixmpy* ou *interp1*, les deux premières contenant des boucles imbriquées, la dernière contenant plusieurs boucles successives. Ils confirment l'intérêt de combiner ces deux transformations.

4.4 En résumé

Ce chapitre s'est consacré à l'exposé de résultats d'expérimentations, obtenus en application les transformations étudiées et développées aux cours de ces travaux. La transformation de tableaux en pointeurs améliore les performances du code initial, contenant boucles et tableaux, de 20% à 30% en moyenne, suivant le processeur cible. L'une des raisons de cette amélioration, outre la réduction des opérations dans les corps de boucles, est l'augmentation du taux de parallélisme au niveau instruction. Comparée à de précédents travaux, cette optimisation produit des résultats comparables, voire légèrement supérieurs, pour des programmes acceptés par les deux prototypes de transformation. La raison en est principalement la possibilité d'explorer un espace de solutions plus large. Enfin, l'application conjointe de cette optimisation et de l'emploi de dispositifs de boucles matérielles dans la machine cible, améliore encore les performances du code final, pour atteindre 50% en moyenne. Ces résultats confirment l'intérêt de la transformation de tableaux en pointeurs telle que pratiquée, et prouvent la faisabilité de son automatisation.

Chapitre 5

Conclusion

Ces travaux de thèse se sont consacrés à l'étude d'optimisations de code à haut niveau d'abstraction, s'appliquant à des applications essentiellement de type traitement du signal, destinées à être exécutées sur un processeur embarqué. L'approche proposée permet de résoudre en partie le problème lié à l'incapacité des compilateurs actuels à générer un code efficace, à partir d'un programme exploitant les caractéristiques de haut niveau d'un langage comme C.

Ces travaux se sont plus précisément attachés au développement d'un algorithme augmentant les performances d'un programme utilisant des tableaux indexés dans des boucles. Le prototype développé, *arT*, agit sur une représentation intermédiaire de haut niveau, sur la base d'informations issues d'une analyse de code approfondie préalable. Cette analyse examine les boucles du programme à transformer et en extrait les variables d'inductions et les expressions d'inductions fonctions affines de ces variables. *arT* regroupe tout d'abord les références tableaux dont l'évolution est parallèle au cours de l'exécution de la boucle, en des ensembles dits ensembles de connivence. Les références tableaux appartenant à un même ensemble ont la propriété de pouvoir être remplacés par un pointeur unique dans le programme initial moyennant in/décrémentation de celui-ci.

A la lumière d'informations sur les modes d'adressage disponibles dans la machine cible, un travail peut-être réalisé sur ces ensembles de façon à optimiser pour cette machine la transformation des références tableaux conniventes en pointeurs. Les modes d'adressages les plus avantageux peuvent ainsi être favorisés. Ces informations sur la machine sont données via un fichier de spécification, décrivant les ressources d'adressages dans un langage simple et fonctionnel.

L'algorithme développé se distingue des travaux existant par l'étendue des problèmes qu'il permet de traiter. Le type de boucle considéré ne se restreint pas aux boucles simples et bien formées mais considère tous types de boucles. Les expressions d'inductions indices de tableaux peuvent être des fonctions affines de plusieurs variables d'inductions différentes. Les tableaux muti-dimensionnels sont considérés, de même que les boucles imbriquées. Le traitement de ces dernières, dans l'ordre inverse de leur profondeur d'imbrication, permet d'optimiser en priorité les parties de code les plus critiques. Les modes d'adressages considérés ne se restreignent pas aux modes post-modifiés, mais incluent les modes d'adressage

pré-calculés s'ils sont disponibles. Le prototype permet d'autre part d'explorer un espace de solutions plus large, en proposant certaines variantes à la transformation principale. Enfin cette optimisation se base sur une analyse du code entièrement statique, c'est à dire indépendante d'une quelconque exécution du programme à transformer.

L'application du prototype *arT* sur un ensembles de programmes contenant divers types de boucles, et ciblant deux processeurs spécifiques, montre une amélioration conséquente des performances des codes générés après transformation, principalement en termes de temps d'exécution, et dans une plus faible mesure en terme de taille de code. Ces résultats confirment l'intérêt d'intégrer une telle optimisation au sein des optimisations incontournables appliquées par un compilateur associé à un processeur de type DSP, et notamment ASDSP.

Ces travaux démontrent par ailleurs l'opportunité de substituer à une transformation traditionnellement manuelle, une transformation automatique. Des études supplémentaires ont identifié un certain nombre d'optimisations, en rapport de près ou de loin avec l'algorithme *arT,* et susceptibles d'améliorer plus avant les performances obtenues avec ce dernier. Au sein de celles-ci, l'exploitation automatique de dispositifs de boucles matérielles, souvent disponibles dans les DSPs, est une candidate particulièrement attrayante, comme l'atteste une implémentation préliminaire. Ces optimisations feront l'objet de travaux futurs, afin de proposer à terme une technologie de compilation capable de permettre au concepteur d'applications d'écrire ces dernières à un niveau élevé, favorisant la portabilité et la facilité d'évolution de celles-ci.

Les transformations implémentées au cours de ces travaux de thèse, se sont effectuées de source à source dans un objectif premier d'expérimentation et de prototypage. En extrapolant cette façon de procéder, ces travaux, comme d'autres avant eux, ont démontré la faisabilité d'une telle approche. Il est ainsi possible envisager un outil de transformation frontal source-source plus général, capable de subtituer à des transformations manuelles, source d'erreurs, des transformations automatiques et systématisées. L'objectif d'un tel outil frontal serait de permettre l'exploitation plus rapide et efficace de cœurs de processeurs existants, accompagnés de compilateurs dont les performances imposent une réécriture totale ou partielle d'applications. En effet, même si la tendance est au développement de cœurs de processeurs plus amicaux d'un point de vue compilation, ainsi que de technologies de compilations plus efficaces, il existe une large gamme de processeurs, dont la vie se maintiendra encore longtemps, et dont la reconception systématique du compilateur est inenvisageable.

Partie II

Analyse de la consommation dans un environnement de synthèse comportementale

Chapitre 6

Introduction

6.1 Motivations

Savoir prendre en compte la consommation au cours de la synthèse de circuits numériques CMOS, voire de systèmes embarqués entiers, est un art d'actualité. Cet engouement général pour la réduction de la consommation, académique et surtout industriel, est une preuve s'il en faut que ce problème échappe au qualificatif de "mode passagère", pour endosser celle de "sujet clé" pour le futur de la conception ULSI.

De fait, le concepteur de circuits intégrés, encore récemment confronté aux deux paramètres principaux bien connus de surface et de délais, se trouve avoir à prendre en compte lors de la conception une troisième dimension : la consommation finale de son circuit.

Bien que la technologie CMOS ait semblé résoudre définitivement à une certaine époque les problèmes de consommation, la très grande intégration actuelle, combinée à une vitesse de calcul toujours plus grande, résulte en une consommation toujours plus élevée et en une dissipation de chaleur plus importante. Cette dernière entraîne une augmentation des coûts de mise en boîtier pouvant avoir de graves répercussions sur le marché des microprocesseurs et systèmes à gros volume de production, comme ceux du secteur multimédia, en pleine expansion, et qui connaît une rude concurrence. Le marché des processeurs embarqués à faibles marges n'échappe pas non plus à la règle et la consommation fait là aussi la différence. Par ailleurs, l'émergence au premier plan du marché des appareillages portables pousse vers la mise au point de systèmes économes afin de préserver l'autonomie des batteries. Ce sont là les principales raisons de l'émergence de la consommation au sein de l'ensemble des problèmes de premier plan conditionnant l'avancée de la microélectronique.

Avant de pouvoir agir sur le phénomène consommation, il faut pouvoir l'observer c'est-à-dire l'estimer. L'objectif est double : constater si cette consommation dépasse ou non les contraintes fixées au départ, et surtout vérifier que les actions menées pour la réduire ont bien eu les effets escomptés. Tel est l'objet des travaux décrits dans cette partie.

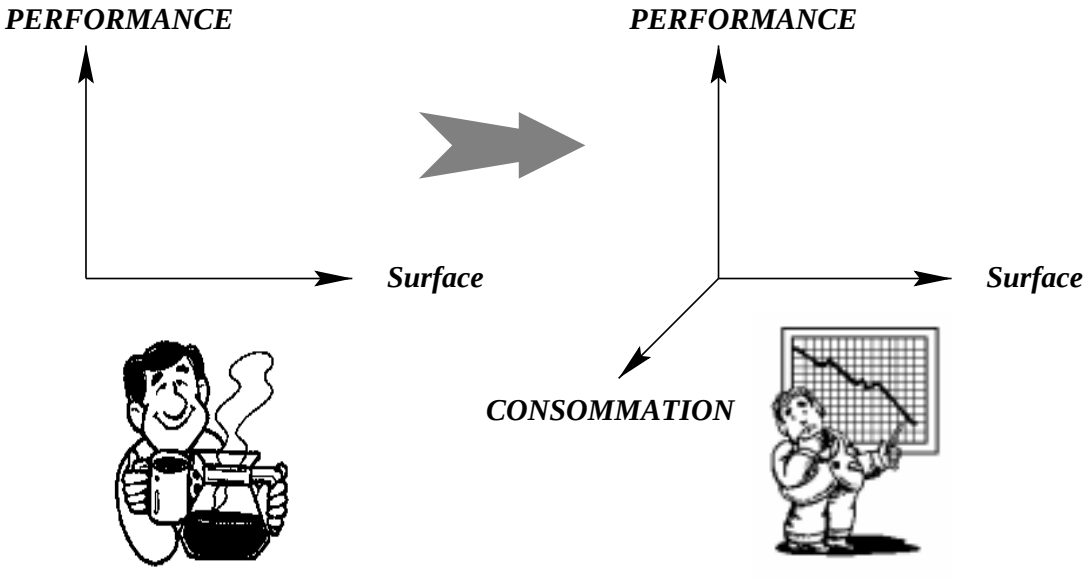


Figure 6.1: Evolution vers une troisième dimension conceptuelle

6.2 Objectifs et organisation de l'exposé des travaux

Les travaux de thèse exposés se consacrent à l'étude et au développement d'une méthodologie d'estimation de la consommation accompagnant un outil de synthèse de haut niveau. Les objectifs sous-tendant ces travaux sont multiples.

- Comprendre les problèmes posés par l' estimation de la consommation. Celle-ci se distingue nettement de l'estimation de la surface et du délai. En effet, ces deux estimations peuvent se baser sur des données constantes dans le cas de la surface, et pire cas dans le cas du délai. A contrario, l'estimation de la consommation dépend d'un paramètre mouvant par définition: l'activité intrinsèque du circuit et notamment l'activité statistique des données manipulées par celui-ci. C'est ainsi que pour une technologie choisie, la consommation d'un composant, par exemple un additionneur, n'est pas la même suivant le circuit dans lequel il est utilisé, alors que son délai maximum ou sa surface sont des données figées.

- Développer une méthodologie d'estimation permettant de rester dans un environnement de synthèse de haut-niveau. Il s'agit d'éviter de devoir descendre jusqu'aux niveaux d'abstraction tel que porte logique ou layout afin d'obtenir les mesures souhaitées. Cela permet en outre d'itérer la boucle *synthèse* $\leftrightarrow$ *estimation* de façon rapide et efficace.

- Fournir une méthodologie capable de prendre en compte les spécificités des circuits fortement orientés contrôle, spécialité de l'équipe au sein de laquelle ces travaux se sont déroulés. Ce type de circuit présente la caractéristique d'obéir à un comportement dépendant des données, à priori imprévisible.

Le chapitre 7 présente un état de l'art des concepts accompagnant la consommation de circuits CMOS, détaillant brièvement les différentes visions de la consommation propres aux niveaux d'abstraction rencontrés au cours d'un flot de conception classique. L'accent est mis sur les travaux relatifs à l'estimation de la consommation au niveaux comportemental et transfert de registres. Le chapitre 8 décrit de façon détaillée la méthodologie développée, explicitant les choix effectués. Des résultats d'estimations, résultant d'expérimentations sur quelques circuits, donnent la mesure de l'impact de ces choix chapitre 9. Le chapitre 10 conclut cet exposé.

Chapitre 7

La consommation: troisième dimension de conception

7.1 Préambule

Ce chapitre brosse un tableau rapide de ce qu'est la consommation, du pourquoi de l'intérêt qu'elle suscite et des moyens d'action à la disposition du concepteur de circuits aujourd'hui. Ce dernier se doit de travailler à ce que son circuit soit une entité non plus seulement sportive et mince, mais aussi économe dans l'effort. Un bref historique tentera de retracer les étapes qui ont remis sur le devant de la scène un problème qui était supposé ne plus en être un, du moins dans les domaines de la conception de circuits classiques. Ce chapitre se propose par ailleurs, de rappeler brièvement les moyens propres à chaque niveau d'abstraction permettant au concepteur de connaître l'importance de la consommation de son circuit, et la manière dont celui-ci consomme. L'accent sera mis essentiellement sur l'état de l'art de l'estimation associée à la synthèse de haut niveau.

7.2 Un bref historique

L'histoire de la microélectronique basse puissance peut se retracer depuis l'invention du transistor en 1947. Le passage de la lampe au transistor, c'est-à-dire d'une dissipation de chaleur se traduisant en watts à quelques dizaines de milliwatts, fut prépondérant. La capacité d'exploitation de ce potentiel basse puissance ne devint réelle qu'en 1958, date de l'invention du circuit intégré, et depuis lors n'a cessé de s'affirmer. Dès 1963, le potentiel de micro-consommation lié à la technologie CMOS était clairement formulé par G. Moore *et al.* [83]. Depuis, la microélectronique a évolué en productivité et performance jusqu'à un degré relatif jamais atteint dans l'histoire de la technique, au point d'avoir un profond impact sur notre vie quotidienne. La loi de Moore, tirée de l'observation de Gordon Moore[1] sur le doublement annuel[2] du nombre de transistors par puce, ainsi que la plupart des analyses,

[1] Intel Corporation

[2] qui a évolué aux alentours de 1,5

n'envisagent que peu de frein à cette incroyable expansion. Il est d'ailleurs plus adapté de parler actuellement de conception ULSI[3] et non plus VLSI: le 21ème siècle verra sans doute l'avènement de la conception GSI[4], soit plusieurs milliards de transistors par puce.

Ainsi [120], alors qu'il y a 20 ans une machine proposant 10 MIPS (Millions d'instruction Par Seconde) coûtait \$10M et consommait 10kW, la génération actuelle propose des microprocesseurs réalisant la même puissance de calcul, disponibles à \$10 ou moins et présentant un pic de puissance de moins de 1W. Ce progrès n'aurait pas été possible sans la réduction en taille des puces, mais aussi sans l'apparition de nouvelles architectures, de nouveaux concepts dans la conception de jeux d'instructions réduits[5] , et d'outils de CAO[6] puissants et performants.

Selon [124], la poussée générale actuelle vers une plus basse consommation tourne autour de trois pôles, correspondant à trois façons d'envisager la conception de systèmes :

1. Le marché des appareils portables tels que les PDAs. Les problèmes clés de la conception de tels appareils sont ici relatifs à l'autonomie de la batterie associée, au poids de l'appareil et au prix du boîtier supportant la puce. Par exemple, une puissance moyenne de 1 à 2 W autorise l'emploi de boîtiers plastiques très bon marché. Il est préférable ici de parler de consommation en terme d'énergie plutôt qu'en terme de puissance, étant entendu qu'une batterie recèle en son sein une certaine quantité d'énergie, et qu'il faut donc tenter de diminuer l'énergie globale consommée par le système pour effectuer une tâche donnée.

2. Les ordinateurs portables haute performance. La conception se focalise sur la réduction de la puissance consommée par la partie purement électronique, de telle sorte que la puissance globale consommée ne soit pas dominée par celle-ci, mais plutôt par les autres parties du système telles que écran et disque dur. Bien entendu, la durée de vie de la batterie est ici aussi prépondérante.

3. Les systèmes indépendants d'une batterie tels que stations de travail, ordinateurs personnels, serveurs ... Ici, l'objectif est d'avoir une consommation ne dépassant pas certaines limites imposées par les problèmes liés à la production de chaleur par effet Joules, tels que fiabilité, refroidissement pouvant entraîner des niveaux de bruit important dans les bureaux, ou simplement liés à la capacité de l'environnement à fournir la puissance requise.

Les méthodes développées et généralisées aujourd'hui visant à réduire la consommation globale d'un circuit s'inspirent, comme il est montré par [83], de concepts connus depuis longtemps. Ce sont l'utilisation de la tension d'alimentation la plus faible possible, l'emploi de dimensions de transistor les plus petites disponibles, le recours à la parallélisation. Un principe directeur émerge, qui est de concevoir un circuit respectant juste les contraintes de

[3]Ultra Large Scale Integration
[4]Gigascale Integration
[5]RISC: Reduced Instruction Set Computer
[6]Conception Assistée par Ordinateur

performances requises, en se focalisant sur une puissance consommée minimum. Autrement dit, une solution viable semble être de concevoir un circuit le plus performant possible, puis descendre ce niveau de performance jusqu'à atteindre la limite inférieure des contraintes du cahier des charges afin d'obtenir un minimum de consommation.

7.3 Consommation et niveaux d'abstraction

A chaque niveau d'abstraction du flot de conception classique, correspond une certaine vision de la consommation et certains moyens d'actions pour la réduire. Cette vision est relative à la précision des informations disponibles, propres au niveau auquel on se place.

La croissance avec le niveau d'abstraction du manque d'information concrètes sur le circuit final , ainsi que la prépondérance générale de la consommation dynamique de commutation par rapport aux autres composantes, ont entraîné la plupart des auteurs à ne considérer que la consommation due aux commutations internes d'un circuit, soit la puissance capacitive.

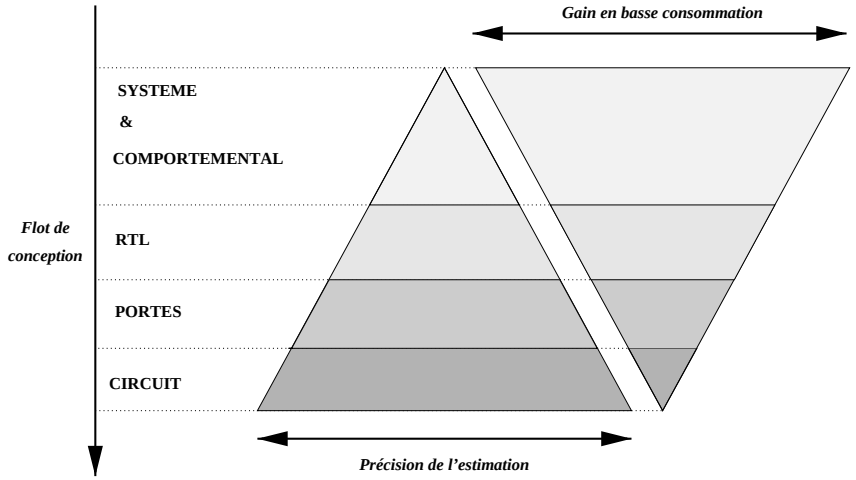


Figure 7.1: Opposition de phase entre estimation et gain

De façon synthétique, alors que le gain potentiel des optimisations et décisions orientées basse-consommation augmente proportionnellement avec le niveau d'abstraction, l'estimation de l'impact de ces décisions voit sa précision décroître tout aussi proportionnellement. C'est cette opposition de phase qu'exprime la figure 7.1.

7.3.1 Niveau circuit

C'est le niveau d'abstraction le plus bas où, à la suite de l'étape de placement-routage, toutes les informations nécessaires à une vision précise de la consommation et de sa répartition sont disponibles. En effet, on connaît alors avec précision la taille des transistors, la taille des interconnexions et donc toutes les capacités parasites qui conditionnent

notamment la dissipation dynamique du circuit. La consommation liée au courant de fuite peut alors être déterminée précisément, ainsi que celle liée au courant de court-circuit. Ces deux dernières composantes dépendant pour une grande part de l'état du circuit à un instant donné, il est nécessaire de les étudier par le biais de simulations. C. Piguet [97] donne l'exemple de la conception d'un microprocesseur où la capacité par porte est de 0.09 pF avant routage, et de 0.14 pF après, soit 35% de la capacité globale due au routage. Le principal problème à ce niveau, aussi bien pour l'analyse que pour l'optimisation, est l'énorme quantité d'information disponible, qui ne permet de traiter le plus souvent qu'une petite partie du circuit global. Certains outils d'analyses[7] remédient en partie à ce problème au prix d'une très faible perte en précision, ce qui est une performance, mais sont loin de permettre la manipulation des dizaines de millions de transistors caractérisant les puces complexes actuelles.

Ce niveau se caractérise donc par une précision très grande, mais une faible marge de manœuvre quant à la révision de la conception. Malgré tout, un certain nombre d'auteurs dont J. Cong [97] fondent quelques espoirs sur des techniques d'optimisation agissant sur la taille des transistors, de l'interconnexion, de l'horloge.

Un autre moyen d'agir à bas niveau est de concevoir des bibliothèques de cellules basse consommation, ainsi que de modifier les procédés de fabrication afin de permettre une tension de seuil V_t plus faible, associée à une tension plus faible.

7.3.2 Niveau portes logiques

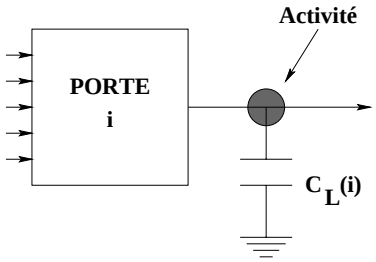


Figure 7.2: Modélisation simple d'une porte logique

Au niveau portes logiques, la plupart des auteurs abandonnent la dissipation statique et celle liée aux courts-circuits, pour se consacrer exclusivement à la consommation capacitive. Une porte i d'un circuit comportant N portes est ainsi vue comme un module réalisant une fonction logique, module pondéré par deux types de valeurs, comme illustré par la figure 7.2 : sa capacité de charge ramenée en sortie $C_L(i)$, et son activité en sortie $\alpha(i)$ soit le nombre moyen de commutations par cycle au nœud de sortie. La puissance globale capacitive moyenne consommée par le circuit est alors :

$$P_{moy} = \sum_{i=1}^{N} \frac{1}{2} \times \alpha(i) \times C_L(i) \times V_{dd}^2 \times f$$

[7] tels que PowerMill ou TimeMill d'Epic

$$= \frac{1}{2} \times C_{eff_{global}} \times V_{dd}^2 \times f$$

où $C_{eff_{global}} = \sum_{i=1}^{N} \alpha(i) \times C_L(i)$ est la capacité commutée moyenne globale du circuit. Les transitions prises en compte en sortie sont supposées complètes, c'est-à-dire correspondant à un écart de tension égal à V_{dd}. Du modèle de délai choisi pour les portes dépend la précision et la fiabilité des estimations basées sur cette formule. Le modèle à délais nuls permet une estimation ne tenant pas compte des transitions superflues pouvant advenir au sein d'un circuit, et par la même s'avère peu fiable, si ce n'est pour fournir une première approximation de ce que la consommation devrait être pour un fonctionnement optimum. Ces transitions superflues sont par définition des transitions logiques inutiles d'un point de vue fonctionnel, et sont de deux sortes :

- Les transitions complètes, c'est-à-dire présentant un écart complet de V_{dd}, mais inutiles.
- Les transitions incomplètes, c'est-à-dire n'atteignant pas V_{dd} ou V_{ss}, autrement nommées *glitches* ou *hazards*.

Dans le dernier cas, bien que d'un point de vue logique l'évènement ne soit pas forcément visible, il crée malgré tout une consommation qui au total, peut ne pas être négligeable. En effet, la capacité de sortie se décharge ou se charge en partie, ce qui provoque des transferts de charges, et ce qui peut occasionner des tensions intermédiaires entre V_{dd} et V_{ss} pendant une période au cours de laquelle le courant de court-circuit ou de fuite résultant peut être conséquent.

L'estimation de la puissance consommée au niveau logique s'attachera donc à déterminer principalement l'activité à chaque nœud du circuit, la charge étant supposée être accessible via des cellules pré-caractérisées en bibliothèque, ou via des annotations provenant du niveau circuit[8]. L'optimisation s'attachera à ce niveau à éviter les transitions logiques inutiles en diminuant la profondeur de la logique, et en équilibrant les chemins d'exécutions. Elle tentera d'autre part de réduire la charge parasite des nœuds fortement actifs.

7.3.3 Niveau transfert de registres

A ce niveau architectural, dit couramment RTL[9], comprenant diverses unités inter-agissantes comme blocs fonctionnels, registres, ou unités d'interconnexions, la modélisation est entièrement axée sur la puissance capacitive. On va donc considérer, ainsi qu'il est représenté sur la figure 7.3 :

- des modules fonctionnels pré-caractérisés en terme de consommation. Le modèle de macro-modélisation de la consommation pour les modules joue un rôle capital.

[8] backannotation
[9] Register Transfer Level

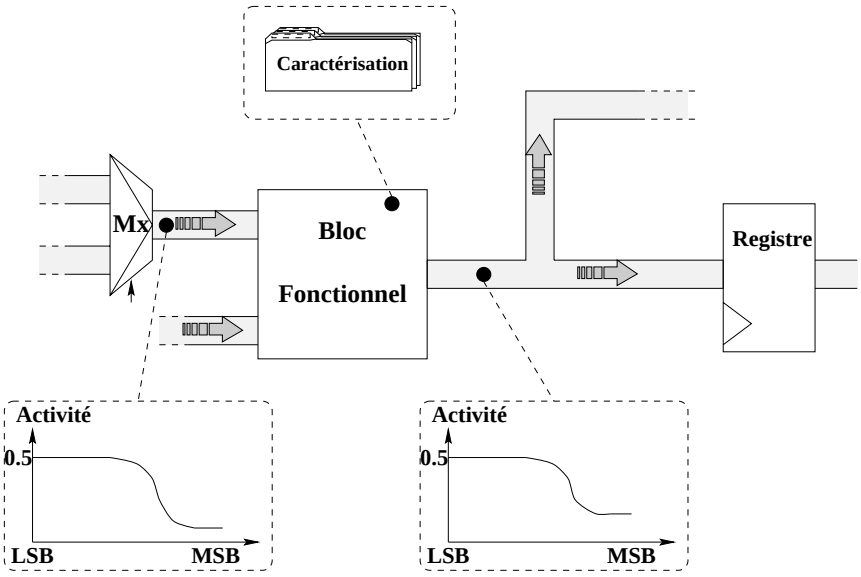


Figure 7.3: Modélisation au niveau transfert de registres

- un réseau d'interconnexions par le biais duquel les transferts de données pourront s'effectuer. Le modèle de ce réseau ainsi que l'activité des données le parcourant conditionneront la part qu'il pourra prendre dans la consommation capacitive globale (un bus, suivant sa charge et son activité, pourra peser lourdement sur cette dernière).
- un modèle statistique des données transférées.

L'optimisation dans un objectif de basse consommation peut se faire en parallélisant certaines parties critiques du circuit, afin de pouvoir baisser la fréquence d'exécution et donc la tension d'alimentation, sans pour autant constater une perte en performance. La technique du pipeline sera employée avec le même effet. Il est également possible de tenter d'isoler des parties du circuit inactives au cours de l'exécution, et de couper l'horloge dans ces parties aux moments opportuns. Le remplacement d'opérateurs très actifs et coûteux par des opérateurs équivalents mais consommant moins, est aussi une solution efficace.

7.3.4 Niveau comportemental

Matériel L'abstraction de ce niveau est source de difficultés pour obtenir la consommation précise liée à un algorithme destiné à être synthétisé en un ASIC. Une solution est de considérer les opérations exécutées et leur possible implémentation physique, de façon à comparer grossièrement l'efficacité de deux solutions algorithmiques en terme de consommation, en ajoutant simplement leur puissance. Par contre, il s'avère que des modifications appliquées à ce niveau peuvent avoir un impact certain sur la consommation finale. Un ensemble de transformations [22] à appliquer sur des algorithmes de traitement du signal peuvent, utilisées conjointement, réduire d'un facteur important la consommation du circuit final.

Logiciel Pour l'analyse et l'optimisation d'un code destiné à être implanté dans un microprocesseur, la métrique employée sera la suivante :

- pour chaque instruction, le coût énergétique associé si elle est utilisée;
- le coût de l'emploi d'une instruction relativement à l'instruction précédente (inter-instruction effect).

Ces deux types de coût sont accessibles dans la mesure où l'on connaît le processeur cible, bien que difficile à obtenir avec précision, à moins que le fabricant ne s'y emploie, et ne fournisse l'information dans l'ouvrage de référence associé au processeur. Partant de là, des techniques d'allocation d'instructions et d'ordonnancement peuvent être appliquées afin d'obtenir un code consommant moins, cela pendant ou après l'étape de compilation.

7.3.5 Niveau système

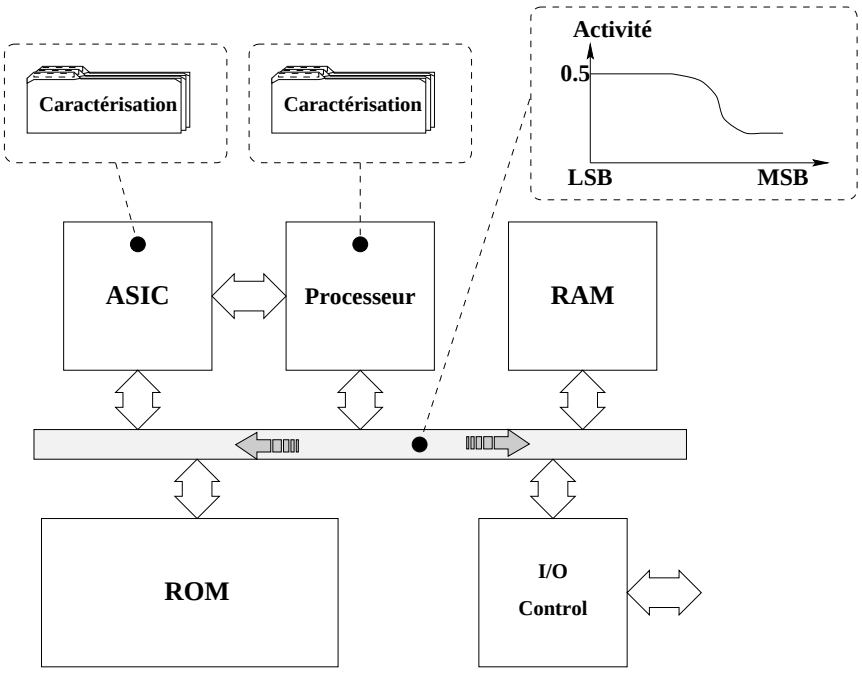


Figure 7.4: Modélisation au niveau système

C'est le niveau le plus haut dans la hiérarchie d'abstraction caractérisant le flot de synthèse descendant[10] classique. A ce niveau, un très grand degré de liberté est associé à un manque critique d'informations physiques issues de l'implémentation du système. C'est à ce niveau que les plus grandes améliorations peuvent être obtenues, comme en témoignent les quelques publications relatant de telles expériences [84], [68].

[10] Top-Down

La modélisation d'un système en terme de consommation est donc similaire à celle du niveau transferts de registres, dans le sens où les deux niveaux sont des niveaux architecturaux : on parle de blocs fonctionnels interconnectés entre eux, comme illustré sur la figure 7.4. Cependant, les blocs fonctionnels au niveau système sont des macro-blocs complexes - microprocesseurs, RAM, ROM, contrôleurs, ASICs, ASIPs, ASDSPs ... - qui supposent une caractérisation énergétique préalable, fonction là aussi de l'activité en entrée.

Les données échangées entre les macro-blocs le sont par l'intermédiaire de bus de données et/ou d'adresses dont le poids peut aussi avoir une grande importance sur la consommation. L'activité des données se succédant sur un bus lourdement chargé (en terme de capacité parasite), est fortement dépendante de l'application visée.

La puissance moyenne consommée par un système n'est généralement pas la somme des puissances moyennes de chaque bloc pris séparément. En effet, suivant le taux d'activité de chaque bloc et l'activité des données en entrée, certains blocs consommant beaucoup peuvent finalement ne pas être critiques face à un bloc consommant moins, mais beaucoup plus actif.

7.4 Estimer la consommation au niveau comportemental

7.4.1 Préambule

Placer la consommation au sein des contraintes de conception au même statut que performance - en terme de vitesse de calcul - et surface, est un tournant relativement récent dans le monde de la CAO de circuits. Compte tenu de la proportionnalité existant entre la hauteur du niveau d'abstraction de la description du circuit, et l'efficacité de décisions orientées basse consommation, concepteurs et architectes doivent pouvoir agir très tôt dans la chaîne de conception d'un circuit, et donc constater très tôt l'impact de telle ou telle décision.

Comme introduit judicieusement dans [27], l'estimation ou l'analyse de la consommation est un premier pas vers la mise en œuvre de techniques d'optimisations basse consommation dans les outils de synthèse de haut niveau. Ces outils se placent suffisamment haut dans le flot de conception pour pouvoir explorer rapidement plusieurs solutions architecturales avec un impact conséquent sur les performances générales à bas niveau.

Au niveau d'abstraction qui est celui de la synthèse de haut niveau, il serait vain d'espérer obtenir une précision absolue parfaite. Rares sont ceux prétendant obtenir de telles précisions, globalement en dessous de 5 à 10 %, et lorsque c'est le cas, ces mesures ne sont significatives que pour des cas très particuliers. Ce qui importe à ce niveau du flot de conception, c'est de pouvoir obtenir rapidement une mesure relative pour une solution donnée d'un circuit, afin d'être en mesure de pouvoir comparer deux solutions entre elles, de pouvoir indiquer les parties d'un circuit, voire les étapes au cours du fonctionnement du circuit, susceptibles d'avoir le plus grand impact sur la consommation. On peut alors agir.

La fourchette de précision doit rester cependant honorable : en dessous de 20% d'erreur semble correspondre aux conditions requises.

Afin d'atteindre le niveau requis de fiabilité, il n'est pas suffisant de simplement considérer des données statiques au cours du processus d'estimation. Cela convient pour estimer la surface, ou encore les performances temporelles en s'attachant à des délais pire-cas, mais pas la consommation. En effet, celle-ci est très dépendante des données fournies au circuit, qui conditionnent son activité interne. Cela est d'autant plus vrai pour les circuits fortement orientés contrôle, puisque leur comportement même est conditionné par les données qu'ils reçoivent. Pour ceux-ci, l'estimation de la consommation sera donc d'autant plus sensible à l'obtention préalable d'informations "dynamiques", c'est-à-dire liées à une activité du circuit. A ces informations dynamiques viendront se juxtaposer des données statiques, c'est-à-dire figées, indépendantes du circuit. L'obtention des données dynamiques et statiques, et leur exploitation en vue de l'estimation font l'objet du chapitre suivant.

7.4.2 Travaux connexes

Peu de travaux tentent de résoudre le problème de l'estimation d'une description comportementale pure [32, 82, 107]. A vrai dire, la tâche est ardue : la description fonctionnelle d'un circuit est particulièrement éloignée de la forme concrète du circuit à venir. L'estimation à ce niveau donne souvent des résultats très relatifs. Elle se base dans ses grandes lignes sur des techniques statistiques, permettant d'extraire l'activité des opérateurs de la description comportementale, auxquels sont associées des mesures de consommation par activation. Néanmoins, son grand intérêt, dans la mesure où l'estimation est suffisamment fiable pour autoriser des comparaisons, est de pouvoir tester dès le niveau algorithmique l'effet de transformations et de réécritures orientées basse-consommation.

De ce point de vue, la description transfert de registre, bien que pouvant être d'un niveau élevé quasi-comportemental, saute un grand fossé vers la forme finale du circuit : le fossé temporel. En effet, tandis que l'exécution de la forme comportementale d'un circuit est cadencée par l'événement - événement de contrôle ou de donnée - on trouve la notion d'horloge au niveau RT (Register Transfert). D'une façon générale, la description RT d'un circuit se prête donc mieux à l'estimation à tous les niveaux, et notamment d'un point de vue consommation. De nombreux travaux ont donc adopté le compromis consistant à estimer la consommation d'une description comportementale *dans le cadre d'un outil de synthèse comportementale*. A l'intérieur de ce cadre, puisque la synthèse comportementale génère une description RT à partir d'une description fonctionnelle pure, l'architecture plus ou moins approximative du circuit final est connue : l'estimation devient plus abordable. Comme la très grande majorité des outils de synthèse fonctionnelle se base sur une bibliothèque de composants, qui sont assignés aux opérations intervenant dans la description comportementale, se pose le problème de la macro-modélisation de ces composants d'un point de vue consommation : c'est là l'un des thèmes majeurs des travaux dans le domaine. Ce problème se pose dans les mêmes termes pour l'estimation d'une description RTL pure.

La stratégie de macro-modélisation la plus simple consiste à considérer la consommation d'un module comme une constante. Cette approche conduit donc à estimer la consomma-

tion d'un module une fois pour toute à bas niveau, généralement en appliquant au module des vecteurs d'entrée purement aléatoires, correspondant à une distribution uniforme des commutations sur les vecteurs d'entrée. Cette mesure est ensuite employée telle qu'elle au cours de l'estimation de la consommation du circuit.

Une approche de macro-modélisation beaucoup plus sophistiquée, est celle développée dans [60] et [59] : c'est l'approche bits-duaux. Elle part du principe que la consommation d'un module est fonction de sa taille, donc entre autres de la largeur des vecteurs d'entrée, et de l'activité transitionnelle sur ses entrées. Cette méthode propose donc des équations donnant la capacité commutée effective du module. Les cœfficients de ces équations, dits cœfficients capacitifs, sont déterminés en fonction de certaines activités sur les entrées du composant. Sont notamment distingués les bits de signe, qui correspondent aux bits de poids forts, dont l'influence est démontrée comme primordiale sur la consommation. Ce modèle permet de tenir compte de la corrélation temporelle d'une séquence de valeurs appliquées en entrée d'un module afin de caractériser plus précisément sa consommation.

[90] et [91] dérivent des modèles de consommation de puissance pour des unités fonctionnelles en fonction de l'activité des opérandes et de la répétition des opérandes, c'est-à-dire de la propension d'une opérande à rester identique entre deux activations de l'unité fonctionnelle.

L'une des applications de la macro-modélisation et de la rapidité d'estimation qu'elle implique, est la mise au point d'outils de synthèse comportementale orientés basse consommation. La métrique la plus couramment rencontrée est la capacité commutée estimée après les étapes de synthèse afin de donner une idée de l'impact de ces dernières sur la consommation. Les modules sont en général caractérisés de façon simple, en fonction de la largeur de leurs entrées, à laquelle est associée une valeur unique issue de l'application de séquences aléatoires uniformément distribuées au cours de simulation de bas niveau.

PDSS [56] est un outil de synthèse comportementale qui se base sur un profilage du graphe de flot de données de la description comportementale. Ce dernier est simulé afin d'obtenir des informations sur l'activité intrinsèque du circuit. Le nombre d'invocation de chaque opération est une des informations obtenues. Ces données sont ensuite employées pour estimer la capacité commutée totale du circuit synthétisé. La stratégie de macro-modélisation adoptée est simple.

[110] fournit une méthode d'allocation basse consommation basée sur une simulation fonctionnelle du CDFG ou graphe de flot de contrôle et de donnée (Control Data Flow Graph).

GAUT_W [34] est un outil de synthèse de haut-niveau de circuits de type DSP, comprenant des modules d'optimisation basse-consommation, et notamment un outil d'estimation au niveau comportemental, considérant une macro-modélisation simples des opérateurs.

HYPER-LP, outil de synthèse développé à Berkeley [22], contient un modèle relativement complet d'estimation de la consommation. Les modules sont caractérisés en fonction de la largeur de leurs entrées. Ici encore, la stratégie de macro-modélisation est la stratégie simple décrite ci-dessus.

Excepté l'approche bits-duaux, la plupart des méthodologies employées considèrent donc un modèle de macro-modélisation simple. La méthodologie expliquée au cours du

prochain chapitre propose une stratégie de macro-modélisation un peu plus complexe dans le but de raffiner l'estimation. Par ailleurs, elle propose une représentation de la consommation détaillée, propre à mieux mettre en évidence les parties critiques du circuit d'un point de vue consommation. Enfin, cette méthodologie tient compte de la consommation parasite du circuit, ce qui n'est pris en compte dans aucunes des méthodes présentées ci-dessus.

7.5 En résumé

Ce chapitre a présenté une vision photographique de la consommation, vision qui est différente suivant le niveau d'abstraction auquel on se place. De façon générale, les niveaux d'abstraction les plus élevés sont ceux présentant le plus grand potentiel d'un point de vue réduction de la consommation. La difficulté à ces niveaux élevés réside dans l'estimation de la consommation, en raison du peu d'information disponible, qui nécessite d'extrapoler la forme finale du circuit. Ce chapitre s'est attaché par ailleurs à présenter un état de l'art des techniques d'estimation de la consommation à un niveau structurel et comportemental. La plupart de ces techniques emploient une stratégie de macro-modélisation simplifiée, et fournissent un résultat d'estimation global. Le chapitre suivant expose une technique tentant de répondre de façon plus précise à ce problème.

Chapitre 8

Méthodologie d'estimation

8.1 Préambule

Ce chapitre détaille la méthodologie d'estimation de la consommation développée au cours de ces travaux. Trois points sont préalablement détaillés : la stratégie adoptée de macro-modélisation des composants du circuit, l'extraction d'informations dynamiques par le biais de simulations de la description comportementale, et le calcul des activités dans le chemin de données afin d'affiner l'estimation. Puis la méthodologie d'estimation elle-même est présentée. Celle-ci exploite les informations fournies par les trois points préalablement détaillés, et fournit une vision détaillée par cycle d'exécution de la consommation énergétique du circuit, ainsi que la puissance moyenne.

8.2 Vue générale de la méthodologie

La technique employée ici en vue d'estimer la consommation, se veut être une synthèse des idées maîtresses qui se sont affirmées dans le domaine, tout en adoptant des voies d'investigation parallèles. Un des objectifs est de fournir un modèle d'estimation valable non seulement pour les circuits orientés flot de données, au comportement régulier et prévisible, mais aussi pour les circuits orientés flot de contrôle au comportement a priori imprévisible car dépendant des données ou de signaux externes. Il s'agit d'estimer la consommation à un stade initial du flot de conception, dans un environnement de synthèse comportementale. L'outil de synthèse considéré, évoqué dans ses grandes lignes dans l'annexe E, étant spécialisé dans les circuits orientés contrôle, cette estimation est orientée plus précisément vers ce type de circuit, mais peut être adaptée dans ses principes à des environnements de synthèse de circuits de type flot de données.

Plus précisément, l'objectif est d'estimer l'impact sur la consommation du chemin de données, de l'horloge et du réseau d'interconnexions, laissant de côté la partie contrôle. L'annexe E expose les caractéristiques de l'architecture cible sur laquelle se base la présente méthodologie. Nous verrons une façon particulière de représenter la consommation d'un circuit après synthèse comportementale, propre à mieux mettre en lumière les parties cri-

tiques de celui-ci. La prise en compte de la double dépendance aux données des circuits orientés contrôle, dépendance en terme de consommation, et en terme de comportement global, sera détaillée. Deux types de données seront nécessaires au processus d'estimation, données dites statiques, c'est-à-dire déterminées au préalable et indépendantes dans leur essence du circuit à estimer, et données dites dynamiques, c'est-à-dire caractéristiques du circuit et surtout de son comportement. Le premier type est lié au processus de caractérisation de la librairie, tandis que le second évoque la notion d'extraction d'information par le biais de simulation du circuit.

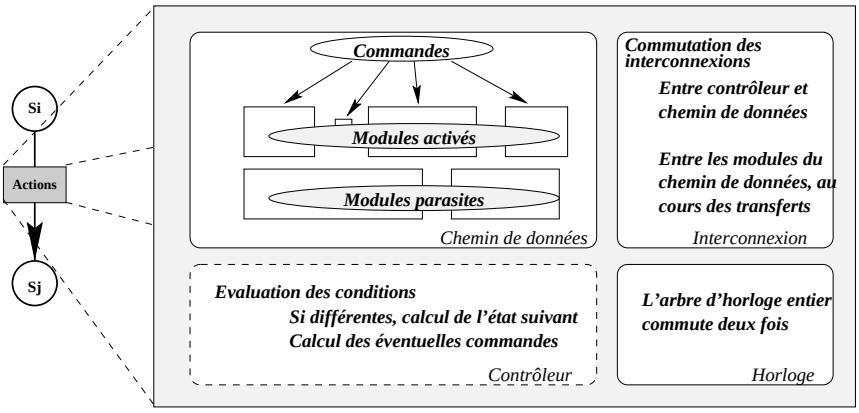


Figure 8.1: Actions intervenant au cours d'un cycle

La logique de l'estimation est directement dépendante de ce qu'il advient dans le circuit au cours d'un cycle de son fonctionnement une fois ce dernier synthétisé, comme représenté sur la figure 8.1. Ce qui est appelé ici cycle correspond aussi à une transition de la machine d'état du circuit, machine de Mealy (cf. annexe E). Au sein du chemin de données, un certain nombre de modules sont activés, délibérément ou non : chacun d'eux consomme donc de l'énergie. Celle-ci est dépendante des statistiques de distribution des données, i.e de la façon dont ces données se succèdent, entraînant des commutations plus ou moins importantes sur les entrées du module; sa valeur est extraite de la bibliothèque de composants préalablement paramétrés, identique à celle à laquelle la synthèse de haut niveau fait appel. Les interconnexions entre les différents modules supportent le passage des données échangées. L'activité moyenne des données échangées, et une capacité moyenne d'interconnexion, s'associent pour consommer de l'énergie à leur tour. La troisième composante de la consommation, et non des moindres, est l'arbre d'horloge qui consomme en fonction du caractère touffu de ses ramifications.

Le flot global est décrit sur la figure 8.2. La description au niveau transfert de registres du circuit synthétisé est fournie en entrée de l'estimateur. Il comprend la machine d'états finis décrivant le comportement de la partie contrôle, et la *netlist*[1] du chemin de données. Sont aussi fournies les données extraites de la simulation de la description VHDL comportementale. L'estimation se déroule en trois phases :

[1]description du circuit sous forme de composants et de leurs interconnexions

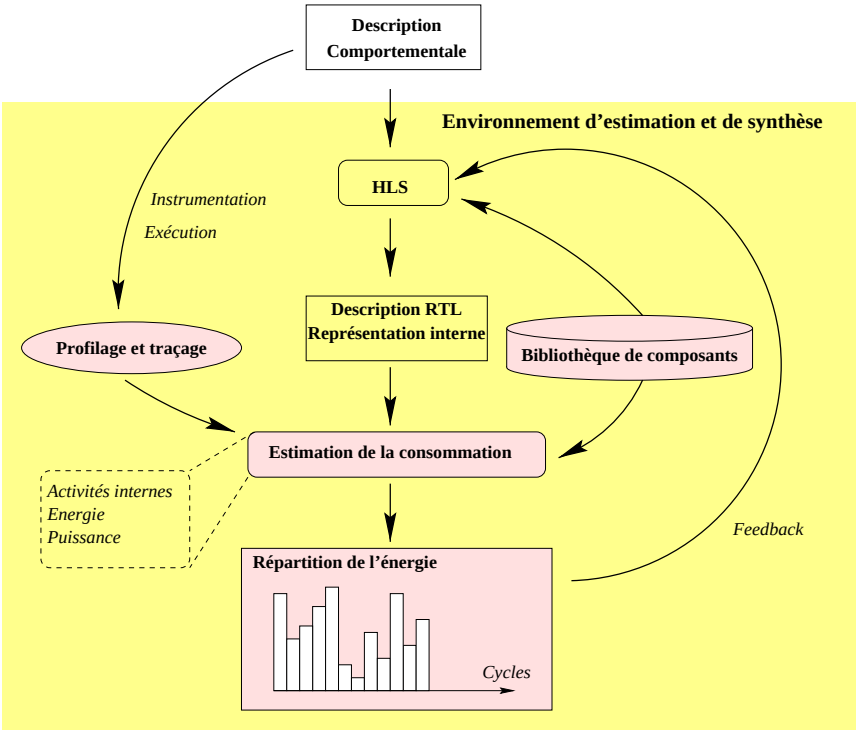


Figure 8.2: Flot global d'estimation

1. évaluation (propagation) des activités internes du chemin de données
2. estimation de l'énergie consommée par chaque partie déjà citées du circuit, incluant l'énergie parasite, à l'aide notamment de la librairie de composants paramétrés
3. estimation de l'énergie consommée au cours de chaque cycle de contrôle et évaluation de la puissance moyenne consommée.

8.3 Paramétrisation de la bibliothèque de composants

Tout procédé d'estimation d'un circuit décrit sous forme modulaire, est basé sur une bibliothèque paramétrée. De cette façon, les données appropriées à l'estimation (voire sensibles au contexte) peuvent être extraites pour chaque module composant le circuit. Pour le problème particulier de la consommation et de son estimation au niveau transfert de registres, la métrique souvent adoptée est la capacité commutée d'un composant. Nous lui préférerons ici *l'énergie par activation,* où activation signifie présence de nouvelles données sur les entrées. Cette métrique est totalement indépendante du temps, ou de la fréquence à laquelle les données se succèdent en entrée. Elle représente simplement la quantité consommée moyenne par un module pour fournir une sortie lorsque se produisent des changements sur ses entrées. Cette quantité est, par contre, dépendante des statistiques de commutation et de distribution des valeurs se succédant en entrée, comme nous le verrons

plus bas. A noter qu'employer énergie par activation et capacité commutée conduit à une équivalence de par la proportionnalité existant entre ces deux entités, due à l'approximation consistant à ne considérer que la consommation commutative. Cette section traite donc du procédé de macro-modélisation (cf. section 7.4.2 page 125) des composants du circuit à estimer qui est employé dans la méthodologie exposée ici.

8.3.1 Influence de l'activité des données échangées sur la consommation

L'estimation de la puissance et de l'énergie consommée par un circuit est une cible mouvante: elle dépend de l'activité du circuit. Cette dernière est elle-même assujettie à l'activité moyenne manifestée par les différentes données se succédant sur les entrées[93] : l'expérience montre que plus le nombre de bits commutant est important entre deux vecteurs consécutifs appliqués à l'entrée d'un module quelconque, plus l'activité du circuit est intense. Cette caractéristique a été confirmée par la plupart des travaux dans le domaine [60, 59, 90, 91], et est considérée comme un acquis dans le reste du document. Ce point est illustré sur la figure 8.3.a) où l'activité, c'est-à-dire la consommation, est symbolisée par un éclair. Cet état de fait ne facilite pas la paramétrisation des composants d'une bibliothèque en vue d'estimer l'énergie ou la puissance.

Comme nous l'avons déjà évoqué précédemment, cette difficulté est très souvent contournée en choisissant une distribution uniforme ou bruit-blanc[2] de vecteurs d'entrée pour caractériser une bibliothèque de modules. Cette solution très simplificatrice permet de se ramener à une logique d'estimation similaire à celle employée pour la surface ou le délai, mais peut entraîner une imprécision importante dans les mesures globales, car elle ne correspond pas le plus souvent à la réalité de l'activité intrinsèque d'un circuit.

La distance de Hamming est une mesure particulièrement intéressante dans ce contexte. Elle traduit entre deux vecteurs consécutifs de N bits, le nombre de transitions de bit à bit entre ces deux vecteurs. Exemple : $H(11010011, 10110100) = 5$. La distance de Hamming moyenne donne la moyenne de la distance de Hamming entre chaque vecteur d'une séquence de n vecteurs appliqués successivement.

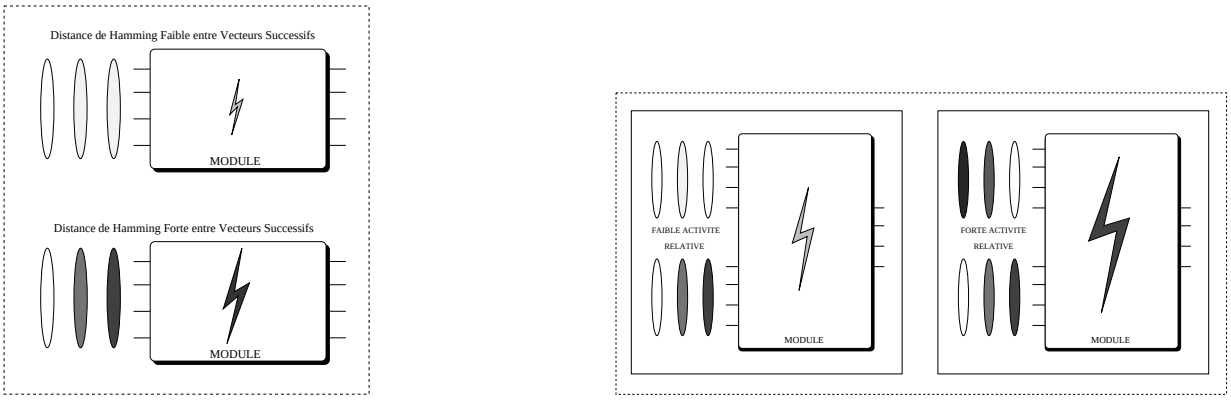
Soit x l' opérande d'un module quelconque; la distance de Hamming moyenne pour cet opérande est donnée par :

$$H(x) = \lim_{n \to +\infty} \frac{\sum_{i=1}^{n} H(x_i, x_{i-1})}{n} \tag{8.1}$$

où x_i est la valeur de l'opérande x durant le cycle i, soit la valeur du ième vecteur de la séquence.

Remarque :

[2] séquence entièrement aléatoire



a) b)

Figure 8.3: Illustration de la dépendance de l'activité intrinsèque d'un circuit en fonction a) de la distance entre les vecteurs d'une séquence appliquée et b) de l'activité relative de ses opérandes.

Une séquence entièrement aléatoire de vecteurs de largeur N bits présentera une distance de Hamming moyenne égale à :

$$H_{Aléatoire}(x) = \frac{N}{2} \tag{8.2}$$

Il est donc aisé de relier distance de Hamming moyenne et activité moyenne voire distributions statistiques des commutations. La distance de Hamming moyenne contient en fait la synthèse des informations sur la taille des entrées en bit et l'activité. Un autre point à considérer pour les modules ayant plus d'une opérande est l'activité relative entre celles-ci : l'activité interne et donc l'énergie consommée est d'autant plus grande que l'activité relative entre chaque opérande augmente comme illustré sur la figure 8.3.b).

8.3.2 Stratégie de macro-modélisation en termes de consommation

La stratégie de macro-modélisation adoptée ici, considère trois types de représentation de l'énergie consommée :

Constante L'énergie d'un composant est donné comme un paramètre constant dans la bibliothèque. Cette représentation est réservée aux unités très simples, ou aux unités dont la consommation est globalement peu dépendante de l'activité sur ses entrées. Elle est aussi employée dans une stratégie d'estimation grossière mais rapide, autorisant le concepteur à donner les mesures qu'il estime plus ou moins correctes de la consommation d'un composant.

Expression L'énergie d'un composant est contenue dans une expression affine simple. Cette représentation est réservée aux unités simples et surtout régulières, dont l'énergie tracée en fonction de certains paramètres se rapproche d'une droite, comme multiplexeurs, registres et certaines unités fonctionnelles. L'énergie E est donnée en fonction de l'activité en entrée et de la taille des entrées, c'est-à-dire de la distance de Hamming moyenne qui contient ces deux paramètres. L'expression utilisée est alors $E = \alpha \times AHD + \beta$ où α et β sont des cœfficients déterminés à l'issue de simulations réalisées à bas niveau[3], et AHD[4]est la distance de Hamming moyenne en entrée du composant.

Liste de valeurs Cette représentation est réservée aux unités complexes, essentiellement aux unités fonctionnelles. Pour des composants complexes comme des unités fonctionnelles spécifiques issues par exemple d'une synthèse précédente, il est difficile voire impossible de trouver une fonction simple générant son énergie consommée. Dans un tel cas, la solution adoptée ici, est de caractériser chaque composant complexe par un ensemble de valeurs extraites de simulation à bas niveau. En bibliothèque, ces valeurs sont stockées et classées en fonction des activités correspondantes sur les entrées. L'activité considérée pour chaque entrée comprend deux termes, l'un pour les bits les plus significatifs[5], l'autre pour les bits les moins significatifs[6], cela afin de prendre en considération de façon particulière, l'effet important de l'activité des bits de poids fort sur la consommation dans le modèle de macro-modélisation. La démarche est similaire à celle décrite dans [60], mais il s'agit ici d'un modèle simplifié. L'activité d'un vecteur de bits en entrée d'un module est coupée en deux parties égales comme le représente la figure 8.4. b). Dans [60], l'activité est considérée de façon beaucoup plus détaillée, comme deux plateaux séparés par une rampe entre les points BP0 et BP1 comme l'illustre la figure 8.4. a).

Figure 8.4: Découpage de l'activité en entrée des modules

A noter d'autre part, qu'est associée à l'énergie consommée d'un module en fonction de l'activité de ses entrées, l'activité en sortie du module. Cela prendra tout son sens dans la

[3] c'est à dire au niveau porte logique, commutateur (modèle de transistor simplifié) ou layout

[4] Average Hamming Distance

[5] ou MSB (Most Significant Bits). Ce sont les bits de poids fort qui comportent notamment les bits de signe. Ces derniers ont une saveur particulière dans le cadre de la consommation dans la mesure où leur variation influe grandement sur l'activité interne des modules à l'entrée desquels ils varient.

[6] ou LSBs (Least Significant Bits). Ce sont les bits de poids faible.

section traitant de la propagation des activités au sein du chemin de données (cf. section 8.5 p. 139).

L'énergie moyenne consommée par chaque module est donc extraite de simulations à bas niveau, par exemple au niveau logique. Dans ce cas, il est préférable de simuler avec un modèle temporel le plus complet possible, et notamment capable de mettre en évidence les commutations non désirées pouvant intervenir au sein du module simulé. Il est d'autre part nécessaire d'avoir à sa disposition un générateur de stimuli qui ait la particularité de fournir des séquences dont la distribution des commutations est réglable. Il faut enfin simuler avec une séquence suffisamment importante pour que les mesures de consommation aient un sens. La méthodologie adoptée ici est illustrée sur la figure 8.5. Le module est stimulé avec des séquences de stimuli dont les statistiques de commutation sont fixées. Les commutations de chaque nœud interne du module décrit au niveau logique, sont fournies à l'issue de la simulation à un outil d'estimation donnant la consommation liées aux charges capacitives. La série d'outils de simulation et de synthèse logique de synopsys[7] a été employée à des fins de simulation et d'estimation au cours de ces travaux de thèse, afin notamment de caractériser les modules de bibliothèque.

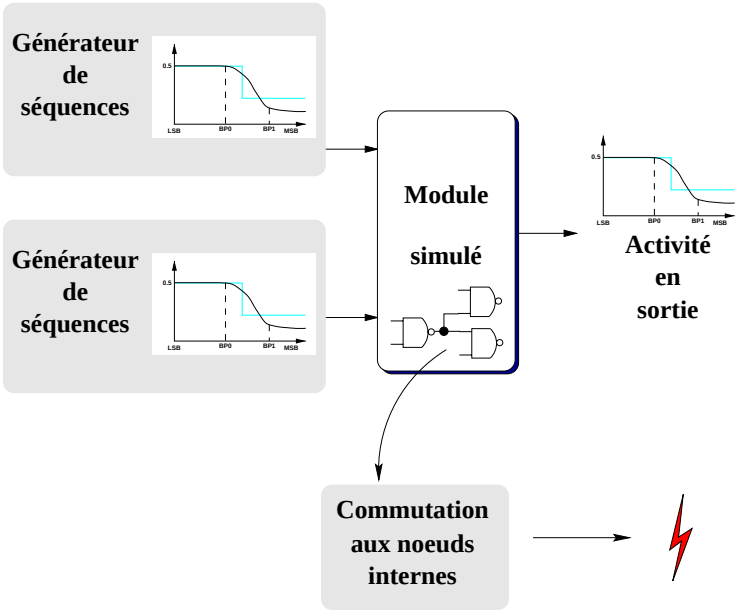


Figure 8.5: Méthodologie de caractérisation des modules de la bibliothèque

8.4 Image dynamique du circuit

Comme mentionné précédemment, la seule façon valable d'estimer la consommation d'un circuit donné passe par la connaissance de son activité intrinsèque qui conditionne l'importance

[7] VSS, Design Analyser et Design Power

de cette consommation. La méthode adoptée ici, consiste à extraire cette activité par le biais d'une simulation du circuit. Dans les contraintes attachées à l'estimation de la consommation sous-tendant ces travaux, l'indépendance vis-à-vis de simulations de plus bas niveau, niveau RTL compris, tient une place prépondérante. Il faut pouvoir estimer la consommation d'un circuit généré par la synthèse comportementale, sans sortir de la boucle de synthèse. C'est la raison pour laquelle simuler la description comportementale pré-synthèse et en extraire des informations suffisantes pour tenir compte de l'activité intrinsèque du circuit au niveau RTL post-synthèse est un choix qui s'impose. Deux types d'information sont extrapolés lors de cette simulation comportementale : les statistiques de commutations des données échangées au noeuds du circuit, et un comportement typique du circuit en termes d'exécution.

Dans la totalité des méthodologies où des données dynamiques issues de simulations sont employées, se pose le problème de la dépendance des résultats relativement aux stimuli fournis. Le principe général, adopté ici de-même, est que l'utilisateur a une connaissance intime du type de données manipulées par le circuit qu'il synthétise. La pertinence des stimuli qu'il fournit au circuit est donc laissée à son seul jugement. Une règle évidente veut que les données fournies au circuit donnent une observabilité de la totalité de ses états de fonctionnement. Cette règle est entachée d'un poids tout particulier dans le cas de l'estimation de la consommation.

8.4.1 Statistiques de commutation des données échangées

Elles sont obtenues à l'issue d'un traçage de la description VHDL comportementale à synthétiser. Ce traçage consiste à simuler cette description, en considérant toutes les valeurs prises successivement par chaque variable. On en déduit une activité moyenne de chaque variable, activité qui peut être ensuite transposée au circuit obtenu après synthèse comme l'activité en sortie des registres générés et donc en entrée des unités alimentées par ces registres.

Ceci est illustré par la figure 8.6. Celle-ci montre sur la gauche, la description comportementale VHDL d'un circuit, le GCD, donnant le pgcd (plus grand commun diviseur) de deux nombres entiers donnés en entrée. Sur la droite, est représenté le chemin de données du circuit après synthèse. Il s'agit d'un circuit très simple, typique d'un circuit orienté contrôle dans le sens où les données fournies conditionnent son comportement. En effet, suivant les entiers fournis, le calcul se fera en peu d'itérations (ex : 25 et 15) ou en un grand nombre d'itérations (ex : 99 et 2). Le nombre d'itérations est impossible à déterminer à l'avance, à moins d'exécuter l'algorithme à la main.

Durant cette simulation, la distance de Hamming moyenne est calculée pour chaque variable par comparaison de ses valeurs successives. L'objectif est d'obtenir un critère d'arrêt de la simulation, par le calcul du taux de convergence de la distance de Hamming moyenne. Le résultat d'un tel traçage est représenté sur la figure 8.7.a) où 3000 couples de vecteurs sont appliqués en entrée du circuit décrit au niveau comportemental. En l'occurence, la simulation aurait pu être interrompue à 2000 voire 1500, le nombre 3000 étant dû à la précision spécifiée. L'exécution du traçage donne pour les variables x et y

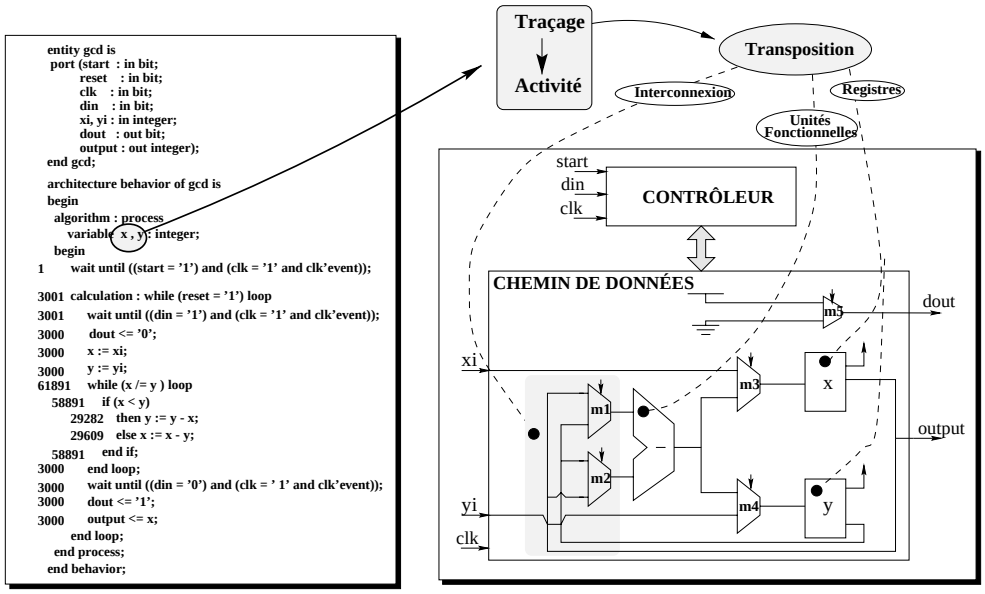
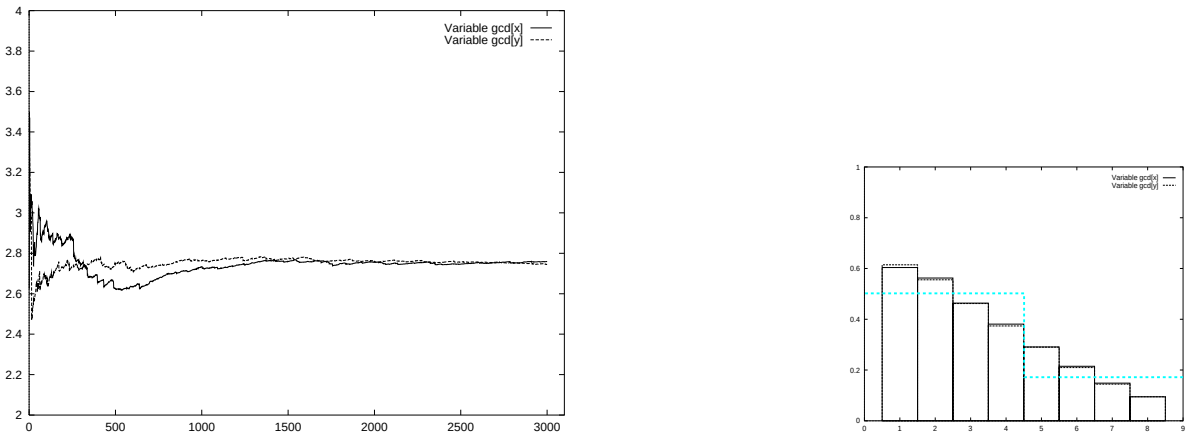


Figure 8.6: Traçage de la description comportementale VHDL

codées sur 8 bits une distance de Hamming moyenne convergeant vers environ 2.7 avec une précision (taux de convergence de la moyenne) d'environ 1/10000. Autrement dit, moins de 3 bits sur 8 commutent en moyenne entre chaque valeur prise par chaque chaque variable au cours de l'exécution du circuit. La figure 8.7.b) montre la distribution moyenne des commutations des deux variables et l'approximation considérée, liée au découpage du vecteur en deux parties, comme indiqué au cours de la section précédente.

A noter que pour cet exemple, utiliser une séquence de vecteurs entièrement aléatoire, soit une distance de Hamming moyenne de 4 pour des vecteurs de 8 bits, pour caractériser chaque module de la bibliothèque comme c'est souvent le cas, est loin de la réalité du circuit considéré : cela conduirait dans ce cas à une surévaluation de l'énergie consommée.

Une précision importante concerne la tansposition directe des activités moyenne des variables de la description comportementale aux sorties des registres de la description structurelle. Cette transposition directe est possible ici, car l'outil de synthèse considéré au cours de ces travaux génére un registre pour chacune des variables de la description initiale, sans chercher à minimiser leur nombre par un partage de registres entre plusieurs variables. Le type de circuits considéré, orienté contrôle, justifie en partie ce choix, par des opportunités de partage plus faibles. Dans le cas de circuits orientés flot de données, le partage de registres entre plusieurs variables afin de minimiser leur nombre est plus systématique. Dans ce cas, et dans l'idée d'exporter la méthodologie d'estimation ici décrite à des environnement de synthèse comportementale pratiquant le partage des registres, il faudra tenir compte de ce partage au cours de la transposition des activités, qui en devient bien moins triviale. En effet, chaque variable partageant un registre donné présentant un certain type d'activité, l'activité finale moyenne en sortie du registre partagé est le résultat de la succession des activités différentes de ces variables au cours du fonctionnement. Ce cas



a) b)

Figure 8.7: Extraction par simulation de l'activité moyenne des variables d'une description comportementale

de figure n'a pas été considéré au cours de ces travaux. Sa résolution dépend essentiellement de la façon dont les variables sont allouées et assignées aux registres au cours de la synthèse.

8.4.2 Comportement typique du circuit au cours de l'exécution

Le contrôleur correspondant à la description du pgcd de la figure 8.6 est représenté sous forme de graphe sur la figure 8.8. Une correspondance directe existe entre la description comportementale VHDL et le graphe, ainsi qu'il est illustré sur cette figure.

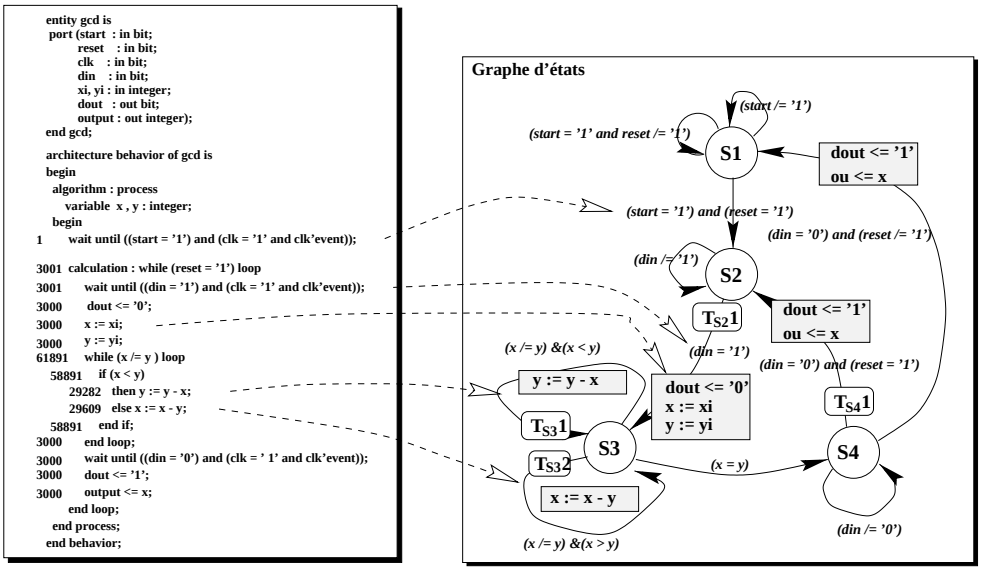


Figure 8.8: Profilage de la description comportementale VHDL

En appliquant la même séquence que précédemment au circuit, et étant donné les relations entre la description comportementale initiale et les opérations activées au niveau de chaque transition du contrôleur du circuit au niveau RTL, un profilage du code comportemental fourni les informations suivantes :

1. la probabilité d'exécution des transitions à partir d'un état donné : pour cet exemple, les probabilités de transition correspondant à $T_{S3}1$ et $T_{S3}2$ sont égales respectivement à 0.47 et 0.48 et reliées à la condition $if x < y$... (on retrouve dans ces deux valeurs la symétrie typique de la description); la probabilité de la transition correspondant à la condition $x = y$, quant à elle, est 0.05.

2. Le nombre d'exécutions moyen de chaque transition : c'est cette information qui est exploitée ici. Ainsi, pour un couple de valeurs qui provoque l'exécution du circuit, les instructions $y <= y - x$ et $x <= x - y$ sont exécutées en moyenne 9.8 fois chacune, ce qui correspond aux transitions rebouclantes $T_{S3}1$ et $T_{S3}2$ de l'état $S3$. Les autres transitions donnant lieu à des transferts au sein de la partie opérative, $T_{S4}1$ et $T_{S2}1$, sont elles, exécutées une fois par couple de valeurs.

Bien sûr, l'exemple utilisé ici est très simple et ne traduit pas certaines difficultés de transposition des données de profilage à la description RTL. Parmi celles-ci, la plus critique est relative à l'ordonnancement appliqué, qui brise la logique d'exécution séquentielle de la description comportementale. En effet, l'algorithme d'ordonnancement appliqué dans l'outil de synthèse comportementale utilisé au cours de ces expérimentations, qui appartient à la famille des algorithmes d'ordonnancement basés sur les chemins appliqués aux circuits orientés contrôle, génère des chemins en fonction de certaines contraintes comme les dépendances de données et les expressions conditionnelles[21], [111]. C'est ainsi qu'une séquence d'expressions VHDL comportementale peut finalement correspondre à plusieurs chemins et donc plusieurs transitions de la machine d'états finis générée. Les opérations correspondant à la séquence comportementale initiale sont évidemment exécutées le même nombre de fois avant et après synthèse (sinon, ce n'est plus le même circuit !). Cependant, ce nombre doit être découpé en autant de portions que de transitions finales. Pratiquement parlant, le plus simple est d'effectuer cette correspondance au cours du processus d'ordonnancement, pour peu que les données issues du profilage soient fournies en entrée de celui-ci, puisque c'est lui qui génère les chemins d'exécutions en se basant sur les expressions conditionnelles.

8.5 Propagation des activités intrinsèques

Afin d'employer la valeur d'énergie la plus exacte possible au cours du processus d'estimation, il est important de connaître l'activité moyenne sur les entrées de chaque unité de la partie opérative, afin d'extraire de la bibliothèque la valeur de l'énergie consommée correspondante la plus proche de la réalité. Il est aisé d'obtenir ces activités à partir d'une simulation de la description RTL du circuit, mais cela va à l'encontre du principe énoncé plus haut,

consistant à estimer la consommation du circuit sans sortir de l'environnement de synthèse de haut niveau.

La simulation comportementale permet d'extraire une partie de l'information nécessaire. A partir du traçage des variables de la description, l'activité moyenne des données stockées par celles-ci est extraite, activité transposée aux sorties des registres de la description RTL finale, et donc aux entrées des modules directement alimentés par les registres. Par ailleurs, l'activité des ports d'entrée du circuit est aussi connue, puisque l'on connaît l'ensemble des stimuli alimentant le circuit au cours de la simulation. A partir de ces informations, l'activité en entrée du restant des modules du chemin de données est extrapolée en propageant les activités connues à travers celui-ci. Considérant les sorties de registres et les ports d'entrée comme point de départ, leurs activités sont propagées à travers la partie opérative en largeur d'abord, jusqu'à atteindre les ports de sorties et les entrées des registres.

Au cours de la propagation, l'activité sur le port de sortie d'une unité donnée ne peut être calculée/propagée que lorsque les activités de la totalité de ses entrées sont connues. Deux types d'unités présentent un traitement particulier : les multiplexeurs et les unités fonctionnelles.

8.5.1 Propagation des activités à travers les multiplexeurs

Considérons le cas des multiplexeurs (mux). Leur principal désavantage est de briser la corrélation temporelle pouvant exister sur leurs entrées. La corrélation temporelle traduit une relation entre les vecteurs d'entrée se succédant au cours du temps. Un exemple classique de séquence de vecteurs présentant une forte corrélation temporelle, est celui d'une séquence issue d'un compteur. En termes de consommation, une entrée présentant une corrélation temporelle aura tendance à manifester une faible activité de commutation.

Ce problème est illustré simplement sur la figure 8.9, représentant un $MUX \times 2$ dont le signal de sélection est α et dont l'activité sur chaque entrée est donnée de façon simplifiée en deux parties, comme expliqué plus haut. Le problème consiste à estimer l'activité en sortie du multiplexeur connaissant les activités sur ses entrées.

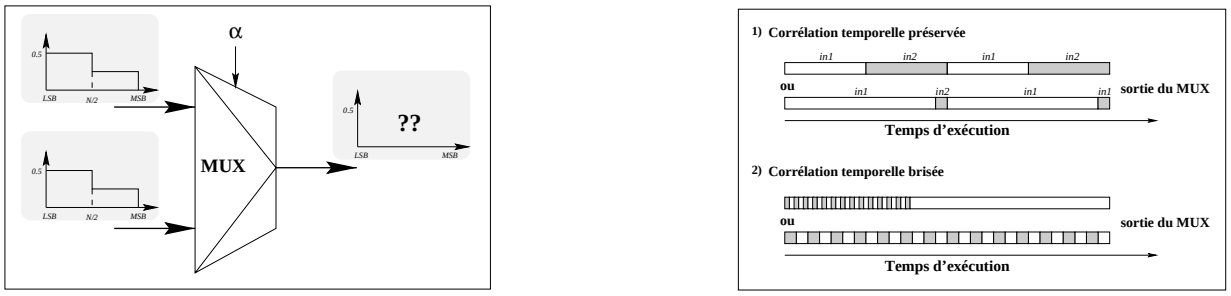


Figure 8.9: Exemple de propagation des activités à travers les multiplexeurs

Soient P_{in_i} la probabilité pour une entrée i d'être sélectionnée (ouverte) au cours de l'exécution, P_α la probabilité de commutation de α, P_{α_i} la probabilité que α soit à la valeur

sélectionnant l'entrée i, A_{in_i} et A_{out} les activités d'une entrée i et de la sortie. A strictement parler, $P_{in_i} = P_{\alpha_i}$. A présent, étudions les cas suivants :

1. $P_\alpha$est très petite, autrement dit le signal de sélection présente une activité faible ou moyenne au cours de l'exécution, comme illustré sur la figure 8.9.b.1) où sont représentés deux types de sorties possibles. Dans ce cas, on peut estimer A_{out} comme étant la somme des activités sur ses entrées, celles-ci étant pondérées par les probabilités pour les entrées correspondantes d'être sélectionnées au cours de l'exécution. C'est une approximation, mais elle est proche de la réalité dans le cas présent : le comportement du multiplexeur préserve dans une large mesure l'activité des entrées, dont la corrélation temporelle n'est que faiblement brisée.

$$A_{out} \equiv P_{in_1} \times A_{in_1} + P_{in_2} \times A_{in_2} = \sum_{i=1}^{N_{input}} P_{in_i} \times A_{in_i} \tag{8.3}$$

2. P_α traduit une activité frénétique du signal de sélection, comme montré sur les deux exemples de la figure 8.9.b.2). L'approximation ci-dessus (eq. 8.3) donne alors des résultats plus grossiers, dans la mesure ou l'activité en sortie s'éloigne des activités en entrées, dont la corrélation temporelle potentielle est partiellement brisée.

Afin d'appliquer cette formule, il est nécessaire de connaître les probabilités de sélection de chaque entrée au cours de l'exécution pour tous les multiplexeurs du circuit. Ces probabilités sont calculées à l'aide de la connaissance des fréquences d'exécution de chaque transition de la partie contrôle. En effet, en analysant la machine d'états finis, il est possible de savoir au cours de quelles transitions un multiplexeur donné est activé explicitement, et quelle est l'entrée sélectionnée dans chacune d'elles. Par ailleurs, une analyse plus approfondie donne les transitions au cours desquelles un multiplexeur est parasite, c'est-à-dire non invoqué explicitement, mais indirectement, au cours de transferts de données inutiles changeant la valeur de ses ports d'entrée de façon non-contrôlée (cf. section 8.5.3 plus bas). Le nombre de cycles d'exécution au cours desquels le multiplexeur est vivant, est égal à :

$$N_{vivant} = \sum_{k=1}^{N_{act}} N_{exec_k} + \sum_{l=1}^{N_{par}} N_{exec_l} \tag{8.4}$$

où N_{act} est le nombre de transitions au cours desquelles le multiplexeur est explicitement activé, N_{par} est le nombre de transitions au cours desquelles le multiplexeur est parasite, N_{exec_k} et N_{exec_l} sont les nombres moyens de fois où les transitions k et l sont exécutées au cours du fonctionnement du circuit.

La probabilité pour une entrée d'être ouverte est donnée par:

$$P_{Mux_x}(in_i) \equiv \frac{\sum_{k=1}^{N_{in_i\,ouverte}} N_{exec_k}}{N_{vivant}} \tag{8.5}$$

où $N_{in_i\ ouverte}$ est le nombre de transitions au cours desquelles le multiplexeur est activé et son entrée in_i ouverte. On a la relation $\sum_{i=1}^{N_{input}} = N_{act}$ où N_{input} est le nombre d'entrées du multiplexeur.

Dans le cas où le multiplexeur est parasite, il est difficile voire impossible de savoir quelle est l'entrée du multiplexeur ouverte et donc susceptible de provoquer un changement de valeur sur les ports de sortie : cela dépend du ou des états précédents, et les possibilités sont nombreuses. Le multiplexeur peut alors propager les activités, ou non (cf. 8.5.3 plus bas). Le parti pris est alors de considérer le pire cas : le multiplexeur est considéré comme actif et l'activité considérée est la plus forte des activités présentées par les entrées du multiplexeur. La probabilité qu'un multiplexeur x soit parasite au cours du fonctionnement du circuit est donnée par:

$$P_{Mux_x}(par) \equiv \frac{\sum_{l=1}^{N_{par}} N_{exec_l}}{N_{vivant}} \tag{8.6}$$

Finalement, l'activité en sortie d'un multiplexeur x est donnée par la formule suivante:

$$A_{mux}(out) = \sum_{i=1}^{N_{input}} P_{Mux_x}(in_i) \times A_{in_i} \quad + P_{Mux_x}(par) \times max(A_{in_i})$$

où A_{in_i} est l'activité sur l'entrée i du multiplexeur.

Ce sont ces formules qui sont appliquées au cours de la propagation des activités à travers les multiplexeurs.

8.5.2 Propagation des activités à travers les unités fonctionnelles

Il convient de distinguer les unités fonctionnelles mono-opération, des unités multi-opérations plus commodément appelées ALU. Dans le cas des unités simples, l'activité propagée en sortie d'une unité fonctionnelle mono-opération, est considérée comme l'activité de son unique entrée ou l'activité la plus forte de ses entrées. Dans le cas des unités complexes, la valeur de l'activité en sortie est extraite de la bibliothèque, car cette information y est stockée au même titre que l'énergie consommée, en fonction des activités sur ses entrées.

Pour ce qui est des unités fonctionnelles multi-opérations, le problème se pose dans des termes très proches de celui se posant pour les multiplexeurs. En effet, l'activité propagée en sortie va alors dépendre à la fois de l'activité en entrée, et de l'opération sélectionnée à un instant donné de l'exécution du circuit. Sachant qu'à une certaine activité sur les entrées et à une opération donnée, est associée une certaine activité en sortie, l'activité moyenne en sortie est calculée comme étant la moyenne des activités en sortie pour chacune des possibles opérations de l'unité fonctionnelle, pondérée par la probabilité de sélection de celles-ci au cours d'une exécution du circuit. Cela vaut pour l'activation explicite de l'unité fonctionnelle. De façon similaire au cas des multiplexeurs, lorsque l'unité considérée est parasite, le choix se porte sur l'opération consommant le plus (pire cas) et son activité

en sortie correspondante. Cette façon de procéder est résumée par la formule suivante, donnant l'activité en sortie propagée pour une unité multi-opérations :

$$A_{FU}(out) = \sum_{i=1}^{N_{operation}} P_{FU}(op_i) \times A_{op_i}(out) \quad + P_{FU}(par) \times A_{max(E_{op_i})}$$

$P_{FU}(op_i)$ est la probabilité que l'opération op_i soit explicitement sélectionnée au cours d'une exécution du circuit. Elle se calcule de façon similaire à celle employée pour calculer la probabilité pour une entrée d'un multiplexeur d'être sélectionnée (cf. équation 8.5). $P_{FU}(par)$ est la probabilité que l'unité soit parasite au cours d'une exécution et se calcule de façon identique à l'équation 8.6. Enfin $A_{op_i}(out)$ est l'activité en sortie de l'unité fonctionnelle dans le cas où l'opération op_i est sélectionnée, et $A_{max(E_{op_i})}$ est l'activité en sortie de l'opération consommant le plus pour la valeur d'activité des entrées considérée. Ces deux valeurs sont issues de la bibliothèque de composants paramétrés.

8.5.3 Détermination des unités parasites

Au cours du fonctionnement d'un circuit, un certain nombre d'unités du chemin de données consomment inutilement. Cette consommation inutile est due aux effets collatéraux de transferts de données, activant des unités ne participant pas aux calculs voulus. Il est important de tenir compte de cette consommation inutile si l'on veut obtenir une image la plus réaliste possible au cours de l'estimation, et c'est ici le parti pris.

La détermination des unités parasites s'effectue en deux phases :

1. Analyse de la machine d'états finis et extraction des transferts ouverts au cours de chaque cycle d'exécution élémentaire. Par ce moyen, les unités officiellement impliquées dans les transferts sont connues.

2. Parcours en profondeur d'abord du chemin de données à partir des unités initiales des transferts intervenant au cours d'un cycle élémentaire, et recherche de chemins divergents. Toutes les unités rencontrées sur les chemins divergents sont des unités parasites. Le parcours des chemins divergents stoppe à la rencontre d'un registre ou d'un port de sortie.

Ce procédé est illustré sur la figure 8.10.a), où l'opération $c = FU1(a, b)$ est explicitement autorisée au cours du cycle d'exécution correspondant. Un chemin divergent à partir du registre *a,* provoque un changement potentiel sur le second port d'entrée de l'unité FU2 qui est alors parasite, de même d'ailleurs que le réseau d'interconnexions impliqué.

La figure 8.10.b) propose une variante à la situation précédente : cette fois, la sortie de *a* diverge vers un multiplexeur. A strictement parler, on ne peut pas savoir si l'entrée du multiplexeur ouverte est justement celle provenant de *a*. Mais, à la lumière du principe de pire cas, le multiplexeur et l'unité FU2 sont là encore considérés comme parasites, bien qu'ils ne le soient que potentiellement. A noter que ce dernier cas illustre une façon

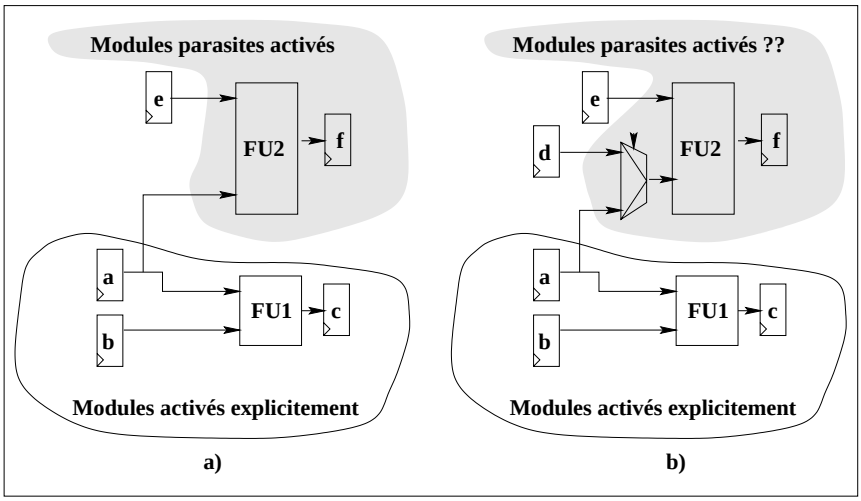


Figure 8.10: Activation d'unités parasites

efficace d'éviter une part de la consommation due aux unités parasites. En effet, puisque le multiplexeur considéré ne fait pas partie des unités impliquées au cours du transfert, rien n'interdit que l'on s'assure que celui-ci ne propage pas les données en forçant son signal de sélection à s'ouvrir sur le registre d, qui reste inchangé. Par ce simple biais, ce multiplexeur, de même que l'unité $FU2$ et leur interconnexion, ne sont plus parasites [109].

8.6 Estimation de la consommation

Le procédé d'estimation vise deux cibles. La première est la répartition de l'énergie consommée moyenne entre les cycles élémentaires d'exécution. Un cycle élémentaire d'exécution correspond à une transition de la machine d'états finis du contrôleur. Certains cycles de calcul consomment plus d'énergie que d'autres, et cette estimation se donne pour objectif de présenter au concepteur le poids énergétique de chacun d'eux. Accompagné de cette répartition de l'énergie, le poids en terme d'exécutions est fourni par la donnée du nombre moyen d'exécutions de chaque cycle élémentaire au cours d'un fonctionnement typique du circuit. Cela permet de mettre en balance l'énergie moyenne consommée au cours d'un cycle et la part de celui-ci dans le fonctionnement global. La seconde cible est la puissance consommée moyenne du circuit. Le procédé d'estimation dans son ensemble utilise les informations issues des simulations comportementales, de la caractérisation des composants de la bibliothèque et de la propagation des activités dans le chemin de données.

8.6.1 Estimation de la répartition de l'énergie consommée moyenne entre les cycles élémentaires d'exécution

Les parts respectives des composants du chemin de données, du réseau d'interconnexions et de l'horloge sont considérées au cours de l'estimation de l'énergie moyenne consommée

au cours d'un cycle d'exécution.

8.6.1.1 Chemin de données

Par chemin de données, le procédé entend les unités qui le composent. Au cours d'une transition de la machine d'états finis, sont activées les unités entrant en jeu dans le calcul voulu d'une part, et les unités parasites s'il y en a d'autre part.

Et de façon générale, si i est le cycle d'exécution courant, nous aurons:

$$E_{DP}(i) = \sum_{k=1}^{N_{vivants}} E_k \tag{8.7}$$

où N_{vivant} est le nombre de composants vivants au cours du cycle i, E_k étant l'énergie du composant k.

L'énergie de chaque composant est issue de la bibliothèque, et sa valeur dépend de l'activité sur les entrées du composant, activité qui est connue partout dans le circuit à l'issue de l'étape de propagation décrite précédemment. La façon dont l'énergie est extraite pour les composants complexes est un peu particulière et mérite de s'y attarder. Les composants complexes voient leur énergie stockée en bibliothèque sous forme de listes de valeurs données en fonction de l'activité sur les entrées. Sachant que l'activité aux portes d'un composant dans le circuit n'a pas forcément son équivalent en bibliothèque, il faut trouver le moyen de fournir une valeur présente en bibliothèque la plus proche possible de la valeur réelle. Dans cet objectif, le concept de distance entre activités est employé. Celle-ci est calculée comme la moyenne de la valeur absolue des différences entre l'activité de chaque entrée du composant dans le circuit, et l'activité de chaque entrée d'une position en bibliothèque. Les bits de poids fort et de poids faible sont distingués dans cette opération dont les formules sont les suivantes :

$$AD_{MSB} = \frac{\sum_{i=1}^{N_{input}} |A_{MSB}(in_i) - A_{lib_{MSB}}(in_i)|}{N_{input}}$$

$$AD_{LSB} = \frac{\sum_{i=1}^{N_{input}} |A_{LSB}(in_i) - A_{lib_{LSB}}(in_i)|}{N_{input}}$$

où AD_{MSB} et AD_{LSB} sont respectivement les distances entre activités pour les bits de poids fort et de poids faible, $A_{MSB}(in_i)$ et $A_{LSB}(in_i)$ les activités des bits de poids fort et de poids faible de l'entrée in_i du module, et $A_{lib_{MSB}}(in_i)$ et $A_{lib_{LSB}}(in_i)$ les activités des bits de poids fort et de poids faible de l'entrée in_i du module en bibliothèque. La position en bibliothèque pour laquelle la distance entre activités est la plus faible correspond à la valeur de l'énergie choisie. La comparaison est d'abord effectuée entre les bits de poids fort, dont l'influence sur la consommation est notoirement plus grande, puis entre les bits de poids faible. La figure 8.11 illustre ce procédé de recherche de la valeur la plus adéquate en bibliothèque.

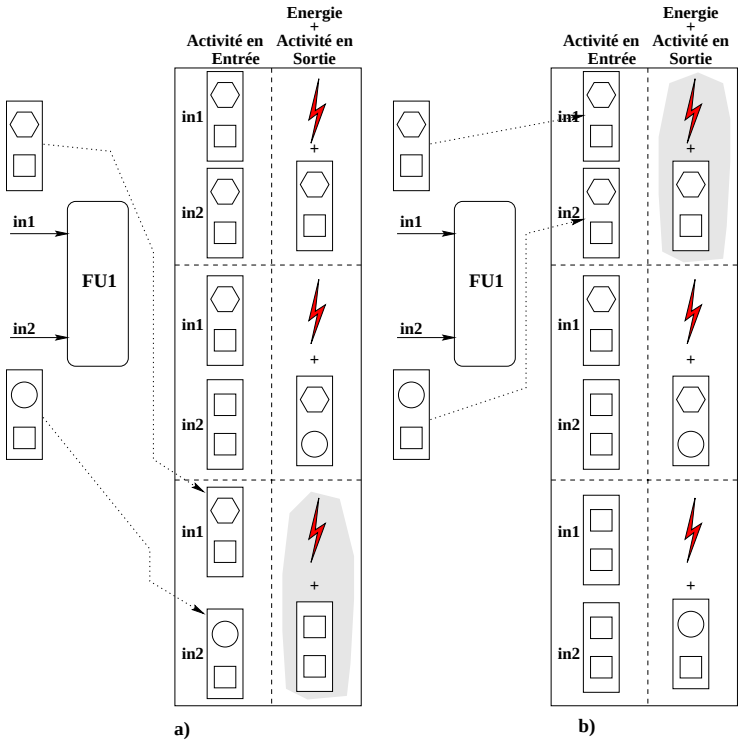


Figure 8.11: Recherche de la valeur d'énergie la plus pertinente en bibliothèque pour un module complexe

Sur la figure 8.11.a), les activités aux portes du module trouvent une exacte correspondance en bibliothèque, ce qui implique une distance entre activités nulle. Le choix coule alors de source. Par contre, ce n'est pas le cas sur la figure 8.11.b). Le choix se porte alors vers celui donné par la distance entre activités la plus faible[8].

8.6.1.2 Réseau d'interconnexions

Deux sortes d'interconnexions sont distinguées au cours du processus d'estimation : le réseau d'interconnexions entre les composants du chemin de données assurant les transferts de données, et les interconnexions entre la partie opérative et la partie contrôle supportant les signaux de contrôle. A chaque type est associé une capacité moyenne dont la commutation provoque une certaine consommation au cours des transferts de données.

Dans le premier cas, l'étape de propagation des activités assure la connaissance sur toutes les connexions inter-composants d'une activité moyenne. D'autre part, les chemins activés et parasites ouverts sont connus, et cela pour chaque cycle d'exécution. Combiné avec la donnée d'une capacité moyenne d'interconnexion inter-composant, il est aisé d'en déduire la part de l'énergie consommée par ce réseau au cours de chaque cycle. Comme

[8]Il est implicite au cours de cette recherche de correspondance, qu'il existe une relation d'équivalence, relayée par l'expérience, entre l'activité constatée aux portes du composant et l'énergie consommée. Dans le cas contraire, cette recherche n'a pas de sens.

l'on connaît par ailleurs les signaux de contrôle activés, l'énergie dépensée par le réseau d'interconnexion au cours du cycle d'exécution k est donnée par l'équation suivante:

$$E_{Interconnexion}(k) = \frac{1}{2} \times V_{dd}^2 \times \left[\sum_{i=1}^{N_{act+par}} C_{Net1} \times A_i + N_{controls} \times C_{Net2} \right]$$

où $N_{act+par}$ est le nombre de connexions inter-composants vivantes, c'est à dire activés et parasites, A_i est l'activité moyenne portée par la connexion i, exprimée comme la distance de Hamming moyenne, $N_{controls}$ est le nombre de signaux de contrôle envoyés et C_{Net1} et C_{Net2} sont respectivement les capacités moyennes des connexions inter-composants et des connexions entre parties contrôle et opérative.

8.6.1.3 Horloge

L'horloge correspond à un arbre, dont la consommation peut avoir une influence loin d'être négligeable en raison de la charge capacitive qu'il représente et de sa fréquence de commutation (deux fois par cycle) [133, 130]. Un exemple connu est le processeur Alpha, de DEC, dont l'horloge totalise 32% de la puissance consommée.

Dans le cas présent, chaque branche de l'arbre d'horloge est relié à une bascule, le nombre de ces dernières conditionnant l'énergie dissipée. Si l'on considère une capacité de charge moyenne $C_{bascule}$ associée à chaque branche, il suffit de multiplier celle-ci par le nombre $N_{bascule}$ de bascules dans le circuit pour connaître $C_{SW_{Horloge}}$, la capacité chargeant l'ensemble de l'arbre d'horloge. Dans le calcul de $N_{bascule}$, sont considérés les registres du chemin de données, mais aussi les bascules présentent dans la partie contrôle et servant à stocker la valeur de l'état courant. Le nombre de bascules dans le contrôleur est déduite du nombre d'états N_{state} par la formule $INT[log_2 N_{state}]$. La part de l'énergie consommée par l'arbre d'horloge est alors donnée par:

$$E_{Horloge} = \frac{1}{2} \times V_{dd}^2 \times \sum_{i=1}^{N_{bascules}} C_{SW_{Horloge}} \times 2$$

sachant que cet arbre commute deux fois à chaque cycle d'exécution.

8.6.1.4 Contrôleur

Bien que le contrôleur ne soit pas considéré au cours de l'estimation, quelques mots à son propos.

La partie contrôle est un bloc dont la structure finale n'est réellement connue qu'à partir du niveau porte. Par ailleurs, les informations dont on dispose au niveau auquel cette méthodologie d'estimation se place sont peu nombreuses. Pour inclure le contrôleur dans un outil d'estimation, quelle que soit la cible (surface, performances temporelles ou consommation), une approximation grossière s'impose.

Certains travaux réduisent les difficultés en supposant l'implémentation du contrôle dans une PLA[9] ou une ROM. L'avantage est la régularité de cette structure, qui permet d'extraire de mesures à bas niveau, des équations traduisant la consommation en fonction des paramètres connus à haut niveau : nombre d'entrées, nombre de signaux de contrôle, nombre d'états.

Dans le cas où l'implémentation est sous forme de portes logiques, la démarche adoptée ci-dessus est plus difficile. On peut alors essayer d'estimer le nombre de portes composant le bloc final et décider, comme mentionné dans [36], que 50% de celles-ci commutent à chaque cycle élémentaire. Les expérimentations effectuées au cours de ces travaux tendent plutôt à montrer que cela est surestimé, et que le nombre réel est plus proche de 25%. En connaissant la valeur de la capacité moyenne par porte, il est aisé alors d'obtenir l'énergie consommée approximative.

Un autre moyen consiste à découper la partie contrôle comme illustré sur la figure E.5 (cf. annexe E), en registre d'états, unités de comparaison, logique combinatoire calculant l'état suivant et logique combinatoire calculant les sorties, et de déterminer pour chacune d'elles des équations donnant leur énergie moyenne consommée. L'avantage est que cela permet d'isoler chacune des parties, qui ne seront pas forcément impliquées au cours de tous les cycles d'exécution. Par exemple, si au cours d'un cycle, l'état suivant est identique à l'état présent, et que les entrées changent, alors seule la logique combinatoire calculant les sorties agit.

L'intégration de la partie contrôle n'ayant pu se faire dans le modèle présent, celle-ci pourra fructueusement faire l'objet de travaux futurs

8.6.1.5 Synthèse

L'énergie consommée moyenne au cours de chaque cycle d'exécution est obtenue en ajoutant les trois parties détaillées précédemment :

$$E(i) = E_{DP}(i) + E_{Interconnexion}(i) + E_{Horloge}$$

A l'utilisateur, est fournie la vision dans son ensemble du poids énergétique de chaque cycle d'exécution. Il peut d'autre part accéder à la répartition au sein du cycle de l'énergie consommée en fonction des trois parties qui la composent.

8.6.2 Estimations globales

En pondérant l'énergie $E(i)$ consommée au cours de chaque cycle i, par le nombre moyen d'exécutions $n(i)$ de celui-ci, est obtenue l'énergie moyenne globale consommée durant le fonctionnement du circuit.

$$E_{Globale} = \sum_{i=1}^{M} E(i) \times n(i) \tag{8.8}$$

[9]Programmable Logic Array

où M est le nombre de cycles élémentaires d'exécutions, ou encore le nombre de transitions de la machine d'états finis.

Par la connaissance de la fréquence de fonctionnement du circuit, donc de la période T de chaque cycle, le temps d'exécution typique du circuit est:

$$T_{Global} = \sum_{i=1}^{M} T \times n(i) \tag{8.9}$$

A l'aide de ces deux valeurs (8.8 and 8.9), le calcul de la puissance moyenne consommée par le circuit est simplement:

$$P_{Average} = \frac{E_{Global}}{T_{Global}} \tag{8.10}$$

8.7 En résumé

Ce chapitre s'est focalisé sur la description de la technique d'estimation de la consommation mise au point au cours de ces travaux de thèse. Cette technique considère la description d'un circuit au niveau transfert de registre, générée par un outil de synthèse de haut niveau, et s'appuie sur une image dynamique du circuit, issue de simulation préalable au niveau comportemental. Cette technique s'appuie en outre sur une stratégie de macro-modélisation, prenant en considération l'activité en entrée des modules du circuit. L'image dynamique fournit des informations partielles sur l'activité interne du circuit, ainsi que des données sur la fréquence d'exécution des cycles d'exécution de celui-ci. Une étape préalable de propagation des activités permet d'obtenir une activité en chacun des noeuds du circuit. Cette activité dirige le choix en librairie de mesures plus précises de l'énergie consommée par chaque module fonctionnel. La méthodologie d'estimation donne l'énergie consommée au cours de chacun des cycles d'exécution, et fournit par ailleurs la puissance moyenne consommée par le circuit. Le chapitre suivant expose le résultat d'expérimentations effectuées en vue de vérifier la justesse des choix effectués.

Chapitre 9

Résultats d'expérimentations

9.1 Préambule

Ce chapitre se propose de présenter les résultats d'expérimentation obtenus sur trois circuits décrits au niveau comportemental, synthétisés au niveau RTL, puis au niveau logique. Un premier jeu de résultats expose la précision obtenue par la propagation des activités internes du circuit. Puis les résultats d'estimation de la consommation proprement dite sont dévoilés. Ils concernent l'estimation de l'énergie consommée au cours des cycles d'exécution, ainsi que l'estimation de la puissance consommée.

9.2 Prototype d'estimation

Un prototype implémentant la méthodologie d'estimation décrite dans le détail dans ce document, à été réalisé afin de tester pratiquement l'intérêt de celle-ci. Le prototype, baptisé *SYNRJ*, est un programme indépendant représentant environ 10000 lignes de code, écrit en C++. Ce prototype accepte en entrée la description du circuit RTL décrit dans un langage intermédiaire[1]. Outre les estimations fournies par *SYNRJ*, ce dernier permet à l'utilisateur de naviguer dans le circuit généré au moyen d'un "shell" de commandes ou de "scripts" de commandes. Il lui est ainsi possible de connaître par exemples les unités parasites dans chaque cycle élémentaire d'exécution. C'est par l'intermédiaire de ces commandes que les activités initiales, ainsi que les divers renseignements utiles, sont donnés à l'estimateur. Il est aussi possible de rentrer les activités en chaque point du chemin de données, par exemple après simulation du circuit RTL, se suppléant aux activités données par l'étape de propagation, bien que ce soit pas le but initial de cet outil. Les résultats d'expérimentations menées à l'aide de quelques exemples, sont condensés dans ce chapitre.

[1] il s'agit du langage SOLAR

9.3 Evaluation de la précision de l'estimation des activités internes

Mis à part les valeurs des activités en sortie des registres et sur les entrées principales, qui sont les valeurs exactes fournies par la simulation comportementale, les activités dans le reste du circuit sont la résultante de l'étape de propagation qui précède l'estimation proprement dite. Cette section s'attache à montrer la précision d'estimation des activités, et plus particulièrement en sortie des multiplexeurs.

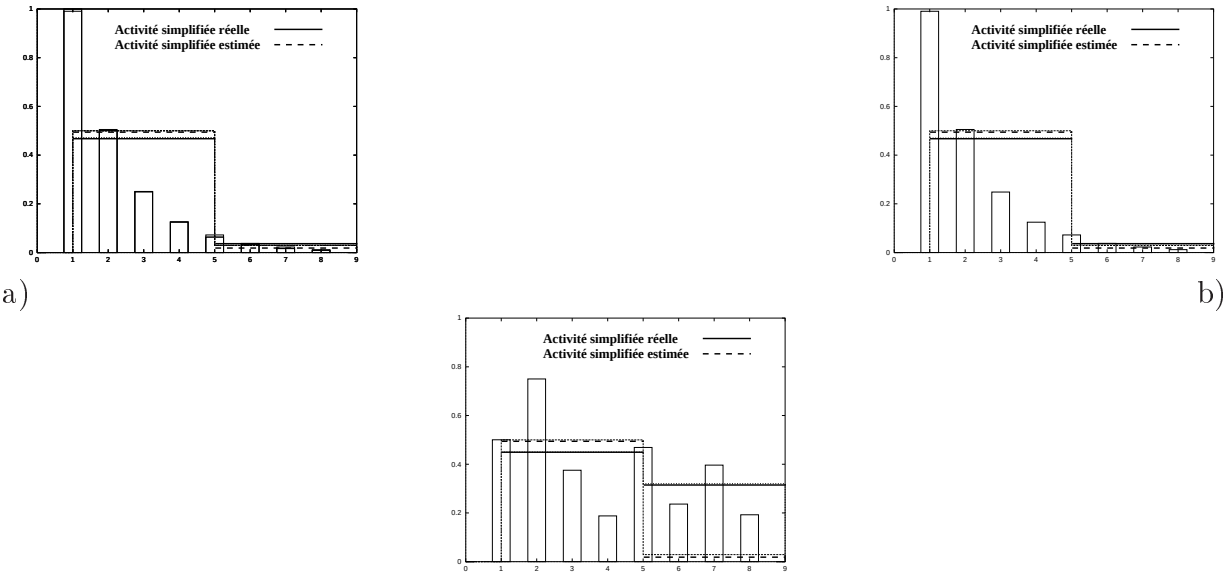


Figure 9.1: Résultats d'expérimentations donnant l'activité en sortie d'un multiplexeur x2 en fonction d'activités corrélées en entrée et de la variation du taux de commutation du signal de contrôle.

Les figures 9.1 et 9.2 mettent en évidence le résultat d'expérimentations réalisées avec un $MUX \times 2$, auquel ont été appliquées diverses séquences en faisant varier leur activité moyenne. La figure 9.1.a) montre l'activité en sortie du multiplexeur correspondant à des entrées très corrélées à la fois temporellement et spatialement (forte corrélation entre 2 vecteurs de même position dans chaque séquence) issues de compteurs synchronisés. L'activité est représentée sous la forme simplifiée définie plus haut : le premier plateau correspond à l'activité moyenne des bits de poids faible, tandis que le second donne l'activité moyenne des bits de poids fort. Chaque barre représente l'activité de chacun des bits du vecteur de sortie. L'application de la formule 8.3 donne une activité très proche de la réalité en dépit d'un taux de commutation élevé du signal de contrôle, égal à $\frac{1000}{2000}$ (une commutation après application de deux paires de valeurs). Cela est lié à la forte corrélation spatiale qui préserve la corrélation temporelle en sortie. La différence entre activités estimée et réelle reste faible sur le graphe 9.1.b), pour lequel les deux compteurs sont désynchronisés

(décalage de 50) ce qui brise la corrélation spatiale, mais le taux de commutation est plus faible (égal à $\frac{20}{2000}$, soit une commutation tous les 100 couples de vecteurs). Une différence est manifeste sur le graphe 9.1.c) pour lequel le décalage des compteurs est maintenu, mais le taux de commutation est poussé à la valeur extrême d'une commutation à chaque application d'un couple de vecteurs : la corrélation temporelle est brisée et l'application de la formule 8.3 donne un résultat approximatif.

Les figures 9.2.a) et 9.2.b) donnent des résultats similaires aux précédents, avec cette fois l'une des entrées présentant une distribution uniforme, l'autre toujours fortement corrélée temporellement. Le premier graphe correspondant à un taux de commutation relativement faible de $\frac{1}{100}$, l'autre extrême de 1 par paire. Même constatation pour ce qui est de la fiabilité de l'application de la formule 8.3. Les graphes 9.2.c) et 9.2.d) conduisent aux mêmes conclusions, avec cette fois des entrées pseudo-uniformes par palier, chaque plateau (LSBs et MSBs) correspondant à une distribution uniforme différente.

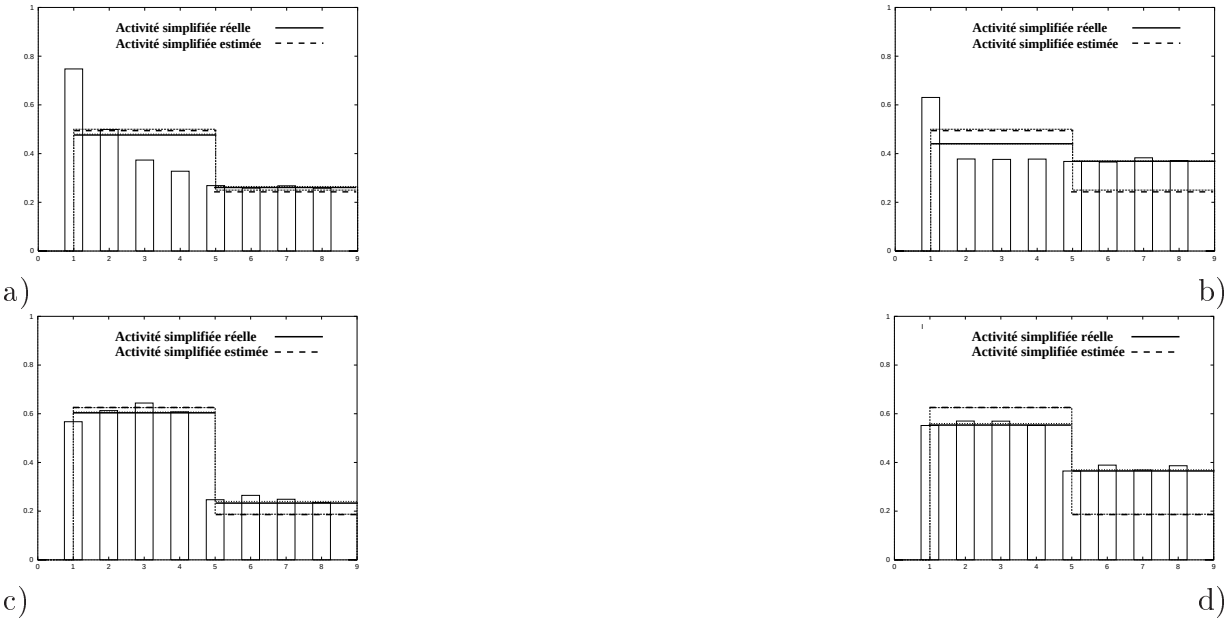


Figure 9.2: Résultat d'expérimentations sur l'activité en sortie d'un multiplexeur x2 en fonction de diverses activités en entrée et du taux de commutation du signal de contrôle.

Pour finir, le graphe de la figure 9.3 donne le tracé des deux distributions simplifiées (LSBs et MSBs), estimées et réelles, en sortie d'un multiplexeur,. Elles sont données en fonction du taux de commutation du signal de contrôle, et pour des entrées hautement corrélées temporellement, car issues de deux compteurs, mais décorrélées spatialement (compteurs décalés de 50). Les mesures se basent sur une séquence de 4000 couples de vecteurs. Comme prévisible, la divergence de l'activité estimée apparait d'autant plus forte que le taux de commutation augmente. A noter qu'il s'agit là, comme dans les expériences relatées ci-dessus, d'un cas limite : les activités ne présentent que rarement de telles corrélations temporelles. Et la figure 9.2 montre que l'approximation est bien meilleure, même en cas

de fort taux de commutation du signal de contrôle, lorsque les entrées sont faiblement corrélées.

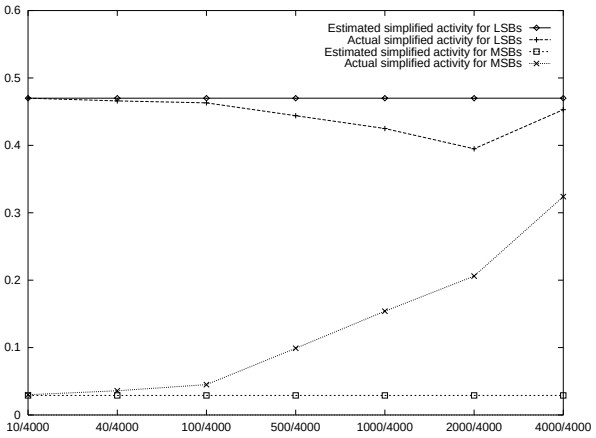


Figure 9.3: Activités en sortie d'un multiplexeur x2 en fonction du taux de variation du port de sélection. Entrées hautement corrélées temporellement (compteurs désynchronisés).

En résumé, l'application de la formule 8.3 pour l'estimation de l'activité en sortie des multiplexeurs au cours de la propagation des activités au sein de la partie opérative, formule appliquée effectivement dans cette méthodologie, fournit des résultats valables pour peu que la probabilité de commutation du signal de contrôle reste dans des proportions faibles voire moyenne. Il est intéressant de constater que l'activité en sortie d'un multiplexeur tends vers l'uniformité lorsque le signal de contrôle commute fréquemment, et ce, quelque soit l'activité sur les entrées.

Circuits	Erreur moyenne
GCD	5.1 %
ABMODN	6.4 %
BUBBLE	6.1 %
Moyenne des erreurs moyennes	5.9 %

Tableau 9.1: Résultat de l'estimation des activités (Distance de Hamming moyenne) en sortie des multiplexeurs.

La table 9.1 donne l'erreur moyenne entre activités estimées et réelles, c'est-à-dire issues de simulation RTL du circuit après synthèse à l'aide des mêmes stimuli, en sortie des multiplexeurs des trois circuits tests. Les activités sont exprimées en terme de distance de Hamming moyenne. Ces résultats montrent que l'estimation de l'énergie du réseau d'interconnexion inter-composants se base sur des estimations d'activités plutôt bonnes, y compris en sortie des multiplexeurs, puisque c'est la distance de Hamming moyenne qui importe dans ce cas.

Circuits	Erreur moyenne
GCD	8.5%
ABMODN	15.3%
BUBBLE	13.3 %
Moyenne des erreurs moyennes	12.4 %

Tableau 9.2: Résultat de l'estimation des activités en sortie des multiplexeurs, en distinguant LSBs et MSBs.

La table 9.2 donne l'erreur moyenne obtenue en sortie des multiplexeurs, mais cette fois en distinguant au cours du calcul les bits de poids fort et de poids faible. L'estimation est moins bonne que précédemment, en raison de la cassure des corrélations temporelles en sortie des multiplexeurs, cassure non prise en compte dans la formule utilisée. Cette erreur a un impact dans la sélection de la valeur d'énergie la plus adaptée dans la bibliothèque de composants, et se retrouve dans l'erreur moyenne globale de l'estimation de la consommation.

Si l'on considère le calcul de l'erreur issue de la propagation des activités sur l'ensemble du circuit, où les valeurs exactes issues de la simulation viennent tempérer les erreurs en sortie essentiellement des multiplexeurs, les résultats sont bien meilleurs, avec une erreur moyenne avoisinant les 2%. En conclusion, on peut affirmer sans trop s'avancer que la propagation des activités mise en œuvre ici est fiable. Cependant, il faut garder à l'esprit que ces résultats, relativement précis comparés à ceux issus de la simulation RTL, peuvent être éloignés des activités constatées au niveau porte, où interviennent des effets de transitions logiques non voulues impossibles à prendre en compte au niveau auquel se place cette méthodologie. Cette dernière, comme ses sœurs développées au cours de travaux antérieurs, assume donc un circuit parfaitement conçu au niveau porte.

9.4 Répartition de l'énergie en fonction des cycles d'exécution

Les exemples des figures 9.4, 9.5 et 9.6 représentent le résultat de l'estimation de la répartition de l'énergie consommée en fonction des cycles d'exécutions d'une part, et le poids sur le fonctionnement du circuit de chaque cycle, donné par le nombre moyen d'exécutions de ceux-ci. Un pic important correspondant au même cycle d'exécution sur les deux graphes de chaque figure, met en évidence la part de la consommation de ce cycle dans la consommation globale du circuit, indiquant à l'utilisateur de concentrer ses efforts sur la réduction de la consommation de celui-ci, par exemple en sélectionnant des composants consommant moins.

La figure 9.4.a) correspond à l'exemple simple du GCD utilisé à des fins d'illustration au cours des sections précédentes. Chaque barre représente l'énergie consommée par chaque cycle, exprimée en pico Joules (pJ). Du bas vers le haut, l'énergie correspondant aux

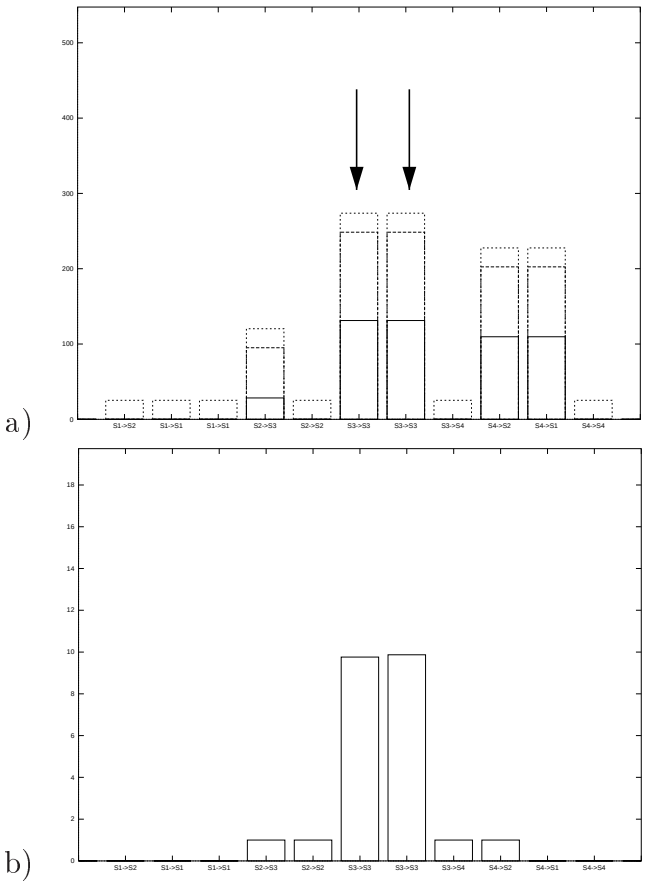


Figure 9.4: Répartition de l'énergie consommée par cycle et pondération par la fréquence d'exécution de chaque cycle : GCD.

composants du chemin de données, au réseau d'interconnexion et à l'horloge, est distinguée si elle existe. L'énergie dépensée par la partie contrôle ne fait pas partie de l'estimation. Les expérimentations menées indiquent que cette énergie est proportionnelle à celle calculée par la méthodologie d'estimation dont les résultats sont condensés ici. On distingue sur cette figure deux pics d'énergie indiquant la prépondérance des cycles correspondant sur la consommation du circuit, ce qui est confirmé par le graphe 9.4.b) qui indique que ces cycles sont les plus exécutés.

Les figures 9.5 et 9.6 montrent les résultats de l'estimation pour des circuits un peu plus conséquents que le premier. L'intérêt de pondérer la consommation de chaque cycle par son nombre d'exécutions est particulièrement évident sur la figure 9.5. Le pic d'énergie le plus conséquent (A) n'est que peu exécuté, tandis qu'un pic de moindre importance (B) pondéré par un grand nombre d'exécutions met en évidence le cycle le plus critique pour la consommation globale du circuit. Même constatation sur la figure 9.6 où les pics A et B donnent les cycles consommant le plus, tandis que le pic C et ses voisins immédiats sont manifestement les cycles ayant le plus grand impact sur la consommation globale du circuit.

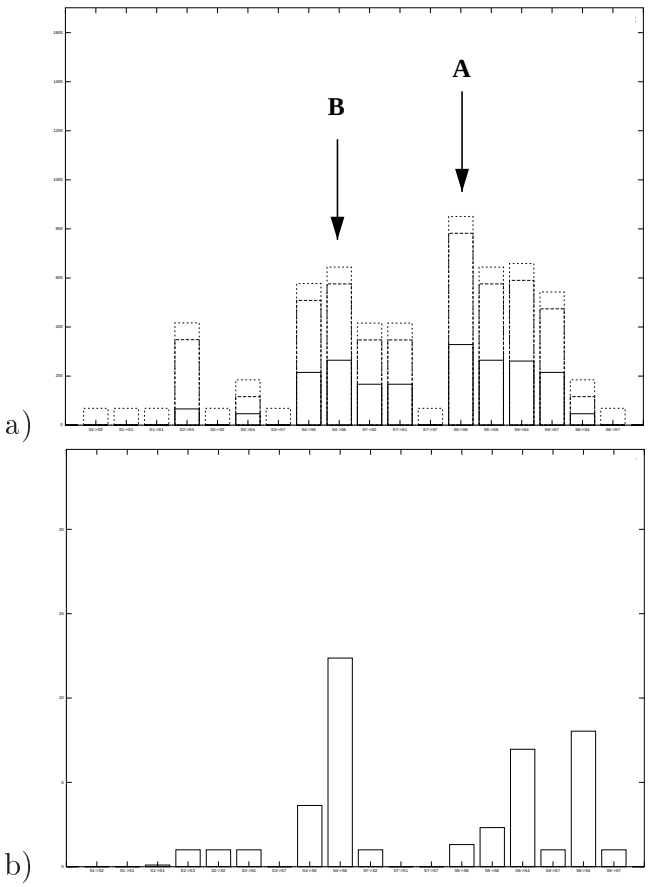


Figure 9.5: Répartition de l'énergie consommée par micro-cycle et pondération par la fréquence d'exécution de chaque cycle : ABMODN.

9.5 Puissance moyenne consommée et précision de l'estimation.

Les figures 9.7, 9.8 et 9.9 montrent les résultats de l'estimation de la puissance consommée par le circuit et la comparaison avec des résultats estimés au niveau porte logique[2]. Chaque graphe présente donc quatre couples de barres, la barre de droite dans chaque couple étant la puissance estimée à l'issue de la synthèse comportementale et celle de gauche la puissance issue d'une estimation au niveau porte logique. De gauche à droite sont représentées la puissance globale, la puissance consommée par les composants du chemin de données, la puissance consommée par le réseau d'interconnexions et l'horloge, et enfin la puissance consommée par l'horloge seule.

En ce qui concerne la répartition de la puissance consommée, une première constatation est l'importance relative de la consommation de l'horloge seule sur la consommation

[2]les estimations au niveau porte logique sont issues de l'emploi de la série d'outils de synopsys: VSS, Design Analyzer et Design Power

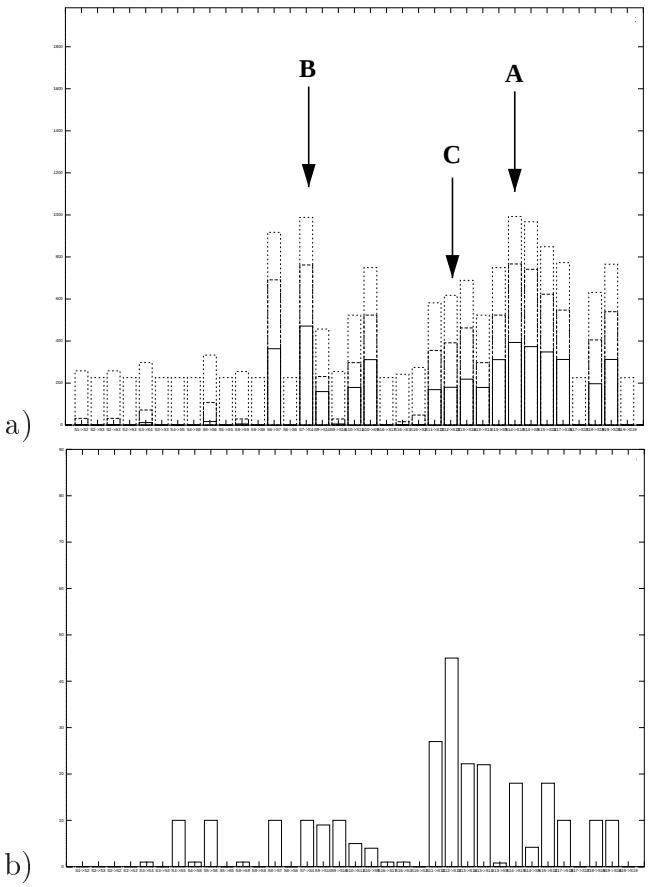


Figure 9.6: Répartition de l'énergie consommée par micro-cycle et pondération par la fréquence d'exécution de chaque cycle : BUBBLE.

globale. La seconde constatation mise en lumière par ces travaux, est la prépondérance de la consommation du réseau d'interconnexion (comprenant la consommation de l'horloge qui finalement en fait partie), sur la consommation due aux seuls composants de la partie opérative, et cela pour les trois exemples présentés ici.

Un point important qui n'apparaît pas sur ces graphes, est le poids de la puissance inutile consommée, c'est-à-dire liée à l'activation de parties du circuit par effet de bord durant l'activation des transferts de données. La table 9.3 donne le pourcentage de la puissance parasite sur la puissance consommée par les seuls composants du chemin de données. Ces résultats sont édifiants et confirment l'intérêt d'en tenir compte au cours de l'estimation. Ils confirment d'autre part la nécessité de travaux dans le sens d'une désactivation des composants inutiles au cours des transferts de données, voire de parties entières du circuit sachant que le réseau d'interconnexions parasite consomme lui aussi de façon importante. Un premier moyen très simple et déjà mentionné consiste par exemple à commander les multiplexeurs n'intervenant pas au cours des transferts, de telle sorte que ceux-ci ne propagent pas les données dans les parties du circuit inutiles. Un autre moyen repose sur l'utilisation d'unités fonctionnelles contrôlées, insensibles aux changements de

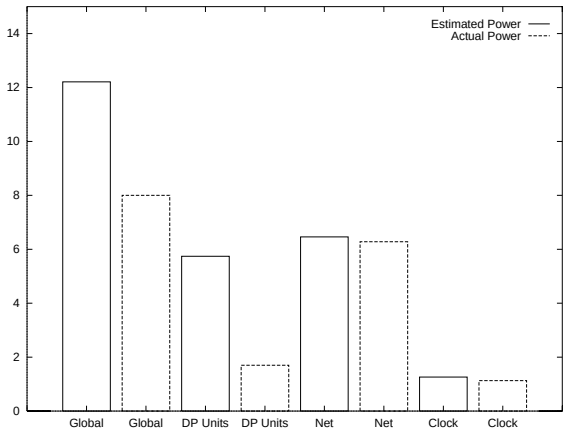


Figure 9.7: Répartition de la puissance consommée: GCD.

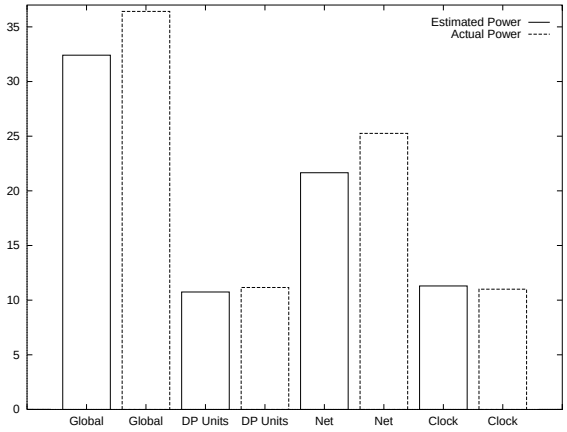


Figure 9.8: Répartition de la puissance consommée: BUBBLE.

valeurs en entrées par le rajout d'interrupteurs commandés. Alors que la première solution représente un coût nul, la seconde peut être onéreuse de part la nécessité de signaux de contrôles supplémentaires, accompagnés de la logique de calcul requise.

En ce qui concerne la précision des estimations, la puissance due aux unités du chemin de données est surestimée pour le GCD. Cela se traduit sur la table 9.4 par une erreur importante dans l'estimation de la puissance consommée par ce circuit, vraisemblablement liée à sa très petite taille, qui amplifie l'effet des nécessaires approximations effectuées au cours de l'estimation. L'erreur pour les circuits plus imposants que sont ABMODN et BUBBLE reste tout à fait acceptable. On peut supposer que l'erreur moyenne globale tourne autour des 10 % plutôt que des 17% obtenus en comptabilisant le GCD.

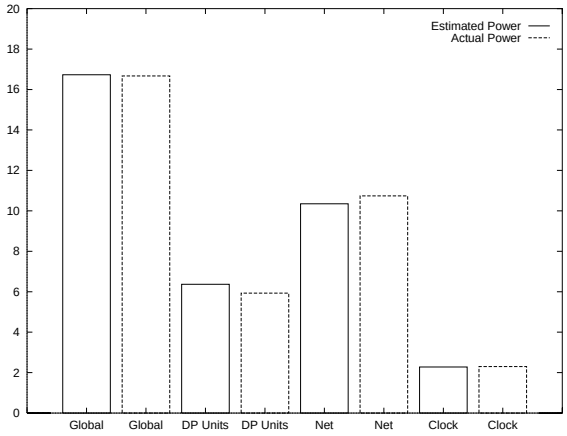


Figure 9.9: Répartition de la puissance consommée: ABMODN.

Circuits	Puissance parasite
GCD	10.7 %
ABMODN	34.1 %
BUBBLE	25.3 %
Moyenne	23.36 %

Tableau 9.3: Part de la puissance parasite dans la puissance consommée par les composants de la partie opérative.

9.6 En résumé

Un ensemble de résultats d'expérimentation ont été exposé dans ce chapitre, fournissant des mesures de la pertinence des choix ayant dirigé le développement de la méthodologie d'estimation de la consommation, et de la précision des estimations obtenues. La propagation des activités donne des résultats fiables, malgré l'approximation de certains calculs d'activités, notamment en sortie des multiplexeurs du circuit. La répartition de l'énergie consommée par cycle d'exécution, pondérée par la fréquence d'exécution de ces cycles, met directement en évidence les cycles critiques pour la consommation globale du circuit. La précision de l'estimation de la puissance moyenne, comparée à une estimation au niveau portes logiques, est globalement bonne. Elle semble proportionnelle à la taille du circuit, une taille plus importante diluant les approximations effectuées.

Circuits	Erreur moyenne
GCD	34.5 %
ABMODN	6.3 %
BUBBLE	11.0 %
Moyenne des erreurs moyennes	17.3 %

Tableau 9.4: Précision de l'estimation de la puissance.

Chapitre 10

Perspectives et conclusion

10.1 Adaptation de la méthodologie d'estimation

10.1.1 Evolution de la synthèse comportementale et du modèle d'architecture généré

Les travaux autour de la synthèse comportementale menés par l'équipe au sein de laquelle ces travaux se sont déroulés, ont subit une évolution constante. Les directions adoptées reposent tout d'abord sur la reconnaissance de la puissance de la synthèse logique disponible aujourd'hui, capable de synthétiser une description fonctionnelle au niveau transfert de registres très abstraite, en un circuit au niveau porte logique. Cette constatation a pu conduire à une évolution importante du modèle architectural généré.

La nouvelle approche s'appuie exclusivement sur l'étape d'ordonnancement pour générer une description RTL, sans passer par l'étape de génération d'architecture [122]. Cette description RTL du circuit consiste en une machine d'états finis de haut niveau, plate. L'allocation des opérations de base (+, -, * ...) est laissée aux bons soins de la synthèse logique : les étapes d'allocation et de génération des interconnexions, autrefois nécessaires, n'ont plus cours. Les opérations complexes, auparavant considérées comme des unités fonctionnelles, sont à présent traitées comme des appels de procédures qui sont mises à plat au cours de l'ordonnancement. Les boucles sont déroulées, tandis que les dépendances de données, déterminantes au cours de la méthodologie de synthèse précédente puisque génératrices d'états supplémentaires, sont résolues par le biais du chaînage d'opérations.

L'exemple de la figure 10.1 représente l'application de la nouvelle méthodologie de synthèse sur l'exemple simple du GCD, déjà mis à contribution dans ce document. Il s'agit de la machine d'états finis plate générée, synthétisable.

Comparée à la machine d'états finis (FSM) de la figure 8.8 p.138, celle de la figure 10.1 est totalement différente. Dans la première, les opérations effectuées au cours de chaque transition sont élémentaires, parallèles s'il y en a plusieurs, et sont traduites sous formes de signaux de commandes envoyés au chemin de données au sein duquel les chemins d'exécutions nécessaires sont ouverts. Les transitions de la nouvelle FSM conti-

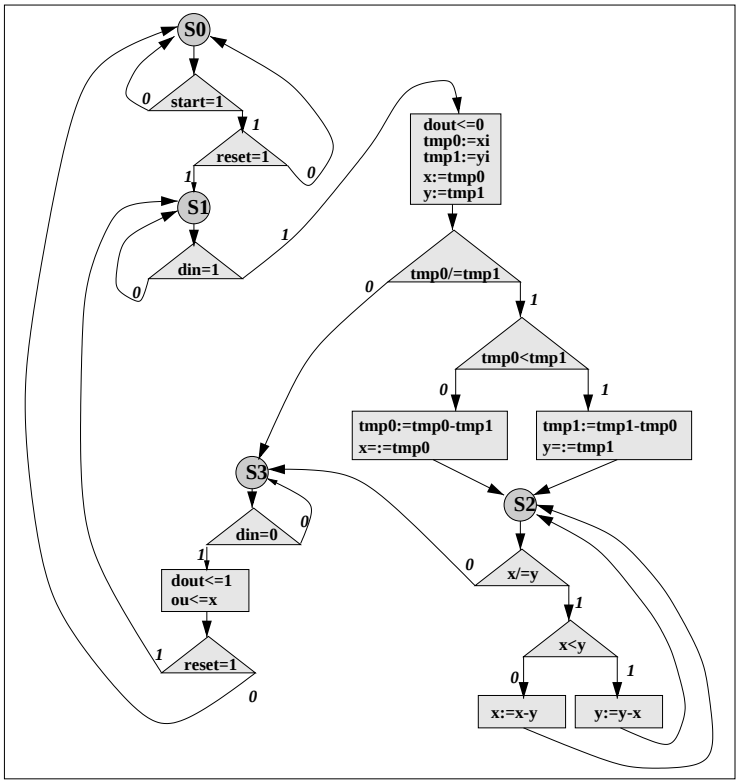


Figure 10.1: Machine d'états finis plate pour le gcd

ennent des graphes de flot de contrôle et de données (CDFG) entiers, correspondant à plusieurs chemins d'exécutions possibles.

10.1.2 Nécessité d'un nouveau modèle d'estimation de la consommation

Le modèle de consommation développé au cours de ces travaux s'appuie sur une architecture partie opérative - partie contrôle ou PO-PC, donnant accès aux détails architecturaux du circuit généré. Cette architecture et sa description étant suffisamment détaillées, les modules fonctionnels et leur nombre connus, les chemins d'exécutions accessibles, le modèle élaboré a pu prendre en compte notamment la part parasite de la consommation. La nouvelle architecture générée par l'outil de synthèse, par contre, est très abstraite. La topologie du circuit final après synthèse logique consiste en un seul bloc logique. Le nombre d'opérateurs fonctionnels ainsi que leur choix est laissé à la charge de la synthèse logique. Par conséquent, un modèle d'estimation se basant sur la seule machine d'états finis abstraite générée ne peut être que plus grossier. Les quelques points suivant sont le résultat de réflexions sur une extrapolation du modèle d'estimation mis au point au cours de ces travaux de thèse, à la nouvelle architecture en sortie de la synthèse comportementale. A vrai dire, les mêmes principes peuvent être employés :

- Une bibliothèque minimaliste, comportant les opérateurs élémentaires (+, -, *, comparaison, ...) est de rigueur. Ces opérateurs élémentaires peuvent être paramétrés suivant une stratégie de macro-modélisation similaire.

- Le même type d'information peut être extrait d'une simulation comportementale, à savoir :

  - Les statistiques relatives aux activités des données stockées par les variables de la description comportementale, extraites d'un traçage de ces variables au cours de l'exécution.
  - les fréquences d'exécution de chaque chemin d'exécution de la description comportementale, par le biais d'un profilage du code.

- La méthode d'estimation peut se baser sur un parcours des graphes de flot de contrôle et de données (CDFG) composant chacune des transitions de la machine d'états finis :

  - Chaque nœud du graphe est une opération : il peut être pondéré par son énergie consommée. Cette énergie est issue de la bibliothèque en fonction de l'activité moyenne de chaque variable intervenant dans l'opération.
  - Le chemin le plus lourd dans le CDFG d'une transition, c'est-à-dire le chemin de plus grand poids du graphe, donne la consommation pire cas de la transition. Appliqué à toutes les transitions, cela permet de connaitre l'énergie moyenne maximale qui peut être consommée en un cycle dans le circuit (poids maximum des poids des chemins les plus lourds). Il ne s'agit pas du pic d'énergie absolu, puisque l'énergie de chaque nœud est une énergie moyenne. Ce serait le cas si chaque nœud était pondéré par l'énergie maximale pouvant être consommée par l'opération correspondante.
  - La somme des poids (de l'énergie) de chaque chemin du CDFG d'une transition, pondérés par leurs probabilités d'exécution (la somme des probabilités d'exécution de tous les chemins du CDFG d'une transition vaut 1), donne l'énergie moyenne consommée lorsque cette transition est exécutée. Les probabilités d'exécution sont extrapolées à partir des données issues du profilage de la description comportementale.
  - La somme des énergies moyenne de chaque transition pondérée par la fréquence d'exécution de chaque transition, elle aussi extrapolée à partir des données de profilage, donne l'énergie moyenne consommée totale au cours d'une exécution du circuit. La puissance moyenne est obtenue en divisant ce dernier résultat par la somme des fréquences d'exécution de chaque transition multipliée par la durée d'un cycle d'exécution.

Il est ainsi possible d'obtenir des mesures donnant une certaine idée de la consommation du circuit généré à partir de la machine d'états finis abstraite. Ces mesures ne tiennent compte

que de la part des opérateurs sur la consommation, ce qui est trop insuffisant pour espérer obtenir une très grande précision. Des expérimentations sont donc nécessaires afin d'ajouter empiriquement à cette estimation la part de l'interconnexion. La part des registres peut aussi être ajoutée, leur nombre et leur taille pouvant être connus, ou du moins estimés. On peut enfin rajouter la part estimée de l'horloge, là aussi de façon empirique, connaissant le nombre de registres, leur taille, et le nombre d'états de la machine.

A noter que l'estimation particulièrement intéressante de la période minimum de l'horloge, c'est-à-dire du chemin critique du circuit, peut être mise en place d'une façon similaire. Il suffit de pondérer chaque nœud du CDFG d'une transition par le délai pire cas de l'opération correspondante. La recherche du chemin de poids maximum donne le délai maximum de la transition. Appliqué à toutes les transitions, le chemin critique est le maximum des délais maximums de chaque transition. Là encore, des expérimentations sont nécessaires de façon à tenir compte de façon empirique du délai dû aux interconnexions.

10.2 Conclusion

La consommation est aujourd'hui un paramètre incontournable dans le domaine de la conception de circuits ULSI. Elle est dans certains cas déterminante pour la vie et la survie de systèmes intégrés, ce qui justifie les efforts engagés afin de prendre en compte ce paramètre très tôt au cours du cycle de conception. Ces travaux de thèse se sont inscrits dans ce cadre, par l'étude et le développement d'une méthodologie d'estimation de la consommation d'un circuit au cours de la synthèse comportementale.

L'un des intérêts de la méthodologie d'estimation de la consommation développée au cours de ces travaux, est de démontrer la possibilité d'obtenir une vision de la consommation d'un circuit au niveau d'abstraction autorisé par la synthèse comportementale, et cela avec une bonne fiabilité. La technique se base sur une description partie contrôle - partie opérative du circuit après synthèse, ainsi que sur des informations dynamiques issues de l'exécution du circuit décrit au niveau fonctionnel. Elle exploite par ailleurs les informations contenues dans la bibliothèque de composants disponible au cours de la synthèse comportementale, préalablement paramétrée d'après une technique de macro-modélisation permettant la prise en compte de l'activité à l'entrée des composants. Le prototype d'estimation, *SYNRJ*, donne une vision détaillée par cycle élémentaire d'exécution de l'énergie consommée par le circuit. Cette consommation inclut la part due aux unités fonctionnelles du chemin de données, la part due au réseau d'interconnexion entre ces unités et celle due à l'horloge. L'estimation prend par ailleurs en compte la partie parasite de la consommation, c'est à dire les éléments actifs de façon involontaire et inutile au cours du fonctionnement du circuit. *SYNRJ* permet d'obtenir d'autre part la puissance moyenne consommée par le circuit. Les résultats obtenus d'après quelques exemples montrent une précision permettant d'envisager une comparaison fiable entre différentes versions d'un même circuit au cours de la boucle de synthèse comportementale.

De façon plus globale, ces travaux ont permis de mettre en évidence les points importants devant être considérés afin d'estimer avec la fiabilité requise. L'un de ces points est

la prise en compte au cours de l'estimation de la consommation parasite due aux unités auxquelles s'applique le même adjectif. Cette consommation parasite pèse lourdement sur la consommation totale, ce qui met l'emphase sur l'intérêt du développement de techniques permettant de la réduire. Un second point est relatif à la caractérisation des composants de la bibliothèque. La solution adoptée considère l'activité en entrée des composants, ce qui est essentiel. Celle-ci est simplificatrice comparée par exemple à la méthode des bits duaux décrite dans [60], qui est très complète mais relativement complexe à mettre en œuvre de façon pratique. La solution employée est une alternative relativement simple. La caractérisation des composants en vue de l'estimation de la consommation est bien plus complexe et lourde qu'elle peut l'être pour la surface ou l'estimation temporelle. En conséquence, cette étape nécessite la mise en place de moyens nécessaires pour être complètement automatisée, ce qui n'a été réalisé que partiellement au cours de ces travaux, par exemple en réalisant un générateur de séquences de stimuli VHDL à statistique de commutation paramétrable.

L'effort fourni au cours de cette étude s'est concentré exclusivement sur l'impact de la partie opérative, de l'arbre d'horloge et du réseau d'interconnexions sur la consommation. La précision obtenue pour cette partie permet d'atteindre l'un des objectifs initiaux, qui est de permettre une comparaison entre plusieurs instances d'un même circuit au cours des itérations de la boucle de synthèse comportementale. Par contre, l'omission du contrôleur dans l'estimation ne permet pas de comparer la consommation de deux circuits différents, ou avec une fiabilité moindre. Les expériences menées montrent que la consommation du contrôleur est proportionnelle à la consommation de la partie du circuit généré ciblée par ces travaux. Cela est somme toute logique dans la mesure où la consommation du contrôleur dépend de sa complexité, qui est elle-même proportionnelle à la complexité du chemin de données dont il orchestre le fonctionnement. Des travaux futurs pourront être à même de combler ce manque, afin d'autoriser des mesures de la consommation absolue plus fiables. L'évolution de la synthèse de haut niveau, telle que mise en oeuvre depuis le développement de ces travaux de thèse, ouvre là aussi la porte à une adaptation de la méthodologie *SYNRJ* afin de prendre en compte ces évolutions.

A vrai dire, ces travaux se placent dans la perspective de travaux plus larges : ils se veulent être les prémices du développement d'un environnement de synthèse comportementale orienté basse consommation. C'est dans cet objectif qu'il fallait un modèle suffisamment précis, avec toute la relativité présente dans ce terme compte tenu du niveau d'abstraction considéré, pour autoriser la comparaison de deux instances d'un même circuit au cours des boucles de raffinement de la synthèse comportementale. L'étude réalisée a notamment mis en lumière l'importance de la consommation parasite qui peut être la cible d'optimisations relativement simples, comme un raffinement de la machine d'état finis générée, ou le rajout "d'anti-parasites" sur les opérateurs potentiellement parasites, de façon à contrôler leur activité. D'autres optimisations sont envisageables, relatives à l'optimisation de la consommation au cours des étapes de synthèse comportementale comme l'ordonnancement ou l'allocation, dont l'impact pourra être mis en lumière par l'estimation étudiée et mise en oeuvre au cours de cette thèse.

Bibliographie

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *"Compilers: Principles, Techniques and Tools"*. Addison Wesley Publishers - InterEditions, 1990. (french version).

[2] F. Anceau. *"The Architecture of Microprocessors"*. Addison-Westley, 1986.

[3] F. Anceau. "Architecture des processeurs VLSI". Cours - Ecole Polytechnique - Département des mathématiques appliquées, 1996.

[4] G. Araujo. *"Code Generation Algorithms for Digital Signal Processors"*. PhD thesis, DEE, Princeton University, june 1997.

[5] G. Araujo, A. Sudarsanam, and S. Malik. "Instruction Set Design and Optimizations for Address Computation in DSP Architectures". In *Proceedings of 9th ISSS*, 1996.

[6] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler Transformations for High-Performance Computing". *ACM Computing Surveys*, 26(4): 345–419, December 1994.

[7] J. Bayco. "Great Microprocessor of the Past and Present". www, 1997. v9.9.0.

[8] P. A. Beerel, K. Y. Yun, S. M. Nowick, and P. Yeh. "Estimation and Bounding of Energy Consumption in Burst-Mode Control Circuits". In *International Conference on Computer-Aided Design*, pages 26–33, 1995.

[9] L. Benini, M. Favalli, P. Olivo, and B. Ricco. "A novel approach to cost-effective estimate of power dissipation in CMOS ICs". In -, pages 354–360, 1993.

[10] M. E. Benitez and J. W. Davidson. "The Advantages of Machine-Dependent Global Optimization". In *Proceedings*, Zurich, Switzerland, March 2-4 1994. Conference on Programming Languages and System Architectures.

[11] J. Benkoski and S. Napper. "Design Techniques and CAD For Low Power Design - Effect of Submicron on Design and Design Tools". Conference DEA - Slides, 1995.

[12] O. Benz, D. B. Lidsky, and J. M. Rabaey. "Information Based Design Environment". Technical report, University of California, Berkeley, 1995. Technical Report.

[13] S. Bhattacharya, S. Dey, and F. Brglez. "Performance Analysis and Optimization of Schedules for Conditional and Loop-Intensive Specifications". In *Design Automation Conference*, pages 491–496. 31st, 1994.

[14] G. Blalock. "General-purpose micro-P for DSP applications: consider the trade-offs". *EDN*, 1997.

[15] A. Bogliolo, B. Ricco, L. Benini, and G. DeMicheli. "Accurate Logic-level Power Estimation". In *International Symposium on Low Power Electronics*, pages 40–41, 1995.

[16] V. Bouchitte, P. Boulet, A. Darte, and Y. Robert. "Evaluating array expressions on massively parallel machines with communication/computation overlap". *Int. Journal of Supercomputer and High Performance Computing*, 1995.

[17] B. Boulanger and F. Sulmont. "Réalisation d'un outil d'analyse de flot de données et applications". Technical report, ENSIMAG, Juin 1998.

[18] Preston Briggs. *"Register Allocation via Graph Coloring"*. PhD thesis, Rice University, 1992.

[19] P. Buch, S. Lin, V. Nagasamy, and E. S. Kuh. "Techniques for Fast Circuit Simulation Applied to Power Estimation of CMOS Circuits". In *International Symposium on Low Power Design*, pages 135–138, April 1995.

[20] J. P. Calvez, D. Heller, and O. Pasquier. "System performance modeling and analysis with vhdl : benefits and limitations". In *VHDL-Forum Europe Conference*. Ireste, Nantes, France, April 1995.

[21] R. Camposano and R. A. Bergamaschi. "Synthesis using Path-based Scheduling : Algorithms and Exercises". In *Design Automation Conference*, pages 450–455. 27th, 1990.

[22] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen. "Optimizing Power Using Transformations". *IEEE trans. on CAD of Integrated Circuits and Systems*, 14(1), Jan 1995.

[23] J. Chang and M. Pedram. "Register Allocation and Binding for Low Power". In *Design Automation Conference*, pages 29–35. San Francisco, June 1995.

[24] T-L. Chou and K. Roy. "Statistical Estimation of Sequential Circuit Activity". In *International Conference on Computer-Aided Design*, 1995.

[25] M. Cornero and M. Santana. "General Purpose Compiler Environment for Application Specific Processors". Embedded Systems Technologies internal notes - STMicroelectronics, 1998.

[26] M. Cornero, M. Santana, and P. Paulin. "A Flexible Environment for the Development of Application-Specific Hardware and Software". Embedded Systems Technologies internal notes - STMicroelectronics, 1998.

[27] S. Devadas and S. Malik. "A Survey of Optimization Techniques Targeting Low Power VLSI Circuits". In *Design Automation Conference*, pages 242–247. 32th, San Francisco, USA, June 1995.

[28] M. Dion, J-L. Philippe, and Y. Robert. "Parallelizing Compilers: what can be achieved? ". Technical report, LIP, Lyon, 1994. ref. RR94-11.

[29] F. Dresig, P. Lanchès, O. Reittig, and U. G. Baitinger. "Simulation and Reduction of CMOS Power Dissipation at Logic Level". In -, pages 341–346, 1993.

[30] Paul M. Embree and Bruce Kimble. *"C Language Algorithms for Digital Signal Processing"*. Prentice Hall, 1991.

[31] C. N. Fischer and R. J. Leblanc. *"Crafting a Compiler with C"*. Benjamin-Cummings Publishing Company Inc., 1991.

[32] S. Gailhard, O. Ingremeau, J-Ph. Diguet, N. Julien, and E. Martin. "Une méthode probabiliste pour estimer la consommation à un niveau algorithmique". In *Colloque CAO de circuits intégrés et systèmes*, 1997.

[33] S. Gailhard, N. Julien, J-Ph. Diguet, and E. Martin. "Methods to transform easily classical architectural synthesis tools to low power ones". In *8th IEEE Great Lakes Symposium on VLSI*, 1998.

[34] S. Gailhard, N. Julien, and E. Martin. "Intégration de méthodes d'optimisation faible consommation dans l'outil de synthèse architecturale GAUT_W". In *AAA'97*, 1998.

[35] D. Gajski, N. Dutt, A. Wu, and Y. Lin. *"High-Level Synthesis : Introduction to Chip and System Design"*. Kluwer Academic Publishers, Boston, Massachusetts, 1992.

[36] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin. *"HIGH-LEVEL SYNTHESIS : Introduction to Chip and System Design"*. Kluwer Academic publisher, 1993.

[37] A. Ghosh, S. Devadas, K. Keutzer, and J. White. "Estimation Of Average Activity in Combinational and Sequential Circuits". In *Design Automation Conference*, pages 253–259. 29th, 1992.

[38] G.J.Chaitin, A.Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. "Register Allocation via Coloring". *Computer Languages*, 6: 47–57, 1981.

[39] G. Goossens, J. v. Praet, D. Lanneer, W. Geurts, and F. Thoen. "Programmable Chips in Consumer Electronics and Telecommunications". In *Hardware-Software Co-design*. Kluwer Academic Publishers, 1996.

[40] Stanford Compiler Group. *"The SUIF Library"*. Stanford University, 1994. Version 1.0.

[41] L. Guerra, M. Potkonjak, and J. Rabaey. "System-Level Design Guidance using Algorithm Properties". In *International Workshop on VLSI Signal Processing*, pages 73–82. San Diego, Oct 1994.

[42] P. Guillaume. "AMICAL extension for handling post synthesis analysis and energy estimation: overview of SYNRJ concepts". Technical report, lab. TIMA, INPG, Grenoble, 1997. System Level Synthesis group internal report.

[43] P. Guillaume. "La dimension consommation en conception de circuits numériques CMOS - Une synthèse des techniques d'estimation et d'optimisation". Technical report, lab. TIMA, INPG, Grenoble, 1997. System Level Synthesis group internal report.

[44] P. Guillaume. "Compilation: Some Advanced Optimization Techniques Overview". Embedded Systems Technology internal notes - STMicroelectronics, 1998.

[45] P. Guillaume, B. Boulanger, M. Santana, M. Cornero, P. Paulin, and A. A. Jerraya. "Exploitation au niveau source des ressources d'adressage machine dans le cadre d'applications embarquées". In *Colloque CAO de circuits intégrés et systèmes*, 1999.

[46] P. Guillaume and A. A. Jerraya. "Caractérisation de la consommation associée à la synthèse architecturale : une méthodologie". In *Colloque CAO de circuits intégrés et systèmes*, Janvier 1997.

[47] J. L. Hennessy and D. A. Patterson. *"Architecture des ordinateurs: Une approche quantitative"*. McGraw-Hill, 1992. French version of "Computer architecture, a quantitative approach. Translated by D. Etiemble and M. Israel.

[48] C. Y. Hitchcock and D. E. Thomas. "A Method Of Automatic Data Path Synthesis". In *Design Automation Conference*, 1983.

[49] M. A. Hopper. "Register Allocation. Technical report, School of Electrical Engineering, Georgia Institute of Technology, 1994. http://www.ee.gatech.edu/users/mhopper.

[50] Synopsys Inc. *"Synopsys Behavioral Compiler User Guide"*, October 1994.

[51] A. A. Jerraya, H. Ding, P. Kission, and M. Rahmouni. *"Behavioral Synthesis and Component Reuse with VHDL"*. Kluwer Academic Publishers, 1997. "Contribution by: E. Berrebi, W. Cesario and P. Guillaume".

[52] A.A. Jerraya, I. Park, and K. O'Brien. "AMICAL: An Interactive High Level Synthesis Environment". In *European Conference on Design Automation*, February 1993.

[53] P. K. Jha and N. D. Dutt. "Rapid Estimation for Parameterized Components in High-Level Synthesis". *IEEE Trans. on VLSI Systems*, 1(3): 296–303, Sept 1993.

[54] S. Katkoori, N. Kumar, and R. Vemuri. "High Level Profiling Based Low Power Synthesis Technique". In *International Conference on Computer Design*, 1995. Submitted.

[55] B. W. Kernighan and D. M. Ritchie. *"Le langage C"*. Prentice Hall, 1990. french version translated by J-F. Groff and E. Mottier.

[56] N. Kumar, S. Katkoori, L. Rader, and R. Vemuri. "Profile-Driven Behavioral Synthesis for Low-Power VLSI Systems". *IEEE Design and Test of Computers*, pages 70–84, Fall 1995.

[57] P. E. Landman, R. Mehra, and J. M. Rabaey. "An Integrated CAD Environment for Low-Power Design". *IEEE Design and Test of Computers*, pages 72–82, Summer 1996.

[58] P. E. Landman and J. M. Rabaey. "Power Estimation for High Level Synthesis". In *European Conference on Design Automation*, pages 361–366. Paris, Feb 1993.

[59] P. E. Landman and J. M. Rabaey. "Black-Box Capacitance Models for Architectural Power Analysis". In *International Workshop on Low Power Design*, April 1994.

[60] P. E. Landman and J. M. Rabaey. "Architectural Power Analysis : The Dual Bit Type Method". *IEEE Trans. on VLSI Systems*, 3(2): 173–187, June 1995.

[61] M. T-C. Lee, V. Tiwari, S. Malik, and M. Fujita. "Power analysis and Low-Power Scheduling Techniques for Embedded DSP Software". In *International Symposium on System Synthesis*, pages 110–115. 8th, 1995.

[62] Karen A. Lemone. *"Design of Compilers - Techniques of programming langage translation"*. CRC Press Inc., 1992.

[63] T. Lepley. "ILP, Scheduling and Loop Optimization Techniques". Embedded Systems Technologies internal notes - STMicroelectronics, 1998.

[64] R. Leupers. *"Retargetable Code Generation for Digital Signal Processors"*. Kluwer Academic Publishers, 1997.

[65] Markus Levy. "C Compilers for DSP flex their muscles". *EDN - Design Feature*, pages 93–106, June 1997. web: www.ednmag.com.

[66] Markus Levy. "EDN's 1997 DSP-architecture Directory". *EDN*, 1997.

[67] S. Liao, S. Devadas, K. Keutzer, A. Wang, G. Araujo, A. Sudarsanam, S. Malik, V. Zivojnovic, and H. Meyr. "Code generation and optimization techniques for embedded digital signal processors". In G. De Micheli and M. Sami, editors, *Hardware Software Co-design*. Kluwer Academic Publisher, 1996.

[68] D. B. Lidsky and J. M. Rabaey. "Low Power Design of Memory Intensive Functions Case STudy : Vector Quantization". In *International Workshop on VLSI Signal Processing*, pages 378–379. San Diego, Oct 1994.

[69] C. Liem, P. Paulin, M. Cornero, and A. Jerraya. "Industrial Experiance Using Rule-driven Retargetable Code Generation for Multimedia Applications". In *Proc. of ISSS*, 1995.

[70] C. Liem, P. Paulin, and A. Jerraya. "Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures". In *Proc. of the*, pages 597–600. Design Automation Conference, 1996.

[71] C. Liem, P. Paulin, and A. Jerraya. "Compilation Methods for the Adress Calculation Units of Embedded Processor Systems". *Design Automation for Embedded Sytems*, pages 61–77, 1996.

[72] Clifford Liem. *"Retargetable Compilers for Embedded Core Processors - Methods and Experiences in Industrial Applications"*. Kluwer Academic Publisher, 1997.

[73] D. Liu and C. Svensson. "Power Consumption Estimation in CMOS VLSI Chips". *IEEE Journal of Solid-State Circuits*, 29(6): 663–670, June 1994.

[74] S. Manne, A. Pardo, R. I. Bahar, G. D. Hachtel, F. Somenzi, E. Macci, and M. Poncino. "Computing the Maximum Power Cycles of a Sequential Circuit". In *Design Automation Conference*, pages 23–28. 32nd, June 1995.

[75] D. Marculescu, R. Marculescu, and M. Pedram. "Information Theoritic Measures of Energy Consumption at Register Transfer Level". In *International Symposium on Low Power Design*, 1995.

[76] R. Marculescu, D. Marculescu, and M. Pedram. "Switching Activity Analysis Considering Spaciotemporal Correlations". In -, 1994.

[77] E. Martin and J-L. Philippe. *"Ingénierie des systèmes à microprocesseur"*. Dunod, 1996. Collection technique et scientifique des télécommunications.

[78] R. San Martin and J. P. Knight. "A Tutorial on Behavioral Synthesis Power Optimization". Technical report, Carleton University, Ottawa, Nov 1994. Technical Report on internet (http: //www.doe.carleton.cal/ rsm/Power/tutorial.html).

[79] R. San Martin and J. P. Knight. "Power-Profiler : Optimizing ASICS Power Consumption at the Behavioral Level". In *Design Automation Conference*, pages 42–47. 32nd, June 1995.

[80] R. San Martin and J. P. Knight. "Optimizing Power in ASIC Behavioral Synthesis". *IEEE Design and Test of Computers*, pages 58–70, Summer 1996.

[81] P. Marwedel and G. Goossens First International Workshop on Code Generation for Embedded Processors Selected Papers. *"Code Generation for Embedded Processors"*. Kluwer Academic Publishers, 1995.

[82] R. Mehra and J. M. Rabaey. "Behavioral Level Power Estimation and Exploration". In *International Workshop on Low Power Design*, pages 197–204, April 1994.

[83] J. D. Meindl. "Low Power Microelectronics : Retrospect and Prospect". *Proceedings of the IEEE*, 83(4): 619–635, April 1995. (Special Issue on Low Power Electronics).

[84] T. H. Meng, B. M. Gordon, E. K. Tsern, and A. C. Hung. "Portable Video-on-Demand in Wireless Communication". *Proceedings of the IEEE*, 83(4): 659–680, April 1995. (Special Issue on Low Power Electronics).

[85] P. Michel, U. Lauther, and P. Duzy. *"The Synthesis Approach to Digitial System Design"*. Kluwer Academic Publishers, 1992.

[86] J. Monteiro and S. Devadas. "Techniques for the Power Estimation of Sequential Logic Circuits Under User-Sprecified Input Sequences and Programs". In *International Symposium on Low Power Design*, April 1995.

[87] J. Monteiro, S. Devadas, and A. Ghosh. "Retiming Sequential Circuits for Low Power". In *International Conference on Computer-Aided Design*, pages 398–402, Nov 1993.

[88] J. Monteiro, S. Devadas, and B. Lin. "A Methodology for Efficient Estimation of Switching Activity in Sequential Logic Circuits". In *Design Automation Conference*, pages 12–17. 31th, 1994.

[89] S. S. Muchnick. *"Advanced Compiler Design and Implementation"*. Morgan Kaufmann Publishers, 1997.

[90] E. Mussol and J. Cortadella. "High-Level Synthesis Techniques for reducing the activity of Functional Units". In *International Symposium on Low Power Design*, pages 99–104, 1995.

[91] E. Mussol and J. Cortadella. "Scheduling and Ressource Binding for Low-Power". In *International Symposium on System Synthesis*, 1995.

[92] François Naçabal. *"Outils pour l'exploration d'architectures programmables embarquées dans le cadre d'applications industrielles"*. PhD thesis, Institut National Polytechnique de Grenoble, 1998.

[93] F. N. Najm. "A Survey of Power Estimation Techniques in VLSI Circuits". *IEEE Trans. on VLSI Systems*, 2(4): 446–455, Dec 1994.

[94] F. N. Najm. "Feedback, Correlation, and Delay Concerns in the Power Estimation of VLSI Circuits". In *Design Automation Conference*, pages 612–617. 32nd, June 1995.

[95] F. N. Najm. "Power Estimation Techniques for Integrated Circuits". In *International Conference on Computer-Aided Design*, 1995.

[96] F. N. Najm, S. Goel, and I. Hajj. "Power Estimation in Sequential Circuits". In *Design Automation Conference*, pages 635–640. 32nd, June 1995.

[97] NATO ASI. "Low Power Design in Deep Submicron Electronics". NATO Advanced Study Institute, 1996.

[98] T. Niday and B. Cutler. "Deep pipelines schedule VLIW for multimedia". *Electronic Engineering Times*, November 1998.

[99] S. Ohr. "In DSP development, compilers rule". *Electronic Engineering Times*, November 1998.

[100] I. Park. *"AMICAL: Un assistant pour la synthèse et l'exploration architecturale des circuits de commande"*. Thèse inpg, INPG, Grenoble, Juillet 1992.

[101] P. G. Paulin, G. Goosens, C. Liem, M. Cornero, and F. Naçabal. "Embedded Software in Real-time Signal Processing Systems: Application and Architecture Trends". *Proc. IEEE*, 1997. Special issue on HW/SW Co-design.

[102] M. Pedram. "Power Minimization in IC Design : Principles and Applications". *ACM Transactions on Design Automation of Electronic Systems*, 1(1): 3–56, January 1996.

[103] R. A. Powers. "Batteries for Low Power Electronics". *Proceedings of the IEEE*, 83(4): 687–693, April 1995. (Special Issue on Low Power Electronics).

[104] IEEE Computer Society Press, editor. *"ISSS"*. Proceedings of the Eighth International Symposium on System Synthesis, September 1995.

[105] J. Rabaey. "Basics of Low Power Design". Eurochip Course on Methods and Tools for Digital System Design, Sept 1995.

[106] J. M. Rabaey and L. M. Guerra. "Exploring The Architecture and Algorithmic Space for Signal Processing Applications". In *Technical Digest of Int'l. Conference on VLSI and CAD*, pages 315–319. Taejon, Korea, Nov 1993.

[107] J. M. Rabaey, L. M. Guerra, and R. Mehra. "Design Guidance in the Power Dimension". In *Invited Paper, ICCASP*, May 1995.

[108] D. Rabe, B. Timmermann, and W. Nebel. "CMOS Library-characterization for power consumption". Technical report, University of Oldenburg, Dep of CS, Germany, 1995.

[109] A. Raghunathan, S. Dey, N. K. Jha, and K. Wakabayashi. "Controller re-specification to minimize switching activity in controller/data path circuits". In *International Symposium on Low Power Electronics and Design*, 1996.

[110] A. Raghunathan and N. K. Jha. "Behavioral Synthesis for Low Power". In *International Conference on Computer Design*. Boston, MA, Oct 1994.

[111] M. Rahmouni, K. O'Brien, and A. A. Jerraya. "A Loop-Based Scheduling Algorithm for Hardware Description Languages". *Parallel Processing Letters*, 4(3): 351–364, 1994.

[112] B. R. Rau. "Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops". In *in proc. 27th MICRO conf*, 1994.

[113] B. R. Rau and J. A. Fisher. "Instruction-Level Parallel Processing: History, Overview and Perspective". Technical report, Hewlett Packard - Computer Systems Laboratory, 1992.

[114] J. Monteiro J. Rinderknecht, S. Devadas, and A. Ghosh. "Optimization of Combinational and Sequential Logic Circuits for Low Power Using Precomputation". In *Conference on Advanced Research on VLSI*. Chapel Hill, March 1995.

[115] H. Russ. "Embedded Processor and Microcontroller primer and FAQ". http://www.bookcase.com/library/faq/usenet/microcontroller-faq/primer.html, 1997.

[116] M. Schlansker, T. M. Conte, J. Dehnert, K. Ebcioglu, J. Z. Fang, and C. L. Thompson. "Compilers for Instruction-Level Parallelism". *Computer*, December 1997.

[117] P. H. Schneider, U. Schlichtmann, and B. Wurth. "Fast Power Estimation of Large Circuits". *IEEE Design and Test of Computers*, pages 70–77, Spring 1996.

[118] A. Sharma and R. Jain. "Estimating Architectural Ressources and Performance for High-Level Synthesis Applications". *IEEE Trans. on VLSI Systems*, 1(2): 175–190, june 1993.

[119] A. Shen, A. Ghosh, S. Devadas, and K. Keutzer. "On Average Power Dissipation and Random Pattern Testability of CMOS Combinational Networks". In *International Conference on Computer-Aided Design*, pages 402–407, Nov. 1992.

[120] J. M. C. Stork. "Technology Leverage for Ultra-Low Power Information Systems". *Proceedings of the IEEE*, 83(4): 607–618, April 1995. (Special Issue on Low Power Electronics).

[121] C-L. Su, C-Y. Tsui, and A. M. Despain. "Saving Power in the Control Path of Embedded Processors". *IEEE Design and Test of Computers*, pages 24–30, Winter 1994.

[122] Z. Sugar. "Génération de code pour les système mixtes". Technical report, TIMA, 1999.

[123] C. Teng, A. M. Hill, and S. Kang. "Estimation of Maximum Transition Counts at Internal Nodes in CMOS VLSI Circuits". In *International Conference on Computer-Aided Design*, 1995.

[124] L. M. Terman and R-H. Yan. "Scanning the Issue". *Proceedings of the IEEE*, 83(4): 495–496, April 1995. (Special Issue on Low Power Electronics).

[125] D. Terry. "Choosing a processor for Embedded Real-Time Applications". Heurikon Corporation - www, 1996.

[126] V. Tiwari, S. Malik, and A. Wolfe. "Power Analysis of Embedded Software : A First Step Towards Software Power Minimization". *IEEE Trans. on VLSI Systems*, 2(4): 437–445, December 1994.

[127] C. Tsui, M. Pedram, and A. M. Despain. "Exact and Approximate Methods for Calculating Signal and Transition Probabilities in FSMs". In *Design Automation Conference*, pages 18–23. 31th, 1994.

[128] S. Turgis, N. Azemard, and D. Auvergne. "Explicit Evaluation of Short Circuit Power Dissipation for CMOS Logic Structures". In *International Symposium on Low Power Design*, pages 129–134, 1995.

[129] J. Vanhoof, K. V. Rompaey, I. Bolsens, G. Goossens, and H. De Man. *"High-Level Synthesis for Real-Time Digitial Signal Processing"*. Kluwer Academic Publishers, 1993.

[130] A. Vittal and M. Marek-Sadowska. "Power Optimal Buffered Clock Tree Design". In *Design Automation Conference*, San Francisco, June 1995. 32nd.

[131] N. H. E. Weste and K. Eshraghian. *"Principles of CMOS VLSI Design: A System Perspective"*. Addison-Wesley Publishing Company, 2nd edition, 1995.

[132] M. G. Xakellis and F. N. Najm. "Statistical Estimation of the switching Acivity in Digital Circuits". In *Design Automation Conference*, pages 728–733. 31st, 1994.

[133] J. G. Xi and W. W-M. Dai. "Buffer Insertion and Sizing Under Process Variations for Low Power Clock Distribution". In *Design Automation Conference*, pages 491–496. 32nd, June 1995.

[134] R. Zimmermann and R. Gupta. "Low-Power Logic Styles : CMOS vs CPL". In *European Solid-State Circuits Conference*, pages 112–115, 1996.

Annexe A

Analyse du flot de contrôle et de données

A.1 Entrée en matière

On peut optimiser le code à tous les niveaux (source, assembleur, ...), mais dans tous les cas, la première étape est la recherche et le calcul d'informations pertinentes autour du programme. Il s'agit de donner du sens à ce qui n'est au départ qu'un enchaînement d'opérations. Ce sont ces informations qui permettront par la suite de mettre en œuvre plus efficacement les optimisations, ce qui nécessite des outils performants pour les calculer.

Le plus souvent, deux types d'informations sont distinguées :

- des informations de flot de contrôle, qui indiquent les différents chemins d'exécution possibles d'un programme ainsi que la hiérarchie inhérente à ces chemins.
- des informations de flot de données, afin de connaître la façon dont les variables sont manipulées et dont elles interagissent entre elles.

Un troisième type un peu à part concerne l'analyse des alias entre variables du programme, c'est-à-dire des pointeurs présents dans le programme. Cette analyse, permettant de lever certaines ambiguïtés bloquantes sur les données manipulées, étend le champ d'action de l'analyse de flot de données et donc celui des optimisations qui en dépendent.

Nous tenterons dans cette section de rappeler brièvement la nature de ces analyses généralement nécessaires à tout compilateur optimisant, et nous décrirons la technologie employée au cours de ces travaux afin de les réaliser pratiquement.

A.2 Glaner les informations requises : l'approche classique

Tout en illustrant la signification des différentes analyses ci-dessus mentionnées, nous donnerons dans cette section les moyens classiques de résolution et de représentation des

problèmes tels que la littérature les fournit. Le fil d'ariane sera l'exemple de la procédure calculant la suite de Fibonacci, exemple tiré de [89].

A.2.1 Mettre en évidence les chemins d'exécution : flot de contrôle[1]

L'analyse de flot de contrôle vise à fournir une représentation explicite des différents chemins d'exécution d'un programme. Cette connaissance est essentiellement utile à la mise en œuvre de l'analyse de flot de données, mais ne se borne pas à cette seule utilité. Une représentation courante du flot de contrôle s'appuie sur un graphe de flot de contrôle ou CFG[2] . Les sommets de ce graphe sont des blocs de base, un bloc de base étant une suite d'instructions sans rupture de séquence; les arcs sont les enchaînements possibles de ces blocs de base tels que les impose la sémantique du programme ainsi représenté. Par exemple, le programme de la figure A.1 peut être représenté par le graphe de contrôle de la figure A.2 (on note B0, B1, ... les différents blocs de base significatifs).

```
int fib(int m) {
  int i, f0, f1, f2;
  f0 = 0;
  f1 = 1;
  if (m <= 1) {
     f2 = m;
  }
  else {
     i = 2;
     while (i <= m) {
        f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
        i = i + 1;
     }
  }
  return f2;
}
```

Figure A.1: Code C calculant les nombres de Fibonacci

La construction d'un tel graphe est la première étape de l'analyse du flot de contrôle. Viennent se greffer ensuite deux grands types d'approches, la première, la plus classique, étant décrite ici, tandis que la seconde fait l'objet de la section A.3. A chaque approche sera associé un certain type d'analyse de flot de données. La première approche, la plus simple à mettre en œuvre, prend un tel graphe et y débusque les boucles en utilisant la notion de dominateurs. L'information portée par le graphe, à laquelle vient s'ajouter celle sur les boucles, devient un squelette suffisant pour pouvoir appliquer une analyse de flot de

[1] On pourra fructueusement se reporter notamment à deux ouvrages de référence traitant de ces aspects analytiques pré-transformationnels: [1] et [89]. Le second est particulièrement mis à contribution ici.

[2] Control Flow Graph

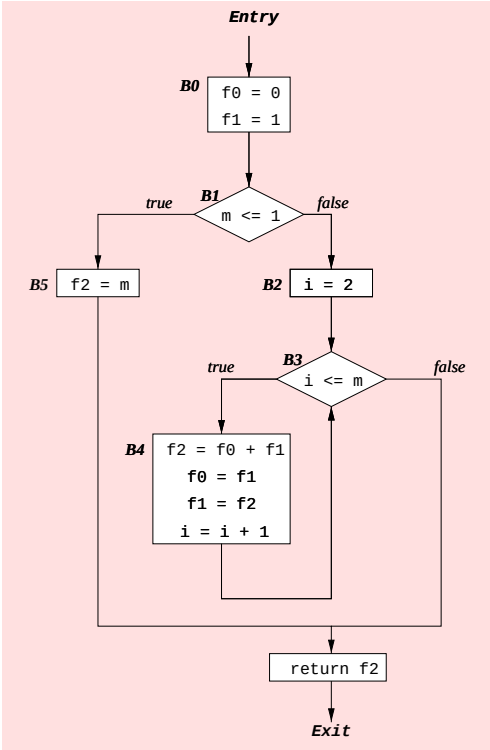


Figure A.2: Graphe de flot de contrôle de la procédure fib

données par la méthode itérative. C'est cette approche qui est utilisée dans cette section pour illustrer les concepts d'analyse de flot de contrôle et de données.

Un dominateur obéit à la définition suivante:

> Un nœud N1 domine un nœud N2 dans un graphe de flot de contrôle, si tous les chemins à partir du nœud d'entrée du graphe jusqu'à N2 passent par N1.

Ici, B1 domine B2 et B5, B2 domine B3 qui domine B4. Si l'on peut trouver un ensemble de nœuds dominés par un seul et même nœud N, à partir desquels on peut atteindre N, et ne comportant qu'un seul arc inverse (arc dont la tête domine la queue), alors cet ensemble de nœuds forme une boucle. Ici, B3 et B4 répondent à ces critères et forment donc une boucle.

Aparté On peut se demander quel est l'intérêt d'une telle recherche de boucle dans la mesure où cette information, dans le cas présent, est portée par la sémantique du code: pourquoi ne pas reporter cette information à plus bas niveau? L'intérêt principal tient dans la généralité de la méthodologie. Si la boucle de l'exemple A.1 est remplacée par un test sur la valeur de i, suivi d'un *goto-label,* on convient aisément que la présence d'une boucle n'est plus aussi évidente et nécessite une analyse particulière pour être mise en lumière.

A.2.2 Connaître la façon dont les données s'échangent et interagissent: flot de données

L'analyse de flot de données se ramène à la recherche en tout point d'un programme d'ensembles de variables ou d'expressions vérifiant certaines propriétés. On peut par exemple vouloir connaître l'ensemble des expressions utilisant une variable avant que celle-ci soit redéfinie, ou l'ensemble des variables d'induction d'une boucle. Par la suite, ces ensembles permettront de décider comment et où une transformation pourra modifier le code pour l'optimiser, ou plus simplement, de calculer d'autres informations plus complexes. Réaliser une analyse de flot de données consiste à résoudre des équations ensemblistes associées généralement aux instructions ou aux blocs de base du graphe de contrôle. Les éléments ciblés par l'analyse et présents dans le code constituent un *univers* représenté le plus souvent et de façon pratique par un vecteur de bits. Les équations traduisent l'effet d'une instruction ou d'un bloc de base sur cet univers. A l'entrée de chaque bloc de base B, on aura l'ensemble IN(B) donnant l'état de l'univers à l'entrée du bloc (en pratique, un masque donnant les éléments de l'univers vivants en entrée du bloc). En sortie de chaque bloc, l'ensemble OUT(B) traduira l'action du bloc sur l'univers de l'analyse. Typiquement, une équation de flot sera de la forme:

$$OUT(B) = (IN(B) - KILL(B)) \cup GEN(B) \tag{A.1}$$

avec:

- $KILL(B)$ ensemble des éléments "tués" par le bloc B. La définition de "tué" sera propre à chaque type d'analyse, comme nous le verrons plus loin.
- $GEN(B)$ ensemble des éléments "créés" dans le bloc B. Idem pour "créé" .
- $IN(B)$ ensemble des éléments de l'univers actifs en entrée du bloc B.
- $OUT(B)$ ensemble des éléments actifs en sortie du bloc B.

Traduite de façon littérale, cette équation signifie simplement : "l'état de l'univers en sortie de bloc correspond à l'état de l'univers en entrée de celui ci, diminué des éléments de l'univers tués par le bloc et enrichi des éléments de l'univers générés par celui-ci."

L'influence des blocs du graphe de contrôle les uns sur les autres doit être par ailleurs modélisée, ce qui conduit à des équations du type:

$$IN(B) = \bigcup_{P \in Pred(B)} OUT(P) \tag{A.2}$$

où $Pred(B)$ est l'ensemble des blocs de base prédécesseurs du bloc B.

Les équations du type A.1 sont appelées **équations de transfert**, tandis que celles du type de A.2 sont appelées **règles de confluence**. Ces équations sont de type avant, c'est-à-dire que l'on calcule les ensembles OUT à partir des ensembles IN ce qui correspond à un parcours avant du graphe. Il est des analyses où un parcours arrière voire bidirectionnel est

requis. L'opérateur utilisé plus haut dans l'équation de confluence inter-bloc ,∪, modélise l'effet de la combinaison d'informations de flot de données provenant de sources différentes en entrée d'un bloc.

Afin d'illustrer ces notions abstraites, nous nous focaliserons sur un exemple très connu et usité d'analyse de flot de données : la recherche des définitions accessibles[3]. C'est une analyse avant ∪ comme opérateur de confluence.

> Une définition d'une variable x, c'est-à-dire l'affectation d'une valeur à x, est dite accessible en un point P du programme s'il existe dans le graphe de flot de contrôle un chemin qui permet d'aller de cette définition au point P sans que x soit redéfinie.

Pour cette analyse particulière, appliquée à l'exemple de la figure *A.2*, la première étape consiste à déterminer l'univers de celle-ci. Ensuite, il convient de formuler les équations de flot. L'univers est ici l'ensemble des variables définies dans le programme, sans distinction. Le tableau ci-dessous regroupe ces définitions:

Expression	Bloc
f0=0	B0
f1=0	B0
i=2	B2
f2=f0+f1	B4
f0=f1	B4
f1=f2	B4
i=i+1	B4
f2=m	B5

Dans ce contexte, "tué" signifie redéfini, tandis que "crée" signifie défini.
On aura pour cette procédure:

$$\begin{cases} GEN(B0) = \{f0 = 1, f1 = 1\} \\ GEN(B1) = \{\} \\ GEN(B2) = \{i = 2\} \\ GEN(B3) = \{\} \\ GEN(B4) = \{f2 = f0 + f1, f0 = f1, f1 = f2, i = i + 1\} \\ GEN(B5) = \{f2 = m\} \end{cases}$$

et

$$\begin{cases} KILL(B0) = \{f0 = f1, f1 = f2\} \\ KILL(B1) = \{\} \\ KILL(B2) = \{i = i + 1\} \\ KILL(B3) = \{\} \\ KILL(B4) = \{f0 = 1, f1 = 1, i = 2, f2 = m\} \\ KILL(B5) = \{f2 = f0 + f1\} \end{cases}$$

[3]reaching definitions

Note: On peut être surpris de voir que les définitions de B4 *'f0=f1'* et *'f1=f2'* sont tuées par le bloc B0: cela n'a guère de sens si l'on observe le programme. C'est simplement lié à ce que l'enchaînement des blocs n'est pas considéré au cours de cette première étape: on considère l'univers constitué de toutes les définitions de variables dans le code, et on détermine l'effet de chaque bloc sur cet univers dans l'absolu. La résolution des équations fait effectivement intervenir l'ordre d'exécution des blocs dans la phase suivante, où entrent en jeu les équations de confluence.

Entre les blocs, les équations suivantes interviennent:

$$\begin{cases} IN(B1) = OUT(B0) \\ IN(B2) = OUT(B1) \\ IN(B3) = OUT(B2) \cup OUT(B4) \\ IN(B4) = OUT(B3) \\ IN(B5) = OUT(B1) \end{cases}$$

Tous les éléments sont à présent regroupés pour la résolution des équations. La première chose à faire est d'initialiser $IN(B_i) = \{\}$ (pas de définitions accessibles en entrée des blocs au départ). Ce système d'équations peut simplement être résolu en itérant jusqu'à ce que l'on ne constate plus de changements dans les ensembles calculés. Après la première application des équations, on obtient:

Blocs	IN	OUT
B0	{}	f0=1,f1=1
B1	f0=1,f1=1	f0=1,f1=1
B2	f0=1,f1=1	f0=1,f1=1,i=2
B3	f0=1,f1=1,i=2	f0=1,f1=1,i=2
B4	f0=1,f1=1,i=2	f2=f0+f1,f0=f1,f1=f2,i=i+1
B5	f0=1,f1=1	f0=1,f1=1,f2=m

Comme OUT(B4) n'est disponible qu'après la première application des équations, impossible de tomber sur l'équilibre du premier coup. Dans tous les cas, au moins deux itérations sont toujours nécessaires lorsqu'une telle méthode itérative est appliquée. La deuxième itération donne:

Blocs	IN	OUT
B0	{Φ}	f0=1,f1=1
B1	f0=1,f1=1	f0=1,f1=1
B2	f0=1,f1=1	f0=1,f1=1,i=2
B3	f0=1,f1=1,i=2,f2=f0+f1,f0=f1,f1=f2,i=i+1	f0=1,f1=1,i=2,f2=f0+f1,f0=f1,f1=f2,i=i+1
B4	f0=1,f1=1,i=2,f2=f0+f1,f0=f1,f1=f2,i=i+1	f2=f0+f1,f0=f1,f1=f2,i=i+1
B5	f0=1,f1=1	f0=1,f1=1,f2=m

Dans le cas présent, une troisième itération, nécessaire pour déterminer la stabilité, donne le même résultat. En termes de mise en œuvre pratique, la représentation par vecteur

de bits présente un intérêt évident: l'application des équations de flot en devient triviale (ET, OU logiques sur les vecteurs). Au final, les ensembles IN et OUT sont des masques de bits où un 1 indique la variable ou l'expression de l'univers activée.

A l'issue de cette analyse, est bien obtenu l'ensemble des définitions accessibles en entrée et sortie des blocs. A noter l'ambiguïté en entrée et sortie du bloc B3 concernant les définitions de f0, f1 et i. Et effectivement, à strictement parler, nul ne peut dire en sortie de B3 quelle est la bonne définition. Cela illustre le parti pris du *sans risque* dans une telle analyse et dans les transformations qui en dépendent. Il est moins grave d'avoir une surestimation des informations, qu'une sous-estimation. En l'occurence, à titre d'exemple, la non détection de l'accessibilité de la définition *i=i+1* en entrée des blocs B3 et B4 pourrait entraîner une transformation future à remplacer i par la valeur 2, ce qui serait faux. Les informations sur les définitions de variables peuvent être utilisées dans le cadre notamment du débogage, puisque l'on peut identifier les variables utilisées mais non définies, ou encore dans le cadre de la propagation de constantes. Dans le cas de problèmes nécessitant un parcours arrière, le procédé est identique en inversant IN et OUT dans les équations. On peut ainsi citer la petite sœur de l'analyse des définitions accessibles, l'analyse des utilisations accessibles qui se propose de déterminer l'ensemble des utilisations d'une variable accessible à partir de sa définition.

L'approche itérative a ceci d'avantageux qu'elle reste simple à mettre en œuvre. Elle est cependant moins performante que d'autres types d'analyses telles les analyses basées sur les intervalles et notamment l'analyse structurelle qui en est une version sophistiquée. Ce type d'analyse fait l'objet de la section suivante et du choix d'implémentation privilégié ici.

A.2.3 S'assurer que les variables contiennent bien les valeurs escomptées: analyse des alias

Sans vouloir outre mesure entrer dans les détails d'une analyse qui est complexe, il est néanmoins bon de rappeler ou d'introduire les principes d'un problème crucial dans le contexte de la transformation optimisante de code: l'analyse des alias.

Une étrange aura entoure le monde des pointeurs, notamment en C, teintée de méfiance et de respect: les bêtes sont utilisées massivement mais on s'en méfie tant il est vrai qu'un code C faisant usage de pointeurs est plus efficace mais bien moins lisible et maintenable qu'un code s'en passant, quand c'est possible. Mais outre ces aspects quelque peu philosophiques, un fait subsiste : les pointeurs existent et sont utilisés. Il faut donc que les compilateurs optimisants s'en arrangent et fassent de leur mieux. Ce n'est pourtant pas une mince affaire. Quelques exemples en illustreront mieux la difficulté, principalement due au simple fait suivant : un pointeur référence à priori n'importe quelle valeur. L'analyse des alias a donc pour but de déterminer, pour un pointeur donné, l'ensemble des variables - en fait des positions mémoires correspondantes - que celui-ci pourra référencer.

La méconnaissance de l'ensemble des valeurs pouvant être représentées par un pointeur, c'est-à-dire des alias, influence la plupart des optimisations de code les plus connues. Un

exemple peut être la propagation de constantes:

```
int * p;
int a,b;
.
.
.
b = 1;
*p = 10;
a = b;
```

Ici, on ne peut décemment pas remplacer a = b par a = 1: la présence juste avant de l'affectation *p = 10 empêche d'agir puisque rien ne prouve que p ne pointe pas sur b, auquel cas la transformation conduit à une erreur. Cela va à l'encontre de l'esprit conservateur, voire réactionnaire qui sévit dans le monde des compilateurs. Et il se justifie ici. Le problème est similaire pour la propagation de copie, soeur jumelle de la propagation de constantes, en remplaçant ces dernières par des variables, ou l'élimination de sous expressions communes. Pour ces optimisations, étant donné que l'analyse d'alias va pouvoir préciser si les variables à propager ou à remplacer sont susceptibles d'être modifiées de façon indirecte, elles auront toute liberté d'agir.

Un autre cas où l'analyse qui fait l'objet de cette section prend tout son sens est l'ordonnancement des instructions. Cette transformation a pour but d'agencer les instructions de façon à mettre en évidence le parallélisme inhérent au jeu d'instruction s'il existe[4] et de façon à faire en sorte que le pipeline marche le mieux possible, c'est-à-dire qu'il travaille à temps plein. Pour ce dernier cas, l'optimiseur peut être amené à changer la séquence d'exécution suivant le contexte, afin par exemple de remplir un délai ou bulle dans le pipeline causé par une dépendance de donnée. Pour mener à bien cette tâche, l'optimiseur doit avoir une connaissance minutieuse des dépendances de données entre les différentes instructions. La méconnaissance des positions référencées par les pointeurs peut empêcher celui-ci d'agir par la méconnaissance des dépendances de données dès qu'un pointeur est présent entre deux instructions. D'une façon similaire, l'analyse d'alias peut permettre de lever les ambiguïtés sur les références mémoires, de telle sorte que des références à des positions mémoires distinctes puissent être réordonnées sans danger.

En résumé, la plupart des optimisations fonctionnent sans bénéficier d'une analyse des alias préalable, pour peu qu'elles s'assurent de restrictions draconiennes préservant la fonctionnalité du programme. Mais c'est au prix d'une perte de performance, puisque les optimisations sont moins efficaces, perte qui peut être cruciale.

A.3 Un outil pour l'analyse approfondie

A.3.1 Description

Un inconvénient de l'algorithme itératif tel que décrit précédemment est son coût, qui n'est pas fixé à l'avance et qui n'est pas forcément optimal : pour détecter la stabilité, on fait

[4] On parlera de parallélisme au niveau instruction ou ILP (Instruction Level Parallelism)

au minimum une itération de trop, et certains ensembles sont recalculés même s'ils sont stables bien avant la fin des itérations. Face à cet inconvénient, un algorithme alternatif, dont le coût est a priori minimal, est l'algorithme structurel, qui s'inspire fortement de celui décrit dans [89].

Cet algorithme s'appuie sur les informations générées par l'analyse structurelle du flot de contrôle afin de produire des équations de flot de données. Ces dernières représentent à proprement parler les effets flot de données des structures de contrôle. Une structure de contrôle sera un bloc de base, une structure if_then_else, une boucle do_while ou while_do. Un super-bloc pour résumer.

Plus précisément l'idée de l'analyse structurelle est de construire pour chaque structure de contrôle une fonction f telle que $OUT = f(IN)$ (ou $IN = f(OUT)$). En combinant ces fonctions (appelées fonctions de flot de données ou flow functions) et les règles de confluence, on peut calculer les ensembles IN et OUT en un unique parcours du graphe de flot de contrôle. On retrouve à vrai dire les même concepts que précédemment, étendus à des structures de contrôles plus importantes que les simples blocs de base.

Cet algorithme présente plusieurs avantages : d'une part et moyennant une bonne implémentation, il peut être optimal en terme de vitesse d'exécution; d'autre part, il n'est pas beaucoup plus complexe à implémenter que l'algorithme itératif. En effet, pour les deux algorithmes, il est nécessaire d'implémenter des ensembles et une représentation des équations (sous forme de fonctions ensemblistes pour la méthode structurelle). Les différences proviennent surtout de la manière de résoudre les équations.

A.3.2 Structure générale de l'outil d'analyse et son application

L'outil développé selon les principes ci-dessus, baptisé *Athlon*, sera décrit dans ses grandes lignes au cours de cette section. Pour plus de détails, il peut être intéressant de consulter [17]. L'organisation globale liée à l'utilisation de l'outil d'analyse est représentée sur la figure A.3, ainsi que ses interactions avec la structure de données initiale, et le module d'optimisation . A partir d'une structure intermédiaire[5] représentant le programme à analyser et à optimiser, le graphe de flot de contrôle (GFC) est généré, ce qui conduit à une structure intermédiaire de plus haut niveau. C'est cette dernière qui contient les informations calculées par le moteur d'analyse de flot de données (DFA). L'optimiseur transforme la représentation intermédiaire initiale en se basant sur les informations qu'il soutire au GFC structurel.

L'analyse de flot de données s'exécute à la requête de l'optimiseur, en se basant sur des propriétés définies par ailleurs et répondant aux besoins particuliers de l'optimiseur. Un certain nombre de propriétés fréquemment utilisées, telles les définitions accessibles, sont disponibles sous forme de bibliothèque. Les propriétés plus spécifiques doivent être créées en parallèle à l'optimiseur. Ce sont en pratique deux fonctions à créer, appelées successivement

[5] On pourra fructueusement se reporter à [40] pour une description détaillée de la structure intermédiaire SUIF (Stanford University Intermediate Format). Il s'agit d'un environnement de compilation dédié à l'expérimentation et développé à l'université de Stanford.

par l'analyseur de flot de données. Ces fonctions expriment les propriétés de l'analyse à réaliser. Les informations calculées lors d'une précédente requête sont réutilisables à volonté.

L'un des intérêts de la représentation structurelle du flot de contrôle est de pouvoir relancer une analyse sur un seul super-bloc ou structure de contrôle, voire une partie du GFC lui même. Ainsi, après transformation de la structure intermédiaire, il est possible à l'optimiseur de relancer une analyse sur la nouvelle structure dans son ensemble, ou sur une partie de celle-ci (la partie transformée) afin d'appliquer une nouvelle passe d'optimisation. Cette caractéristique s'avère extrêmement utile en pratique puisqu'elle implique un gain important en termes de performance et de souplesse d'utilisation.

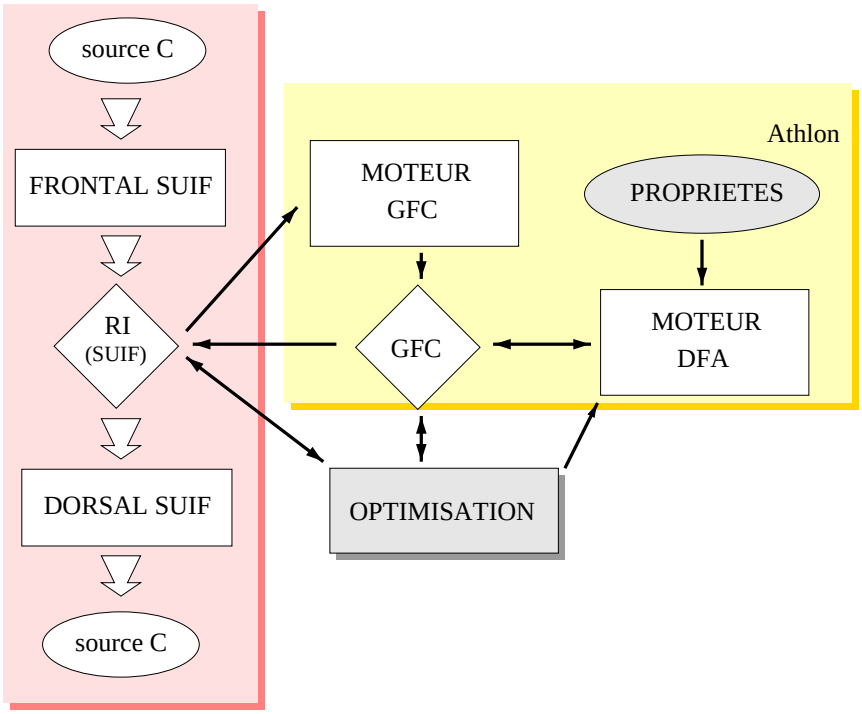


Figure A.3: Représentation générale du flot de données dans l'outil Athlon.

A.3.3 Représentation du flot de contrôle

A.3.3.1 Flot d'entrée du module

Le premier module de l'outil d'analyse construit le graphe de flot de contrôle adapté aux besoins de l'analyse. Considérons la procédure C suivante (*cf.* A.2.1 page 180) :

```
int fib(int m)
/* calcule la suite
   de Fibonacci */
{
  int i, f0, f1, f2;
  f0 = 0;
  f1 = 1;
  if (m <= 1) {
    f2 = m;
  }
  else {
    for (i = 2;  i<=m;  i++)
     {
       f2 = f0 + f1;
       f0 = f1;
       f1 = f2;
     }
  }
  return f2;
}
```

interprétée de la façon suivante par le module de construction du GFC:

```
int fib(int m)
/* calcule la suite
   de fibonacci */
{
  int i, f0, f1, f2;
  f0 = 0;
  f1 = 1;
  if (m <= 1) {
    f2 = m;
  }
  else {
    i = 2;
    while (i <= m) {
       f2 = f0 + f1;
       f0 = f1;
       f1 = f2;
       i = i + 1;
    }
  }
  return f2;
}
```

Les deux procédures sont équivalentes mais non identiques. Deux raisons majeures à cela, liées à la représentation intermédiaire primaire[6] dont dépend l'ensemble de l'outil et qui impose ses propres restrictions:

1. Un de ses diktats est l'expansion systématique des expressions du style i++; i += 2; ... en i=i+1; i=i+2; ... Un autre de ses diktats est la transformation systématique des boucles *while-do* en des boucles *do-while* accompagnées des nécessaires expressions conditionnelles.

2. Les ingrédients associés aux boucles *for*, boucles qui ont une saveur toute particulière et auxquelles de nombreuses restrictions sont associées, ces ingrédients donc ne sont pas explicites pour l'analyse. Ils doivent donc le devenir par le biais d'une représentation *while-do* créée pour l'occasion de toutes pièces dans le GFC. Des liens avec la représentation primaire sont établis de façon à préserver la cohérence. Ainsi, les instructions d'initialisation et d'incrémentation de boucle sont-elles créées, de même que l'instruction de test de fin de boucle. Le corps est préservé. Pour la petite histoire, la représentation intermédiaire préserve les boucles *for* dans la mesure où celles-ci sont "bien formées", c'est-à-dire mettent en œuvre une seule et unique variable d'induction de base, initialisée et incrémentée explicitement et qui entre en jeu dans le test de sortie.

Ce qui précède est très dépendant de l'implémentation de l'outil d'analyse et en l'occurence de la représentation intermédiaire primaire choisie. Ces précisions n'ont qu'un intérêt anecdotique et illustratif des problèmes spécifiques rencontrés au cours de la mise en œuvre de

[6] voir note de bas de page 5

cet outil d'analyse. Néanmoins, ces quelques détails seront utiles dans la compréhension des exemples qui suivent.

A.3.3.2 Graphe de contrôle: représentation structurelle

Le graphe obtenu à partir de la procédure *fib* est représenté sur la figure A.4.

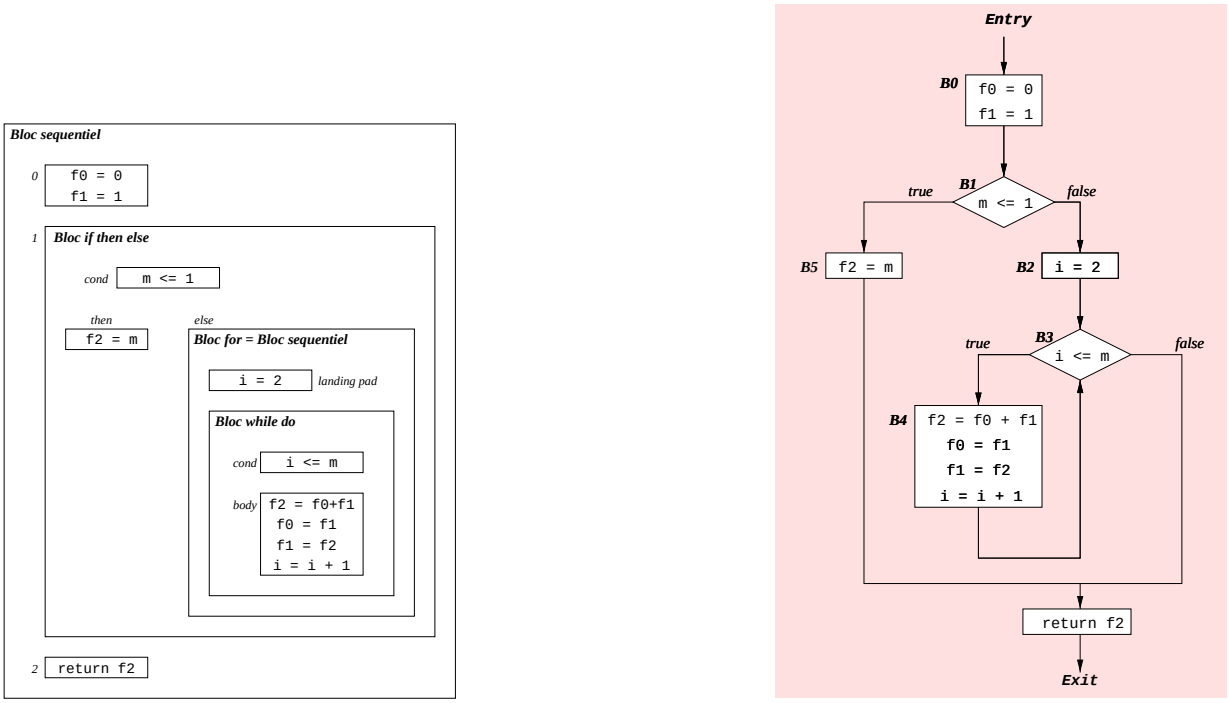


Figure A.4: Graphe de flot de contrôle de la procédure *fib*.

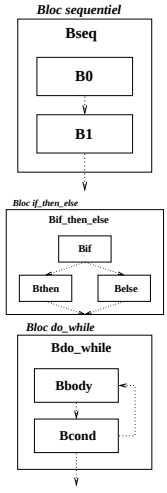
Sur cette figure, on peut voir à gauche le graphe tel qu'il est construit par l'outil d'analyse. La partie de droite correspond au graphe tel que peut l'afficher ce dernier. Cette dernière représentation est plus classique et plus parlante. La première représentation permet néanmoins de bien comprendre l'architecture retenue pour le graphe de flot de contrôle, qui, comme son nom l'indique, attache une grande importance aux structures de contrôle et à la façon dont celles-ci s'emboîtent.

Comme introduit plus haut, à chaque structure de contrôle est associé un *super_bloc* particulier, qui reprend la structure de la partie de code concernée. En guise d'exemple un *super_bloc if* a trois fils, chacun d'eux correspondant respectivement à la condition, au "*then*" et au "*else*", et chacun d'eux étant lui-même un *super_bloc*. Les liens entre les blocs sont alors implicites et la procédure (ou la portion de code) dont on construit le graphe n'est plus qu'un *super_bloc* elle-même, comme l'illustre la figure A.4. Les différents types de super_blocs sont au nombre de six:

- bloc de base

- bloc séquentiel : deux super_blocs de type quelconque agencés l'un après l'autre
- bloc do_while
- bloc while_do
- bloc if
- bloc for : c'est en fait un bloc séquentiel comprenant un bloc de base contenant l'initialisation de l'index de boucle, suivi d'un bloc while_do.

Pour comprendre l'intérêt de cette structure, outre la manipulation systématique du même type d'objet quelle que soit sa nature, il faut revenir à l'algorithme choisi: celui-ci repose sur des fonctions ensemblistes associées à chaque structure de contrôle, fonctions qui peuvent s'exprimer comme composition d'autres fonctions. Comme le montre la figure A.5, l'architecture de graphe telle que représentée permet de connaître facilement les fonctions rattachées à un *super_bloc* différent d'un simple bloc de base.



$$f_{Bseq} = f_{B1} \bullet f_{B0}$$

$$f_{Bif\_then\_else} = (f_{Bthen} + f_{Belse}) \bullet f_{Bif}$$

$$f_{Bdo\_while} = (f_{Bcond} \bullet f_{Bbody})^{+} = f_{Bcond} \bullet f_{Bbody}$$

Figure A.5: Les fonctions ensemblistes de quelques structures de contrôle

Pour ces structures, on a donc immédiatement les fonctions et ces dernières peuvent être mises à jour automatiquement si le bloc d'un niveau inférieur est modifié. Si l'on détaille ces trois structures de contrôle, la fonction résultant de deux super_blocs s'exécutant en séquence est la composée (au sens mathématique du terme) des fonctions de ces deux blocs. Pour le bloc if, + représente l'opérateur de confluence déjà mentionné dans la section A.2.2.

A.3.3.3 Informations rattachées au graphe

L'information de flot de données doit pouvoir être associée au GFC . Pour ce faire, un champ est prévu pour chaque bloc (en fait pour chaque "*super-bloc*"), permettant de stocker et d'accéder rapidement à des fonctions de flot de données et à des ensembles. A tous les

niveaux du graphe, les informations de flot de données peuvent être accédées. Ainsi, il est possible de "court-circuiter" une partie du code si le détail de celle-ci ne nous intéresse pas. Autrement dit, on peut "encapsuler" une portion de code dans un même bloc et ne considérer que celui-ci par la suite.

La figure A.6 illustre cette possibilité : on peut accéder aux informations de la boucle *for* (en fait de blocs en séquence) sans avoir connaissance du contenu de celle-ci.

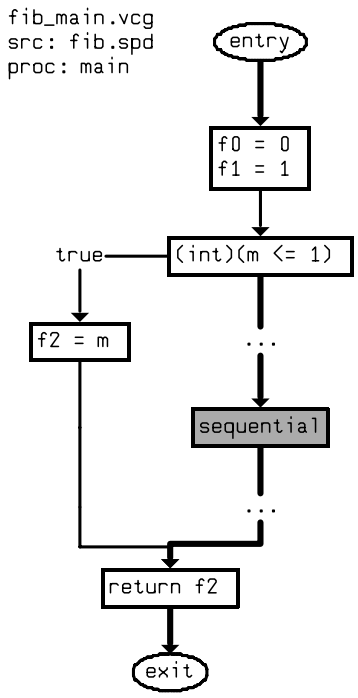


Figure A.6: "Courtcircuit" de la boucle for dans la procédure *fib*

A.3.4 Analyse de flot de données

Le moteur d'analyse de flot de données implémente l'algorithme d'analyse structurelle et doit donc calculer et associer au graphe de flot de contrôle les fonctions ensemblistes, puis les ensembles recherchés. A titre illustratif, l'exemple de la recherche des définitions accessibles est à nouveau mis à contribution.

La première chose que doit assurer le moteur est le calcul pour chaque bloc d'une fonction ensembliste, de la forme $OUT = f(IN)$. Un premier problème se pose ici: comment représenter une fonction ensembliste et plus simplement, comment représenter un ensemble ? Après avoir répondu à cette question, sera détaillée la construction proprement dite des fonctions. Nous terminerons en étudiant plus précisément le calcul des ensembles à partir des fonctions.

A.3.4.1 Les ensembles

Afin d'accélérer les opérations ensemblistes comme l'union, l'intersection ou la différence, les ensembles sont représentés par des vecteurs de bits: on associe chaque élément à une position unique dans un vecteur de bits (un tel vecteur est en pratique une suite d'entiers). L'élément est alors dans l'ensemble si le bit correspondant est à 1, et inversement si le bit est à 0. Les opérations ensemblistes se traduisent alors par des opérations logiques entre les vecteurs de bits. Par exemple, l'union de deux ensembles correspond à un OU binaire entre les vecteurs représentant ces ensembles. Bien sûr, cela suppose que l'univers, autrement dit l'ensemble des éléments susceptibles d'appartenir aux ensembles, soit connu et que chacun de ses éléments ait une position unique dans tous les vecteurs de bits.

Pour ce faire, on définit un univers qui peut s'enrichir au fur et à mesure que l'on découvre de nouveaux éléments en parcourant le graphe (ou plus précisément les instructions). On définit alors les ensembles comme des masques de cet univers, comme illustré par la figure A.7.

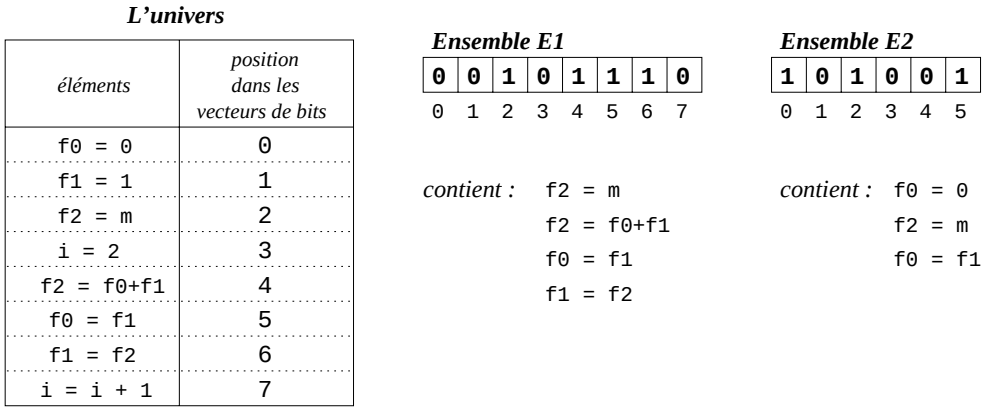


Figure A.7: Les ensembles de données : exemple

A.3.4.2 Les fonctions de flot de données

Une fonction de flot de données est une représentation, pour un bloc particulier, de l'ensemble des équations de flot de données traduisant une certaine propriété (la solution du système d'équations donne les éléments vérifiant cette propriété). Dans la représentation choisie, chaque super_bloc aura sa propre fonction de flot construite soit directement, soit par composition d'autres fonctions de flot. Ces fonctions sont de deux types:

1. Au niveau bloc de base, on trouve les fonctions de flot primaires. Ces fonctions doivent être capables de décrire l'influence du bloc sur un ensemble passé en entrée, et de façon plus précise, l'influence des expressions composant le bloc sur cet ensemble. Cette influence peut se résumer à la question: pour chaque élément de l'univers, selon sa présence ou non dans l'ensemble d'entrée, ou ensemble IN, est-il ou non dans l'ensemble renvoyé en sortie, ou ensemble OUT? Pour représenter ces fonctions

primaires, on utilise une table associant les éléments à une valeur booléenne, qui s'interprète comme suit : si l'élément est associé à vrai, alors il sera dans l'ensemble de sortie, et inversement s'il est associé à faux. Si l'élément n'est pas dans la table, alors il n'est pas influencé par le bloc. L'élément peut être aussi associé à vrai ou faux en fonction d'un ou de plusieurs autres éléments de l'ensemble d'entrée: sa valeur associée est donc indéterminée. On trouvera donc en outre des éléments de la table associés à des fonctions booléennes simples égales par exemple à la valeur d'un élément, ou à des opérations booléennes sur les valeurs d'éléments comme $Val_{elt1} \wedge Val_{elt2}$. Les fonctions de flot primaires sont les plus délicates à construire.

2. Au niveau des blocs supérieurs, les fonctions de flot seront construites en composant simplement les fonctions de flot suivant le type de bloc et d'analyse. Cette composition étant hiérarchique, le premier niveau correspond à la composition des fonctions de flot primaires. Cette composition de blocs déjà abordée est représentée sur la figure A.5. Elle reste relativement triviale.

Deux passes seront nécessaires afin de construire les fonctions de flot de données.

Construction des fonctions de flot de données - première passe La première passe résout plusieurs points de l'analyse. Elle permet tout d'abord de construire l'univers propre à l'analyse visée. C'est aussi au cours de cette passe qu'est initiée la construction des fonctions de flot de données, **mais en restant dans le cadre du bloc**. La figure A.8 permet d'illustrer cette première passe sur l'exemple déjà étudié d'analyse des définitions accessibles (cf. section A.2.2 p. 182). La fonction de flot issue de la première passe se résume donc pour chaque bloc de base à une simple table indiquant quels sont les éléments de l'univers influencés par celui-ci, dans l'absolu et au niveau du bloc: celui-ci est aveugle à l'existence de ses frères. A noter que la figure A.7 représente l'univers obtenu à l'issue de la première passe.

Pour mener à bien cette passe, on doit bien sûr connaître la propriété sur laquelle on travaille. Celle-ci doit être définie pour chaque type d'analyse, d'où la nécessité de mettre en place un formalisme de définition qui soit le plus simple possible. Ce formalisme contient entre autre le sens de l'analyse (direct, inverse ou les deux), les règles de confluence (il s'agit de savoir s'il faut faire l'union ou l'intersection des ensembles entre les blocs ce qui dépend du type d'analyse à réaliser) et la définition d'une fonction capable de retourner l'influence d'une expression. Autrement dit, elle doit renvoyer les éléments de l'expression qui sont susceptibles de vérifier la propriété, en précisant l'effet de l'expression sur ceux-ci. Par exemple, la fonction appelée avec l'instruction *'f0 = 0'* doit renvoyer cette définition de variable, en précisant qu'elle est "générée" par l'instruction. C'est enfin au cours de cette étape que les règles de confluence sont utilisées pour composer les fonctions entre les blocs.

Construction des fonctions de flot de données - deuxième passe Cette passe permet de tenir compte des interactions **entre les expressions**, et raffine la fonction de

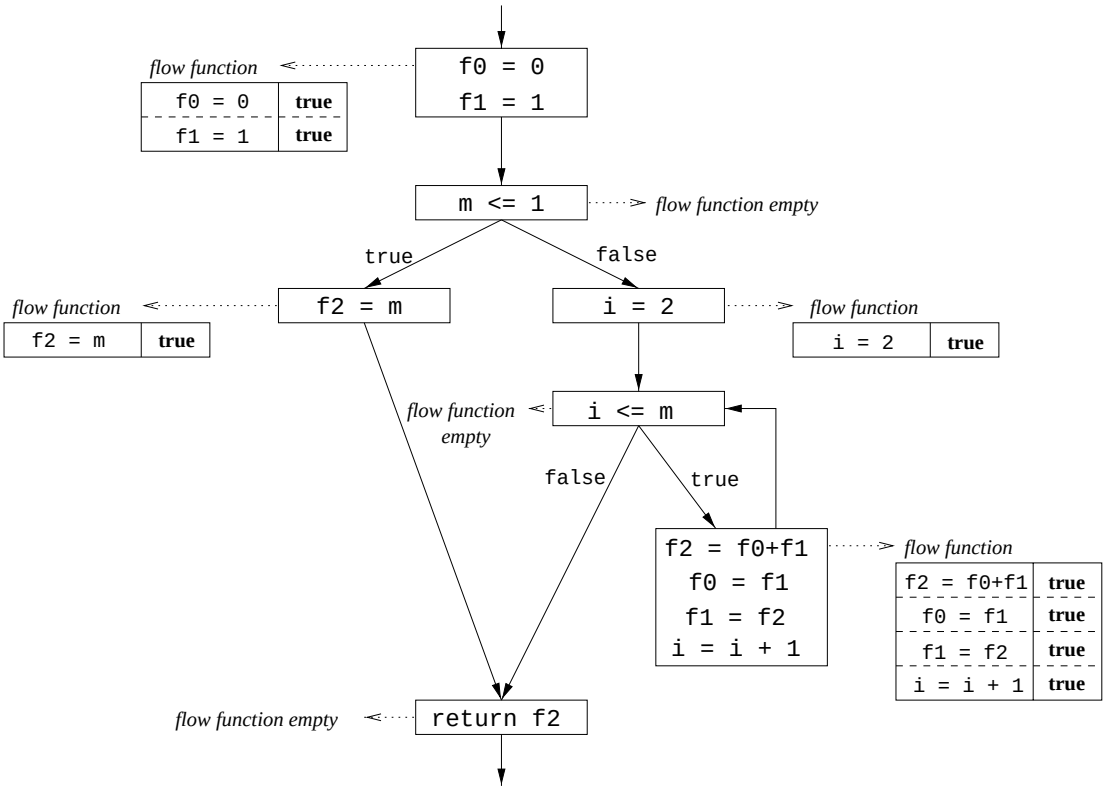


Figure A.8: Construction des fonctions de flot de données, **première passe**.

flot de chaque bloc de base, fonction de flot initiée au cours de la première passe. On utilise ici une deuxième fonction définie dans la propriété : celle-ci est appelée séquentiellement pour tous les couples (expression - élément de l'univers) et peut alors préciser l'influence de l'expression sur l'élément.

La figure A.9 permet de mieux comprendre cette étape. Par exemple, la définition de la variable *f0* dans l'instruction '*f0 = f1*' tue celle de l'instruction '*f0 = 0*'.

A l'issue de cette passe, on obtient donc pour chaque bloc de base une fonction de flot sous forme de table indiquant l'influence réelle du bloc sur l'univers, et incluant l'interaction avec ses congénères. Les fonctions de flot traduisant l'interaction entre les blocs sont celles construites au cours de la première passe: elles restent inchangées. Cette deuxième passe n'est pas indispensable si la première passe suffit à construire entièrement les fonctions de flot de données. Dans ce cas, la propriété n'est jamais tuée, et l'analyse revient à un simple parcours des instructions.

Une petite parenthèse sur les fonctions devant être écrites pour chaque propriété. Un certain formalisme, du reste assez peu contraignant, doit être suivi pour les écrire. En fait, l'interface entre les propriétés (les fonctions décrites ci-dessus) et le moteur d'analyse de flot de données est une simple liste de doublets, facile à utiliser et à mettre en œuvre. Un doublet contient un élément susceptible de vérifier la propriété et une expression booléenne.

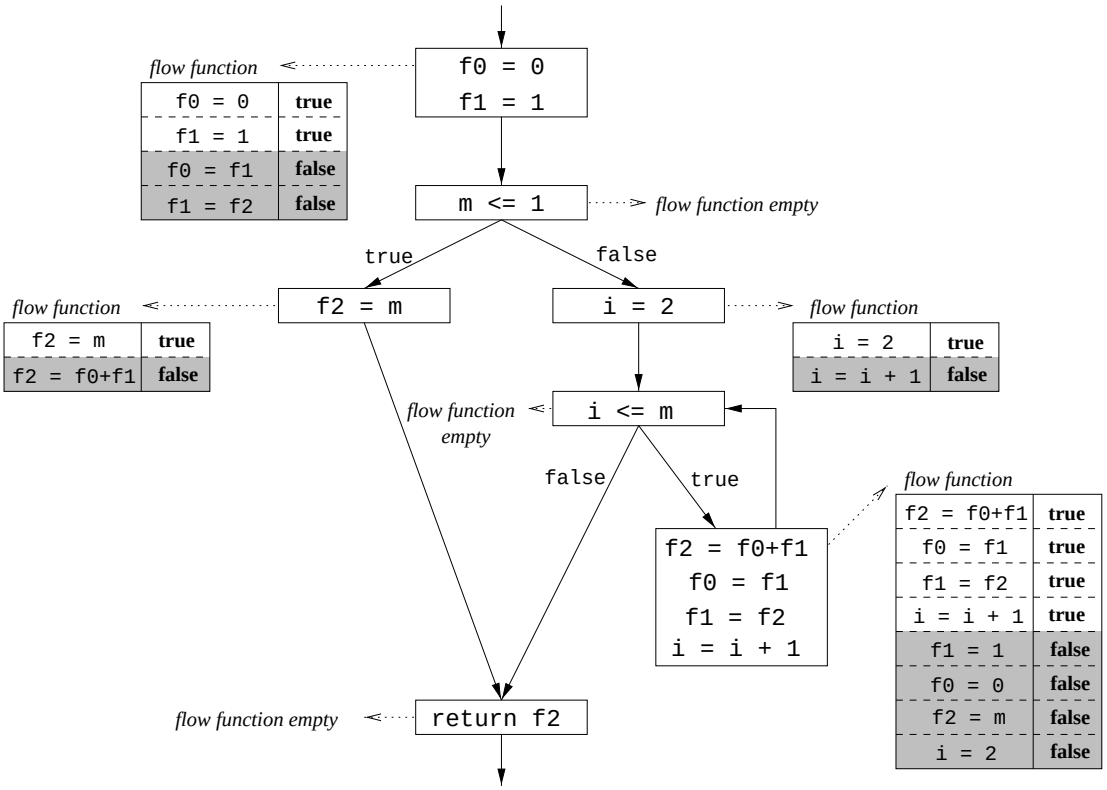


Figure A.9: Construction des fonctions de flot de données, **deuxième passe**.

A.3.4.3 Calcul des ensembles

Une fois les fonctions de flot de données construites, et connaissant parfaitement les structures de contrôle, on peut résoudre les équations, c'est-à-dire calculer les ensembles IN et OUT, but ultime. Pour cela, on commence par fixer l'ensemble initial, souvent l'ensemble vide, ce qui traduit le fait qu'aucun élément vérifiant la propriété n'a encore été trouvé. On effectue ensuite le calcul sur tous les blocs, en propageant les ensembles IN et OUT entre les blocs (par exemple, la sortie d'une condition est égale à l'entrée du "then" ET à celle du "else") et en enrichissant le graphe de flot de contrôle. La propagation des ensembles IN et OUT est régie par les règles de confluence, ce qui illustre un des intérêts de la représentation structurelle, dans la mesure où ces dernières sont implicites. En effet, à tout niveau, des schémas standards sont manipulés et l'enchaînement des blocs est maîtrisé. Après un parcours de ce graphe, on obtient à chaque niveau les ensembles IN et OUT. La figure A.10 montre ces résultats aux niveaux des blocs de base. Au niveau des blocs supérieurs, les ensembles IN et OUT résultent des fonctions de flot composant les ensembles des blocs inférieurs.

Une fois calculées les fonctions de flot, une seule passe est nécessaire pour le calcul des ensembles IN et OUT. Le coût global nécessaire est finalement peu élevé. Grossièrement, si N_{bb} est le nombre total de blocs de base dans le graphe, N_{sp} le nombre de nœuds

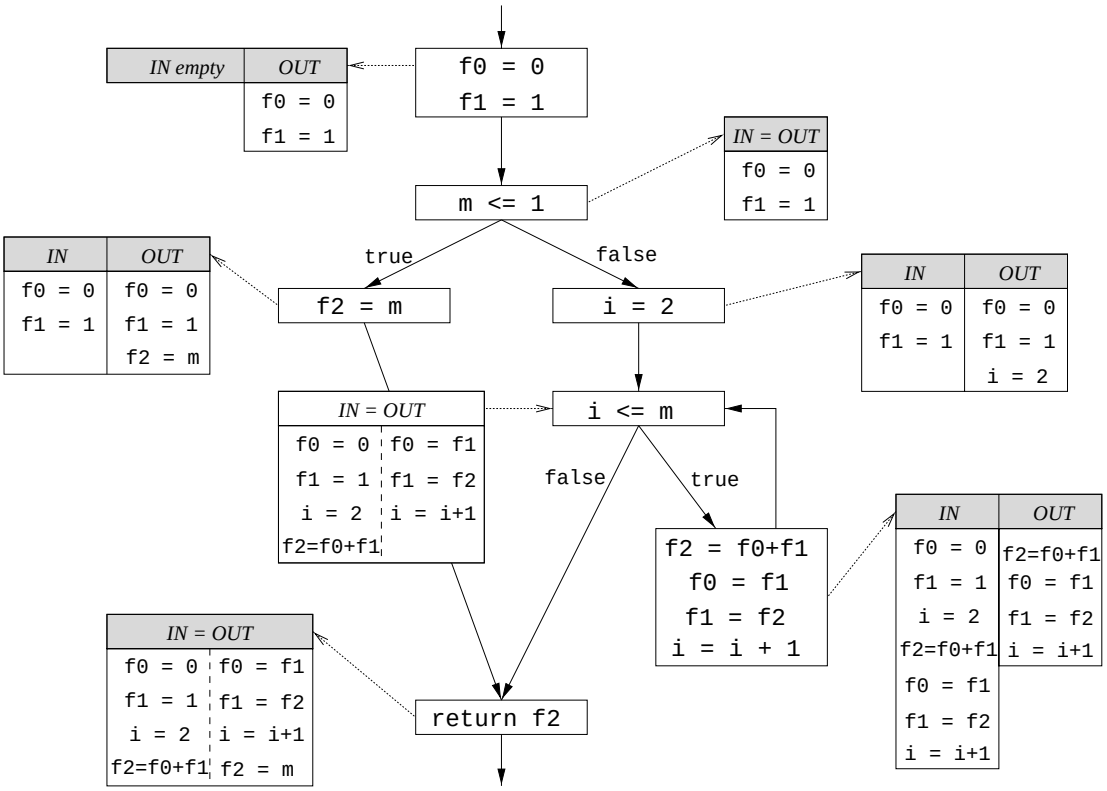


Figure A.10: Calcul des ensembles IN et OUT.

(soit de super-blocs) du graphe, N_{exp_i} le cardinal du bloc de base i c'est-à-dire le nombre d'expressions dans ce bloc de base et N_Φ le cardinal de l'univers pour la propriété à résoudre, le coût peut être estimé à :

$$C = N_{exp} + N_{sp} + N_\Phi \times N_{exp} + N_{sp} = N_{exp} \times (1 + N_\Phi) + 2 \times N_{sp}$$

où $N_{exp} = \sum_{i=1}^{N_{bb}} N_{exp_i}$ est le nombre total d'expressions dans la procédure analysée. Ce coût représente le nombre d'opérations à effectuer pour une analyse. Pour la première expression, le premier terme correspond à la première passe du calcul des fonctions de flot, le second au calcul des fonctions de flot des super-blocs supérieurs par composition ce qui correspond à un parcours de l'ensemble des super-blocs. Le troisième terme représente la deuxième passe du calcul des fonctions de flot primaire; c'est de loin la partie la plus coûteuse de l'algorithme bien qu'il soit utile de préciser que l'on a toujours $N_\Phi \leq N_{exp}$. Enfin, le quatrième terme correspond au calcul des ensembles, qui revient à parcourir à nouveau tous les super-blocs du graphe, en profondeur d'abord. Comparer avec la méthode itérative n'est pas une tâche triviale, surtout sans en avoir implémenté une version. On peut raisonnablement estimer que la méthode présente un coût supérieur de deux ou trois passes par rapport à une itération de la méthode itérative, sachant que cette dernière itère au minimum deux fois. Autrement dit, la présente méthode est particulièrement intéressante pour des analyses un tant soit peu complexes, sur des programmes fortement bouclés,

puisque ce sont les boucles présentes dans le programme qui vont influer sur le nombre d'itérations nécessaires à la méthode classique pour atteindre l'équilibre. Mais c'est ici le domaine de la conjecture faiblement éclairée par l'expérience, mais relayée par des avis experts (cf. [89]).

A.3.5 Pointeurs, appels de procédures et ruptures de séquences

A.3.5.1 Influence des pointeurs et des appels de fonctions

Les pointeurs et appels de fonctions posent des problèmes particuliers dans le cadre de l'analyse de flot de données. Ceux-ci peuvent en effet provoquer des effets collatéraux remettant en question la fiabilité de l'analyse sans traitement particulier. Sur l'exemple A.11.a), i et j sont des variables globales et i est affectée par un appel à la procédure *set*. Imaginons que l'on lance la recherche des définitions accessibles (*reaching definition*) sur le corps de la procédure *main*. En l'absence de traitement spécifique des appels de fonctions, l'analyse de flot de données va trouver la définition '$i = 1$' comme étant accessible en sortie du corps de *main*, ce qui est faux.

```
int i,j;
void set(int x) {
  i = x;
}
int value() {
  return(i);
}
void main(){
   i = 1;
   set(3);
   j = value();
}
```

a)

```
int *p, *i;  int j;
i = p;
do {
  *p = 3*(*p);
  j = j + *i;
} while(1);
```

b)

Figure A.11: Effets collatéraux possibles des pointeurs et appels de procédures

De même les pointeurs peuvent introduire de dangereux effets de bord, comme le montre l'exemple A.11.b), où i et p pointent sur la même zone mémoire. Si l'on cherche les variables d'induction de base, j sera déterminée comme telle puisque i est, selon des critères classiques, une constante de boucle. En réalité l'affectation de *p affecte aussi i et j n'est donc pas une variable d'induction de base.

Afin d'être traité convenablement, le cas de l'exemple A.11.a) nécessite une analyse interprocédurale, tandis que le cas de l'exemple A.11.b) requiert une analyse d'alias. Ces analyses supplémentaires font partie des travaux à mettre en œuvre et ne sont donc pas

disponibles. La solution adoptée, à défaut de traitement salvateur, ne peut être que radicale: elle consiste à considérer par défaut que tout appel de procédure et toute affectation de pointeurs remet en question l'analyse du bloc concerné, ce qui impose une remise à zéro de tous les ensembles issus de cette analyse dans le bloc. Quelques nuances cependant: il est laissé aux bons soins de l'utilisateur de désactiver cette protection par défaut dans le cas où ce dernier estime qu'elle n'est pas nécessaire. Cette désactivation peut être totale, dans le cas où l'utilisateur estime que tous les appels de procédures et/ou toutes les affectations de pointeurs sont sans danger pour l'analyse. Cette désactivation peut être partielle en fournissant à l'analyseur préalablement à toute analyse une liste de pointeurs et/ou procédures n'ayant pas d'effets collatéraux. Ces informations peuvent être issues d'outils d'analyses annexes.

A.3.5.2 Les ruptures de séquences

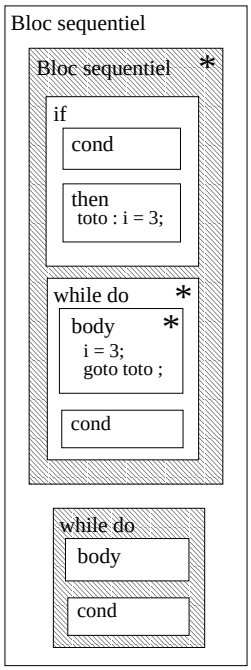


Figure A.12: Les blocs invalidés

Le graphe de flot de contrôle est représenté sous la forme d'un modèle structurel, qui permet de préserver plus efficacement les informations de haut-niveau et qui est adapté au modèle d'analyse de flot de données choisi. Ce modèle structurel de GFC ne prend cependant pas en compte les ruptures de séquences telles que *goto*, *break*, *continue* et autres *return*. En attendant de mettre en œuvre la modification de la structure de base requise, la solution adoptée est relativement extrémiste : le moteur de construction du GFC marque les blocs contenant des ruptures de séquences comme étant impropres, et leur associe une fonction de flot de données nulle (renvoyant toujours l'ensemble vide). En pratique, une telle fonction

signifie que d'une part, les calculs ne sont pas effectués sur les blocs d'un niveau inférieur, et que d'autre part, toutes les informations de flot de données sont remises à zéro lors du passage par ce bloc. Afin de minimiser le nombre de blocs invalidés, le moteur recherche le bloc père contenant à la fois le branchement et sa destination[7], puis invalide le bloc courant ainsi que tous les blocs de hiérarchie supérieure jusqu'à parvenir au bloc englobant. Ceci est illustré sur la figure A.12.

Dans le cas d'un *return* sauvage par exemple, c'est-à-dire un *return* situé ailleurs que sur le nœud final du GFC d'une procédure, toute la procédure est invalidée. Dans le cas contraire, le *return* n'a aucun effet sur l'analyse.

Cette solution reste évidemment une solution temporaire. Le traitement des ruptures de types *return, break* ou *continue* reste facilement solvable: il s'agit de permettre à un bloc de transmettre son ensemble de sortie respectivement en sortie de la procédure traitée, en sortie de la boucle le contenant et en entrée de celle-ci. Le cas des *goto-label* est plus complexe et nécessite une étude plus approfondie, puisque ceux-ci peuvent violer totalement les concepts hiérarchiques associés au graphe.

[7] toute rupture de séquence sauvage peut se ramener à un goto-label

Annexe B

Description des ressources d'adressage

La description des ressources d'adressage passe par l'écriture d'un fichier permettant d'exposer d'une part, les ressources de stockage disponibles (registres d'adresse et d'index) et d'autre part, les opérations autorisées. Les deux exemples ci-dessous représentent les descriptions des ressources et opérations autorisées par les AGU (Address Generation Unit) des processeurs Sapphire et MMDSP. *@ea* représente l'adresse effective et s'emploie dans le cas de modes d'adressage pré-calculés ou non-modifiants. Les registres peuvent être décrits sous forme de banc de registres et employés comme tels dans le cas d'opérations communent à tous les registres, comme c'est le cas pour la description du Sapphire. Lorsqu'un mode d'adressage peut employer un immédiat issu directement de l'instruction, la notation *#imm* est employée, à laquelle est attachée la taille en bits de la valeur possible. Les valeurs minimum et maximum peuvent aussi être fournies.

```
// addressing spec of sapphire
MEMORIES {
      X,Y;
}
REGISTERS {
   AGU_REGISTERS {
      ADDRESS {
          Ax: Ax[0..1];
          Ay: Ay[0..1];
      }
      INDEX {
          Dx: Dx[0..1];
          Dy: Dy[0..1];
      }
   }
}
OPERATIONS {
   AGU_OPERATIONS {
       Ax: ++;
       Ay: ++;
       Ax: +=Dx: ;
       Ay: +=Dy: ;
       @ea = Ax: ;
       @ea = Ay: ;
       @ea = Ax:  + Dx: ;
       @ea = Ay:  + Dy: ;
       @ea = Ax:
+ #imm(4);
       @ea = Ay:
+ #imm(4);
   }
}
```

```
// addressing spec of mmdsp

MEMORIES {
        X,Y;
}

REGISTERS {
   AGU_REGISTERS {
      ADDRESS {
          Ax: AX[1..6];
      }
      INDEX {
          Ix: IX[1..3];
      }
   }
}

OPERATIONS {
   AGU_OPERATIONS {
          Ax: ++;
          Ax: --;
          Ax:  += 2;
          Ax:  -= 2;
          Ax:
+= #imm(16);
          AX[1] += IX[1];
          AX[4] += IX[1];
          AX[2] += IX[2];
          AX[5] += IX[2];
          AX[3] += IX[3];
          AX[6] += IX[3];
          AX[1] -= IX[1];
          AX[4] -= IX[1];
          AX[2] -= IX[2];
          AX[5] -= IX[2];
          AX[3] -= IX[3];
          AX[6] -= IX[3];
   }
}
```

Annexe C

Exemples de codes

C.1 Boucles imbriquées

nest5

Code initial:

```
int i,j;
for(i=0;  i<40;  i++) {
  for(j=20;  j>0; j--) {
    a[i*2+j] = b[i] + a[j]
               + 3*b[i+3];
  }
  b[i] = a[2*i+5] + 4*b[i+2]
         + 3;
}
```

Code *arTé*: 4 pointeurs

```
int i;
int j;
int *ptra;
int *ptra0;
int *ptra1;
int *ptrb;

ptra1 = &a[20];
ptrb = &b[3];
for (i = 0;  i < 40;  i++) {
   ptra = &a[20];
   ptra0 = ptra1;
   for (j = 20;  j > 0;  j--) {
     *ptra0 = *(ptrb - 3) + *ptra
              + 3 * *ptrb;
     ptra = ptra - 1;
     ptra0 = ptra0 - 1;
   }
   *(ptrb + -3) = *(ptra1 + -15)

+ 4 * *(ptrb - 1)
                  + 3;
   ptra1 = ptra1 + 2;
   ptrb = ptrb + 1;
}
```

Ecriture manuelle: 3 pointeurs

```
int * pa, * pa0, * pb;
pa = &a[20];
pb = b;
```

```
for(i=0;  i<40;  i++) {
    pa0=&a[20];
    for(j=20;  j>=0; j--) {
       *pa = *pb + *pa0
             + 3 * *(pb+3);
       pa--;
       pa0--;
    }
   pa+=20;
   *pb =*(pa-15) + 4 * (*pb+2)
         + 3;
   pa+=2;
   pb++;
}
```

convolution

Code initial:

```
   for (n = 0;  n < SIZE;  n++) {
     acc = 0;
     for (i = 0;  i < n;  i++)
       acc += a[i] * b[n-i];
     conv[n] = acc;
   }
```

Code *arTé*: 4 pointeurs

```
int *ptrb;
int *ptra;
int *ptrb0;
int *ptrconv;
ptrb0 = b;
ptrconv = conv;
for (n = 0;  n < 100;  n++)
  {
  acc = 0;
  ptrb = ptrb0;
  ptra = a;
  for (i = 0;  i < n;  i++)
    {
    acc = acc + *ptra * *ptrb;
    ptrb = ptrb - 1;
    ptra = ptra + 1;
    }
  *ptrconv = acc;
  ptrb0 = ptrb0 + 1;
  ptrconv = ptrconv + 1;
}
```

Ecriture manuelle: 3 pointeurs

```
int *ptrb;
int *ptra;
int *ptrconv;
ptrb = b;
ptrconv = conv;
for (n = 0;  n < 100;  n++)
  {
  acc = 0;
  ptra = a;
  for (i = 0;  i < n;  i++)
    {
    acc = acc + *ptra * *ptrb;
    ptrb = ptrb - 1;
    ptra = ptra + 1;
    }
    *ptrconv = acc;
    ptrb = ptrb + n;
    ptrb = ptrb + 1;
    ptrconv = ptrconv + 1;
}
```

C.2 Exemple de génération de C bas-niveau

Les compilateurs utilisés permettent de cibler la machine à l'aide de directives spécifiques, comme illustré sur l'exemple suivant :

matrixmpy

```
register int * ptra At_reg(AX[1]);
register int * ptrb At_reg(AX[2]);
register int * ptrc At_reg(AX[3]);
register int * ptrb0 At_reg(AX[4]);
register int * ptra1 At_reg(AX[5]);
register int * ptrc2 At_reg(AX[6]);

ptra1 = &a[0][0];
ptrc2 = &c[0][0];
loop(10) {
    ptrc = ptrc2;
    ptrb0 = &b[0][0];
    loop(10) {
        *ptrc = 0;
        ptra = ptra1;
        ptrb = ptrb0;
        loop(10) {
            *ptrc = *ptrc + *ptra * *ptrb;
            ptra = ptra + 1;
            ptrb = ptrb + 10;
        }
        ptrc = ptrc + 1;
        ptrb0 = ptrb0 + 1;
    }
    ptra1 = ptra1 + 10;
    ptrc2 = ptrc2 + 10;
  }
```

Annexe D

Consommation en technologie CMOS

On distingue deux types de puissance consommée: la consommation en régime dynamique, c'est-à-dire liée aux commutations internes d'un circuit donné, de loin la plus conséquente en technologie CMOS, et la consommation statique, c'est-à-dire indépendante de toute activité intrinsèque.

D.1 Composante dynamique

La composante dynamique de la consommation peut elle-même être scindée en deux parties; la première est liée aux charges et décharges périodiques des capacités parasites du circuit lors des commutations internes : elle est nommée puissance capacitive ou puissance de commutation[1]; la seconde est liée aux court-circuits entre les lignes d'alimentation créés lors des transitions internes, appelée fort à propos, puissance de court-circuit[2].

La composante dynamique est de loin la composante principale de la consommation d'un circuit. A cette composante est attachée la notion d'activité. En effet, elle résulte des commutations aux nœuds internes, induites par l'état précédent des nœuds et les entrées présentes fournies au circuit. Ce lien entre activité et consommation fait de cette dernière une cible particulièrement difficile à atteindre : donner une consommation instantanée n'a guère de sens par exemple, puisqu'elle représente l'état du circuit à un moment donné de son fonctionnement, alors que l'on cherche en général à caractériser un circuit pour un fonctionnement moyen. Cette consommation instantanée gagnera un sens si l'on recherche les conditions d'un pic de consommation, qui peut poser certains problèmes particuliers.

On sera donc amené à accorder une attention toute particulière au type d'entrées fournies à un circuit donné, puisque ces entrées conditionneront l'activité interne du circuit. On pourra à ce niveau trouver quelques petites analogies avec le test. D'autre part, étant donné le peu de signification du calcul d'une puissance instantanée, on sera amener à considérer principalement des puissances moyennes, c'est-à-dire l'estimation de la puissance

[1] switching power
[2] short-circuit power

moyenne, et sa réduction, ce qui implique l'emploi intensif de statistiques et probabilités, seules à même de rendre compte d'une activité moyenne et donc d'une puissance moyenne.

D.1.1 Charge et décharge des capacités parasites

Modélisation des capacités parasites Il est commun de considérer une capacité parasite de charge reportée au nœud de sortie d'une porte logique. Cette capacité, notée C_L, est elle-même issue de plusieurs composantes. Une modélisation peut en être la suivante. La figure D.1 illustre les principales capacités parasites d'un transistor MOS.

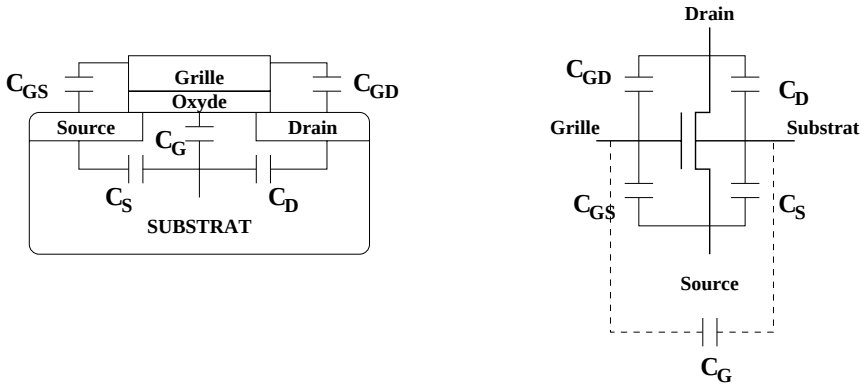


Figure D.1: Capacités parasites d'un transistor MOS

Si l'on considère une porte CMOS, constituée d'un réseau p complémentaire d'un réseau n, la capacité parasite de cette porte, susceptible d'être chargée/déchargée de façon périodique, peut se modéliser par:

$$\begin{aligned} C_L &= C_w + nC_o + mC_i \\ avec: \; C_w &= \textit{capacité de connexion} \\ C_o &= C_D + 2C_{GD} \\ C_i &= C_G + C_{GS} + 2C_{GD} \\ n &= \textit{nombre total d'étages de transistors} \\ m &= \textit{nombre total de transistors de charges.} \end{aligned}$$

Energie de commutation consommée par un inverseur C'est l'exemple de base dont le mérite principal est l'extrême simplicité. Il est généralisable à tout type de porte CMOS, constituée d'un réseau p complémentaire d'un réseau n.

Le modèle capacitif choisi ici considère une seule capacité parasite équivalente en sortie. Une transition descendante en entrée provoque une transition montante en sortie, qui a pour effet de charger la capacité parasite C_L à travers le transistor (réseau) p qui présente une certaine résistivité (fig D.2). La quantité d'énergie stockée dans la capacité est alors $E_{Stock} = \frac{1}{2} \times C_L \times V_{dd}^2$. Etant donné l'énergie fournie par l'alimentation, égale à $E_T =$

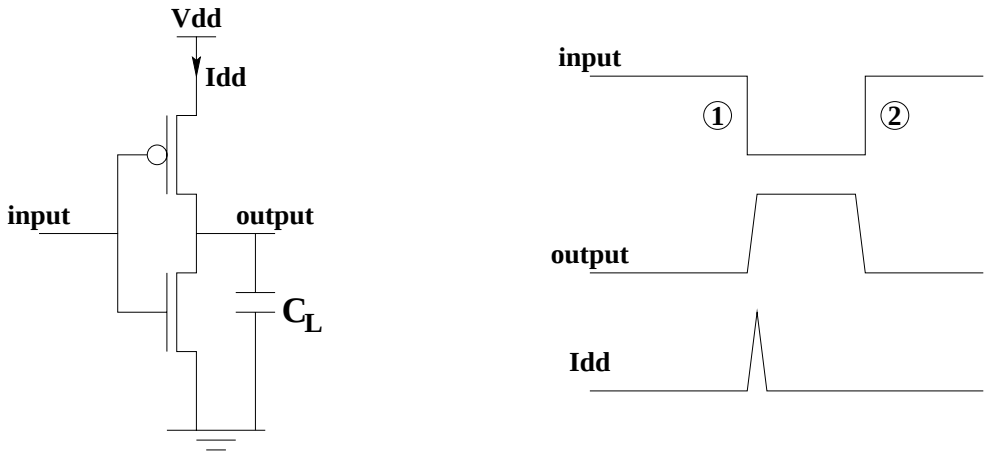


Figure D.2: Energie de commutation d'un inverseur

$V_{dd} \times Q = C \times V_{dd}^2$, où Q est la quantité de charges, on en déduit facilement l'énergie dissipée $E_{Dis} = \frac{1}{2} \times C_L \times V_{dd}^2 = (\sum R) \times I_{dd}$.

Lors d'une transition montante en entrée, provoquant une transition descendante en sortie, la capacité C_L se décharge et l'énergie stockée se dissipe à travers le transistor (réseau) n. A chaque commutation en sortie est donc associée une dissipation énergétique égale à $\frac{1}{2} \times C_L \times V_{dd}^2$. Pour calculer l'énergie de commutation consommée au cours d'une période T, il suffit de donc de compter le nombre N de transitions en sortie et de multiplier par l'énergie consommée au cours d'une transition, soit $\frac{N}{2} \times C_L \times V_{dd}^2$.

> Remarque: Notons qu'au cours d'un cycle complet de charge/décharge, est dissipée la quantité $C_L \times V_{dd}^2$, la moitié dans le réseau n, l'autre dans le p. Partant du principe qu'une charge à un nœud donné X d'un circuit est forcément suivie à un moment ou un autre de la décharge de ce même nœud X, certains auteurs ne considèrent qu'un seul type de transition, montante par exemple, nommant celle-ci transition consommante[3]. Pour calculer l'énergie de commutation consommée, il suffit alors de ne compter que le nombre N' de transitions consommantes durant T, et d'appliquer la formule $N' \times C_L \times V_{dd}^2$. Pour T suffisamment grande, il est clair que $N' \simeq \frac{N}{2}$.

La puissance moyenne consommée par une porte, dont la dépense énergétique est $\frac{1}{2} \times C_L \times V_{dd}^2$, au cours d'une période T, sera $P_{Moyenne} = \frac{1}{T} \times \frac{N}{2} \times C_L \times V_{dd}^2$. Si maintenant T représente la période d'horloge d'un circuit, et si l'on considère un nombre moyen de commutations en sortie de cette porte au cours d'une période d'horloge T égal à α, alors la puissance consommée à ce nœud sera:

$$P_{Moyenne} = \frac{1}{2} \times \alpha \times C_L \times V_{dd}^2 \times f \tag{D.1}$$

[3]power consuming transition

f étant la fréquence d'horloge.

En général, α est inférieur à 1, car il est rare qu'un nœud commute plus d'une fois par cycle, ce qui est déjà une activité conséquente. Cela peut advenir si par mégarde, se produisent à ce nœud des commutations inutiles à la fonctionnalité, dont l'apparition est liée à la propagation des signaux jusqu'aux entrées de cette porte, et au délai de propagation de cette dernière. Cela résulte en une augmentation de la puissance moyenne consommée à ce nœud. Cette activité supplémentaire aura d'autant plus de poids sur la puissance consommée globale que le nœud très actif sera lourdement chargé (cas d'un grand *fanout* par exemple, ou de la sortie d'un gros *buffer*). Une astuce évidente sera alors d'essayer de diminuer l'activité ou la capacité liée à ce nœud, afin de diminuer la consommation du circuit.

Le produit $\alpha \times C_L$ est souvent remplacé par le terme C_{eff} (pour $C_{effective}$) ou $C_{commutée}$ (correspondant à l'anglais C_{sw} pour $C_{switched}$). Ce terme représente donc la capacité commutée moyenne par cycle d'horloge. Quant au produit $\alpha \times f$, qui représente le nombre de commutations par seconde à un nœud donné, il se retrouve dans la littérature sous la dénomination *densité de transition,* notée D.

D.1.2 Courant de court-circuit

La deuxième partie de la composante dynamique de la consommation en CMOS est relative à l'inévitable ouverture simultanée des transistors n et p pendant un temps très court, lors de la commutation d'une porte. Ce court-circuit entre V_{dd} et V_{ss}, qui se produit pendant le délai de transition du signal d'entrée pour passer de V_{tn} à $V_{dd} - \mid V_{tp} \mid$ (et inversement), provoque une courte impulsion de courant. Si la moyenne de ce courant est I_{ccM}, la puissance consommée liée au court-circuit, ou puissance de court-circuit, sera égale à $P_{cc} = I_{ccM} \times V_{dd}$.

Le courant de court-circuit dépend des temps de montée et de descente des signaux d'entrée, de la charge en sortie et de la façon dont la porte est conçue (taille relative des transistors) [131, 128]. Ainsi, ce courant diminue lorsque la charge en sortie augmente, dominé par le courant de charge/décharge des capacités. Il augmente par contre lorsque les délais de transition des signaux d'entrée augmente, ainsi que lorsque le rapport de taille interne k (entre transistors p et n) augmente.

Une formule répandue donne une approximation de la puissance de court-circuit pour un inverseur non chargé [131]: $P_{cc} \cong \frac{\beta}{12} \times (V_{dd} - 2V_t)^3 \frac{\tau}{T}$ avec τ temps de montée ou de descente de la rampe d'entrée, T période du signal d'entrée, et β facteur de gain du transitor dépendant de la taille W de celui ci et de la mobilité μ des porteurs. Cette formule traduit la dépendance de cette consommation aux délais de transitions en entrée (τ), d'où la règle générale de conception de faire en sorte de toujours obtenir des délais de transition les plus bas possibles. Cette formule ne traduit pas, par contre, la diminution de cette consommation en fonction de la charge en sortie.

Partant de la constatation que l'impulsion de courant est fortement triangulaire (fig. D.3.a), J. Figueras [97] développe une méthode analytique de caractérisation de l'énergie consommée au cours du phénomène, calculant la surface présumée de ce triangle égale en

Figure D.3: Courant de court-circuit

fait à l'energie de court-circuit E_{cc} :

$$E_{cc} = \int_{t1}^{t2} V_{dd}.I_{cc}(t).dt = \frac{1}{2} \times V_{dd} \times I_{max} \times (t2 - t1) = \frac{1}{2} \times \tau \times (V_{dd} - \mid V_{tp} \mid - V_{tn}) \times I_{max}$$

[128] propose de considérer une capacité équivalente de court-circuit, C_{cc} n'ayant aucune signification physique, et traduisant simplement le transfert de charge. L'avantage est dans ce cas d'obtenir une expression de la puissance de court-circuit $P_{cc} = \frac{1}{2} \times \alpha \times C_{cc} \times V_{dd}^2 \times f$ soit une expression élégante pour la puissance dynamique $P_{dyn} = \frac{1}{2} \times \alpha \times (C_{eff} + C_{cc}) \times V_{dd}^2 \times f$, ainsi que le souligne [102].

La consommation résultante de ce courant de court-circuit est souvent tenue pour négligeable pour des circuits bien conçus (transitions rapides notamment), de l'ordre de 10 % de la consommation globale. Cependant, J. Figueras ([97]) montre que le poids relatif de la puissance de court-circuit peut être conséquent, particulièrement pour des temps de transition lents et de faibles charges en sortie ($\simeq 40\%$). De plus, ce poids relatif est susceptible d'augmenter avec la diminution de la taille des transistors, ce qui assombrit l'horizon radieux de l'intégration galopante sévissant actuellement.

D.2 Composante statique

La composante statique de la consommation en technologie CMOS est en général trés faible, ce qui a toujours fait l'attrait de cette technologie pour des applications basse-consommation comparée à des technologies comme TTL (Transistor-Transistor Logic) par exemple, basée sur des transistors bipolaires, et pour lesquelles la consommation statique domine. Elle se définit comme la consommation intervenant en régime statique, c'est-à-dire lorsque tous les signaux sont au repos. Se produit alors malgré tout un faible courant de fuite, noté I_{DDQ}, produisant donc une puissance $P_{fuite} = I_{DDQ} \times V_{dd}$.

Ce courant de fuite est lui-même composé de plusieurs affluents, qui sont résumés sur la figure D.3.b). I_{SUB} est le courant de fuite de "sous-seuil[4]". C'est la composante la plus importante du courant de fuite global. Il se produit lorsque la tension V_{GS} entre grille et

[4]subthreshold

substrat est au-dessus du point d'inversion faible du canal, mais en dessous de la tension de seuil. Dans ce cas, le courant résultant à travers le canal est exponentiellement dépendant de V_{GS} ce qui donne à titre purement informatif : $I_{SUB} = \mu_0 . C_{ox} . \frac{W}{L} . V_t^2 . \exp\left[\frac{V_{GS} - V_{th} + \eta . V_{DS}}{n . V_t}\right]$. D'autre part, si la tension de seuil baisse, $V_{GS} - V_{th}$ augmente, et donc I_{SUB}. I_D est le courant de fuite dû aux diodes équivalentes entre drain et substrat. Il peut être décomposé en trois parties : génération, diffusion et effet dit "tunneling". I_{PT} est le courant de "PunchThrough". I_G est le courant de grille. I_{GIDL} est le courant de fuite de drain induit par la grille (Gate-Induced Drain Leakage Current).

J. Figueras souligne que le courant de fuite, bien qu'indépendant de toute activité provoquant des commutations, dépend de l'état du circuit à un instant donné, donc des vecteurs d'entrées amenant le circuit dans cet état. Cela rend le problème de l'estimation du courant de fuite maximum NP-complet. Des techniques d'ATPG peuvent éventuellement être utilisées pour estimer ce dernier.

La puissance statique peut ne pas être insignifiante pour des systèmes présentant de longues périodes d'inactivité, et aura tendance à acquérir de plus en plus d'importance avec la diminution des dimensions de transistors.

D.3 Poids relatif des deux composantes sur la consommation globale

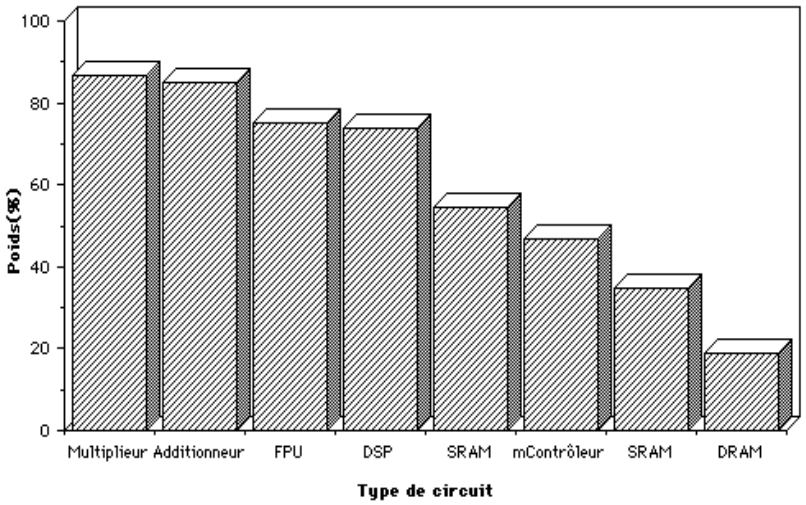


Figure D.4: Poids relatifs de la consommation dynamique (commutations) par rapport à la consommation globale pour différents types de circuits

Ce poids dépend du type de circuit considéré, de la façon dont le circuit est conçu (la priorité aux performances est un facteur qui est en général néfaste en ce sens ...). Ainsi qu'il est cité précédemment, la composante dynamique est en général de loin la plus conséquente. Cet état de fait aura tendance à s'atténuer avec les possibilités d'intégration futures. A titre d'illustration du poids de la puissance liée à l'activité relativement à la puissance consommée globale, [11] fournit un tableau résumé par la figure D.4. Pour des circuits comme une DRAM, dont la consommation statique domine avec une puissance dynamique représentant 18,9% de la puissance globale, l'approche consistant à ne considérer que la consommation liée au fonctionnement peut ainsi amener à de graves erreurs d'appéciations.

D.4 Notions complémentaires

D.4.1 Avantages comparés de la famille logique CMOS

Selon [83], de toutes les familles logiques actuellement utilisées, il s'avère que la famille CMOS statique est la plus prometteuse pour la GSI basse puissance. Les raisons invoquées sont tout d'abord la plus faible consommation en mode d'attente (standby), les plus grandes marges d'opération, la plus grande capacité à être réduite en dimension, et enfin la plus grande souplesse concernant les diverses fonctions qu'elle peut implémenter.

C. Swensson ([97]), compare l'implémentation de fonctions simples à l'aide de logiques différentes, statiques comme CMOS, CVSL (Cascade Voltage Switch Logic) et CPL (Complementary Pass Logic), et dynamiques qui sont les familles logiques à précharge. Ce dernier type de logique s'avère peu recommandable en terme de basse puissance, la palme revenant là aussi à la famille logique CMOS statique. [134] confirme la supériorité du CMOS comparée à la famille CPL.

D.4.2 Distinction énergie/puissance consommée

Il est fréquent de rencontrer dans la littérature une absence de distinction entre énergie et puissance consommée. L'énergie consommée est assimilable au carburant à fournir à un système donné, ou que ce système prélève à une certaine source, afin qu'il puisse exécuter une tâche donnée, cela indépendamment de la vitesse à laquelle cette dernière est exécutée. La puissance, par contre, est relative au débit auquel est délivrée cette énergie, soit à la fréquence à laquelle cette tâche est exécutée. Par exemple, si une technique d'optimisation parvient à réduire de moitié la puissance consommée par un circuit, mais que ce circuit met deux fois plus de temps pour s'exécuter, l'énergie consommée reste la même. Pour les systèmes alimentés par batterie, l'objectif essentiel est une diminution de l'énergie consommée.

> Remarque: Une batterie recèle une certaine quantité d'énergie qu'elle sera amenée à délivrer plus ou moins vite suivant le système à alimenter. Cependant, l'énergie délivrée dépend de la puissance à fournir [103] : en général, alors que le courant augmente, l'énergie delivrée diminue. Cette caractéristique est

souvent représentée par la courbe de l'énergie délivrée par rapport à la puissance fournie, appelée courbe de Ragonne. Les constructeurs donnent ainsi la capacité d'une batterie en Ampère-heures ou Watt-heures (ce qui est une autre façon de représenter l'énergie, traditionnellement donnée en Joules) pour un certain taux de décharge, et une certaine tension.

Différents phénomènes désagréables sont associés à la puissance consommée. Une grande puissance instantanée maximum, liée à des pics de puissance élevés sous certaines conditions, suppose un gros courant (V_{dd} constant) pouvant résulter en des phénomènes d'électromigration qui peuvent atteindre la fiabilité du circuit, ou en des "hot spots" ou pics de chaleur localisés, qui peuvent entamer les performances du circuit, et donc à terme, sa fiabilité. Une puissance moyenne dissipée trop importante résulte en une trop grande chaleur dissipée, ce qui implique l'utilisation de boitier coûteux (céramique) et de systèmes de refroidissements annexes eux-mêmes coûteux.

D.4.3 Notions de limites théoriques et pratiques

[83] expose le point de vue selon lequel les futures opportunités d'intégration pour la GSI (GigaScale Integration) basse puissance seront gouvernées par une hiérarchie de limites théoriques et pratiques : ces limites sont qualifiées de fondamentales, matériaux, transistors, circuits et systèmes. Elles ont l'intérêt de démontrer que les performances maximum liées aux matériaux actuels ne sont pas encore atteintes mais ne sont pas infinies.

Les 3 plus importantes limites fondamentales découlent des principes physiques de base de 1) la thermodynamique, 2) de la mécanique quantique et 3) de l'électromagnétisme. De la première, lié à la notion de puissance de bruit, on déduit une limite minimum pour l'énergie commutée $E_{com} \geq \gamma kT$ soit, pour le facteur $\gamma = 4$ (optimum) et une température de 300 K, $E_{com} \geq 0.104eV$ ce qui peut être interprété comme l'énergie nécessaire pour déplacer un électron à travers une différence de potentiel de 0.104 V. A partir de la mécanique quantique, et plus précisément du principe d'incertitude d'Eisenberg, on montre que la puissance requise pour la transition d'un simple électron au cours d'un temps Δt doit obéir à $P \geq \frac{h}{(\Delta t)^2}$. Quant à l'électromagnétisme, il impose une vitesse de propagation le long d'une interconnexion inférieure à la vitesse de la lumière. Ce sont, évidemment, des limites inférieures théoriques.

Les limites du matériau semiconducteur (Si) sont déterminées par la mobilité des porteurs μ, la vitesse de saturation des porteurs v_s, la force des champs électriques autoionisants E_c et la conductivité thermique K. Pour une interconnexion, la limite matérielle est imposée par la constante diélectrique de son isolant correspondant.

Au niveau du transistor, la limite la plus importante est la longueur de canal minimum d'un MOSFET, L_{min}, qui conditionne son énergie de commutation minimum et son temps de commutation intrinsèque. La réduction de la longueur de canal entraîne des effets appelés "effets de canal court" (short channel effect) : les régions de déplétion de la source et du drain commencent à capturer des ions chargés dans la région centrale du canal, sensée être sous le contrôle strict de la grille. La conséquence de ce type d'effet, qui s'aggrave

lorsque la tension de drain augmente, est de baisser la tension de seuil du transistor affecté, ce qui entraîne une augmentation du courant de fuite de "sous-seuil". L'existence d'une longueur minimum possible pour le canal, limite l'énergie consommée minimum possible pour effectuer une transition. On peut prévoir à terme une longueur de canal inférieure à 60 nm pour des MOSFETs sur substrat, et inférieure à 30 nm pour des MOSFETs SOI (Silicon On Insulator). Pour ce qui est des performances liées à l'interconnexion, la limite principale est relative au temps de réponse d'un réseau RC distribué de façon canonique.

Au niveau circuit, afin d'assurer la restauration du niveau logique en CMOS statique, une tension minimum autorisée est d'environ $\frac{4kT}{q}$, soit environ 0.1 V pour une température T de 300 K, k étant la constante de Boltzmann $(1.38\times10^{-23}J/K)$ et q la charge électronique $(1.6\times10^{-19}C)$. Cette limite de tension est nécessaire si l'on veut qu'une porte CMOS réalise ce pour quoi elle est fabriquée : reconnaître un "zéro" d'un "un". Cependant, autoriser une variation de tension pour une commutation aussi faible impliquerait une tension de seuil si petite, que le courant de fuite résultant serait trop grand pour la plupart des applications. Aussi, un bon compromis est-il de choisir $V_{dd} = V_o = 1.0V$, qui permet de définir des limites d'énergie minimum de commutation acceptables.

D.4.4 Choix d'une métrique

Afin de bien saisir les implications en terme de consommation de décisions prises au cours de la conception, concernant divers choix d'implémentation, le choix de la métrique utilisée revêt une importance capitale. Le problème est donc de choisir une fonction telle qu'elle mette bien en lumière les nécessités propres au niveau de conception concerné. [102] suggère de choisir l'énergie, c'est-à-dire le produit $Puissance \times délai$, comme fonction à minimiser si la vie de la batterie est le principal souci. Si le délai du circuit est aussi primordial, il faut alors s'appliquer à minimiser l'*action*, c'est-à-dire le produit $énergie \times délai$. En fait, la plupart des auteurs parlent souvent de minimisation de la puissance sous des contraintes de délai : il s'agit à strictement parler de minimiser l'énergie consommée, à fréquence fixée.

C. Piguet suggère d'employer un certain nombre de fonctions utiles et élégantes afin de clarifier et préciser les objectifs. Afin de permettre une comparaison aisée entre les diverses solutions s'offrant au concepteur pour implémenter une fonction donnée, une mesure utile est le rapport:

$$\frac{Energie}{Opération} = CPO \times \frac{1}{2} \times C_{eff} \times V_{dd}^2$$

où CPO (Clocks Per Operation) est le nombre de cycles d'horloges nécessaires pour exécuter une opération donnée, et C_{eff}, la capacité commutée moyenne par cycle pour cette opération. Pour les microprocesseurs, un rapport équivalent est:

$$\frac{Energie}{Instruction} = CPI \times \frac{1}{2} \times C_{eff} \times V_{dd}^2$$

où CPI (Clocks Per Instruction) est le nombre de cycles d'horloges nécessaires pour exécuter une instruction donnée. A partir de cette formule simple, on peut distinguer deux principaux cas de figures :

- **Exécution d'une tache donnée nécessitant N opérations.** Le cas se rencontre pour des systèmes travaillant par exemple par à coup (burst mode computation), exécutant un certain nombre d'opérations à la requête de l'utilisateur, et en attente sinon. L'énergie totale sera ici plus intéressante à prendre en considération; l'énergie consommée pour exécuter cette tâche est égale à: $E_{Totale} = \sum^{N} \frac{Energie}{Opération} = \sum^{N} \left(CPO \times \frac{1}{2} \times C_{eff} \times V_{dd^2} \right)$. Une formule similaire est utilisée avec CPI et le nombre N d'instructions pour exécuter une tâche dans un microprocesseur. Des formules ci-dessus, on peut immédiatement déduire qu'il faut réduire le nombre d'opérations par tâche, et le nombre d'étapes par opération.

- Débit d de $\frac{N opérations}{seconde}$ imposé. Si l'on considère une opération nécessitant CPO cycles, la fréquence nécessaire pour respecter une contrainte sur un débit d est $f = CPO \times d$ (dérivée de $d = \frac{f}{CPO}$). Dans ce cas, il faudra considérer une formule tenant compte de ces contraintes de débit. La puissance consommée sera $P = d \times \frac{Energie}{Opération}$. La formule considérée pourra être le produit $Energie \times délai$ égal à $\frac{Energie}{Opération} \times \frac{1}{f}$, ou le $\frac{débit^2}{Watt}$ en $\frac{MOPS^2}{Watt}$ égal à $d \times \frac{10^{-12}}{\frac{Energie}{Opération}}$, $MOPS$ signifiant *Million d'Opérations Par Seconde.* De formules ssimilaires existent pour les microprocesseurs, en considérant les formules précédantes relativement aux instructions (en lieu et place des opérations), et des $MIPS$ ou *Million d'Instructions Par Seconde.*

Annexe E

Synthèse de haut niveau : contexte particulier

E.1 Généralités

Un bref arrêt s'impose sur les concepts de la synthèse de haut niveau tels qu'ils ont pu être appréhendés au cours du développement de ces travaux, et tels qu'ils seront employés par la suite. Il s'agit plus particulièrement des aspects architecturaux tels que l'on pourra les rencontrer, mais aussi d'ordonnancement, qui influenceront l'estimation. Il sera fructueux de se reporter à [51] pour de plus amples détails.

La synthèse de haut niveau, également connue sous le vocable de synthèse comportementale, est employée dans le but de concevoir des circuits intégrés très spécifiques (ASIC). Elle permet la mise en œuvre sous forme silicium d'une application dont on désire des performances élevées. Elle est l'opération consistant à transposer une description comportementale d'une application - futur circuit - en une description au niveau transfert de registres (Register Transfert Level ou RTL) de cette application. Cette description prend la forme d'une architecture le plus souvent de type Von Neuman[1], composée d'une partie contrôle ou contrôleur, et d'une partie opérative ou chemin de données [35, 48, 129, 85, 50].

L'architecture générée est obtenue après application d'un certain nombre d'étapes dont on pourra trouver une description plus précise dans [52, 100], étapes représentées sur la figure E.1.

Cet environnement de synthèse comportementale se base d'une part sur une description comportementale donnée en VHDL, et d'autre part sur une bibliothèque abstraite d'unités fonctionnelles (UFs). Le sous-ensemble VHDL accepté comporte la plupart des instructions séquentielles typiques de ce langage, et autorise des descriptions complexes manipulant par exemple des boucles conditionnelles imbriquées ou des instructions conditionnelles d'attente pré-définies. La bibliothèque, quant à elle, se compose de deux types de modules : des

[1] C'est un modèle d'architecture pour les systèmes de calculs qui a eu ses heures de gloires ([2]) et quasi universellement utilisé, bien que la tendance lui préfère le modèle Harvard, caractérisé par une séparation des données et des instructions du programme, pour la conception des microprocesseurs actuels.

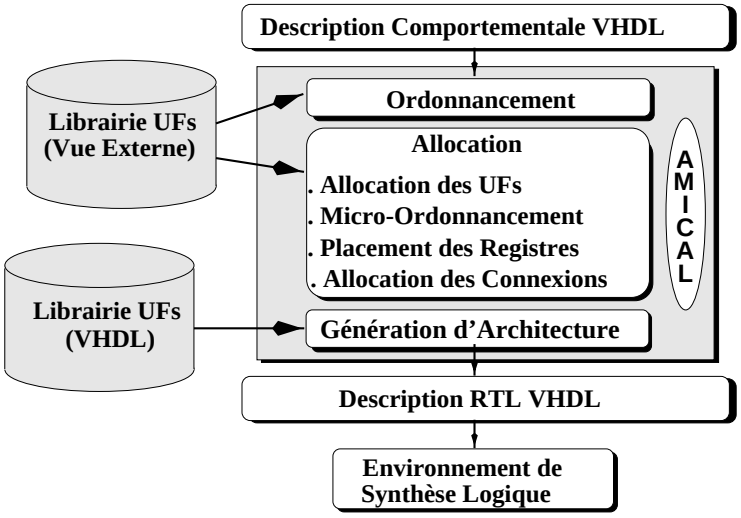


Figure E.1: L'environnement de synthèse de haut niveau AMICAL

modules élémentaires pré-définis au sein du système et donc implicitement présents dans cette bibliothèque (registres, commutateurs, multiplexeurs, ...), et des modules pouvant être complexes et que l'utilisateur a le soin de concevoir et d'abstraire afin de les placer dans celle-ci. Ces composants, pré-définis ou complexes, sont décrits d'une part sous forme de boîtes noires paramétrées et génériques (notamment en ce qui concerne la taille des ports d'entrée), ce qui correspond à l'étape d'abstraction des composants, et d'autre part en VHDL.

L'architecture fournie en sortie du synthétiseur se compose d'un contrôleur et d'une partie opérative décrits au niveau transfert de registres (RTL). Nous reviendrons , au cours des sections suivantes, sur les implications imposées par cette architecture, ainsi que son découpage dans le cadre du modèle présenté dans ce document.

Les possibilités de descriptions comportementales comportant des boucles d'exécutions conditionnelles complexes, et les caractéristiques de l'algorithme d'ordonnancement, font d'AMICAL un outil particulièrement indiqué pour la synthèse de systèmes complexes fortement orientés contrôle, par opposition aux systèmes de synthèse orientés flot de données (DSP). Il est indispensable de prendre en considération cette particularité d'AMICAL dans le cadre de la mise au point d'un modèle tel qu'exposé ici.

On peut dire grossièrement que la partie opérative est un lieu d'action, tandis que la partie contrôle est un lieu de décision. La figure E.2 illustre les principales caractéristiques des deux parties et la façon dont elles se connectent. Le contrôleur consiste en une machine d'états finis ou FSM (Finite State Machine) qui reçoit les signaux de contrôles externes ainsi que les rapports d'exécution du chemin de données, et dirige ce dernier via des signaux de contrôle. Le chemin de données exécute les opérations autorisées par le contrôleur. Il est constitué de plusieurs types d'unités, les unes pour la communication et l'échange des données, les autres pour la manipulation et la transformation des données qui transitent. Cette architecture est synchrone, i. e. un changement d'état du contrôleur, suivi d'actions

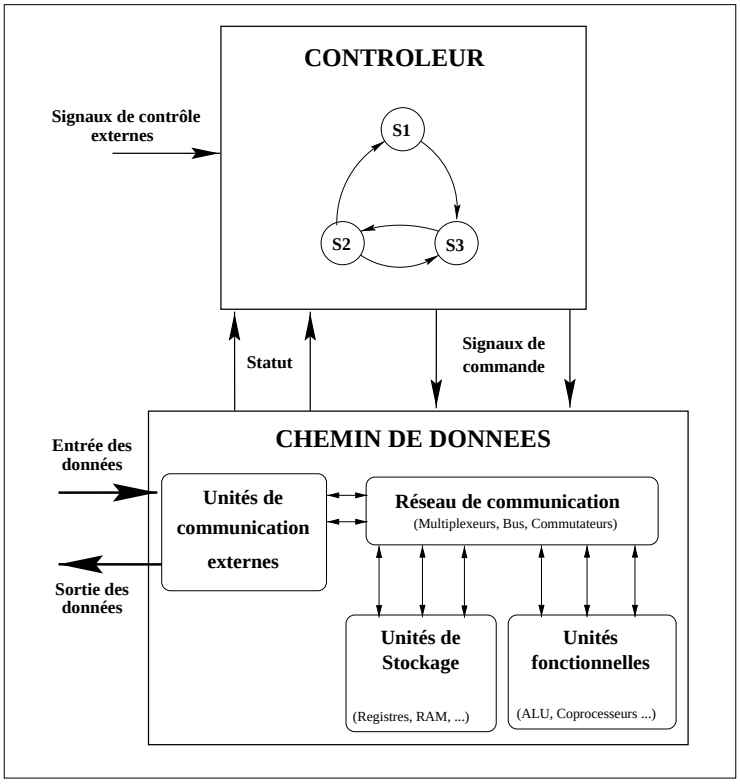


Figure E.2: Modèle général d'architecture

variées au sein des deux parties, intervient sur un événement d'horloge, un front montant le plus communément. Nous considérerons par la suite une horloge à phase unique, et définirons plus précisément le modèle de synchronisation du type d'architecture choisi, c'est-à-dire l'organisation temporelle des évènements des deux parties au cours d'un cycle élémentaire.

E.2 Chemin de données

Conceptuellement, une partie opérative peut se définir par deux aspects: sa structure et son organisation fonctionnelle, c'est-à-dire son modèle de transfert de données. La structure tout d'abord; plusieurs types d'unités se distinguent.

Unités fonctionnelles Appelées aussi unités d'exécution. Ce sont les unités qui vont exécuter les opérations de la description comportementale initiale. De l'opération élémentaire, logique ou arithmétique, mono-cycle, ces unités peuvent gérer une, voire plusieurs, opérations complexes et peuvent s'apparenter à des coprocesseurs ([51]) comportant un contrôle local. Elles peuvent donc être multi-cycles, ou même s'exécuter en un nombre indéfini de cycles. Une unité fonctionnelle peut par exemple être issue

d'une précédente étape de synthèse comportementale, dans un processus de synthèse hiérarchique.

Unités de stockage Registres, banc de registres, RAM, ROM, sont les unités ayant la charge de stocker les données manipulées au sein du chemin de données. Comme elle peuvent manipuler les données via un protocole parfois complexe, RAM et ROM peuvent se décrire comme des unités fonctionnelles particulières exécutant les opérations de stockage de données.

Unités de communication Les unités de communication sont employées pour orienter et dans une certaine mesure contrôler les transferts de données entre les autres unités, cela via le réseau d'interconnexions composé de fils et bus qui sont le support de ces transferts. Nous considérerons ici des multiplexeurs comme unité de communication, et donc la topologie point-à-point en tant que topologie d'interconnexions.

Unités de communication externe Elles réalisent les connexions entre les autres unités et le monde extérieur au circuit.

La puissance de calcul de la partie opérative est fortement dépendante de son modèle de transfert, qui fixe la façon dont les données s'échangent en son sein. Le modèle de transfert considéré ici est relativement général. Traditionnellement, un transfert intervient au cours d'un cycle entre deux registres (nous sommes ici au niveau transfert de registres ...), comme l'illustre l'exemple ci-dessous représentant l'opération c=UF(a,b). Les unités Mux_i sont des multiplexeurs, Reg_i des registres et UF est l'unité fonctionnelle impliquée au cours de ces transferts. Cette opération s'exécute en un cycle, les échanges de données entre registres et entrées de UF intervenant en parallèle.

$$\left[\begin{array}{cc} (1) & (2) \\ \left.\begin{array}{l} Reg_A \rightarrow Mux_1 \rightarrow UF_{in1} \\ Reg_B \rightarrow Mux_2 \rightarrow UF_{in2} \end{array}\right\} & UF_{out} \rightarrow Mux_3 \rightarrow Reg_A \end{array}\right]$$

De façon plus générale, un transfert se définit par une source et une destination, toutes deux unités quelconques du chemin de données. Au cours de celui-ci interviennent de façon séquentielle des échanges et des transformations de données. Plusieurs transferts peuvent s'exécuter en parallèle au cours d'un cycle élémentaire de calcul. Cela suppose l'absence de conflits, c'est-à-dire l'utilisation simultanée des mêmes ressources au cours de transferts différents. Un transfert ne peut avoir lieu que si le chemin correspondant dans la partie opérative est ouvert entre source et destination. Cela concerne le contrôleur qui est en charge d'envoyer préalablement les bon signaux de contrôle. Ces signaux, et le transfert associé, représentent finalement l'expression la plus élémentaire de ce qui est assimilable à une instruction d'un processeur. Dans ce contexte, le jeu d'instructions est l'ensemble des transferts possibles au sein d'un chemin de données.

E.3 Contrôleur

La partie contrôle de l'architecture considérée ici est le centre de décision: elle orchestre tous les transferts dans la partie opérative. Un contrôleur est une machine d'états finis, séquentielle, composée d'états et de transitions entre ces états. De façon plus formelle, une machine d'états finis (FSM) peut être décrite sous forme d'un quintuple:

$$FSM = (S, I, O, FS, FO)$$

où:

$S = \{s_i\}$ est l'ensemble des états;
$I = \{i_j\}$ est l'ensemble des ports d'entrée;
$O = \{o_i\}$ est l'ensemble des ports de sortie;
$FS = \{fs \mid fs : S \times I \rightarrow S\}$ est l'ensemble des fonctions définissant les états suivants;
FO est l'ensemble des fonctions mettant à jour la valeur des ports de sortie

FS, en d'autres termes, est l'ensemble des fonctions donnant l'état suivant en fonction d'un état courant et des entrées. Deux grandes approches définissent de deux façons différentes FO: la première mène aux FSMs appelées automates de Moore, la seconde aux FSMs dites automates de Mealy.

Dans le cas des automates de Moore, FO se définit de la façon suivante: $FO = \{fo \mid fo : S \rightarrow O\}$ c'est-à-dire que la logique de calcul des sorties dépend uniquement de l'état courant de la machine. Dans le cas d'une machine de Mealy, $FO = \{fo \mid fo : S \times I \rightarrow O\}$: la logique de calcul des sorties dépend non seulement de l'état courant de la machine, mais aussi de la valeur des entrées. L'approche de Moore est appelée aussi séquencement conditionnel, tandis que celle de Mealy est parfois nommée génération conditionnelle des commandes.

L'organisation physique d'une FSM peut se décomposer en deux parties distinctes: une partie purement combinatoire, et une partie de stockage d'état courant, cette dernière étant synchronisée par un signal global (horloge) dans le cas de systèmes synchrones. Le nombre N d'états d'une FSM détermine le nombre de ligne d'états requis, et donc le nombre d'éléments de stockage de celle-ci, égal à la partie entière de $log_2(N)$.

Les deux styles de FSMs ne sont pas décrits de la même façon. Deux exemples très simples de machines d'états finis, dont le graphe d'état et la description VHDL sont fournis, seront utiles pour mieux appréhender les différences pouvant exister entre ces deux styles. Un graphe d'états est un graphe où les nœuds représentent les états de la machine, tandis que les arcs sont les transitions entre ces états, généralement associées à des conditions dépendant des entrées. Par exemple, sur la figure E.3, la transition de $S0$ à $S2$ est valide si l'entrée X vaut 1; dans le cas contraire, la machine reste à l'état $S0$. Il s'agit d'une machine de Moore: une valeur de sortie de Z est associée à chaque états. Dans le cas de la machine de Mealy, sur la figure E.4, il est visible que les valeurs de sorties sont associées aux transitions.

Les deux machines ici représentées manifestent le même comportement. Néanmoins, la machine de Moore possède un état supplémentaire. Une autre différence subtile est le degré de réactivité des deux représentations. Alors que pour la machine de Moore, une fois celle-ci dans un état donné, les sorties ne varient pas, il est possible dans le cas d'une machine de Mealy que les sorties varient plusieurs fois avant de se stabiliser, de concert avec une éventuelle variation des entrées.

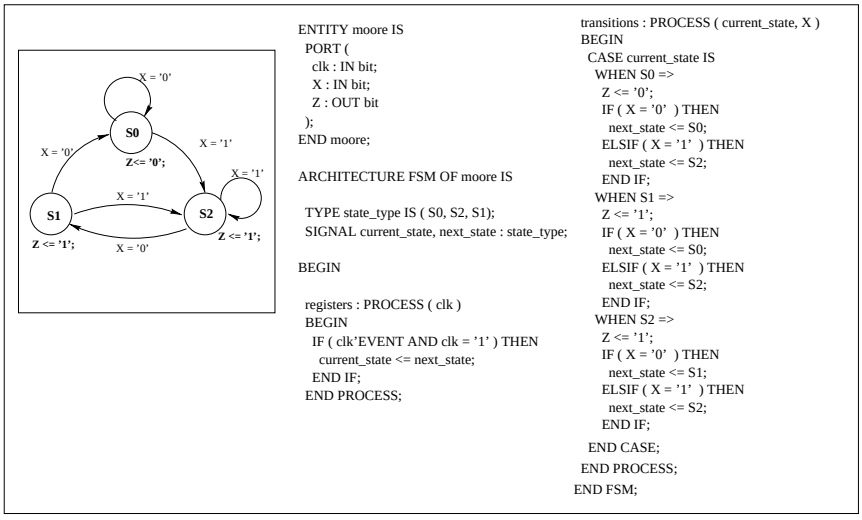


Figure E.3: Automate de Moore: graphe d'état et description VHDL

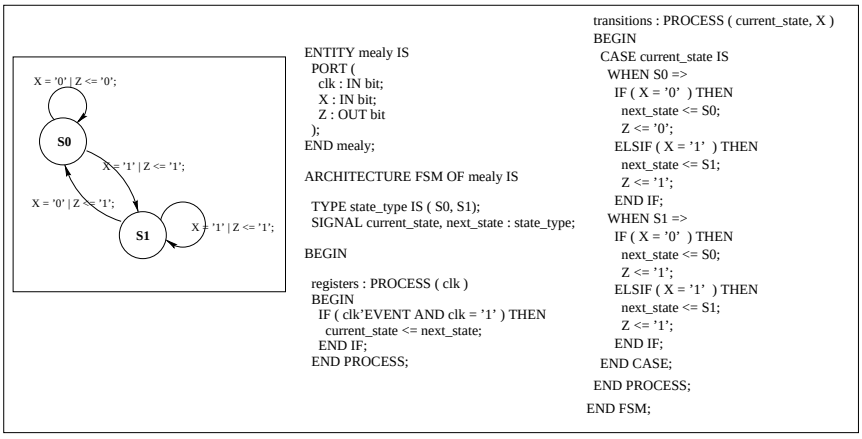


Figure E.4: Automate de Mealy: graphe d'état et description VHDL

Nous considérerons par la suite avoir affaire à des machines de Mealy générant les commandes permettant l'ouverture des transferts au sein du chemin de données. Une telle machine est représentée sur la figure E.5.

La figure E.6 décompose ce qu'il advient dans le circuit au cours d'un cycle d'horloge. Lorsque la FSM entre dans un nouvel état, sur un front d'horloge, l'état suivant est calculé

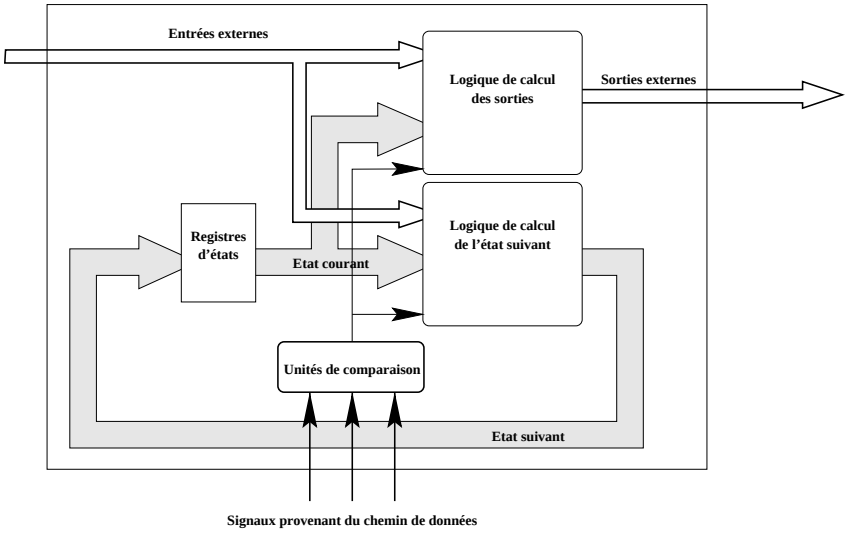


Figure E.5: Représentation d'une partie contrôle de type Mealy

ainsi que les sorties commandant l'ouverture des chemins requis dans la partie opérative. Au cours de ce calcul, il peut se produire un certain flottement au sein du chemin de données. Une fois les signaux de commande stabilisés, les bonnes opérations peuvent s'exécuter.

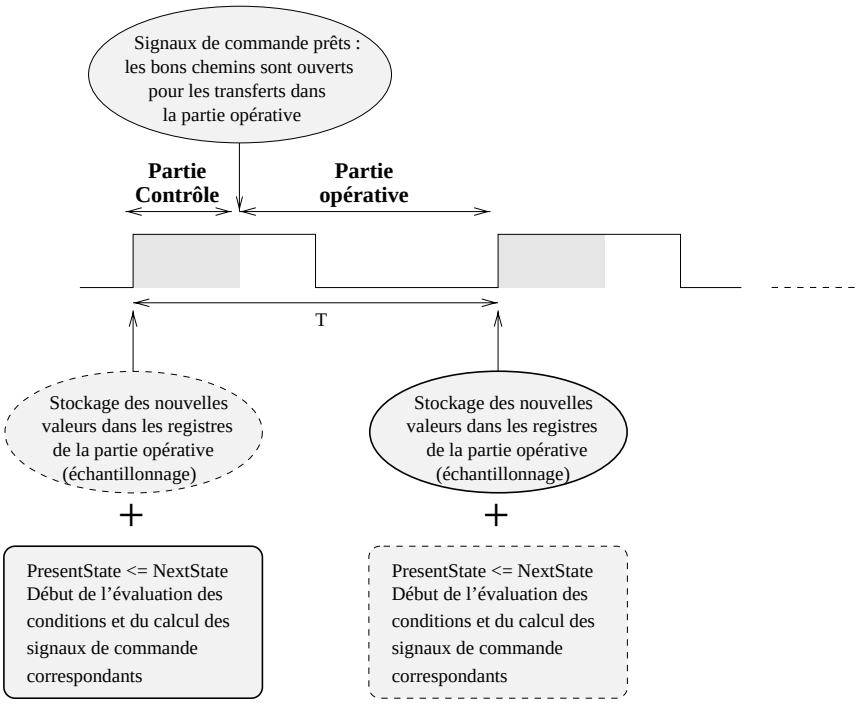


Figure E.6: Illustration du séquencement des tâches entre contrôleur et partie opérative

Annexe F

Code vhdl des circuits d'expérimentation de *SYNRJ*

F.1 gcd.vhd

```
------------------------------------------------------------------------
   --              Description comportementale de la fonction            --
   --              du plus grand commun diviseur, 8bits                  --
   --                                                                    --
   --              derniere modif :  25/11/96                            --
   --                                                                    --
   --              artist :  P. Guillaume                                --
   ------------------------------------------------------------------------


   library IEEE;

   use IEEE.std_logic_1164.all;

   use IEEE.std_logic_arith.all;

   use IEEE.std_logic_unsigned.all;

   use work.types.all;


   entity gcd is
     port (clk   :  in std_bit;
           reset :  in std_bit;
           start :  in std_bit;
           din   :  in std_bit;
           xi, yi:  in std_vector8;
           dout  :  out std_bit;
           ou    :  out std_vector8);
   end gcd;
```

```
architecture behavior of gcd is

begin

  algorithm:  process
    variable x,y:  std_vector8;
  begin

    wait until (start = '1' and rising_edge(clk));

    calculation: while (reset = '1') loop

      wait until (din = '1' and rising_edge(clk));

      dout <= '0';
      x: = xi;
      y: = yi;

      while (x /= y) loop
        if (x < y) then
          y: = y - x;
        else
          x: = x - y;
        end if;
      end loop;

      wait until (din = '0' and rising_edge(clk));

      dout <= '1';
      ou <= x;


    end loop calculation;

  end process algorithm;
end behavior;

configuration cfg_gcd of gcd is
        for behavior
        end for;
end cfg_gcd;
```

F.2 abmodn.vhd

```
------------------------------------------------------------------------
   --              Description comportementale de la fonction             --
   --              a.b modulo n avec 0 <= a,b <= n et log(n) <= 15        --
   --                                                                     --
   --              derniere modif :  06/12/96                             --
   --                                                                     --
   --              artist :  P. Guillaume                                 --
   -------------------------------------------------------------------------

   library IEEE;

   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_unsigned.all;
   use work.types.all;

   entity abmodn is
     port (clk   :  in std_bit;
           reset :  in std_bit;
           start :  in std_bit;
           din   :  in std_bit;
           ai    :  in std_vector8;
           bi,ni :  in std_vector8;
           si    :  out std_vector8;
           dout  :  out std_bit);
   end abmodn;

   architecture behavior of abmodn is

   begin

     algorithm :  process

       variable a, b, n, s :  std_vector10;
       variable i :  std_vector5;
       variable parity :  std_bit;
       constant loop_constant :  integer : = 15;

       function odd (b :  in std_vector10) return std_bit is
       begin
           return b(0);  -- retourne '1' si b impair
       end odd;

       function inc (x :  in std_vector5) return std_vector5 is
       begin
         return x + "00001";
       end inc;

       function mult2 (x :  in std_vector10) return std_vector10 is
```

```
  begin
    return conv_std_logic_vector(conv_integer(x) * 2, 10);
  end mult2;

  function div2 (x :  in std_vector10) return std_vector10 is
  begin
    return conv_std_logic_vector(conv_integer(x)/2, 10);
  end div2;

  function Convert8_2_10 (signal s :  in std_vector8) return std_vector10 is
    variable x :  std_vector10;
  begin
      x (7 downto 0) : = s;
      x (8) : = '0';
      x (9) : = '0';
      return x;
  end Convert8_2_10;

  function Convert10_2_8 (x :  in std_vector10) return std_vector8 is
  begin
      return x (7 downto 0);
  end Convert10_2_8;

begin

  wait until (start = '1' and rising_edge(clk));

  calculation :  while (reset = '1') loop

    wait until (din = '1' and rising_edge(clk));

    dout <= '0';
    a : = Convert8_2_10(ai);
    b : = Convert8_2_10(bi);
    n : = Convert8_2_10(ni);
    s : = "0000000000";
    i : = "00000";
    parity : = '0';  -- pair

    while (i <= loop_constant) loop

      parity : = odd(b);  -- test de la parite de b
      if (parity = '1') then
        s : = s + a;
        if  (s >= n) then
          s : = s - n;
        end if;
      end if;

      i : = inc(i);  -- i : = i + 1;
      b : = div2(b);  -- b : = b/2;
```

```
          a : = mult2(a);  -- a : = a*2;

          if (a >= n) then
            a : = a - n;
          end if;

        end loop;
        wait until (din = '0' and rising_edge(clk));
        dout <= '1';
        si <= Convert10_2_8(s);
      end loop calculation;
    end process algorithm;
end behavior;

configuration cfg_abmodn of abmodn is
  for behavior
  end for;
end cfg_abmodn;
```

F.3 bubble.vhd

```
------------------------------------------------------------------------
   --              Description comportementale de la fonction            --
   --              de tri "bubble sort" a base d'une memoire             --
   --              de 16 octets.                                         --
   --                                                                    --
   --              derniere modif :  03/12/96                             --
   --                                                                    --
   --              artist :  P. Guillaume                                 --
   ------------------------------------------------------------------------

   library IEEE;

   use IEEE.std_logic_1164.all;

   use IEEE.std_logic_arith.all;

   use IEEE.std_logic_unsigned.all;

   use work.types.all;

   entity bubble is
     port(reset :  IN std_bit;
          clk :  IN std_bit;
          start :  IN std_bit;
          validin :  IN std_bit;  -- donnée valide en entrée (extérieur)
          ackout :  IN std_bit;  -- donnée recue
          fill  :  IN std_bit;  -- remplissage de la memoire
          ackin :  OUT std_bit;  -- donnée recue
          validout :  OUT std_bit;  -- donnee valide en sortie (flag)
          done :  OUT std_bit;  -- fin
          datain :  IN std_vector8;
          dataout :  OUT std_vector8);
   end bubble;

   architecture behavior of bubble is

     type memory is array (0 to 15) of std_vector8;

     function inc (x :  in int5) return int5 is
     begin
       return x + 1;
     end inc;

     function decr1 (x :  in int5) return int5 is
     begin
       return x - 1;
     end decr1;

     function decr2 (x :  in int5) return int5 is
```

```
    begin
      return x - 2;
    end decr2;

  begin

    Bubblesort :  process

      variable i, nb_vect, t1, t2, iter :  int5;  -- 5 suffit comme bitwidth
      variable t3, t4 :  std_vector8;
      variable MEM :  memory;

      procedure fill_mem is
      begin

        i : = 0;
        nb_vect : = 0;

        fill_m :  while ((i <= 15) and (fill /= '0')) loop
          wait until (validin = '1') and rising_edge(clk);
          t3 : = datain;
          ackin <= '1';
          wait until (validin = '0') and rising_edge(clk);
          MEM(i) : = t3;
          ackin <= '0';
          i : = inc(i);
          nb_vect : = inc (nb_vect);

       end loop fill_m;

      end fill_mem;

      procedure empty_mem is

      begin
        i : = 0;
        while (i < nb_vect) loop
          wait until (ackout = '0') and rising_edge(clk);
          t3 : = MEM(i);
          validout <= '1';
          dataout <= t3;
          wait until (ackout = '1') and rising_edge(clk);
          validout <= '0';
          i : = inc(i);
        end loop;

      end empty_mem;


    begin       -- corps principal du process
```

```
    ackin <= '0';
    validout <= '0';
    wait until (start ='1' and rising_edge(clk));

    calculation:  while (reset = '1') loop

      wait until fill = '1' and rising_edge(clk);

      done <= '0';
      fill_mem;

      wait until fill = '0' and rising_edge(clk);

                       -- debut du tri

      i : = 1;
      while (i < nb_vect) loop
        iter : = nb_vect;
        while (iter > i) loop
          t1 : = decr1(iter);
          t2 : = decr2(iter);
          t3 : = MEM(t1);
          t4 : = MEM(t2);
          iter : = t1;
          if (t3 < t4) then
            MEM(t1) : = t4;
            MEM(t2) : = t3;
          end if;
        end loop;

        i : = inc(i);
      end loop;

      empty_mem;

      done <= '1';

    end loop calculation;


  end process Bubblesort;   -- fin du process

end behavior;    -- fin de l'architecture

configuration cfg_bubble of bubble is
  for behavior
  end for;
end cfg_bubble;
```

Sigles, acronymes et autres abbréviations

ACU	Address Calculation Unit - Unité de calcul d'adresses (synonyme de AGU)
AGU	Address Generation Unit - Unité de génération d'adresses.
ALU	Arithmetic and Logic Unit - Unité Arithmétique et Logique
ASIC	Application-Specific Integrated Circuit - Circuit intégré dédié
ASIP	Application-Specific Instruction-set Processor - Processeur dont le jeu d'instructions est spécifique à une application
BIV	Basic Induction Variable - Variable d'induction de base
CFG	Control Flow Graph - Graphe de flot de contrôle: Représentation traditionnelle des chemins d'exécution d'un programme.
DSP	Digital Signal processing - Traitement numérique du signal. Cet acronyme est employé pour désigner à la fois la science du traitement du signal, et les processeurs spécialisés dans le traitement des applications de traitement du signal.
DCU	Data Calculation Unit - Unité de calcul de données
EPT	Embedded Processors Team - Groupe spécialisé dans les processeurs dédiés, STMicroelectronics
EST	Embedded Systems Technology - groupe de recherche et développement - Central R&D - STMicroelectronics
GNU	Gnu is Not Unix
GCC	GNU C compiler - compilateur C de GNU
HDL	Hardware Description Language - Langage de description matérielle
IE	Induction Expression - Expression d'induction: expression qui est une fonction affine des IVs d'une boucle.
IIE	Indexed Induction Expression - Expression d'induction indexée c'est-à-dire IE indice d'un tableau.
ILP	Instruction Level Parallelism - Parallélisme au niveau instruction dit aussi parallélisme à grain fin
INPG	Institut National Polytechnique de Grenoble
IP	Intellectual Property - Propriété intellectuelle. S'emploie dans le contexte de la réutilisation et de l'échange de portefeuilles de produits pré-conçus et réutilisés dansles systèmes plus vastes. L'*IP reuse* ou réutilisation de propriétés intellectuelles est une des stratégies adoptées pour répondre à l'accélération des besoins du marché.

ISO	International Standards Organization - Organisation de standardisation internationale
IV	Induction variable - Variable d'induction: variable de boucle dont l'évolution inter-itération correspond à l'ajout ou au retrait d'une constante.
MAC	Multiply Accumulator - Accumulateur Multiplieur. Unité commune à tout DSP digne de ce nom.
MCU	Microcontroller Unit - Unité de micro-contrôle
MIPS	Million Instructions Per Second - Million d'instructions par seconde
MMDSP	MultiMedia Digital processing Processor - Processeur dédié de l'équipe EPT de ST
MMIO	Memory-Mapped Input/Output - Entrée/Sortie adressée par la mémoire
MPEG	Motion Picture Experts Group - Standard européen de codage de vidéo numérique
RAM	Random Access Memory - Mémoire vive
RISC	Reduced Instruction-Set Computer - Ordinateur à jeu d'instructions réduit
ROM	Read Only Memory - Mémoire morte
RTL	Register Transfer Level - Description matérielle au niveau transfert de registres
SLS	System Level Synthesis - groupe de recherche - laboratoire TIMA
ST	STMicroelectronics
SUIF	Stanford University Intermediate Format
TIMA	Techniques de l'Informatique et de la Microélectronique pour l'Architecture d'ordinateurs
UAL	Unité Arithmétique et Logique
VHDL	VHSIC Hardware Description Language - Langage de description de circuits intégrés
VHSIC	Very High Speed Integrated Circuit - circuit intégré à très haute vitesse
VLIW	Very Large Instruction Word - Mot-instruction très large
VLSI	Very Large Scale Integration - Intégration à très grande échelle
VSS	VHDL Synopsys Simulator - Simulateur VHDL de Synopsys