UNIVERSITÉ DE MONTRÉAL

Université de Montréal

DÉVELOPPEMENT D'ESTIMATEURS DE PERFORMANCE POUR DES APLICATIONS DE CO-DESIGN MATÉRIEL/LOGICIEL

LÉVIS THÉRIAULT
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES (M.Sc.A.)
(GÉNIE ÉLECTRIQUE)

AOÛT 2000

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

DÉVELOPPEMENT D'ESTIMATEURS DE PERFORMANCE POUR DES APPLICATIONS DE CO-DESIGN MATÉRIEL/LOGICIEL

présenté par: THÉRIAULT Lévis

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. BOIS Guy, Ph.D., président

M. SAVARIA Yvon, Ph.D., membre et directeur de recherche

M. AUDET Daniel, Ph.D., membre et codirecteur de recherche

M. ABOULHAMID El Mostapha, Ph.D., membre

À mes parents qui m'ont toujours soutenu

À mon épouse pour son soutien et sa patience

À mes proches

Remerciements

Avant de vous proposer le résultat de ma recherche personnelle, qu'il me soit permis ici d'adresser mes remerciements les plus profonds et les plus sincères à quelques personnes qui ont été associées plus étroitement à moi tout au long de ce parcours inoubliable dans ma formation universitaire.

En premier lieu et de façon très particulière, je tiens à remercier Monsieur Yvon Savaria, professeur à l'École Polytechnique de Montréal, qui a bien voulu me servir de premier directeur de recherche, qui m'a offert un indispensable support non seulement scientifique mais également financier et qui en plus m'a constamment nourri de ses précieux conseils dans le cadre de fréquents dialogues fort enrichissants sur le sujet, ce qui a contribué pour une très large part à la réalisation de ce présent mémoire.

En deuxième lieu, je voudrais adresser un merci tout spécial à Monsieur Daniel Audet, professeur à l'Université du Québec à Chicoutimi et co-directeur de mon présent mémoire, qui a bien voulu assurer presque quotidiennement la supervision de cette recherche et qui en outre n'a jamais refusé de répondre à mes nombreuses et incalculables interrogations.

Je m'en voudrais ici de ne pas ajouter un remerciement explicite à plusieurs autres personnes qui, en raison de leur situation particulière, ont joué un rôle significatif dans la réalisation de ce mémoire de maîtrise.

Aux membres du jury de ce présent mémoire, je souhaite exprimer toute ma reconnaissance et ma gratitude.

Au Conseil de Recherches en Sciences Naturelles et en Génie du Canada (CRSNG), je dis un gros merci pour la qualité de son soutien financier.

À Madame Ghyslaine Éthier Carrier, aux étudiants et aux membres du Groupe de Recherche en Microélectronique (GRM) de l'École Polytechnique de Montréal, que j'ai eu la chance de rencontrer et de joindre grâce à mon directeur ainsi qu'à tous les membres de l'Équipe de Recherche en Microélectronique et Traitement Informatique des Signaux (ERMETIS) de l'Université du Québec à Chicoutimi, je voudrais également adresser mes sincères remerciements pour leur soutien technique. Sur ce plan, je ne peux passer sous silence la grande collaboration, l'inconditionnelle disponibilité et l'indispensable aide que m'a offert en tout temps Monsieur André Falgueyret.

Au soutien scientifique, financier et technique vient s'ajouter parfois le support moral, ce dont j'ai eu quelques fois besoin au cours de cet itinéraire, aux courbes souvent imprévues. Pour cette raison, je me dois d'adresser de chaleureux remerciements à Monsieur Jean-Guy Girard et à quelques autres professeurs de l'Université du Québec à Chicoutimi, qui, entre autres, m'ont permis de prendre du recul et de ne jamais baisser les bras. Sous cet angle, je désire également remercier tous ceux et celles qui ont contribué à l'élaboration de ce travail, dont principalement mon épouse bien-aimée.

Ce n'est pas tout de développer son intelligence et de développer son savoir-faire. Il faut également construire constamment son savoir-être. Aussi, au terme de cette recherche, je dois reconnaître devant tous que la réalisation de ce mémoire n'a pas été une tâche facile et qu'il m'a fallu me dépasser moi-même presque à chaque jour. Outre les exigences intellectuelles, qui ne sont pas à négliger, j'avoue bien humblement que j'ai dû faire preuve d'autonomie et de débrouillardise, deux qualités qui sont essentielles pour réussir à mener à terme une recherche de ce type.

À chacun de vous et à tous, je dis merci, un gros merci et mille fois merci.

Résumé

Le partitionnement matériel/logiciel constitue une tâche fondamentale en conception de systèmes basés sur une méthodologie de co-design matériel/logiciel. La pertinence de ces éventuels partitionnements va non seulement dépendre de l'architecture matérielle ciblée, mais aussi des estimateurs de performance considérés. Ces derniers affectent directement l'accélération des calculs selon la qualité du partitionnement matériel/logiciel. Toutefois, peu d'outils de co-design matériel/logiciel sont disponibles et sont souvent trop spécifiques à un environnement donné, ce qui limite leur utilisation et leur performance.

Nous proposons dans ce mémoire un nouvel environnement intégré de co-design matériel/logiciel, conçu essentiellement d'estimateurs ou métriques de performance permettant d'évaluer la structure logicielle d'applications écrites en langage C ANSI. Les métriques permettent de caractériser, en fonction de l'architecture cible, chacun des blocs de base (boucles ou nids de boucles) d'une application en vue de prédire la pertinence de ses éventuels partitionnements matériel/logiciel. Le partitionnement s'effectuera sur une architecture matérielle reconfigurable de type FPGA, afin d'accentuer la vitesse d'exécution des blocs les plus prometteurs et ainsi d'accélérer l'exécution de toute l'application. Ces métriques sont le temps d'exécution requis, l'espace mémoire requis, la bande passante requise et finalement, la surface matérielle utilisée.

Les temps d'exécution considérés incluent le temps requis sur un Pentium pour exécuter chacun des blocs en logiciel et, le temps requis sur un FPGA pour exécuter ces mêmes blocs en matériel. Ce dernier tient compte à la fois du temps de communication, ainsi que du temps de latence s'y rattachant. À ce stade, l'architecture offrant une efficacité d'exécution optimale pour chacun des blocs est connue. Si le temps requis sur un Pentium est inférieur à celui requis sur un FPGA, pour un même bloc, alors le processus d'analyse est terminé et ledit bloc sera éventuellement exécuté en logiciel. Dans le cas contraire, une deuxième phase d'analyse est enclenchée consistant à vérifier si la quantité de mémoire, la bande passante et la surface matérielle requises pour exécuter le bloc en question, sont suffisantes.

Pour ce faire, les séquences d'accès à chacun des tableaux sont enregistrées, dans le but de mesurer la durée de vie de chacune des variables et de déterminer l'espace mémoire minimal requis pour réaliser tous les passages dans la boucle. Par la suite, la bande passante est estimée en fonction du taux de transfert des données fournies à l'unité de calcul (FPGA) pour effectuer une tâche spécifique. Finalement, la surface matérielle du FPGA utilisée est estimée selon le nombre de CLBs requis. Cette métrique permet, d'une part, d'analyser la structure logique qui doit être disponible et, d'autre part, de déterminer si les blocs analysés peuvent être transposés en matériel.

Une fois les valeurs de métriques trouvées, une fonction paramétrique est utilisée pour caractériser chacun des blocs de base. La mise en œuvre de cette fonction repose sur l'algorithme de partitionnement matériel/logiciel développé qui permet de déterminer s'il est possible et profitable de transposer et d'exécuter en matériel chacun des blocs de base analysés. Pour qu'un partitionnement soit possible et profitable, il faut que toutes les conditions soient gagnantes, c'est-à-dire qu'elles respectent les contraintes de temps, ainsi que celles imposées par l'architecture matérielle.

Les métriques de performance que nous avons proposées permettent de caractériser l'exécution de chacun des blocs de base et de détecter les blocs les plus prometteurs ("boucles chaudes"). Ces blocs seront éventuellement transposés et exécutés sur une

architecture matérielle reconfigurable afin d'accentuer la vitesse d'exécution de ces blocs et ainsi d'accélérer l'exécution de toute l'application.

Les résultats obtenus montrent la pertinence de notre travail et constituent une base solide pour le développement futur de nouveaux outils d'analyse plus élaborés. Ces outils nous permettront de prédire la pertinence d'éventuels partitionnements matériel/logiciel sur des architectures matérielles reconfigurables.

Abstract

Hardware/software partitioning constitutes a fundamental task in the design of systems based on a hardware/software co-design methodology. The performance of these possible partitioning will not only depend on targeted architecture, but also on the performance of considered estimators. The latter ones directly speed up the computation according to the hardware/software partitioning quality. However, few hardware/software co-design tools are available and they are often too specific to a given environment, which limits their use and their performance.

This thesis proposes a new integrated co-design environment, which includes hardware/software performance estimators or metrics allowing to evaluate the software structure of applications written in ANSI C language. The metrics allow us to characterize, according to the target architecture, which basic block (loops or nested loops) of an application is worth embedding in hardware. The process targets a FPGA based reconfigurable hardware architecture, in order to increase the execution speed of the most promising blocks and to accelerate the execution of the whole application. These metrics include the required execution time, the required memory capacity, the required bandwidth and finally, the complexity of the hardware partition.

The considered execution time takes into account the time required by a Pentium processor to execute each block in software and, the time required by a suitably

configured FPGA chip to execute these same blocks in hardware. The latter takes into account both the communication time and the corresponding latency time. At this stage, the architecture that offers optimal execution speed for each block is known. If the required time for a given block on a Pentium is lower than that on a FPGA, then the analysis is stopped and this block is left for software implementation. Otherwise, a second phase of analysis is engaged. It checks the availability of the required memory, bandwidth and hardware resources to implement this block.

To realize this, the access sequences to each table are recorded, in order to measure the lifespan of each variable and to determine the minimal memory capacity required to execute all the steps of the loops. Next, the bandwidth is estimated according to the data transfer rate to the processing unit (FPGA) required to carry out a specific task. Finally, the tool estimates the required FPGA complexity measured by the number of configurable logic blocks (CLBs). These metrics allow to determine if the analyzed blocks can be implemented in hardware.

Once the metrics are known, a parametric function is used to characterize each basic block. A block is retained for hardware implementation if its mapping to hardware is advantageous and feasible.

Our proposed metrics allow characterizing the execution of each basic block and to detect the most promising ones ("hot loops"). Mapping these blocks to hardware accelerates the execution of the overall application.

The results obtained show the relevance of the proposed method and constitutes a solid base for future developments of more elaborate tools. Such tools would allow predicting the performance of possible hardware/software partitions on reconfigurable architectures.

Table des matières

Liste des tableaux

Liste des figures

Liste des annexes

Liste des sigles et abréviations

ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
AST	Abstract Syntactic Tree
CD	Control Dependency
CFA	Control Flow Analysis
CFG	Control Flow Graph
CFSM	Codesign Finite State Machine
CLB	Configurable Logic Block
COREGEN	CORE GENerator
CPU	Central Processing Unit
DD	Data Dependency
DFA	Data Flow Analysis

DFG	Data Flow Graph
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
IR	Intermediate Representation
MDFG	Multiple Data Flow Graph
MIPS	Million d'Instructions Par Seconde
PDG	Program Dependency Graph
PDT	Post Dominator Tree
PRAM	Parallele Random Access Machines
RTL	Register Transfer Level
SUIF	Stanford University Intermediate Format
VHSIC	Very High-Speed Integrated Circuits
VHDL	VHSIC Hardware Description Language
VLIW	Very Large Instruction Word

Introduction

Depuis quelques années, les ordinateurs à architectures reconfigurables ont fait leur apparition. Ces machines permettent une répartition de l'exécution du programme sur des plates-formes étroitement couplées. Ces systèmes offrent des possibilités substantielles d'accélération [EDW94, HAR96].

Ces systèmes offrent un grand potentiel, car il est bien connu qu'une grande partie du temps d'exécution des programmes scientifiques et de traitement des signaux est passée dans les nids de boucles [GEN96]. L'accélération globale d'un programme passe nécessairement par l'optimisation de ces portions de code. L'exploitation du parallélisme inhérent aux nids de boucles et la réutilisation des données sont des moyens de réduire les temps d'exécution. La détection des nids de boucles offrant le meilleur potentiel est donc essentielle pour qui veut accélérer l'exécution d'un traitement.

L'approche générale que nous proposons consiste à transporter l'exécution des nids de boucles sur une structure à base de logique reconfigurable. En d'autres termes, il s'agit, pour un traitement donné, de synthétiser l'application décrite par les nids de boucles, puis de l'implanter sur un support reconfigurable.

Pour ce faire, il faut auparavant prédire la pertinence de déporter des segments[1] de code en matériel afin d'accentuer la vitesse d'exécution de certains segments d'une tâche. Des métriques permettant de caractériser chacun des segments analysés doivent donc être définies. Notre objectif consiste à intégrer un certain nombre de métriques de performance à l'intérieur d'un environnement de co-design matériel/logiciel et de voir les différentes adaptations requises en fonction des autres environnements disponibles sur le marché tels que Cosyma, SpecSyn, etc.

L'originalité du travail repose sur le développement de métriques permettant la détection et la caractérisation de "boucles chaudes" à l'intérieur d'une application donnée, écrite en langage C ANSI. Une fois les valeurs des métriques trouvées, suite à une analyse statistique de chacun des blocs de base [BAL92], elles sont intégrées dans une fonction paramétrique. La mise en œuvre des blocs repose finalement sur l'algorithme de partitionnement choisi. La combinaison des métriques et d'un algorithme de partitionnement permet d'établir la pertinence d'un éventuel partitionnement matériel/logiciel dudit bloc de base, accentuant la vitesse d'exécution de l'application.

L'emploi de FPGA (Field Programmable Gate Arrays) pour mettre en œuvre des co-processeurs programmables n'est pas nouvelle. Certains des travaux les plus pertinents sont présentés dans le chapitre 1. Le chapitre suivant (chap. 2) contiendra les différentes étapes de la méthodologie de co-design matériel/logiciel que nous avons élaborée. L'approche suivie et les métriques associés à notre outil sont décrites dans le chapitre 3. La validité de notre approche est démontrée à l'aide d'exemples concrets présentés au chapitre 4.

---

[1] Dans le présent ouvrage, *segment*, *région*, *partie* et *bloc de base* sont synonymes et représentent tous une boucle ou nid de boucles dans une application.

Chapitre 1

Survol des différentes stratégies d'optimisation

Dans le présent chapitre, nous faisons un tour d'horizon des différents ouvrages traitant, d'une part, des méthodes d'évaluation de performance logicielle et matérielle, et, d'autre part, des techniques de partitionnement matériel/logiciel utilisant des co-processeurs reconfigurables et dédiés. Enfin, notre positionnement par rapport à l'ensemble de ces ouvrages sera exposé à la fin de ce chapitre.

1.1 Les méthodes d'évaluation de performance

Cette section fait état des différents ouvrages les plus pertinents sur les techniques utilisées pour mesurer les performances des systèmes. La plupart du temps, les applications scientifiques, exécutées par ces systèmes, ont été mises en œuvre en utilisant une philosophie de conception séquentielle. D'où l'intérêt de mesurer les performances de ce genre d'applications, afin d'exploiter le parallélisme en respectant les dépendances des données, en vue d'un éventuel ordonnancement ou partitionnement du code.

1.1.1 Méthodes logicielles

Plusieurs chercheurs ont proposé un certain nombre d'approches pour caractériser l'exécution des logiciels [GON94]. Une méthode simple d'estimation du temps d'exécution est présentée dans [DAM94]. Dans cette étude, les temps d'exécution sont calculés en effectuant le produit du nombre d'instructions exécutées par le temps moyen d'exécution. Le traitement est effectué sur un processeur MIPS. L'étude se limite à caractériser le temps d'exécution sur ce processeur et n'évalue en aucun temps la pertinence de transposer des segments de code sur un co-processeur de type FPGA.

Dans [HAR93] et [WOL93], on propose des méthodes statistiques pour modéliser l'exécution sur l'unité centrale de traitement de sorte que plusieurs types de CPU peuvent être évalués en regard du programme devant être exécuté. Parallèlement, [SMI91] propose un système de synthèse logiciel, où toutes les primitives pour construire un programme sont définies dans une séquence d'instructions fixe. Le temps d'exécution requis pour effectuer un bloc d'instructions donné est pré-calculé. Par conséquent, ces primitives peuvent être employées pour effectuer des prévisions précises d'exécution.

Un modèle proposé dans [GUP94] estime le temps d'exécution d'un programme par le nombre de cycles d'exécution nécessaires pour chaque instruction dans le programme, la quantité de mémoire en lecture/écriture, et le nombre de cycles pour chaque accès mémoire. Cependant, ce modèle ne tient pas compte des caractéristiques intrinsèques de l'architecture matérielle cible, ce qui est essentiel pour optimiser l'exécution via un partitionnement efficace.

Dans [YE93], la méthode d'évaluation employée dans le système COSYMA est présentée. Dans cette méthode, le système cible exécutant le programme est également synthétisé. L'approche consiste à "exécuter" le code sur un modèle du système cible au niveau RT (Register-Transfer) pour en extraire des caractéristiques de synchronisation à partir des résultats de simulation. Cette méthode d'évaluation considère le code source dans son ensemble et non pas bloc de base par bloc de base.

Une autre variante intéressante pour calculer le temps d'exécution du logiciel est d'utiliser un modèle d'estimation. Il peut être spécifique à un processeur ou générique, de façon à pouvoir être indépendant du processeur choisi [GAJ94c, KNI96]. Un profilage [CON96] peut être fait sur les différents blocs du logiciel, de façon à déterminer le nombre de fois où chacun des blocs sera exécuté, par exemple. Ce profilage peut se faire de façon dynamique en faisant exécuter plusieurs fois les blocs sur le processeur cible ou en utilisant un outil tel que QPT2 [BAL92]. Le profilage statique n'utilise pas les données pour déterminer le nombre de fois où chacun des blocs sera exécuté. Il doit donc être capable de déterminer les bornes supérieures et inférieures. À cet effet, l'outil Cinderella [CIN00] peut être utilisé pour ce type de profilage.

Une étude dans [BAL96] compare PP à QPT2. PP est un outil de profilage qui emploie la bibliothèque EEL [LAR95] pour insérer une instrumentation dans les programmes exécutables. QPT2 est un autre outil de profilage construit avec EEL, qui utilise un algorithme [BAL94] profilant les effets de bord (les débuts et les fins de boucles). QPT2 requiert habituellement moins de temps système, mais les deux systèmes donnent des résultats similaires. Encore une fois, ces outils de profilage ne considèrent pas les caractéristiques de l'architecture matérielle cible, ce qui est un handicap important lorsque l'on désire effectuer un partitionnement matériel/logiciel pertinent et efficace.

Young et Smith ont employé une forme limitée du traçage de programme afin d'enregistrer les voies d'accès permettant ainsi d'étudier la corrélation des branchements [YOU94]. Dans une mémoire tampon de type FIFO, les N derniers branchements sont enregistrés, où chacun d'eux se compose d'un nombre de blocs de base et les résultats de branchements s'y rattachant. Notons que cette technique coûte plus cher que celle du profilage par voie d'accès et exige également un niveau supplémentaire d'adressage indirect, associé au compteur des voies d'accès, composé d'une séquence de nombre de blocs. À la différence du profilage par voie d'accès, cette technique n'a pas besoin de distinguer la répétition des voies d'accès acycliques, puisque cette répétition se tronque de part et d'autre à la borne du FIFO.

Martin Edwards [EDW93] décrit un système en C qui rassemble l'information de profilage par la compilation et l'exécution du programme. Puisqu'il ne considère que les fonctions ou procédures (dans les programmes sources) comme étant des régions à accélérer, il ne peut considérer des plus petites régions plus prometteuses comme les boucles. Comme dans le travail d'Athanas [ATH91], les régions sont choisies par un utilisateur humain et aucune analyse automatique n'est effectuée.

1.1.2 Méthodes matérielles

Il existe principalement deux méthodes permettant d'évaluer les performances d'une application lorsque celle-ci est exécutée sur une plate-forme matérielle. La première utilise une technique de synthèse. Cette technique réalise essentiellement une synthèse rapide sans aucune optimisation. Le bloc matériel est donc transformé depuis sa description comportementale à partir d'un graphe de flots de données. Une allocation d'opérateurs est effectuée pour chacune des opérations. Ces opérateurs sont ensuite ordonnancés. Par la suite, une estimation du nombre d'étapes de contrôle est faite, suite à l'exécution dudit bloc matériel afin d'obtenir un temps d'exécution approximatif. Pour ce faire, un outil de synthèse comme Monet de Mentor Graphics peut être utilisé.

La seconde, consiste à réaliser une traduction du langage C vers un langage de description matérielle, tels que le VHDL ou le Verilog. Différents outils sont disponibles à cet effet [COM90, RUN90], cependant ceux-ci ne supportent qu'un sous-ensemble limité du langage et ils sont souvent orientés vers des architectures de type VLIW, par exemple. De nos jours, on retrouve des outils commerciaux de conception de systèmes sur puce (System-on-Chip, SoC), tel que SystemC [ARN00, SYS00]. L'apparition et la grande popularité de ce type de méthodologie de conception de système sur puce a apporté avec lui une variété de suggestions pour un langage simple qui peut décrire toutes les conditions fonctionnelles pour ces conceptions fortement complexes. Ces systèmes favorisent la conception de blocs matériel qui peuvent être réutilisés éventuellement. Par la suite, différentes techniques peuvent être utilisées pour estimer le temps d'exécution du

bloc matériel. Par exemple, le même programme peut être exécuté plusieurs fois et le temps maximal d'exécution pourra être choisi comme étant le temps d'exécution du bloc matériel. De cette façon, on obtient une quantité considérable de statistiques permettant ainsi d'évaluer les performances de chacun des blocs matériels analysés.

1.2 Les architectures matérielles

Cette section présente une synthèse des différentes plates-formes les plus utilisées lors de la conception de systèmes matériel/logiciel. Principalement, on y retrouvera les circuits reprogrammables ainsi que quelques systèmes embarqués.

1.2.1 Les FPGAs

Bertin, Roncin et Vuillemin [BER92] enregistrent des vitesses impressionnantes pour dix algorithmes réalisés sur des co-processeurs FPGAs. Les algorithmes utilisés ont été soigneusement conçus pour convenir, de façon optimale, à l'architecture matérielle. Certains de leurs exemples présentant des accélérations du traitement considérables sont à l'origine d'une reformulation des algorithmes, afin d'orienter leurs mises en œuvre de façon très étroite avec les caractéristiques intrinsèques de l'architecture ciblée. Leurs exemples sont simples et ils décrivent des applications spécifiques (non génériques), par opposition aux programmes classiques qui seront étudiés dans ce mémoire.

Luk, Lok et Page [LUK93] ont effectué une étude qui montre comment une application logicielle peut être accélérée, en utilisant des FPGAs qui lui sont disponibles, lors d'un partitionnement matériel/logiciel soigneux qui adapte la partition matérielle en fonction de l'architecture cible. Cependant, l'avantage principal repose essentiellement sur un choix judicieux ou une conception soigneuse de l'algorithme en fonction de l'architecture logicielle et matérielle disponible. En d'autres termes, tous ces facteurs doivent être connus de la part de l'utilisateur. Malheureusement, cette approche n'est pas faisable pour un partitionnement matériel/logiciel automatique d'une application donnée

quelconque. Notre but est d'automatiser ce processus, en utilisant des estimateurs de performance afin de prédire la pertinence d'un éventuel partitionnement matériel/logiciel.

Koch et Golze décrivent un système embarqué de co-processeurs [KOC93], lequel est utilisé dans la conception, et le but de son utilisation est semblable à [JAN94a]. Cependant, ils se concentrent d'avantage sur la conception de systèmes embarqués et moins sur des méthodologies de partitionnement. De plus, ils ne mettent en application aucun algorithme de partitionnement.

Le travail sur l'architecture Mark-I par Lewis et al. [LEW93] a pour but l'accélération d'une classe de programmes définissant une application spécifique représentée par un ensemble d'instructions qui sont exécutées sur un support matériel constitué de 16 FPGA Xilinx. La disposition d'un tel positionnement d'instructions, pour un programme donné, est semblable à identifier des régions critiques pour une éventuelle implantation matérielle sur des FPGAs. Lewis et al. n'ont pas automatisé le partitionnement, mais se sont concentrés particulièrement sur le développement de l'architecture matérielle cible.

1.2.2 Les ASICs

Dans [CAR96], la méthodologie de co-design matériel/logiciel est vue comme une approche rentable et prometteuse pour mettre en œuvre des systèmes complexes. La rentabilité est dérivée d'un partitionnement approprié des tâches du système parmi les composantes matérielles (applications spécifiques ASICs) qui sont rapides mais chères et les composantes logicielles (exécutées dans un ou plusieurs dispositifs, habituellement des processeurs standards) qui sont plus lents mais peu coûteux. En outre, certains comportements sont plus efficaces suite à une mises en œuvre matérielles ou logicielles.

Dans la phase de spécification, le système est décrit en utilisant le langage formel LOTOS [BOL87, DEL95]. LOTOS a les caractéristiques requises pour définir les spécifications au niveau du système. Les buts et les contraintes, guidant le processus de

co-design, sont dans la description même du système. Des transformations formelles sont appliquées pour l'optimiser.

Après cette phase, la syntaxe et la sémantique sont analysées afin d'établir une forme intermédiaire avec laquelle le partitionnement matériel/logiciel sera effectué. L'architecture matériel/logiciel cible se compose d'un processeur exécutant le logiciel, d'un ASIC mettant en application le matériel et d'une mémoire partagée consultée par un bus commun. Des modules d'interfaces sont utilisés pour relier le processeur et l'ASIC au bus, tout en tenant compte des transmissions et des transferts de données entre elles. Cette architecture est paramétrable en ce qui concerne le nombre de processeurs et/ou d'ASICs.

1.3 Les stratégies de partitionnement

Cette section énumère les différents outils de co-design matériel/logiciel disponibles sur le marché. Elle présente aussi les avantages et les inconvénients relatifs à chacun d'eux. De plus, elle examine leurs limites de fonctionnement qui sont déterminées par les contextes d'utilisation.

1.3.1 Les outils existants

Plusieurs recherches essayent d'intégrer le matériel et le logiciel dans un même processus de conception : COSMOS de TIMA/INPG [ISM95], SpecSyn de Irvine [GAJ94a], CODES de Siemens [BUC93], Thomas de CMU [THO93], Gupta et De Micheli de Stanford [GUP93b], Ptomely [CHI93] et POLIS [SUZ96] de Berkeley.

COSMOS est un environnement de co-design matériel/logiciel basé sur le format intermédiaire SOLAR [ISM94] pour modéliser et synthétiser des systèmes hybrides. La description intermédiaire peut être produite par différents langages comme SDL, LOTOS, etc. Le partitionnement est effectué en utilisant un ensemble hiérarchique de processus de communication qui caractérise une fonction de coût et de performance. Ce

partitionnement est transformé en appliquant quatre opérations différentes, le résultat de celui-ci est ensuite utilisé lors de la synthèse. De ce partitionnement, une combinaison gagnante et qui satisfait les différentes contraintes de conception est choisie. SOLAR supporte différents niveaux comportementaux des spécifications du système. COSMOS inclut des outils de synthèse, utilisés pour synthétiser la transmission et le partitionnement du système. AMICAL est un outil comportemental de synthèse pour le matériel utilisé dans COSMOS. La version courante de COSMOS procède à partir du langage SDL et produit une architecture hétérogène comprenant des descriptions matérielles en langage VHDL et des descriptions logicielles en langage C.

SpecSyn est un cadre de conception de système qui aide le concepteur à obtenir des descriptions pouvant être synthétisées afin d'obtenir des modules pouvant être mis en œuvre sur une architecture matérielle. Les spécifications du système abstrait sont données par les langages SpecCharts ou VHDL. Les principales tâches de conception de SpecSyn sont: l'allocation des objets structuraux (c.-à-d. modules et bus), le partitionnement des spécifications du système (c.-à-d. que les objets fonctionnels sont alloués aux objets structuraux) et finalement, une liaison entre les objets structuraux et les bibliothèques de composants est effectuée.

CODES est un environnement de conception qui intègre un nouvel outil de modélisation avec plusieurs outils existants permettant de définir les spécifications du système pour la mise en œuvre du matériel et du logiciel. L'outil de modélisation est basé sur le modèle algébrique abstrait PRAM (Parallel Random Access Machines) et sur une extension des réseaux de Pétri. L'entrée pour cet outil peut être un Statemate ou une spécification SDL. Les environnements intégrés dans l'outil existant sont: les générateurs de C et de VHDL pour StateChart et SDL, l'outil de conception et de simulation Matrix, et l'outil SIDECON pour la configuration de base des cartes électroniques. Actuellement, CODES vise la conception de systèmes à base de processeurs.

L'approche de co-synthèse de Thomas consiste à indiquer au système un ensemble de tâches et d'assigner ces tâches aux processus effectuant la mise en œuvre sur le

matériel spécialisé ou sur le logiciel d'application fonctionnant sur un CPU commercial. Thomas propose de spécifier dans le langage de description matérielle Verilog, les processus qui seront mis en œuvre en matériel. Le matériel et le logiciel communiquent à l'aide des sockets BSD disponibles sur Unix et par un module Verilog correspondant à un bus d'interface abstraite. Une co-simulation du système est possible en utilisant un simulateur du langage Verilog.

Gupta et De Micheli [GUP92, GUP93a] proposent une approche orientée matériel qui utilise un sous-ensemble limité du langage C, appelé Hardware C, comme langage décrivant les spécifications du système. Ce dernier est le langage d'entrée du système Olympus. Leur système de conception permet d'entrer graduellement des fonctions, qui seront effectuées en matériel, dans le programme source, tout en considérant les contraintes de temps et de synchronisation. À partir des spécifications du système, on dérive le graphe de flot de données (DFG). Celui-ci est utilisé pour évaluer une fonction de coût de partitionnement matériel/logiciel. L'approche orientée matériel et l'utilisation de Hardware C limitent la complexité du système global.

L'environnement de conception Ptolemy permet le développement et la simulation des modules matériels et logiciels. L'environnement supporte, pour la partie matérielle, les architectures de type "behavioral" et "structural", décrivant le comportement de chaque entité et, pour la partie logicielle, un module de génération du code d'assemblage ciblé pour des microprocesseurs DSP. Un utilisateur de Ptolemy peut synthétiser le logiciel, modéliser le matériel et simuler des algorithmes donnés. Il indique à l'entrée de Ptolemy les structures de flots de données synchrones et les graphiques fonctionnels en utilisant les deux niveaux d'abstraction suivants : au niveau porte et au niveau comportemental. Il peut obtenir comme sortie un code assembleur ciblé pour un processeur DSP, pour la partie logicielle et, pour la partie matérielle, une description "Netlist" pouvant être utilisée pour la synthèse logique.

Dans le système POLIS, la partition, la vérification de la synchronisation et l'optimisation au niveau CFSM (Codesign Finite State Machine) et au niveau des graphes

sont guidés par une évaluation de la mise en œuvre finale. Cette évaluation est basée sur des modèles du système cible et de la structure du code.

Outre ces nombreux projets, d'autres chercheurs ont essayé aussi de concevoir des outils de partitionnement matériel/logiciel intégrant à la fois le logiciel et le matériel dans un même processus de conception. Les travaux les plus pertinents sur le sujet sont présentés ci-dessous, les auteurs ainsi que leurs méthodologies respectives y seront cités.

Barros da Silva [BAR93] présente un outil de partitionnement basé sur une description conforme à UNITY. Cette description peut être séquentielle ou parallèle. Un processus d'affectation permet de rassembler, en deux étapes, d'une part, les différentes mises en œuvre possibles et, d'autre part, les dépendances de données, les ressources partagées ainsi que les performances s'y rattachant. La gestion des affectations est ordonnée pour l'architecture ciblée. Ce processus est réitéré afin de satisfaire toutes les contraintes de conception.

O'Nils et al. [ONI95] présente un outil de partitionnement nommé AKKA. Cet outil est basé sur la même approche que dans [JAN94b], mais cependant, plusieurs améliorations ont été apportées. La première consiste à permettre à l'utilisateur, lors du partitionnement, d'imposer des contraintes en assignant des candidats potentiels au matériel ou au logiciel afin d'obtenir l'information résultante reliée à un éventuel profilage. Cette information est visible via une interface graphique. La seconde repose sur le fait que l'information relative au profilage peut être utilisée lors du transfert des données, ce qui réduit considérablement les goulots d'étranglement lors de la transmission.

L'ambition d'Athanas [ATH91] est le partitionnement automatique d'un programme, écrit en langage C, dans des parties logicielles et matérielles et implanter les parties matérielles sur des FPGAs. Par contre, son approche considère seulement les circuits combinatoires purs, sans utiliser des techniques de synthèse de haut niveau. Il ne permet

pas l'accès mémoire disponible sur les FPGAs et le processus de partitionnement est guidé lui aussi par un utilisateur humain.

Ernst et Henkel [ERN92, ERN98] utilisent une extension du langage C, comme langage de spécification, pour un contrôleur embarqué lors de la conception. Leur objectif est d'extraire des segments de code qui seront éventuellement implantés sur une architecture matérielle. Leur système de partitionnement matériel/logiciel est basé sur le recuit simulé et considère chaque segment de code logiciel comme étant des régions prometteuses. Cette approche est inefficace, puisque la plupart de ces segments de code ne sont pas des régions critiques. Les données de profilage sont recueillies par un simulateur qui est beaucoup plus lent que la compilation et l'exécution du programme source. Ceci est d'autant plus fastidieux lorsque l'on exécute des longs programmes.

Cette revue sur les outils de co-design matériel/logiciel est certes incomplète, puisque ce domaine évolue tellement rapidement, il est fort probable que déjà d'autres types d'outils sont présentement sur le marché et que d'autres ont complètement été laissés de côté. À ce titre, le lecteur peut consulter les ouvrage spécialisés cités aux références [EET00, GAJ00, SUM00]. Ces ouvrages actualisent et répertorient les principaux travaux dans ce domaine.

1.3.1.1 Les outils commerciaux

Il est important de noter que quelques outils de co-design matériel/logiciel ont vu le jours dernièrement et sont présentement disponibles. Parmi les principaux on note CoWare [BOL97, COW00] et Arexsys [ARE00]. On peut les catégorisés comme des outils de partitionnement semi-automatique permettant, entre autre, un raffinement intéressant au niveau des communications entre le logiciel et le matériel.

CoWare est un environnement de co-design matériel/logiciel permettant la mise en œuvre de systèmes hétérogènes basée sur des spécifications hétérogènes. Cet environnement permet à un concepteur de mettre en œuvre une application basée sur une conception de raffinement. Tout d'abord, le système est spécifié complètement en

langage C. Ensuite, selon le choix du partitionnement matériel/logiciel, le C fonctionnel correspondant à la partie matérielle est traduit manuellement en une description comportementale. Puis, CoWare génère automatiquement une description Verilog ou VHDL de niveau RTL équivalente, ainsi que l'interface de communication entre la partie logicielle et matérielle. Finalement, CoWare supporte également la co-simulation avec des simulateurs Verilog et VHDL.

Arexsys est une amorce de technologie sur le marché naissant de la conception assistée par ordinateur de système (System Design Automation, SDA). SDA entoure les outils et les méthodologies qui permettent à des concepteurs, selon des caractéristiques de systèmes électroniques, de mettre en œuvre des modules en logiciel et en matériel. Le processus commence par un cahier des charges formel écrit en langages standard d'industrie tels que C, SDL ou VHDL et fournit des descriptions matérielles au niveau transfert de registre (RTL) ciblées pour les langages VHDL et Verilog et des descriptions logicielles ciblées pour le langage C.

1.3.2 Les algorithmes de partitionnement

Une fois la modélisation du système complétée, un algorithme de partitionnement doit être utilisé afin d'examiner les différentes configurations possibles. Il existe des algorithmes précis comme le "branch-and-bound" [AXE97] qui examine l'ensemble des configurations possibles pour une même modélisation. Cependant, la plupart de ces procédures sont des heuristiques qui essaient, à partir d'une modélisation donnée, de faire des changements à une configuration de départ dans le but de l'améliorer. Les heuristiques les plus répandues sont la recherche taboue [HEN97], le recuit simulé [AXE97], la méthode vorace [GAJ94c] et les procédures génétiques [GAJ94c].

Peng's et Kuchcinski's [ELE94] proposent un partitionnement basé sur le recuit simulé et la représentation interne est basée sur les réseaux de Petri. Elle utilise les informations de profilage, qui sont obtenues par un simulateur, ainsi que l'information reliée à la connectivité statique entre les opérations effectuées.

Jantsch et al. [JAN94a] présente un algorithme de partitionnement basé sur un arrangement hiérarchique de présélection qui utilise des techniques de programmation dynamique. Cette approche permet: (a) une collecte efficace des données relatives au profilage possible grâce à l'utilisation des langages de haut niveau, tels que C et C++, comme langage de spécification; (b) une rapidité du partitionnement due à la présélection des candidats potentiels; (c) un niveau de complexité supérieur du partitionnement matériel permettant une émulation logique du système entier.

Le lecteur intéressé par les méthodes de partitionnement les plus répandues peut consulter l'ouvrage cité à la référence [EDW97].

1.4 Le choix de la méthodologie suivie

Dans cette dernière section, nous allons identifier les spécifications, les objectifs et les contraintes du projet. Nous allons y décrire le choix des fonctionnalités que nous avons implantées, afin de bien faire ressortir la méthodologie préconisée dans ce travail. Mais tout d'abord, présentons une vue d'ensemble du projet CODE.

1.4.1 Le projet CODE

Le projet CODE du Groupe de Recherche en Microélectronique de l'École Polytechnique de Montréal consiste à étudier la structure d'un compilateur C ciblé vers un ordinateur d'une architecture hybride reconfigurable. Ce projet propose de tenter de capter le flux de design de ce type d'application dans un compilateur d'un langage de haut niveau. Cet outil intégrera à la fois le flux de design logiciel et matériel dans un même processus de conception.

1.4.2 Les spécifications de l'outil

1.4.2.1 Les objectifs

Tout d'abord, les spécifications du cahier des charges initiales étaient de concevoir et de mettre en œuvre des estimateurs de performance. Ces estimateurs permettant de prédire la pertinence d'un éventuel partitionnement matériel/logiciel automatique, sur une architecture matérielle reconfigurable de type FPGA, afin d'accentuer la vitesse d'exécution d'une tâche donnée. Ceux-ci identifiant les "boucles chaudes" ou régions critiques présentes à l'intérieur de programmes, écrits en langage C ANSI. En d'autres termes, nous devions développer des outils permettant d'analyser la structure logicielle d'une application quelconque écrite en C et d'extraire les informations relatives aux caractéristiques de cette application lors de son exécution.

Initialement, cinq fonctionnalités de base ont été définies afin d'élaborer un premier embryon d'outil de mesure et d'analyse de performance. Ces fonctionnalités recherchées sont : un compilateur du langage C, un module permettant la génération de deux codes cibles, soit : le C et VHDL, une bibliothèque de modules générique afin de supporter plusieurs architectures matérielles, un module de profilage servant à quantifier le squelette d'exécution de l'application et, enfin, un module d'estimation de la performance de chacune des applications, mesurant ainsi la pertinence d'un éventuel partitionnement matériel/logiciel sur une architecture matérielle reconfigurable.

Cependant, un certain nombre de contraintes s'ajoutent à la complexité de l'outil. Citons les trois plus importantes contraintes qui imposent, en quelque sorte, la structure logicielle que devra prendre l'outil. Tout d'abord, le code cible (C et VHDL) doit pouvoir être synthétisé et compilé par des outils commerciaux de base. On ne veut pas réécrire un "front-end", donc celui-ci doit être récupéré du compilateur C utilisé. Enfin, l'outil doit supporter la simulation et le déverminage des codes cibles générés.

1.4.2.2 La méthodologie

Afin de bien présenter le portrait général de la méthodologie suivie et de la pertinence de celle-ci, on doit se rapporter, de façon très étroite, aux spécifications fonctionnelles du futur outil d'analyse.

Notons que, dans le cadre de cette étude, nous avons focalisé nos efforts essentiellement sur la conception et la mise en œuvre du module effectuant un profilage dynamique et du module permettant d'estimer la performance de chacune des applications analysées. Le module de profilage dynamique sert à quantifier le squelette d'exécution de l'application et le module d'estimation de performance mesure la pertinence d'un éventuel partitionnement matériel/logiciel sur une architecture matérielle reconfigurable.

Pour y parvenir, voici les grandes étapes de la méthode suivie lors de la conception. La première étape du processus est la revue de la littérature sur le sujet des processeurs et des circuits reconfigurables. À ce stade, la décision d'utiliser une architecture reconfigurable de type FPGA ou autre chose, découle en général des contraintes de performance et de coût. Le choix d'un compilateur, du domaine public, devrait normalement découler des spécifications fonctionnelles et des contraintes du système. Le développement d'une stratégie d'optimisation permet de localiser les "boucles chaudes" d'un programme et d'en évaluer les coûts d'exécution selon la quincaillerie disponible. Cela constitue les dernières étapes de la méthodologie de conception.

Le chapitre qui suit passe à travers tous ces points d'une façon plus détaillée, en commençant par la description de la structure générale d'un système hybride reconfigurable.

Chapitre 2

La méthodologie de co-design matériel/logiciel

Dans ce chapitre, nous présenterons d'abord une vue d'ensemble de la problématique de recherche relative au choix d'une méthodologie de co-design matériel/logiciel débouchant sur la réalisation d'un système hybride reconfigurable. Par la suite, nous traiterons de la structure modulaire de l'outil d'analyse, puis nous décrirons les spécifications respectives rattachées à chacun des modules.

Une partie des concepts de base sur le co-design utilisés dans ce chapitre proviennent du travail de Doucet [DOU99] sur la définition de l'architecture générale du système.

2.1 La problématique

La conception d'applications hybrides de type matériel/logiciel a longtemps été un art exercé par de petits groupes de chercheurs universitaires et privés. L'apparition d'une telle philosophie de conception vient probablement du besoin qu'a eu l'industrie, dans le secteur des hautes technologies, de systèmes de plus en plus performants pour être en mesure d'exécuter des applications de plus en plus complexes et ce, le plus rapidement possible. Cependant, le partitionnement s'effectue manuellement et selon l'expérience du concepteur. Ces tâches sont souvent réservées aux ingénieurs les plus expérimentés ayant

la capacité, en fonction de leurs expériences en conception, de déterminer quels seront les blocs qui seront plus efficace en matériel et ceux qui seront plus efficace en logiciel respectivement. Il y a donc quasi absence de procédures automatisées permettant d'obtenir de l'information sur la nature des programmes analysés. Les mécanismes d'interface entre les partitions logicielles et matérielles sont souvent improvisés et sont décrites dans des langages non-formels et non-standardisés. Un langage standard est généralement un gage de compatibilité, de portabilité, de stabilité et de pérennité. Ce manque de formalisme réduit de façon importante la performance globale des systèmes, un certain nombre de facteurs provoquent ces pertes de performance. Le premier de ces facteurs est, sans contredit, que la conception et la mise en œuvre de ces partitions repose sur une méthodologie de conception ponctuelle ne respectant aucune spécification fonctionnelle commune. Le second découle des délais importants reliés à l'intégration de ces partitions afin de respecter, d'une part, les spécifications fonctionnelles respectives à chacune et, d'autre part, la synchronisation de ces deux processus, ainsi que leur mode de fonctionnement spécifique. Pour palier à ces problèmes, les concepteurs ont tendance à mettre en œuvre le plus de segments du système en logiciel et de minimiser les partitions matérielles. Par conséquent, ce genre d'approche résulte très souvent en des partitionnements matériel/logiciel plus ou moins efficaces et non-optimaux. Cependant, on retrouve actuellement sur le marché des outils de co-simulation comme Eagle de Synopsys et Seamless CVE de Mentor Graphics qui permettent de vérifier la synchronisation d'une façon automatisée. Ces outils sont mature et très efficace.

Au cours de la dernière décennie, une nouvelle méthodologie de conception a vu le jour. Cette méthodologie se nomme le co-design et elle inclut, dans un même processus, à la fois les parties logicielles et les parties matérielles. Le co-design permet, lors de la conception, de séparer les parties qui seront en logiciel des parties qui seront en matériel selon des critères rigoureux. Cependant, peu d'outils de co-design sont disponibles et sont souvent orientés pour des applications spécifiques ou des grammaires pré-sélectionnées.

Notre approche propose un certain nombre de critères ou métriques de performance servant à détecter et à caractériser chacun des segments de code d'une application donnée écrite en langage C ANSI. À partir des valeurs de métriques trouvées, une fonction paramétrique a été développée pour quantifier chacun de ces segments afin de prédire la pertinence d'éventuels partitionnements matériel/logiciel sur une architecture matérielle reconfigurable de type FPGA. La mise en œuvre de cette fonction est basée sur un algorithme de partitionnement respectant à la fois les contraintes de temps et les contraintes imposées par l'architecture matérielle ciblée.

2.2 Les spécifications du système

Comme nous venons de le voir, ce travail de recherche consiste à développer un outil d'analyse afin de prédire la pertinence de transposer et d'exécuter des segments de code en matériel afin d'accélérer la vitesse d'exécution de certains segments d'une tâche. Essentiellement, cet outil devra transformer une description dans un système formel de départ vers une description dans un système formel d'arrivée, en utilisant les règles de traduction appropriées selon les résultats des métriques de performance. Évidemment, cette section présente une vue d'ensemble du projet afin de permettre au lecteur de se faire une idée générale des différents facteurs considérés.

2.2.1 Structure générale du système

La structure générale du système, illustrée à la figure 2.1, permet d'identifier les différentes fonctionnalités du système. Les domaines de fonctionnalité couverts par les systèmes formels de départ et d'arrivée ne sont pas les mêmes. On cherche à traduire les segments de code qui sont dans l'intersection des domaines de fonctionnalité. La détermination des segments se trouvant dans cette intersection est obtenue via les résultats des métriques de performance.

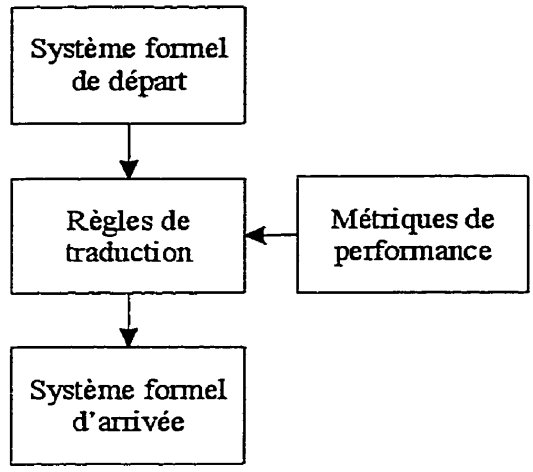


FIG. 2.1 Structure générale du système

2.2.2 Système formel de départ

Le système formel de départ décrit des applications dans un langage de haut niveau ciblé pour un ordinateur avec un microprocesseur séquentiel. La figure 2.2 illustre l'architecture d'un tel ordinateur sous sa forme la plus simplifiée.

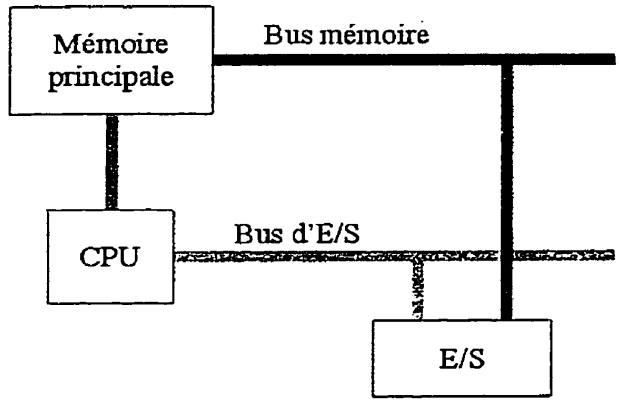


FIG. 2.2 Architecture logique du système initial

La description de chacune des applications séquentielles est écrite en langage C ANSI. La figure 2.3 illustre le flux de compilation traditionnel relatif au système formel de départ.

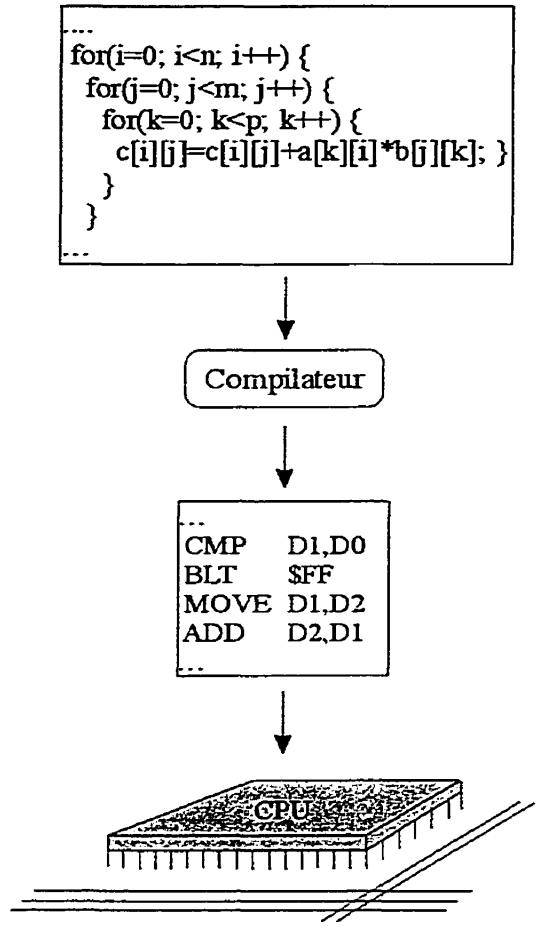


FIG. 2.3 Flux de compilation traditionnel

Essentiellement, la compilation effectue une traduction d'un programme écrit en langage évolué en un programme équivalent en langage machine. Le programme qui exécute la compilation utilise le programme source écrit en langage source comme données et produit, comme résultat, un programme objet en langage objet, ce dernier pouvant être en fait le langage assembleur ou directement le langage machine, qui sera exécuté séquentiellement sur le microprocesseur cible.

2.2.3 Système formel d'arrivée

Le système formel d'arrivée décrit des applications dans un langage ciblé pour un processeur reconfigurable de type FPGA. L'architecture d'un tel processeur est illustrée,

sous sa forme la plus simplifiée, à la figure 2.4. Notons que le bloc de gauche, dans cette figure, intitulé *Architecture logique du système initial*, correspond exactement à ce qui est illustré à la figure 2.2. On veut schématiser, ici, que l'architecture logique cible du système hybride reconfigurable est constituée, d'une part, de l'architecture logique du système initial et, d'autre part, de l'architecture logique du co-processeur FPGA. Rappelons que le FPGA possède à la fois de la mémoire interne et externe.

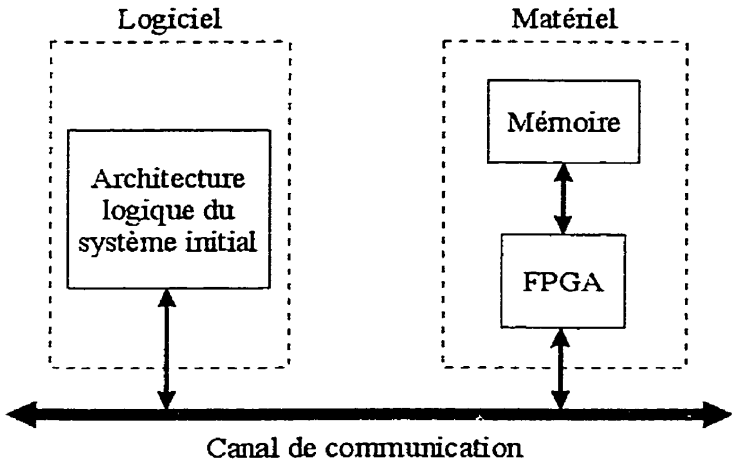


FIG. 2.4 Architecture logique cible du système hybride reconfigurable

La description de chaque application, pour le système hybride, est écrite dans une combinaison de deux langages, soient le langage C ANSI et le langage VHDL. Le VHDL est un langage compilé vers des descriptions matérielles au niveau des registres et des portes logiques. Ces descriptions sont chargées par la suite dans le processeur reconfigurable. La figure 2.5 illustre le flux de compilation relatif au système formel d'arrivée.

2.2.4 L'architecture matérielle

Dans la présente section, nous présentons d'abord une vue d'ensemble des caractéristiques de la plate-forme matérielle. Ensuite, nous traitons des différents éléments programmables présents dans cette architecture. Après quoi, nous abordons le

langage de description matérielle VHDL, puis nous proposons une brève introduction qui permet de mettre en évidence certains facteurs importants reliés au choix de ce langage.

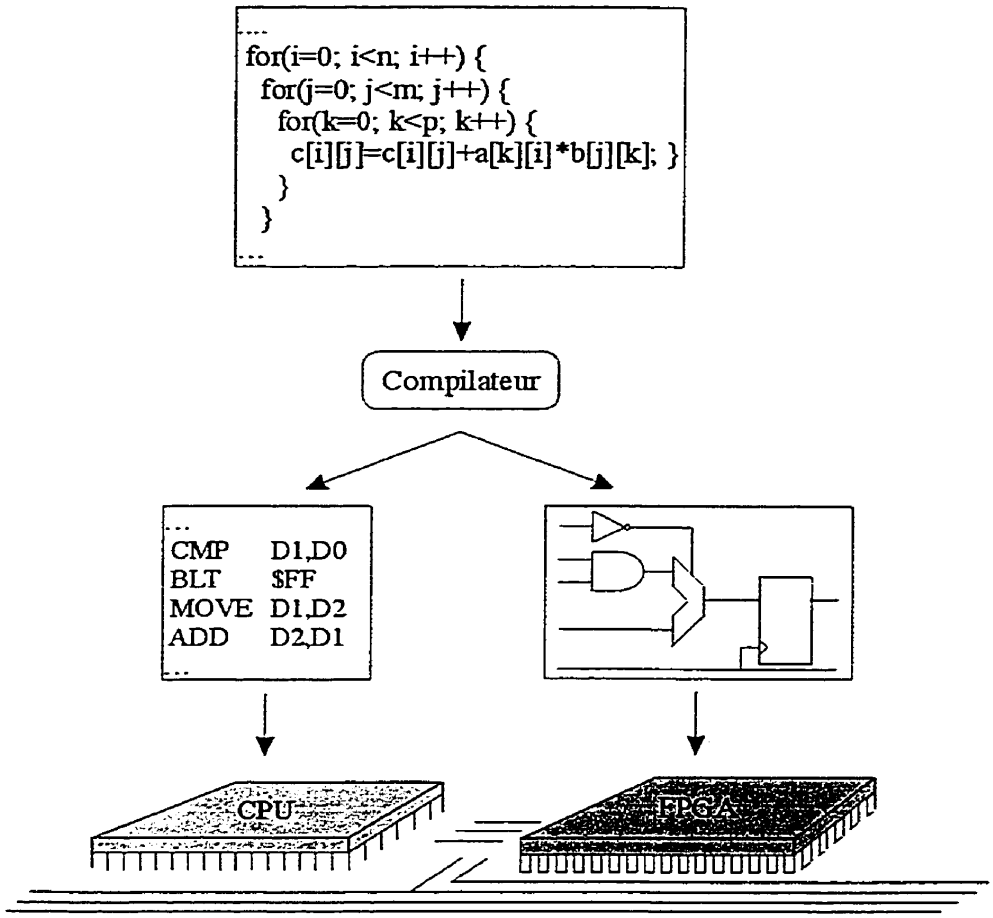


FIG. 2.5 Flux de compilation recherché

2.2.4.1 Les FPGA

Un FPGA est un circuit reconfigurable. Son architecture interne, présentée à la figure 2.6, est constituée essentiellement de blocs logiques programmables (CLBs), de matrices de routage et de blocs mémoire. Un CLB (Configurable Logic Block) est un module logique qui met en œuvre une table de vérité. La figure 2.7 illustre les différents composants constituant un CLB. Les matrices de routage établissent les liens entre les CLBs, la mémoire et les broches physiques de la puce. En combinant chacun des CLBs, on arrive à implanter diverses fonctions logiques. Le lecteur intéressé par les circuits

programmables (FPGAs, ASICs, etc.) et leurs architectures respectives peut consulter divers ouvrages spécialisés [DUT97, SMI97, XIL99a].

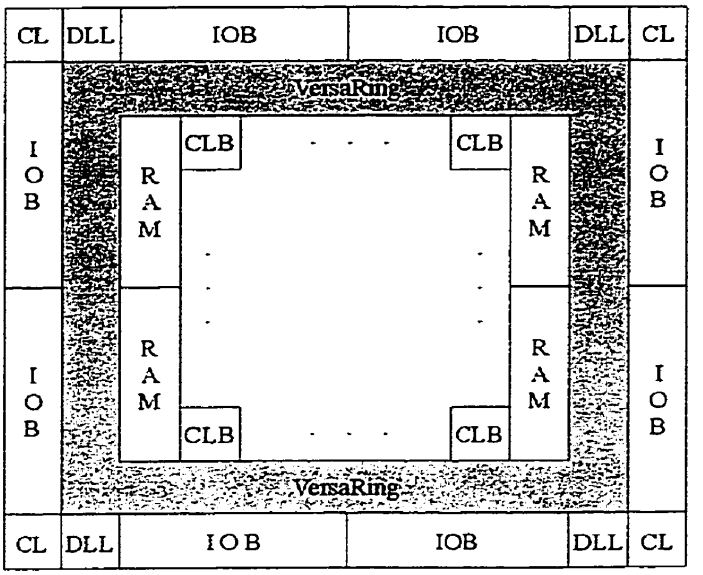


FIG. 2.6 Vue d'ensemble de l'architecture Virtex

Une description en langage VHDL peut être compilée vers une description de séries de tables logiques. Ensuite, un placement et un routage détermine les liens à établir entre les CLBs pour réaliser matériellement les fonctionnalités sur le FPGA.

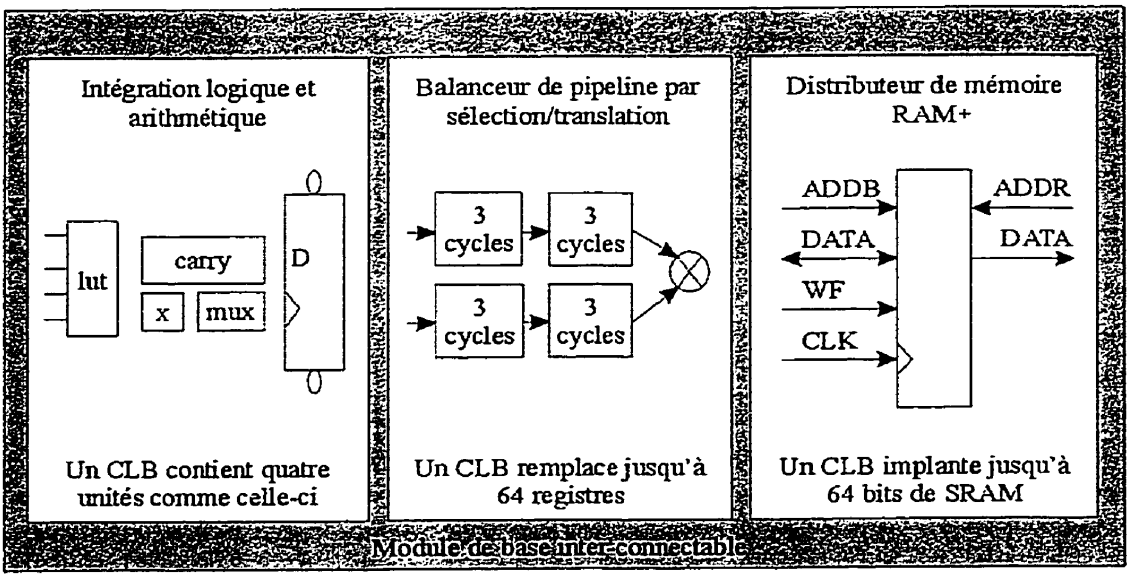


FIG. 2.7 Vue schématique d'un bloc logique programmable (CLB)

2.2.4.2 Le langage VHDL

C'est dans le courant des années 1970 que le ministère américain de la défense (DoD) lança le programme VHSIC (Very High Speed Integrated Circuits), une initiative de grande ampleur destinée à produire des circuits intégrés complexes de très haute performance. Il apparut rapidement que, compte tenu du grand nombre de sociétés concernées, le défi majeur résidait moins dans l'effort technologique que dans la normalisation de la communication entre les différents intervenants. La plus grande diversité régnait parmi les outils utilisés, non seulement entre les différentes compagnies contractées, mais aussi bien entre les différents départements d'une même compagnie, voire entre différents projets gérés par un même département. Cet état de fait a motivé l'émergence du VHDL (VHSIC Hardware Description Language), qui visait entre autres à fournir un environnement de développement matériel complet unifié. Standardisé en décembre 1987 par l'Institute of Electrical and Electronics Engineers sous la référence IEEE-1076, le VHDL se présente donc comme un langage unique pour la description, la modélisation, la simulation, la synthèse et la documentation [LAR97].

Son utilisation pour la synthèse logique, autrement dit la création de circuits à partir de descriptions textuelles, est plus récente, et est notamment reliée à l'essor extraordinaire des composants logiques programmables, qu'ils se nomment CPLD ou FPGA. Il fallait, à ces architectures devenues complexes, un langage descriptif de haut niveau, en remplacement des langages de première génération aux fonctionnalités limitées ou aux méthodes de capture schématique. Il leur fallait également un langage universel, ne verrouillant pas l'utilisateur à une architecture unique. Il leur fallait enfin un langage démocratique, apte à promouvoir des environnements de développement économiques. Toutes ces conditions sont aujourd'hui remplies : le VHDL est devenu le langage descriptif le plus répandu parmi les concepteurs de circuits.

Cette présentation est certes incomplète, puisque nous ne donnons pas nécessairement au lecteur l'information relative à l'utilisation de ce langage. Cependant, le lecteur intéressé peut toujours consulter divers ouvrages [AUM96, LAR97].

2.2.5 Règles de traduction

À cette étape, on cherche à traduire les segments de code de l'application qui sont dans l'intersection des domaines de fonctionnalité. On va pouvoir déterminer quels sont les segments qui sont dans cette intersection à l'aide des informations fournies par les métriques de performance. Ces informations permettront de choisir les règles de traduction appropriées.

2.2.6 Métriques de performance

Les métriques de performance permettent de caractériser chacun des blocs de base de l'application logicielle, ces métriques sont : le temps d'exécution, l'espace mémoire requis, la bande passante requise et la surface matérielle utilisée sur le FPGA. La pertinence d'un éventuel partitionnement matériel/logiciel d'un bloc de base est déterminée par une fonction paramétrique définie à l'aide de ces métriques. Le résultat de cette fonction est ensuite comparé à un seuil d'acceptabilité [MIC94].

Nous consacrons un chapitre entier (chap. 3) à la description et au mode d'évaluation de chacune des métriques.

2.3 La structure logicielle de l'outil d'analyse

Maintenant que le système a été conceptualisé, voyons la structure logicielle de l'outil d'analyse qui permettra d'intégrer lesdites métriques de performance.

2.3.1 Le flux de données du système

La première étape fut de modéliser la structure de l'outil. À cette étape, il faut d'abord dresser la liste des fonctionnalités que l'on désire obtenir. Après quoi, on peut adopter une méthodologie de conception modulaire favorisant l'extensibilité et ainsi différents projets peuvent venir se greffer par la suite. Chacun des projets peut consister

au développement d'un module spécifique. Le présent projet reflète bien cette philosophie de conception.

La figure 2.8 présente le flux de données à travers le système ainsi que l'ensemble des principaux modules. L'AST (Abstract Syntactic Tree) de l'application logicielle en entrée est construit par le module d'analyse lexicale et syntaxique. Par la suite, un autre module construit le CFG (Control Flow Graph). Chaque nœud dans le CFG possède une liste d'instructions dont chacune consiste en une liste d'opérations. Ces opérations ont des pointeurs aux nœuds du DFG (Data Flow Graph).

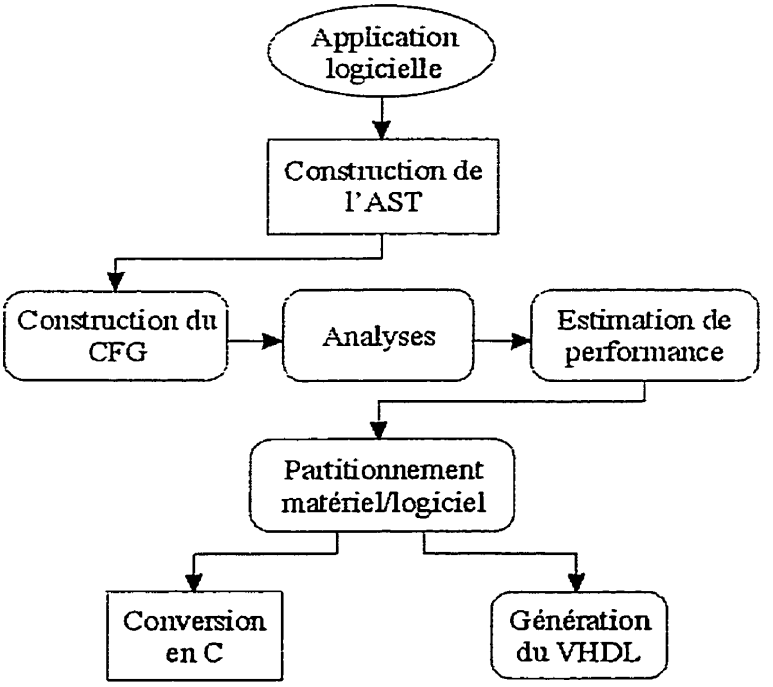


FIG. 2.8 Diagramme de flux de données

Dans le CFG, les nœuds représentent les structures de contrôle comme *if-then-else*, *for*, *while*, etc. Normalement, le CFG correspond exactement à un nœud *Begin* (début du programme) où le flux de contrôle débute et à un nœud *End* (fin du programme) où le flux de contrôle termine. Les arcs dans les graphes sont orientés et représentent le flux de contrôle. Un arc originaire d'un nœud *N1* et pointant vers un nœud *N2* signifie que le nœud *N1* devra être exécuté en premier, et que le nœud *N2* sera exécuté par la suite. À

tout moment, seulement un nœud de contrôle dans le CFG peut être exécuté à la fois. Dans le cas où les nœuds de flux de contrôle ont deux arcs ou plus vers différents nœuds, un seul doit être choisi à la fin de l'exécution de ce nœud. Par exemple, le nœud *if* du CFG possède deux arcs sortants : un qui pointe vers le nœud *then* et l'autre vers le nœud *else*. Tous les nœuds du CFG possèdent une section spécifique qui évalue les expressions représentées par le DFG.

Le DFG est divisé en plusieurs sous-ensembles disjoints. Chaque sous-ensemble décrit une section linéaire du programme (pas de branchement). Un DFG est toujours attaché à un CFG qui décrit les constructions de contrôle alors que le DFG décrit les instructions et les expressions utilisées pour évaluer les conditions de branchement ou autres opérations.

Lorsque ces différents graphes sont construits et disponibles, le module d'analyse de flux et de dépendance suggérées dans [AHO86, FER87] effectue une série de passes pour générer de l'information sur le comportement de l'application analysée. Par la suite, un profilage dynamique est effectué. C'est-à-dire que le même programme est exécuté plusieurs fois sur le processeur cible et le temps moyen d'exécution est choisi comme étant le temps d'exécution du bloc logiciel.

Le module de partitionnement matériel/logiciel effectue le découpage entre le logiciel et le matériel suivant les résultats des métriques, afin de déterminer la partition qui sera la plus désirable. Une fois le partitionnement effectué, les parties de l'AST qui restent en logiciel sont converties en C et, les parties ciblées vers le matériel sont converties en VHDL.

2.3.2 Le compilateur SUIF

Afin de réaliser le "parsing" de l'application logicielle en entrée qui englobe, d'une part, les analyses lexicale et syntaxique et, d'autre part, la construction de l'AST, nous avons choisi d'utiliser le compilateur SUIF [SUI99]. Le lecteur intéressé par ce

compilateur et par son architecture interne peut consulter [KIE98, SUI99, WIL98]. Ce compilateur a été développé par un groupe de recherche de l'Université de Stanford et est une version améliorée du compilateur LCC [FRA95]. Les critères pour ce choix furent la qualité du design et de sa mise en œuvre, les mécanismes d'extensibilité, son utilisation par la communauté scientifique dans ce domaine et, finalement pour la séparation entre le "parsing" et la construction de l'AST. Ce dernier critère garantit que le langage en entrée peut être remplacé par un autre, c'est-à-dire que la construction de l'AST est indépendante de la grammaire. De ce fait, notons que SUIF supporte les grammaires du langage C et Fortran. De plus, ce compilateur est du domaine public et il n'y a pas de restriction sur les droits de copie.

L'idée de réutiliser l'architecture interne dudit compilateur est d'augmenter sa base de données, suite à un profilage effectué antérieurement, en vue de déterminer des indicateurs de performance [HAR96, LIE97] pour quantifier la pertinence d'éventuels partitionnements matériel/logiciel.

2.3.3 Le co-design matériel/logiciel

Les différentes phases d'une conception conjointe matériel/logiciel sont montrées à la figure 2.9 [ADA96]. La conception du système ou, plus particulièrement dans cette étude, de l'application logicielle, n'est pas considérée dans ce travail. Notre mandat est d'analyser ces applications logicielles et de déterminer la pertinence de transposer et d'exécuter certains segments de code afin d'en accélérer la vitesse d'exécution. À cet effet, la granularité des objets, correspondant à la dimension des parties qui seront implantées en logiciel ou en matériel, est représentée par un bloc de base. Chacun des blocs de base est représenté par une boucle ou un nid de boucles. Il y a autant de blocs de base qu'il y a de boucles dans l'application.

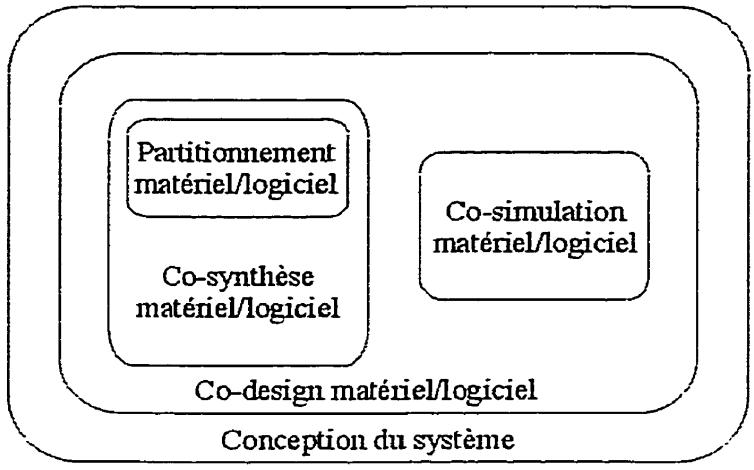


FIG. 2.9 Phases d'une conception conjointe matériel/logiciel

C'est dans la phase de conception conjointe matériel/logiciel que le système est partitionné et que ses différentes parties sont implantées soit en logiciel, soit en matériel.

La phase de partitionnement matériel/logiciel peut se faire de façon formelle. Il s'agit d'utiliser une fonction pour caractériser le choix du partitionnement par une quantité qui est comparée à un seuil d'acceptabilité. Cette fonction prend en compte l'accélération, le coût du matériel, la séquentialité du logiciel, etc. [MIC94]. C'est d'ailleurs dans cette optique qu'ont été développées chacune des métriques de performance permettant justement de tenir compte de ces différents facteurs. Le processus de co-design n'est pas à ce jour un processus automatisé, mais il y a beaucoup de recherche dans cette voie. Pour supporter le co-design, il existe des outils de co-synthèse et de co-simulation comme Monet et Seamless CVE de Mentor Graphics. Ce dernier permet de simuler le matériel et le logiciel simultanément afin de déterminer le meilleur partitionnement.

2.3.4 Le partitionnement matériel/logiciel

Le partitionnement matériel/logiciel [GAJ95, PET95] consiste à porter les segments de code de l'application logicielle en matériel, en fonction des résultats des métriques de performance développées. À cet effet, deux types de modules de génération de code sont

nécessaires. Il y a un module de traduction pour la partie logicielle et un module de traduction pour la partie matérielle, comme l'illustre la figure 2.5.

La figure 2.10 montre la configuration type d'un système de partitionnement [GAJ95]. Un algorithme est utilisé pour déterminer les fonctionnalités qui seront implantées en logiciel et celles implantées en matériel.

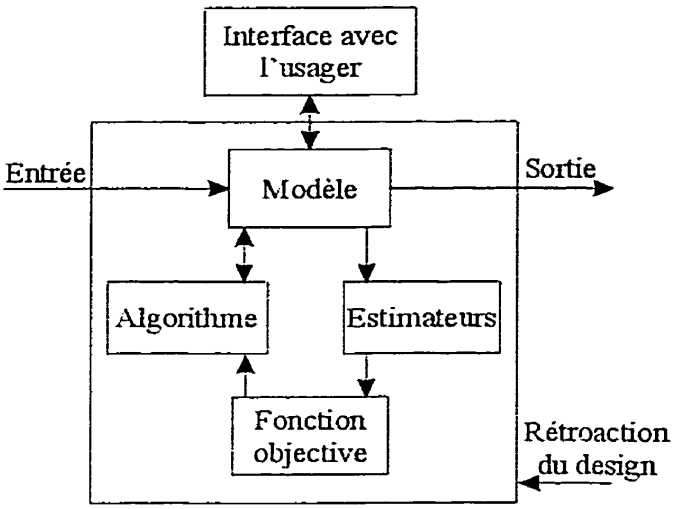


FIG. 2.10 Configuration type d'un système de partitionnement

2.3.5 La partie logicielle

L'émission du code, pour la partie logicielle, est réalisée par une primitive disponible avec le compilateur SUIF [KIE98, WIL98]. Cette primitive permet d'émettre du code C à partir de l'AST. Par la suite, il suffit de compiler ledit code par un compilateur spécialisé pour un processeur DSP (Digital Signal Processing) ou par un compilateur ciblé pour un processeur commercial de type Pentium. Finalement, ce dernier sera exécuté par le processeur cible.

2.3.6 La partie matérielle

L'émission du code, pour la partie matérielle, est réalisée par un module traducteur de C au VHDL RTL [ASH96]. Ce module est présentement en cours de réalisation par un

des membres du projet CODE. Il sera greffé à la structure logicielle de SUIF, car ce type de fonctionnalité n'existait pas dans la trousse de base de SUIF. La traduction terminée, on obtient un code exécutable sur l'architecture matérielle cible. Ce code correspond à la description matérielle du comportement logique des segments de code logiciel déportés vers le matériel.

Afin d'obtenir des versions préliminaires reliées à une éventuelle exécution matérielle, il serait intéressant d'envisager l'utilisation de librairies actuellement disponibles sur le marché comme C Level [BRA00, CLE00] et SystemC [SYS00]. Ces librairies de haut niveau permettent de convertir à partir du langage C ANSI une description matérielle au niveau RTL ciblée pour le langage VHDL ou Verilog. De cette façon, nous serions en mesure de valider dès maintenant notre méthodologie sans attendre que notre module traducteur de C au VHDL RTL soit totalement fonctionnel. Mentionnons, en terminant, que ces outils n'effectuent aucune transformation du type parallélisation, ré-ordonnancement de code, etc.

Chapitre 3
Les métriques de performance

Le présent chapitre donne la description et le mode d'évaluation de chacune des métriques permettant d'identifier des boucles chaudes à l'intérieur d'applications logicielles écrites en langage C ANSI.

3.1 L'approche suivie

3.1.1 Introduction

Plusieurs facteurs peuvent influencer la façon dont le partitionnement entre le logiciel et le matériel sera fait. Les métriques que nous allons utilisées pour nos applications sont le temps d'exécution, l'espace mémoire requis, la bande passante requise et la surface matérielle du FPGA nécessaire. Ces métriques de performance permettent de caractériser chacun des blocs de base d'un programme, représentés essentiellement par une boucle ou nid de boucles. La pertinence d'un éventuel partitionnement matériel/logiciel d'un bloc de base est déterminée par une fonction paramétrique. Cette dernière intègre chacune de ces métriques, afin de caractériser le choix du partitionnement par une quantité, qui est comparée à un seuil d'acceptabilité [DEM94]. Celle-ci sera décrite plus loin dans ce chapitre.

3.1.2 Modèle architectural de l'outil

3.1.2.1 Généralités

Dans les pages précédentes, plus particulièrement aux sections 2.2 et 2.3, nous avons présenté l'architecture à très haut niveau du système[2] tant logiciel que matériel. Ce système utilise le compilateur SUIF qui est utilisé pour générer l'AST des programmes en entrée. L'AST de SUIF est construit en langage C++. L'AST fournit par le compilateur SUIF est à la base de notre outil. Il est important de noter que la représentation intermédiaire de SUIF est, par défaut, sous forme d'AST.

Par dessus celui-ci, le graphe de flux de contrôle (CFG) est annoté (on pourrait aussi dire greffé) sur les nœuds de l'AST. L'information de flux sera annoté sur les nœuds du CFG, sous forme d'ensemble, sur lesquels sont appliqués des opérateurs mathématiques comme l'union et l'intersection.

Un cadre de référence est utilisé pour effectuer les analyses de flux [DOU99]. Elle propose une interface permettant l'intégration de l'information venant à la fois des méthodes SUIF et des nouvelles structures développées. Ainsi, le format de stockage de l'information respectera un standard pour l'ensemble des analyses effectuées. Le lecteur intéressé à approfondir ce sujet peut consulter les ouvrages cités aux références [AHO86] et [FER87].

3.1.2.2 Interface entre les modules

Un diagramme bloc de l'architecture de l'outil apparaît à la figure 3.1. Les détails relatifs à la mise en œuvre du module de profilage et d'estimation de performance, nommé *ProfEstim*, sont présentés plus loin dans cette section. Par ailleurs un diagramme détaillé illustrant le flux de données global entre les principaux modules est reproduit à l'annexe A. Par conséquent, nous limiterons ici la discussion sur l'architecture générale

---

[2] Le terme *système* est utilisé ici pour désigner l'intégration de tous les modules constituant ledit outil.

de l'outil, et nous mettrons d'avantage l'accent sur la conception ainsi que la mise en œuvre du module *ProfEstim* qui constitue, à toutes fins pratiques, l'essentiel de notre recherche. Cependant, les ouvrages cités aux références [DOU98] et [DOU99] discutent plus longuement de cette architecture.

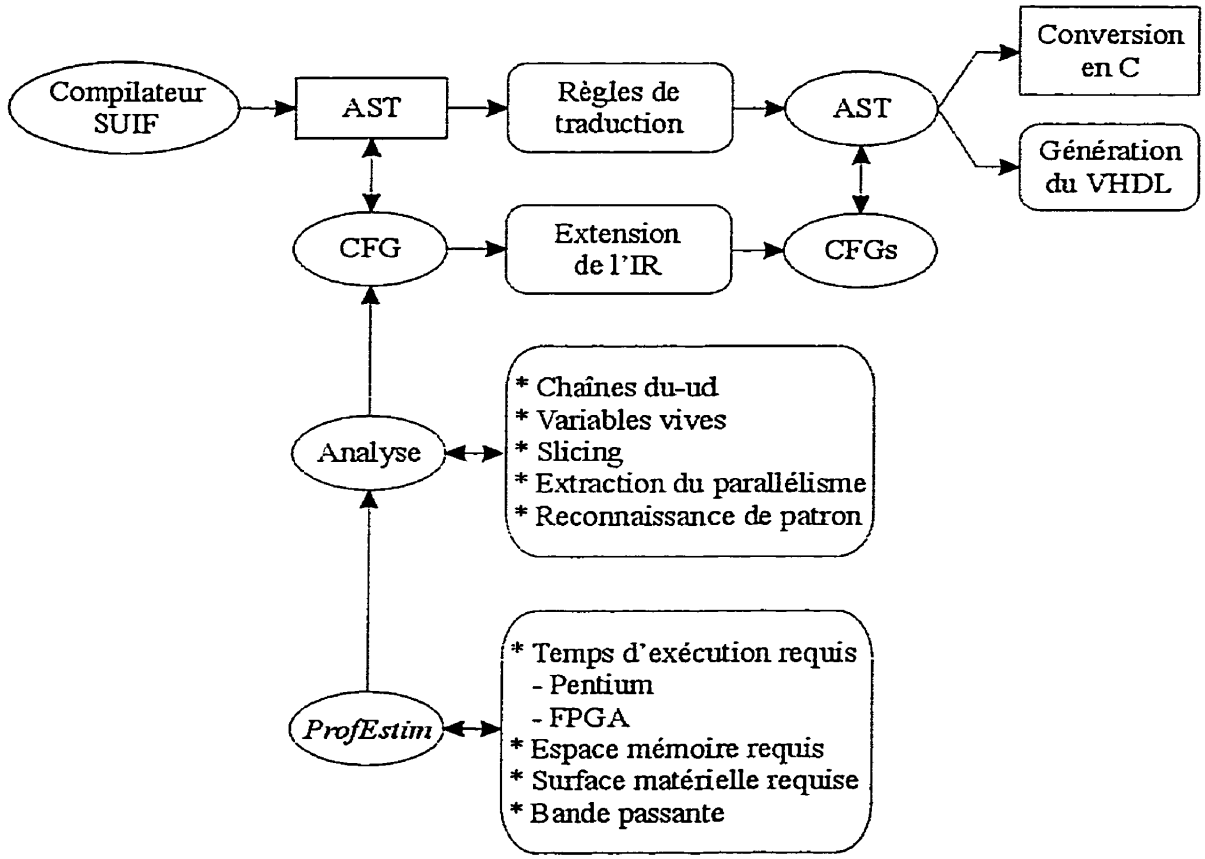


FIG. 3.1 Diagramme bloc du modèle architectural de l'outil

La méthodologie de conception adoptée pour le développement du module *ProfEstim* a été de type modulaire afin de faciliter l'extensibilité (p. ex. l'ajout et/ou la suppression de métriques de performance) à l'intérieur du module.

En pratique, chaque sous-module de ce module principal met en œuvre une métrique. Cette approche permet d'insérer autant de métriques que l'on désire à l'intérieur de l'outil, il suffit simplement d'adapter l'interface du module *ProfEstim* afin de référencer le nouveau sous-module au reste de la structure logicielle.

3.1.2.3 Description du module ProfEstim

Pour obtenir les valeurs de chacune des métriques, des sous-modules ont été développés, en utilisant le compilateur SUIF et le langage C++. L'ensemble de ces sous-modules a été greffé à l'architecture interne du compilateur (fig. 3.1) et forme un prototype d'outil d'analyse de performance permettant d'évaluer chacune des métriques.

Le premier sous-module permet d'estimer les temps d'exécution requis sur un Pentium et sur un co-processeur FPGA. Celui-ci insère automatiquement les fonctions de comptage à l'intérieur du programme source de l'application. Ce sous-module a été développé en C++ et utilise le compteur de cycles du Pentium.

Pour obtenir les valeurs de la métrique concernant l'espace mémoire requis pour exécuter une boucle dans un circuit programmable, un sous-module d'enregistrement des séquences d'accès aux tableaux a été développé en C++ et un module du compilateur SUIF a été modifié afin d'extraire l'information relative aux variables vectorielles, lors de la compilation.

En ce qui concerne la surface matérielle du circuit programmable utilisée pour exécuter une boucle, un autre sous-module a été développée en SUIF. Ce dernier permet, d'une part, de compter le nombre d'opérateurs utilisés et, d'autre part, de déterminer la longueur des mots de chaque opérande de chacune des instructions du bloc de base et de déterminer le nombre de CLB requis pour mettre en œuvre lesdites instructions en matériel.

Finalement, un dernier sous-module évalue la bande passante requise. Celui-ci permet d'estimer le taux de transfert des données que l'on doit fournir à l'unité de calcul pour effectuer une tâche. Ce taux est caractérisé par l'ensemble des variables d'entrée et de sortie par étape de contrôle.

La méthodologie de conception et la mise en œuvre de chacune des métriques de performance fera l'objet de la prochaine section. Pour l'instant, nous allons aborder les

différentes hypothèses de départ qui ont été considérées pour définir les diverses solutions matérielles envisagées.

3.1.3 Modèle de l'architecture matérielle

Comme nous l'avons mentionné précédemment, le but de ce présent mémoire n'est pas de démontrer que l'on peut accélérer des applications en utilisant des ressources matérielles de type FPGA. Ceci a été démontré, entre autre, dans [SAV96]. Nous cherchons à démontrer ici la faisabilité de mesurer la performance en vue d'automatiser le processus de partitionnement matériel/logiciel. Pour l'étude, nous avons considéré un seul FPGA et uniquement un seul opérateur additionneur, soustracteur et multiplicateur est disponible sur le circuit programmable. Les solutions matérielles envisagées dans chacun des cas analysés seront tous basées sur ce modèle simplifié. La conséquence de cette hypothèse simplificatrice se reflétera donc sur la nature des résultats obtenues présentés au chapitre suivant. Évidemment, notre modèle pourra être étendu afin de considérer les cas les plus fréquemment rencontrés dans les programmes scientifiques.

Ainsi, nous proposons un prototype d'un outil permettant l'estimation de métriques de performance en utilisant le compilateur SUIF et le langage C ANSI. Les exemples considérés sont utilisés essentiellement pour valider la faisabilité de mesurer la performance avec cet environnement.

3.2 Les métriques de performance

Dans cette section, nous allons décrire chacune des métriques d'une façon plus détaillée, en commençant par la description des métriques. Nous allons expliquer la façon de les évaluer pour la partie logicielle et pour la partie matérielle. Puis, nous allons enchaîner avec la fonction paramétrique et finalement, l'algorithme de partitionnement.

Il est important de mentionner que l'ensemble des exemples qui seront présentés dans ce mémoire ont été étudiés en utilisant, comme processeur standard, un Pentium 233

MHz et, comme plate-forme matérielle, un FPGA XCV300-200MHz de la famille Virtex de XILINX. Cependant, on aurait pu utiliser une variété d'autres plates-formes.

3.2.1 Le temps d'exécution

La première métrique de performance consiste à estimer le temps d'exécution requis sur le Pentium et sur le FPGA afin de déterminer laquelle des deux approches est la plus efficace pour effectuer la tâche.

3.2.1.1 Sur le Pentium

Pour déterminer le temps d'exécution sur le Pentium, on peut utiliser un compteur de cycles CPU. Sur un Pentium [INT99], ce dernier se nomme RDTSC (ReaD Time Stamp Counter). L'accessibilité à ce compteur se fait via un appel de fonction, écrite en assembleur. Il suffit donc, au début et à la fin de chacun des blocs de base, de placer les fonctions *startChrono()* et *stopChrono()* respectivement. Ceci permettra de compter le nombre de cycles requis pour effectuer la ou les sous-tâches qui nous intéressent. Pour obtenir le nombre moyen de cycles, on exécute le même programme un certain nombre de fois. Dans le tableau 3.1, on a exécuté 3 fois le programme.

Si on désire déterminer le temps d'exécution, en unité de temps, il suffit d'utiliser l'équation suivante :

$$T_{EXE} = \frac{1}{f} C \tag{3.1}$$

où T_{EXE} est le temps d'exécution exprimé en unité de temps, f est la fréquence du processeur et C le nombre de cycles requis.

Le calcul de la suite de Fibonacci sera utilisée ici comme exemple pour illustrer chacune des métriques. La figure 3.2 présente le code source de ce calcul. On y retrouve les appels de fonction du compteur RDTSC. Le tableau 3.1 présente les résultats obtenus après l'exécution de ce programme (fig. 3.2).

TABLEAU 3.1 Cycles d'horloge requis sur le Pentium 233 MHz pour exécuter le programme qui calcule la suite de Fibonacci lorsque *Fibnum* = 45

Cpt	Essai 1	Essai 2	Essai 3	Moy	%
C1	8607	8594	8300	8500	100
C2	2	2	2	2	0.02
C3	580	580	580	580	6.82
C4	8012	7999	7705	7906	93.01

Dans le tableau 3.1, *C1*, *C2*, *C3* et *C4* sont des compteurs calculant respectivement le nombre de cycles requis pour effectuer le programme en entier, le traitement avant la boucle, le traitement de la boucle et le traitement après la boucle. Notons que le temps d'appel à *startChrono()* et *stopChrono()* est nul, tous les calculs requis pour soustraire leur temps réel d'exécution ont été insérés dans leur mise en œuvre respective.

Pour déterminer le temps d'exécution sur le Pentium, on utilise l'équation 3.1. Ainsi pour la boucle #1 :

$$T_{EXE\_PENT} = \frac{1}{f}C = \frac{1}{233X10^6}(580) = 2.49\mu s \qquad (3.2)$$

3.2.1.2 Sur le FPGA

Pour déterminer le temps d'exécution sur le FPGA, on utilise l'outil COREGEN de Xilinx [XIL99b]. Le nombre de cycles requis en fonction du type d'opérateur (additionneur, soustracteur, multiplicateur) et de la taille des opérandes y est présenté. Notons qu'un facteur de correction peut être apporté pour estimer réellement le nombre de cycles requis sur un véritable FPGA. Les valeurs respectives de chacune des combinaisons sont présentées dans le tableau 3.2.

```
#include <stdio.h>
#include <stdlib.h>
#include "timer.c"
void fib(long Fibnum)
{
   long i;
   unsigned long F[Fibnum];
   startChrono(C1);          /* départ du compteur C1 */
   startChrono(C2);          /* départ du compteur C2 */
   F[0] = 0;
   F[1] = 1;
   stopChrono(C2);           /* arrêt du compteur C2 */
   startChrono(C3);          /* départ du compteur C3 */
   for(i = 2; i < Fibnum; i++)
      F[i] = F[i - 1] + F[i - 2];      } Boucle #1
   stopChrono(C3);           /* arrêt du compteur C3 */
   startChrono(C4);          /* départ du compteur C4 */
   printf("Fib = %6d\n", F[Fibnum - 1]);
   stopChrono(C4);           /* arrêt du compteur C4 */
   stopChrono(C1);           /* arrêt du compteur C1 */
}
main(int argc, char *argv[])
{
   long Fibnum;              /* Fibnum doit être plus grand ou égal à 3 */
   SetPriorityClass(GetCurrentProcess(), 31);
   init_timer();
   C1 = 0; C2 = 0; C3 = 0; C4 = 0;
   if(argc == 2)
   {
      Fibnum = strtol(argv[1], NULL, 0);
      fib(Fibnum);
      printf("C1 = %d\n", (int)C1);
      printf("C2 = %d\n", (int)C2);
      printf("C3 = %d\n", (int)C3);
      printf("C4 = %d\n", (int)C4);
   }
}
```

FIG. 3.2 Code source augmenté des fonctions de comptage

TABLEAU 3.2 Cycles d'horloge requis sur le FPGA XCV300-200 MHz en fonction des opérateurs et de la taille des opérandes

Paramètre	**Additionneur / Soustracteur**					**Multiplicateur**				
TDO	8	16	24	32	64	8	16	24	32	64
C	1	1	1	2	2	2	3	4	5	6

Dans le tableau 3.2, *TDO* est la taille des opérandes et *C* est le nombre de cycles requis pour effectuer l'opération donnée par rapport à la taille des opérandes.

Pour déterminer le temps d'exécution réel requis sur le FPGA, trois temps doivent être considérés, soit le temps de traitement de la boucle (T_B), le temps de communication (T_{COM}) et le temps de latence (T_{LAT}). Le chronogramme de synchronisation représentant chacun de ces temps est illustré à la figure 3.3. Notons que ce chronogramme permet de modéliser, d'une façon rigoureuse, l'application en cause. Il ne s'agit pas là d'un modèle générique applicable à l'ensemble des applications scientifiques.

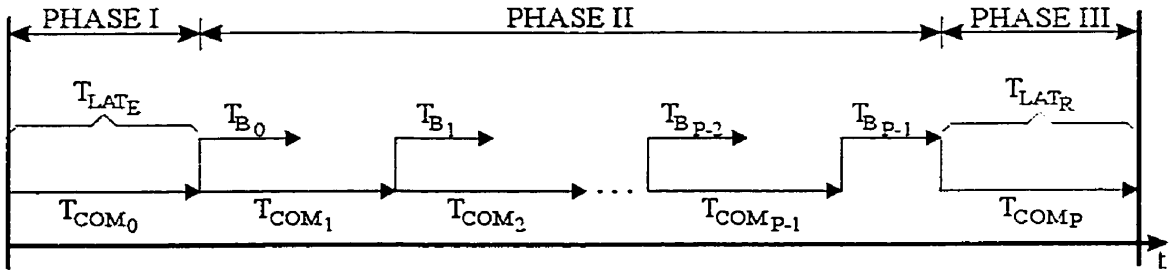


FIG. 3.3 Chronogramme de synchronisation des différents temps considérés

On distingue trois phases permettant de définir d'une façon plus détaillée le temps d'exécution réel requis sur le FPGA. La première phase illustre le temps requis et nécessaire pour envoyer les premières données initiales T_{COM_0}, du Pentium vers le FPGA. Dans cette même optique, on retrouve la réciproque de cette phase pour renvoyer les dernières données traitées T_{COM_P}, du FPGA vers le Pentium, à la fin du calcul (phase III). Or, on remarque que dans chacune de ces phases, il n'y a aucun traitement effectué, c'est-

à-dire que le FPGA est au repos. Par conséquent, on doit tenir compte de ce temps de repos et celui-ci s'appelle le temps de latence. On observe deux types de latences dans cet exemple; soit le temps de latence lors de l'envoi des données initiales T_{LAT_E} et lors du renvoi des résultats T_{LAT_R}. Le temps de latence est donné par :

$$T_{LAT} = T_{LAT_E} + T_{LAT_R} \tag{3.3}$$

Dans la deuxième phase, le FPGA effectue le traitement proprement dit de la boucle T_{B_i} sur les données qui lui sont envoyées à chaque T_{COM_i}, où i représente chacun des P passages dans la boucle. Par exemple, les données envoyées à T_{COM_1} sont traitées sur le FPGA à T_{B_1} et ainsi de suite. Par conséquent, d'après la figure 3.3, les temps effectifs de traitement et de communication que nous considérons sont donnés par les deux expressions respectives suivantes :

$$T_B = \sum_{i=0}^{P-2} T_{B_i} + T_{B_{P-1}} \tag{3.4}$$

$$T_{COM} = \sum_{i=1}^{P-1} T_{COM_i} \tag{3.5}$$

Il est facile de démontrer que la valeur du temps d'exécution réel requis sur le FPGA est donné par :

$$T_{EXE\_FPGA} = MAX\left(\sum_{i=0}^{P-2} T_{B_i}, \sum_{i=1}^{P-1} T_{COM_i}\right) + T_{B_{P-1}} + T_{LAT} \tag{3.6}$$

Nous allons aborder chacun de ces temps d'une façon plus détaillée, en commençant par la description du temps de traitement de la boucle.

3.2.1.2a) Temps de traitement de la boucle

On distingue deux termes dans le calcul de T_B. Tout d'abord, le premier terme de l'équation 3.4, représentant les $P-1$ premiers passages dans la boucle, est donné par l'équation 3.7. Cette équation est une forme équivalente de l'équation 3.1.

$$\sum_{i=0}^{P-2} T_{B_i} = \frac{1}{f} \sum_{i=0}^{P-2} C_{B_i} \tag{3.7}$$

où C_{B_i} est le nombre de cycles requis sur le FPGA pour exécuter un passage dans la boucle, c'est-à-dire pour effectuer une addition avec des opérandes entières représentées sur 32 bits (réf. tableau 3.2) et P est le nombre de passages dans la boucle ($P = 43$).

Pour déterminer la valeur de ce premier terme, on utilise l'équation 3.7 en posant 200 MHz comme fréquence d'opération du FPGA et 2 comme nombre de cycles requis.

$$\sum_{i=0}^{P-2} T_{B_i} = \frac{1}{f} \sum_{i=0}^{P-2} C_{B_i} = \frac{1}{200X10^6} \sum_{i=0}^{41} 2 = 0.42\mu s \tag{3.8}$$

Le second terme de l'équation 3.4, qui représente le dernier passage dans la boucle, est donné par l'équation suivante :

$$T_{B_{P-1}} = \frac{1}{f} C_{B_{P-1}} \tag{3.9}$$

Enfin, pour déterminer la valeur de ce terme, on pose les mêmes paramètres que dans l'équation 3.8.

$$T_{B_{P-1}} = \frac{1}{f} C_{B_{P-1}} = \frac{1}{200X10^6}(2) = 0.01\mu s \tag{3.10}$$

3.2.1.2b) Temps de communication

Il est important d'ajouter les temps de communication au temps d'exécution requis par le FPGA pour avoir une meilleure estimation du temps d'exécution. Pour ce faire, la

quantité de données à transmettre, ainsi que la largeur du bus, doivent être connus. L'équation suivante peut être utilisée pour faire une estimation rapide du temps de communication [MAD98] :

$$C_{COM} = \frac{B}{L} C \tag{3.11}$$

où C_{COM} est le nombre de cycles requis par le bus de communication pour transférer les données requises pour effectuer un passage dans la boucle, B est le nombre de bits à transférer, L est la largeur du bus et C est le nombre de cycles requis pour un transfert.

À partir des équations 3.5 et 3.11, on obtient une expression de la forme :

$$T_{COM} = \sum_{i=1}^{P-1} T_{COM_i} = \frac{1}{f}\left(\sum_{i=1}^{P-1} \frac{B_L}{L} C_{L_i} + \sum_{i=1}^{P-1} \frac{B_E}{L} C_{E_i} \right) \tag{3.12}$$

où chacune des sommations signifient respectivement les temps de communication requis pour des opérations de lecture et d'écriture sur lesdites données.

Les termes B_L et B_E signifient respectivement le nombre de bits à transférer. De même que C_{L_i} et C_{E_i} signifient respectivement le nombre de cycles requis pour un transfert.

Pour les fins de l'exemple, nous supposerons que 2 cycles sont nécessaires pour une opération de lecture ($C_{L_i} = 2$) et 3 cycles sont nécessaires pour une opération d'écriture ($C_{E_i} = 3$).

De l'équation 3.12, on obtient l'équation simplifiée suivante servant à évaluer le temps total requis pour les communications.

$$T_{COM} = \sum_{i=1}^{P-1} T_{COM_i} = \frac{1}{f}\frac{1}{L}\left(B_L \sum_{i=1}^{P-1} C_{L_i} + B_E \sum_{i=1}^{P-1} C_{E_i} \right) \tag{3.13}$$

Par exemple, supposons un bus de 32 bits de largeur ($L = 32$) dont la vitesse de transfert pourrait être de 100 MHz. Dans notre exemple, on transfère 2 opérandes entières en lecture ($B_L = 64$) et une opérande entière en écriture ($B_E = 32$) représentées chacune sur 32 bits.

Par conséquent, le temps de communication est donné par l'équation 3.14 en posant 100 MHz comme fréquence du bus et 43 comme nombre de passages dans la boucle.

$$T_{COM} = \sum_{i=1}^{P-1} T_{COM_i} = \frac{1}{100X10^6}\frac{1}{32}\left(64\sum_{i=1}^{42}(2)+32\sum_{i=1}^{42}(3)\right) = 2.94\mu s \qquad (3.14)$$

3.2.1.2c) Temps de latence

Le temps de latence est défini comme étant le temps entre l'envoi du premier ou dernier résultat nécessaire pour effectuer un calcul et le retour de la première ou dernière donnée traitée. On estime ce temps en utilisant les équations 3.3 et 3.11.

$$T_{LAT} = \frac{1}{f}\left(\frac{B_L}{L}C_L + \frac{B_E}{L}C_E\right) \qquad (3.15)$$

En utilisant les mêmes paramètres que dans la section précédente, on obtient le temps de latence suivant :

$$T_{LAT} = \frac{1}{100X10^6}\frac{1}{32}(64(2)+32(3)) = 0.07\mu s \qquad (3.16)$$

3.2.1.2d) Calcul du temps d'exécution réel

Finalement, on estime le temps d'exécution réel requis sur le FPGA, en utilisant l'équation 3.6.

D'après cette équation, le temps d'exécution sur le FPGA est le maximum entre le temps de traitement de la boucle (0.42 µs) et le temps de communication (2.94 µs). On

retiendra donc pour le temps d'exécution sur le FPGA 3.02 µs (c.-à-d. 2.94 µs + 0.01 µs + 0.07 µs).

Le tableau 3.3 synthétise les deux temps d'exécution relatifs à une éventuelle exécution, sur le Pentium et sur le FPGA, du calcul de la suite de Fibonacci.

TABLEAU 3.3 Temps d'exécution relatif à une éventuelle exécution logicielle et matérielle

Numéro de la boucle	T_{EXE_PENT} (µs)	T_{EXE_FPGA} (µs)
#1	2.49	3.02

Suite aux résultats obtenus, on est en mesure de déterminer quelle architecture est la plus performante. Or, on remarque qu'il n'est pas efficace de porter et d'exécuter la boucle sur le FPGA. On observerait un gain $G_{PENT/FPGA}$ égal à :

$$G_{PENT/FPGA} = \frac{T_{EXE\_PENT}}{T_{EXE\_FPGA}} = \frac{2.49}{3.02} = 0.82 \tag{3.17}$$

3.2.2 L'espace mémoire

Lors d'un traitement, on peut enregistrer les séquences d'accès au tableau *F[i]* afin de mesurer la durée de vie de chacune des variables (p. ex. *F[0]*, *F[1]*, etc.) et ainsi déterminer l'espace mémoire minimal requis devant être disponible pour réaliser la boucle.

Pour ce faire, la fonction *AddAccess(int no_table, int dim_table, int i, int j, ...)* est appelée à chaque fois qu'une opération de lecture est effectuée sur une des variables du tableau. À cet effet, la figure 3.4 met l'accent sur le passage de chacun des paramètres au moment de l'appel de cette fonction.

```
void fib(int Fibnum)
{
   int i;
   for(i = 2; i < Fibnum; i++)
   {
      AddAccess(1, 1, i - 1);          /* correspond à F[i - 1] */
      AddAccess(1, 1, i - 2);          /* correspond à F[i - 2] */
   }
}
```

FIG. 3.4 Code source modifié illustrant les fonctions d'accès au tableau *F[i]*

La première valeur transmise au moment de l'appel est alors reçue dans la variable *no_table*, identifiant chaque tableau par une valeur numérique unique. Dans notre exemple, le tableau *F[i]* est identifié par la valeur 1 (*no_table* = 1). La seconde valeur transmise au moment de l'appel est alors reçue dans la variable *dim_table*, considérée comme la dimension du tableau. Ce paramètre peut prendre les valeurs numériques suivantes : 1 (tableau unidimensionnel), 2, 3, ... *N* (tableau multidimensionnel). Étant donné que le tableau *F[i]* est unidimensionnel, sa valeur sera égale à 1 (*dim_table* = 1). Les valeurs subséquentes transmises au moment de l'appel sont alors reçues dans les variables *i*, *j*, ..., considérées comme les vecteurs d'itération respectifs à chacun des tableaux qui seront accédés.

Le rôle de cette fonction est d'enregistrer, lors de son exécution, les moments où ont été accédées chacune des variables (cases mémoires) afin de déterminer les séquences d'accès respectives à chacune d'elles.

Afin de visualiser ces enregistrements, la fonction *PrintAccess(int no_table, int i, int j, ...)* est appelée pour chacune des variables du tableau. La figure 3.5 montre une méthode permettant d'afficher, pour l'ensemble des variables d'un tableau, la liste de ces séquences d'accès.

```
#include <stdio.h>
#include "varsize_table.cc"
#include "output.c"
main(int argc, char *argv[])
{
   long i, Fibnum;
   if(argc == 2)
   {
      Fibnum = strtol(argv[1], NULL, 0);
      fib(Fibnum);
      for(i = 0; i < Fibnum; i++)
      {
         printf("F[%d]  ", i); PrintAccess(1, i);
      }
   }
}
```

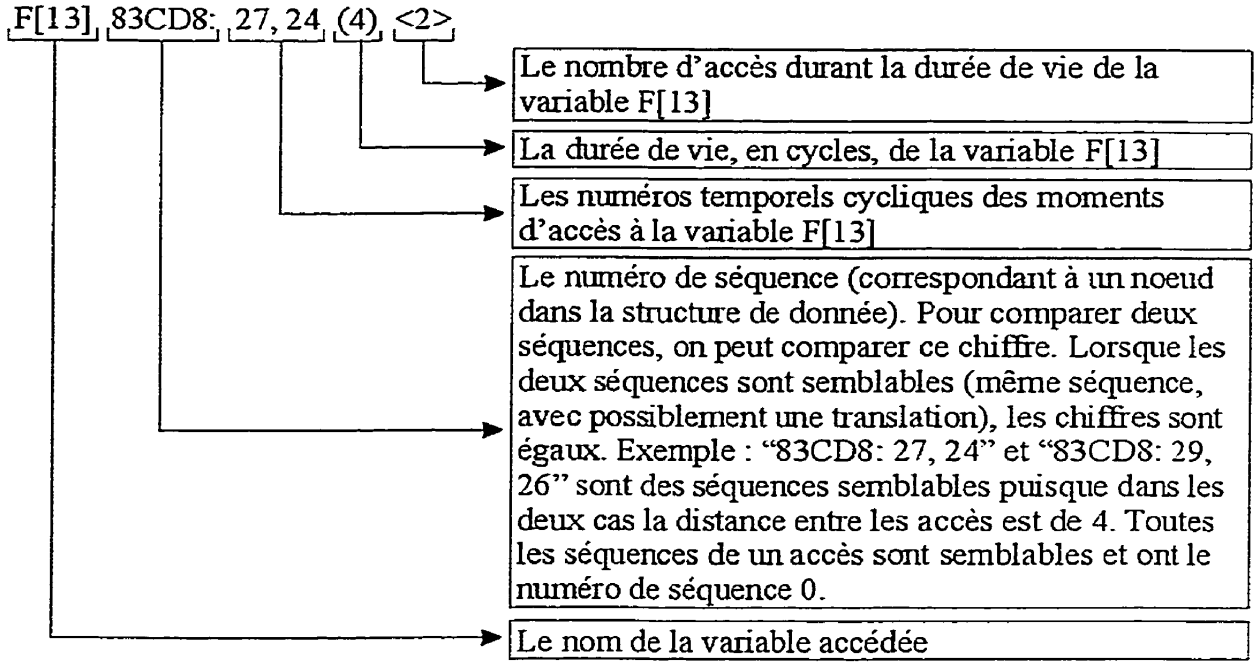
FIG. 3.5 Programme permettant de visualiser les séquences d'accès au tableau *F[i]*

La première valeur transmise au moment de l'appel est alors reçue dans la variable *no_table*, identifiant chaque tableau avec la même valeur que celle utilisée lors de l'appel de la fonction *AddAccess(...)*. À ce moment, on réfère, via cette valeur unique, au tableau qui sera analysé par les fonctions appelées. Les dernières valeurs transmises au moment de l'appel sont alors reçues dans les variables *i*, *j*, ..., considérées comme les vecteurs d'itération respectifs à chacun des tableaux.

Suite à l'enregistrement de ces séquences d'accès, on identifie les patrons relatifs aux instructions du traitement. Plus il y a de types de patrons différents, plus l'espace mémoire nécessaire sera important.

Reprenons notre exemple et analysons les séquences d'accès au tableau *F[i]*, tels qu'illustrées à la figure 3.6, suite à la compilation et à l'exécution du code présenté à la figure 3.5.

Cette figure met en évidence chacun de ces enregistrements, qui sont regroupées en ordre croissant, par rapport au vecteur d'itération *i*. Afin de mettre en lumière l'ensemble des informations recueillies pour chacun de ces enregistrements, prenons, à titre d'exemple, l'enregistrement présentant les divers moments d'accès de la variables *F[13]*.



La figure 3.7 illustre, cycle par cycle, ces séquences d'accès. De plus, on peut observer la durée de vie de chacune des variables. On détermine la durée de vie de chacune des variables en analysant les séquences d'accès de chacune d'entre elles. Par exemple, la variable *F[1]*, accédée une première fois à $t = 0$, va être accédée une seconde et dernière fois à $t = 3$, ce qui correspond à une durée de vie de 4 cycles avant que l'espace mémoire ne puisse être réutilisé. Dans un autre cas, la variable *F[0]*, accédée une première et dernière fois à $t = 1$, a une durée de vie de 1 cycle. Notons que les zones ombragées identifient les espaces mémoires utilisés. Ceci permet de visualiser l'utilisation des espaces mémoire dans le temps. À cet effet, on note que les espaces devant être lus sont en caractères gras (ex. **F[1]**) et les espaces occupés (requis pour une lecture subséquente) sont en caractères italiques (ex. *F[1]*).

```
F[0] 0: 1 (1) <1>
F[1] 83CD8: 3, 0 (4) <2>
F[2] 83CD8: 5, 2 (4) <2>
F[3] 83CD8: 7, 4 (4) <2>
F[4] 83CD8: 9, 6 (4) <2>
F[5] 83CD8: 11, 8 (4) <2>
F[6] 83CD8: 13, 10 (4) <2>
F[7] 83CD8: 15, 12 (4) <2>
F[8] 83CD8: 17, 14 (4) <2>
F[9] 83CD8: 19, 16 (4) <2>
F[10] 83CD8: 21, 18 (4) <2>
F[11] 83CD8: 23, 20 (4) <2>
F[12] 83CD8: 25, 22 (4) <2>
F[13] 83CD8: 27, 24 (4) <2>
F[14] 83CD8: 29, 26 (4) <2>
F[15] 83CD8: 31, 28 (4) <2>
F[16] 83CD8: 33, 30 (4) <2>
F[17] 83CD8: 35, 32 (4) <2>
F[18] 83CD8: 37, 34 (4) <2>
F[19] 83CD8: 39, 36 (4) <2>
F[20] 83CD8: 41, 38 (4) <2>
F[21] 83CD8: 43, 40 (4) <2>
F[22] 83CD8: 45, 42 (4) <2>
F[23] 83CD8: 47, 44 (4) <2>
F[24] 83CD8: 49, 46 (4) <2>
F[25] 83CD8: 51, 48 (4) <2>
F[26] 83CD8: 53, 50 (4) <2>
F[27] 83CD8: 55, 52 (4) <2>
F[28] 83CD8: 57, 54 (4) <2>
F[29] 83CD8: 59, 56 (4) <2>
F[30] 83CD8: 61, 58 (4) <2>
F[31] 83CD8: 63, 60 (4) <2>
F[32] 83CD8: 65, 62 (4) <2>
F[33] 83CD8: 67, 64 (4) <2>
F[34] 83CD8: 69, 66 (4) <2>
F[35] 83CD8: 71, 68 (4) <2>
F[36] 83CD8: 73, 70 (4) <2>
F[37] 83CD8: 75, 72 (4) <2>
F[38] 83CD8: 77, 74 (4) <2>
F[39] 83CD8: 79, 76 (4) <2>
F[40] 83CD8: 81, 78 (4) <2>
F[41] 83CD8: 83, 80 (4) <2>
F[42] 83CD8: 85, 82 (4) <2>
F[43] 0: 84 (1) <1>
F[44] index (i=44) non référencée
```

FIG. 3.6 Liste des enregistrements des séquences d'accès au tableau *F[i]*

À l'aide de cette analyse, on peut estimer la taille minimale de la mémoire pour effectuer la boucle. Pour notre exemple, la taille minimale est égale à 2, c'est-à-dire 64 bits, étant donné que l'on traite des entiers *M1* et *M2* représentés sur 32 bits.

t	*M1*	*M2*
0	F[1]	
1	F[3]	F[0]
2	F[5]	F[2]
3	F[7]	F[4]
4	F[9]	F[6]
5	F[11]	F[8]
6	F[13]	F[10]
7	F[15]	F[12]
...	...	...
80	F[81]	F[80]
81	F[83]	F[82]
82	F[85]	F[84]
83	F[87]	F[86]
84	F[89]	F[88]
85		F[90]

FIG. 3.7 Séquences d'accès au tableau *F[i]*

À ce stade, on est en mesure de déterminer quelle architecture offre une efficacité d'exécution optimale et d'estimer la quantité de mémoire requise pour effectuer la boucle analysée. Par la suite, il est essentiel de s'assurer que l'on possède un nombre suffisant de blocs logiques programmables (CLBs), sur le FPGA, pour implanter chacune des

instructions de ladite boucle. La section suivante présente la façon d'évaluer cette métrique.

3.2.3 La surface matérielle du FPGA

La surface matérielle requise par l'application peut être estimée à partir du nombre de CLBs requis. Cette métrique nous informe sur la structure logique (portes, bascules, unités d'interconnexion, etc.) devant être implantée sur le FPGA et, par le fait même, permet de déterminer si le bloc de base analysé peut être transposé en matériel. Elle tient compte de la longueur des mots de chaque opérande et du type de mise en œuvre de chaque opérateur. L'estimation de la surface de l'architecture matérielle est fondamentale et peut être faite de façon systématique.

Les tableaux 3.4 et 3.5 présentent le nombre de CLBs nécessaires pour mettre en œuvre chacun des opérateurs ainsi que les temps de réponse de chacun (additionneurs, soustracteurs, etc.). Ces valeurs ont été obtenues en utilisant, encore une fois, l'outil COREGEN de Xilinx [XIL99b].

Analysons maintenant la surface matérielle du FPGA nécessaire pour porter la boucle qui calcule la suite de Fibonacci. Les extrêmes ont été considérés afin d'estimer la surface minimale et maximale. On notera, qu'en général, plus le nombre de CLBs est élevé, plus la mise en œuvre est rapide. Le tableau 3.6 présente les valeurs obtenues en fonction des valeurs contenues dans le tableau 3.4.

TABLEAU 3.4 Nombre de CLB nécessaires pour mettre en œuvre les opérateurs additionneur et soustracteur en fonction des différentes configurations possibles sur le FPGA

Taille des opérandes (en bits)		**Additionneur / Soustracteur** (en CLBs)			
A	*B*	*BSA*	*RCA*	*LACA*	*CSA*
8	8	16	16	17	21
16	16	32	32	33	44
24	24	48	48	52	64
32	32	64	64	73	87

Dans le tableau 3.4, *A* et *B* sont les opérandes. Les termes *BSA*, *RCA*, *LACA* et *CSA* signifient respectivement "Bit-Serial Adder", "Ripple-Carry Adder", "Look-Ahead-Carry Adder" et "Conditional-Sum Adder".

TABLEAU 3.5 Nombre de CLB nécessaires pour mettre en œuvre l'opérateur multiplicateur en fonction des différentes configurations possibles sur le FPGA

Taille des opérandes (en bits)		**Multiplicateur** (en CLBs)		
A	*B*	*CCM*	*CCMP*	*PMAO*
8	8	19	21	54
16	16	75	83	213
24	24	163	197	489
32	32	285	363	606

Dans le tableau 3.5, *A* et *B* sont les opérandes. Les termes *CCM*, *CCMP* et *PMAO* signifient respectivement "Constant Coefficient Multiplier", "Constant Coefficient Multiplier Pipelined" et "Parallel Multiplier Area Optimized".

TABLEAU 3.6 Nombre minimum et maximum de CLBs pour mettre en œuvre la suite de Fibonacci sur le FPGA

Paramètre	Minimum	Maximum
Opérande	32 bits	32 bits
Opérateur	*BSA*	*CSA*
Nombre de CLBs	64	87

3.2.4 La bande passante

La bande passante représente le taux de transfert des données que l'on doit fournir à l'unité de calcul pour effectuer une tâche. Elle est caractérisée par l'ensemble des variables d'entrée et de sortie. La figure 3.8 illustre le modèle qui sera utilisé dans notre analyse.

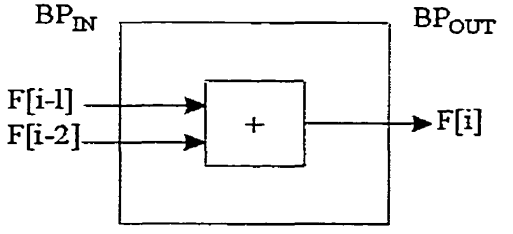


FIG. 3.8 Modèle de calcul de la bande passante

En se basant sur la suite de Fibonacci, on constate que la bande passante entrante (BP_{IN}) est égale à 2 et que la bande passante sortante (BP_{OUT}) est égale à 1. On définit donc la bande passante totale comme étant la somme de BP_{IN} et de BP_{OUT} . En effet, les entrées et les sorties utilisent souvent sur le même bus. On aura donc une bande passante totale égale à 3 étapes de contrôle.

3.3 La fonction paramétrique

À partir des valeurs de métrique trouvées, une fonction paramétrique est requise pour caractériser chacun des blocs de base. Il est important de normaliser les valeurs des

estimateurs de façon à pouvoir les comparer à l'intérieur d'une même fonction. Une pondération peut être ajoutée aux valeurs des métriques selon l'importance de l'aspect qui est évalué par celui-ci [GAJ94b, GAJ95]. Pour l'instant, les valeurs générées par la fonction paramétrique sont binaires (0 et 1). Cette fonction détermine donc si une partition considérée, respecte ou non les contraintes matérielles de l'architecture (c.-à-d. quantité de mémoire, quantité de CLBs et bande passante suffisantes) et, bien entendu, le temps d'exécution estimé sur le FPGA doit être inférieur au temps d'exécution estimé sur le Pentium. Ces valeurs sont obtenues en utilisant l'algorithme décrit au point 3.4. Éventuellement, cette fonction paramétrique pourrait être une représentation en quatre dimensions, où chacun des axes représenterait une métrique et la valeur générée par cette fonction correspondrait à la coordonnée vectorielle de ce point. La plage de chacun des axes varieraient entre 0 et 100, où 0 représenterait une borne inférieure de non-pertinence et 100 une borne supérieure de pertinence reliée à une éventuelle exécution matérielle.

3.4 L'algorithme de partitionnement

L'algorithme, présenté à la figure 3.9, permet de déterminer s'il est possible de transposer en matériel le bloc de base analysé en fonction des contraintes de l'architecture matérielle cible.

On sait qu'il est pertinent de transposer et d'exécuter en matériel un bloc de base analysé, lorsque l'ensemble des quatre conditions sont gagnantes. Dans d'autres cas, le bloc de base doit rester en logiciel et être exécuté sur le processeur considéré, qui dans le cas présent est le Pentium.

Même si cet algorithme n'a pas été mis en œuvre dans le cadre de ce projet, il en demeure pas moins intéressant de caractériser ses limitations. Tout d'abord, il est important de spécifier que sa granularité est au niveau des boucles. De plus, les boucles traitées sont des boucles parfaites, c'est-à-dire que toutes les boucles sont imbriquées et bornées. Prenons, à titre d'exemple, une application logicielle décrite par cinq boucles.

Dans un premier temps, chacune de ces boucles sont analysées individuellement en vue de déterminer la pertinence d'un éventuel partitionnement matériel/logiciel de chacune d'entre elles en matériel. Les boucles les plus prometteuses sont transposées et exécutées en matériel, les unes après les autres. Dans le cas où toutes les boucles sont indépendantes, cette approche ne cause aucun problème, on va les transposer tant qu'il y a de l'espace disponible sur l'architecture matérielle. Cependant, dans le cas où les boucles sont dépendantes les unes aux autres, il se peut qu'une boucle soit considérée non profitable en matériel à cause du coût inhérent aux communications lors de la transposition de cette dernière en matériel. Mais que la transposition simultanément d'une ou plusieurs boucles en matériel donne un scénario vraiment beaucoup plus profitable. Dans ce cas, les coûts de communication entre ces boucles ont été considérablement minimisés, rendant du fait même le partitionnement profitable. Il est donc fortement recommandé de considérer tous ces facteurs lors d'une éventuelle utilisation de cet algorithme. Soulignons, par le fait même, que si nous voudrions traiter des boucles non bornées, il faudrait changer le type d'analyse de performance au niveau logiciel en utilisant une approche comme celle présentée dans Cinderella [CIN00].

Jusqu'à présent, on n'a présenté que notre méthodologie de conception et de réalisation. Cependant, peu d'information sur l'utilisation de ce prototype sont abordées dans ce présent ouvrage, nous renvoyons donc le lecteur intéressé à consulter [THE99]. Un sommaire du guide d'utilisation de ce prototype est reproduit à l'annexe B.

Le chapitre suivant (chap. 4) présente une synthèse des différents tests qui ont été effectués afin de valider notre approche. Ces tests illustrent les différents facteurs pouvant être à l'origine d'un bon ou d'un mauvais partitionnement matériel/logiciel en fonction des contraintes imposées par le système. Une analyse mettant en lumière les forces et les faiblesses de notre modèle y sera aussi présentée.

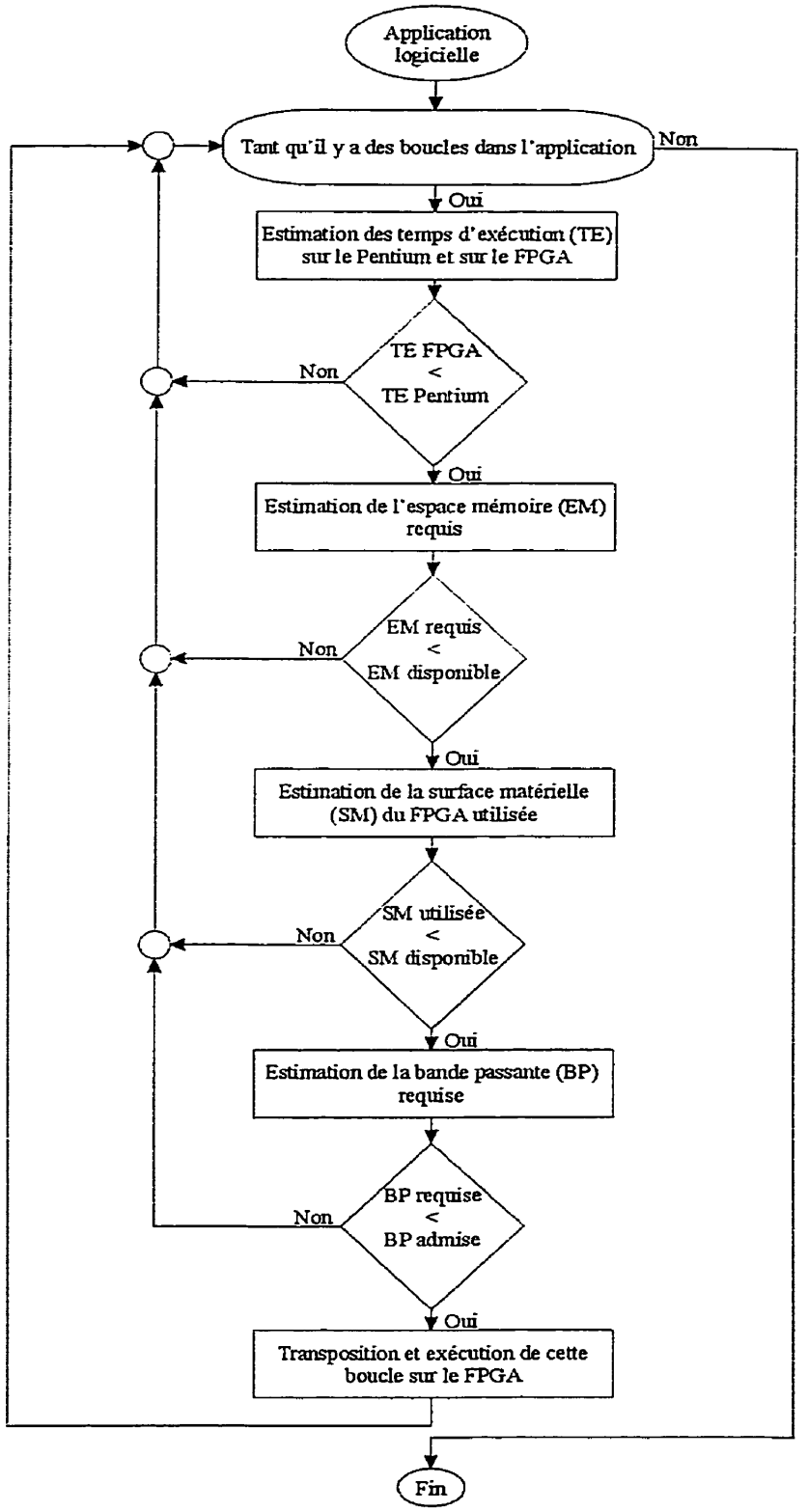


FIG. 3.9 Algorithme de partitionnement développé dans le cadre de cette étude

Chapitre 4

Les résultats et analyses

Cette section présente les divers résultats obtenus avec les différents programmes de test utilisés. Trois applications types ont été analysées : le calcul de la valeur du $N^{ième}$ terme de la suite de Fibonacci, le calcul du produit de deux matrices carrées de dimension N et le calcul de la convolution de deux vecteurs de dimension N.

4.1 Paramètres initiaux

Trois applications scientifiques ont fait l'objet d'analyses de performance permettant de valider l'approche suivie dans le cadre de ce mémoire de maîtrise. Notre approche repose essentiellement sur le développement de quatre métriques de performance. Ces métriques permettent d'analyser le caractère de chacun des programmes mettant en œuvre l'application proprement dite afin d'extraire les caractéristiques intrinsèques lors de leur exécution. Les premiers résultats seront obtenus en utilisant la suite de Fibonacci comme application de départ. Le calcul de celle-ci consiste en une sommation des n-1 et n-2 termes. Rappelons que la suite de Fibonacci peut s'écrire de la façon suivante :

$$y_n = \begin{cases} 0 & pour\ x = 0 \\ 1 & pour\ x = 1 \\ y_{n-1} + y_{n-2} & pour\ x = 2,\ 3,\ ...,\ N \end{cases} \qquad (4.1)$$

À partir de cette équation, on peut déduire les premiers termes de la suite de Fibonacci qui sont les suivants : 0, 1, 1, 2, 3, 5, 8, 13, 21, etc.

La multiplication de deux matrices carrées de dimension N sera la seconde application analysée. Pour multiplier deux matrices quelconques, on convient de multiplier successivement chaque ligne de la première matrice par les colonnes de la seconde : ce processus revient à une suite de multiplications d'une matrice ligne par une matrice colonne, cette multiplication est donnée par :

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{ki} \qquad \text{où } i = 1, 2, \ldots, N \text{ et } j = 1, 2, \ldots, N \tag{4.2}$$

Finalement, le dernier cas considéré dans cette étude est la convolution de deux vecteurs de dimension N. La convolution est donnée par l'équation suivante :

$$z(n) = \sum_{m=1}^{N} x(n)\, y(n-m) \qquad \text{où } n = 0, 1, \ldots, 2N\text{-}1 \tag{4.3}$$

Dans tous les cas, on essayera de déterminer la relation qui existe entre la variable indépendante N en fonction des quatre métriques et ainsi de déterminer la zone de fonctionnalité pertinente et susceptible d'être éventuellement transposée et exécutée sur l'architecture matérielle reconfigurable. Ce processus est appliqué à chacune des applications.

4.2 Résultats obtenus

Dans cette section, les équations de chacune des courbes ont été déterminées à l'aide de méthodes numériques classiques.

4.2.1 Temps d'exécution requis

Quelques valeurs du temps requis en fonction de N sont présentées au tableau 4.1. Dans ce tableau, les termes T_{EXE_PENT} et T_{EXE_FPGA} signifient respectivement le temps requis suite à une éventuelle exécution sur le Pentium et sur le FPGA. L'expression $G_{PENT/FPGA}$ exprime le gain observé, tel que défini à l'équation 3.17.

Les figures 4.1 à 4.3 illustrent graphiquement la variation du temps requis appelée T, sur le Pentium et sur le FPGA en fonction de la variable indépendante N pour chacun des cas analysés.

La figure 4.1 présente cette variation en fonction du $N^{ième}$ terme de la suite de Fibonacci. Il est intéressant de souligner que la variation de T en fonction de N a un comportement linéaire lorsque N est supérieur à 1000, et ce, tant pour le Pentium que pour le FPGA.

On constate que lorsque que le $N^{ième}$ terme est petit, le gain $G_{PENT/FPGA}$ est inférieur à 1. Ceci signifie qu'il est plus rapide d'exécuter ce segment de code sur le Pentium. D'après la figure 4.1, on observe un gain qui tend vers 1.1 lorsque le $N^{ième}$ terme est supérieur à 193, cette valeur correspond au point d'intersection entre les deux courbes. À ce moment, on observera une accélération reliée à une éventuelle exécution sur le FPGA. Le gain observé dans ce cas est minime.

La figure 4.2 présente la variation du temps requis (T) en fonction de la dimension (N) des deux matrices carrées multipliées entre elles. La courbe de T en fonction de N peut être représentée par un polynôme d'ordre 3. Encore une fois, on observe que le Pentium et le FPGA suivent un comportement similaire. Dans ce cas, il est toujours plus rentable de transposer et d'exécuter le segment de code de l'application sur le FPGA. D'après le tableau 4.1, on observe que le gain est considérable et varie entre 4.34 et 4.66. Il est possible de démontrer que le gain tend vers 4 lorsque N est grand.

TABLEAU 4.1 Valeurs relatives à la métrique du temps d'exécution requis pour effectuer chacun des cas sur le Pentium et sur le FPGA en fonction de la valeur de N

Application	N	T_{EXE_PENT} (µs)	T_{EXE_FPGA} (µs)	$G_{PENT/FPGA}$
$N^{ième}$ terme de la suite de Fibonacci	25	1.33	1.62	0.82
	50	2.79	3.37	0.83
	100	5.69	6.87	0.83
	1000	75.50	69.87	1.08
	5000	388.82	349.87	1.11
	10000	777,81	699.87	1.11
Produit de deux matrices carrées de dimension N	5	41.00	8.79	4.66
	10	313.86	70.04	4.48
	15	1038.79	236.29	4.40
	20	2452.39	560.04	4.38
	30	8260.97	1890.03	4.37
	50	38000.76	8750.03	4.34
Convolution de deux vecteurs de dimension N	50	0.56	0.35	1.6
	100	2.22	1.40	1.59
	200	8.84	5.60	1.58
	500	55.09	35.00	1.57
	1000	220.78	140.00	1.58
	2000	891.15	560.03	1.59

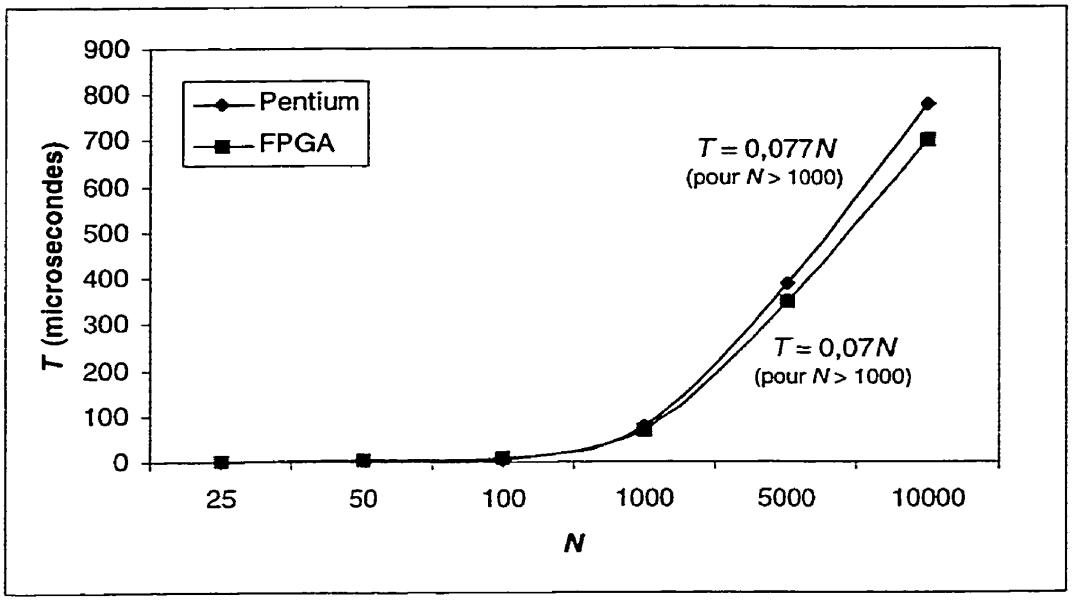


FIG. 4.1 Variation de T en fonction du $N^{ième}$ terme

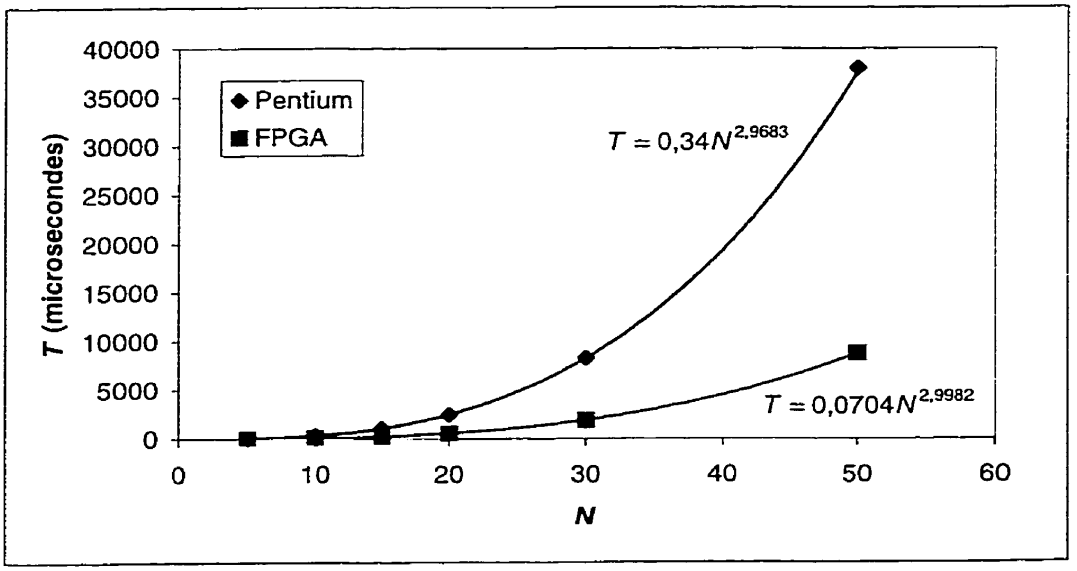


FIG. 4.2 Variation de T en fonction de la dimension N des deux matrices

Finalement, la figure 4.3 montre la variation du temps requis (T) en fonction de la dimension (N) des deux vecteurs convolués. La variation de T en fonction de N est représentée par un polynôme d'ordre 2. Encore une fois, on observe que le Pentium et le FPGA suivent un comportement similaire. Il est encore toujours plus rentable de

transposer et d'exécuter le segment de code de l'application sur le FPGA. D'après le tableau 4.1, on observe que le gain oscille entre 1.57 et 1.6.

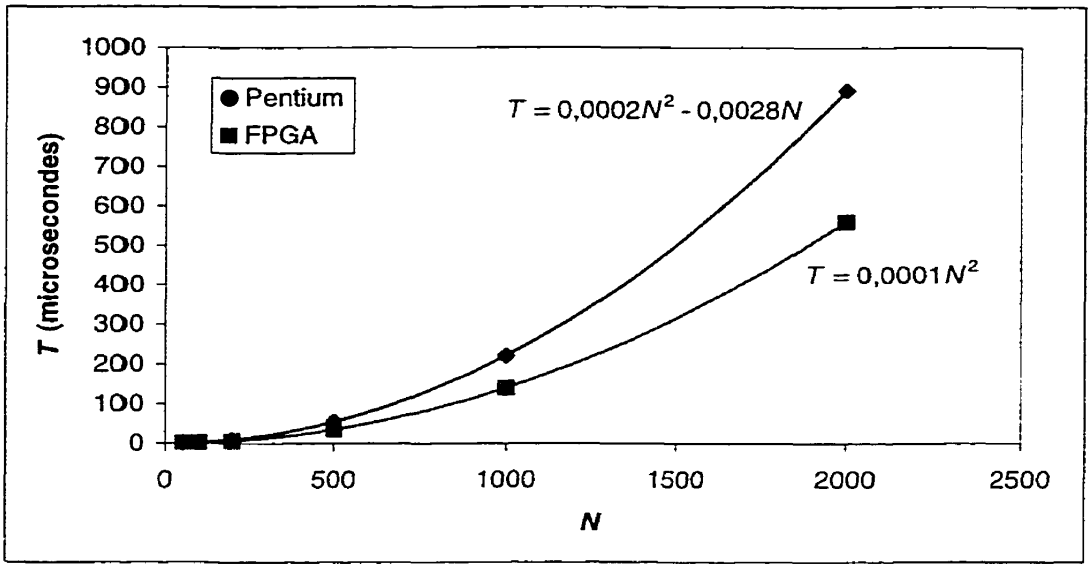


FIG. 4.3 Variation de T en fonction de la dimension N des deux vecteurs

4.2.2 Espace mémoire requis

Le tableau 4.2 présente la quantité de mémoire requise en fonction de N. Dans ce tableau 4.2, QT_M signifie la quantité de mémoire requise pour effectuer tous les passages dans la boucle.

Les figures 4.4 à 4.6 illustrent graphiquement l'espace mémoire requis (E) sur le FPGA en fonction de la variable indépendante N pour chacun des cas analysés.

Notons que le FPGA considéré possède ***8 KB*** (65536 bits) de mémoire disponible. Donc, dès que la valeur de QT_M dépasse ce seuil, un partitionnement peut ne plus être pertinent, car il faudrait utiliser une mémoire auxiliaire externe. Ceci implique un coût additionnel en temps de communication, ce qui peut diminuer de façon importante la performance globale.

TABLEAU 4.2 Valeurs relatives à la métrique de l'espace mémoire requis pour réaliser chacun des cas sur le FPGA en fonction de la valeur de *N*

Application	*N*	*QT_M* (bits)
$N^{ième}$ terme de la suite de Fibonacci	25	64
	50	64
	100	64
	1000	64
	5000	64
	10000	64
Produit de deux matrices carrées de dimension *N*	5	992
	10	3552
	15	7712
	20	13472
	30	29792
	50	81632
Convolution de deux vecteurs de dimension *N*	50	3200
	100	6400
	200	12800
	500	32000
	1000	64000
	2000	128000

D'après la figure 4.4, on observe que l'espace mémoire requis (*E*) pour réaliser le calcul du $N^{ième}$ terme de la suite Fibonacci est fixe et égal à 64 bits. Il est facile de montrer que l'espace mémoire requis est égal à 64 bits car, pour calculer n'importe quel

terme, on a uniquement besoin des deux termes précédents, tel que défini à l'équation 4.2. Comme on vient de le voir, cette application n'est nullement limitée par la quantité de mémoire requise et ce, indépendamment de la valeur de *N*.

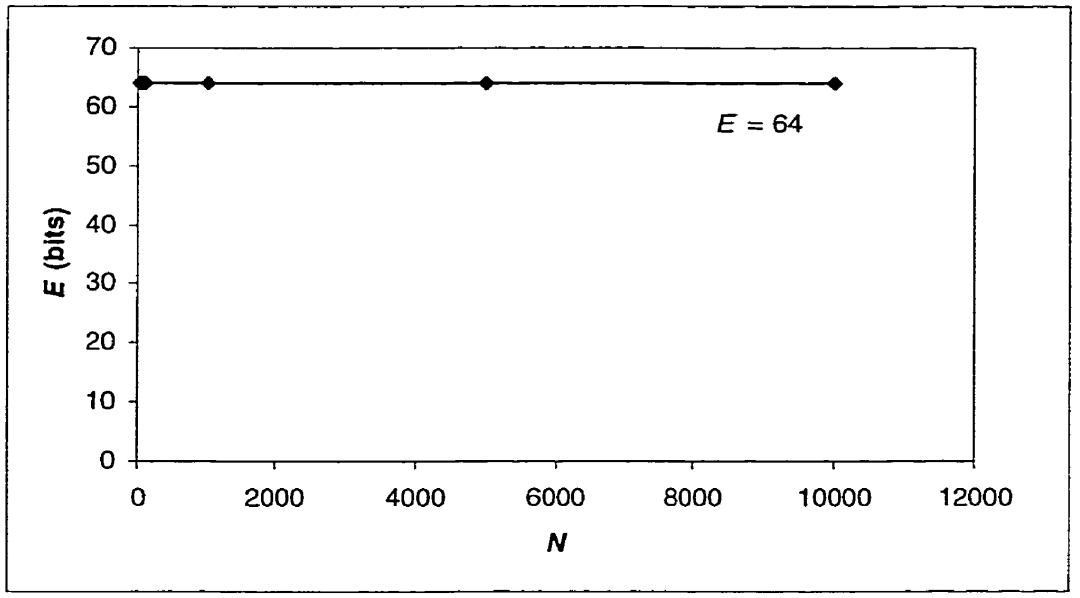


FIG. 4.4 Variation de *E* en fonction du $N^{ième}$ terme

La figure 4.5 représente la variation de l'espace mémoire requis (*E*) mais cette fois-ci en fonction de la dimension (*N*) des deux matrices carrées multipliées entre elles. La variation de *E* en fonction de *N* est représentée par un polynôme d'ordre 2. Dans ce cas, dès que *N* est supérieur à 44, on observe que la quantité de mémoire requise est supérieure à la quantité de mémoire disponible sur le FPGA, rendant du fait même l'éventuel partitionnement moins profitable.

Finalement, la figure 4.6 représente la variation de l'espace mémoire requis (*E*) en fonction de la dimension (*N*) des deux vecteurs convolués. La variation de *E* en fonction de *N* est représentée par une droite de pente égale à 64. De ce fait, on observe que lorsque *N* est supérieur à 1024, la quantité de mémoire requise est supérieure à la quantité de mémoire disponible sur le FPGA, rendant encore une fois un éventuel partitionnement moins profitable.

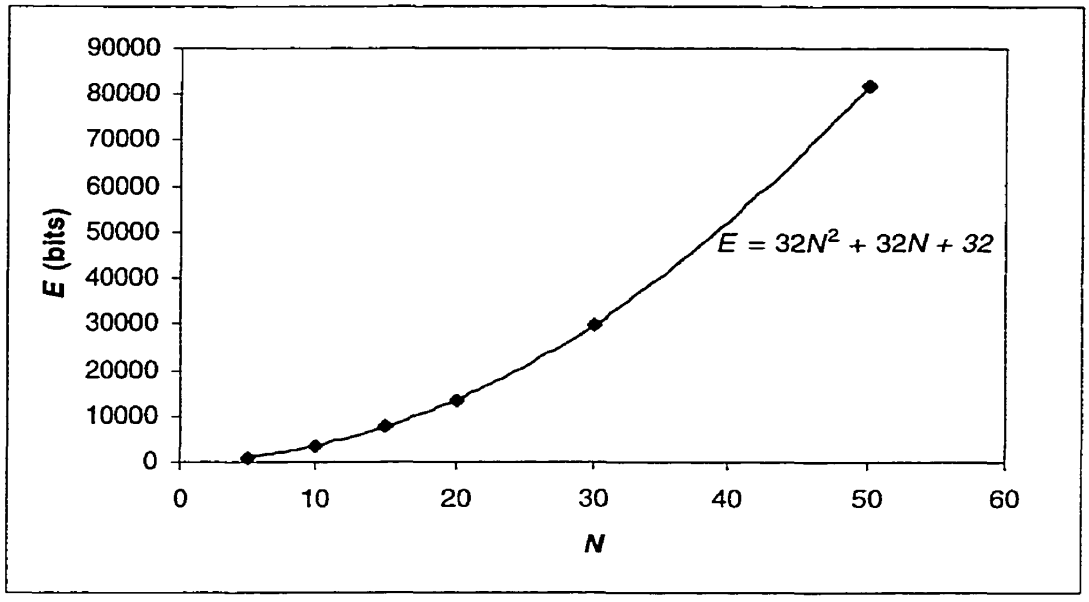


FIG. 4.5 Variation de E en fonction de la dimension N des deux matrices

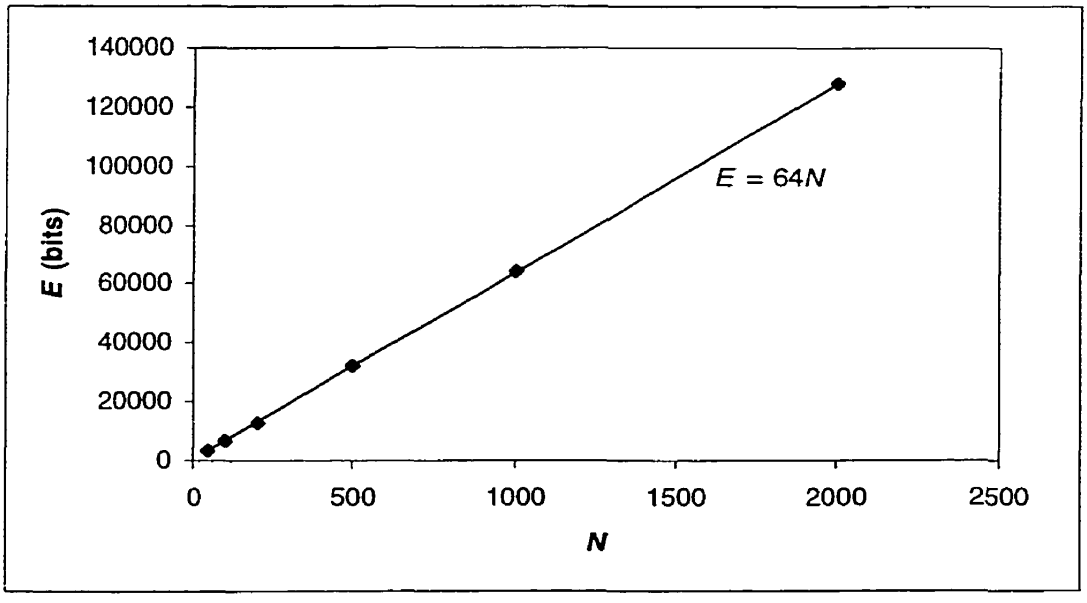


FIG. 4.6 Variation de E en fonction de la dimension N des deux vecteurs

4.2.3 Surface matérielle utilisée

Le tableau 4.3 présente la surface matérielle requise pour une application en fonction de N. Dans ce tableau, *NB_CLB* signifie le nombre de CLBs requis pour mettre en œuvre

la boucle sur le FPGA. La surface matérielle utilisée (S) en fonction de la variable indépendante N est illustrée graphiquement aux figures 4.7 à 4.9.

TABLEAU 4.3 Valeurs relatives à la métrique du nombre de CLBs requis pour mettre en œuvre les fonctionnalités de chacun des cas sur le FPGA

Application	N	*NB_CLB* minimum	*NB_CLB* maximum
$N^{ième}$ terme de la suite de Fibonacci	25	64	87
	50	64	87
	100	64	87
	1000	64	87
	5000	64	87
	10000	64	87
Produit de deux matrices carrées de dimension N	5	349	693
	10	349	693
	15	349	693
	20	349	693
	30	349	693
	50	349	693
Convolution de deux vecteurs de dimension N	50	349	693
	100	349	693
	200	349	693
	500	349	693
	1000	349	693
	2000	349	693

Il est important de noter que le FPGA considéré dans cette étude comporte ***6912*** blocs logiques programmables (CLBs). Ainsi, dès que la valeur de *NB_CLB* minimum dépasse ce seuil alors un partitionnement matériel/logiciel devient difficile car il faudrait utiliser plusieurs circuits.

Soulignons que chacun des temps d'exécution requis sur le FPGA, tel que rapportés au tableau 4.1, sont obtenus en ciblant une mise en œuvre nécessitant le nombre minimum et maximum de CLBs. La seule raison pour laquelle nous avons présenté le nombre maximal de CLBs pour mettre en œuvre lesdites applications est uniquement dans le but de prévoir le pire cas.

D'après la figure 4.7, on observe que la surface matérielle utilisée (S) pour mettre en œuvre la fonction réalisant le calcul du $N^{ième}$ terme de la suite Fibonacci est au minimum égal à 64 CLBs et au maximum égal à 87 CLBs. On a donc un nombre de CLB disponibles largement suffisant sur le FPGA pour mettre en œuvre cette application et comme ce nombre est indépendant de N, les contraintes matérielles n'induisent donc pas de restriction en fonction de la valeur de N.

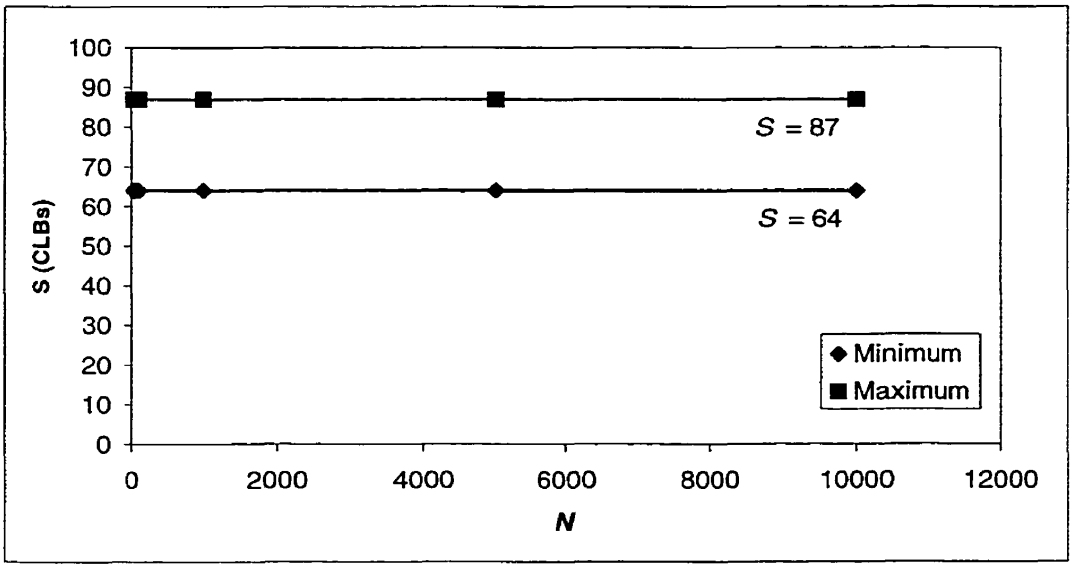


FIG. 4.7 Variation de S en fonction du $N^{ième}$ terme

La figure 4.8 présente la surface matérielle utilisée (S) pour mettre en œuvre la fonction réalisant le calcul du produit de deux matrices carrées de dimension N.

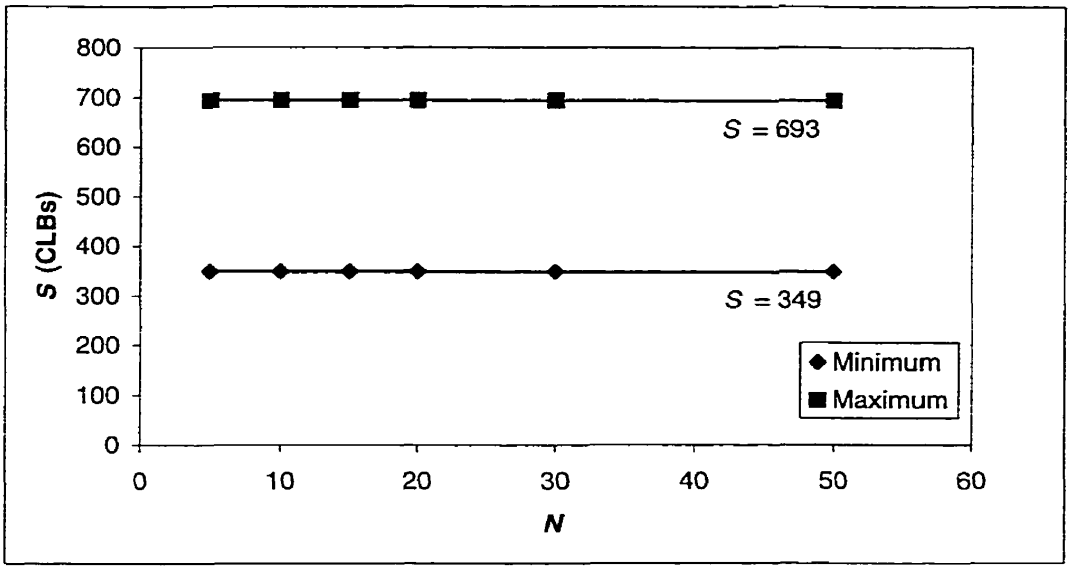


FIG. 4.8 Variation de S en fonction de la dimension N des deux matrices

Le nombre de CLB requis est fixe et égal au minimum à 349 et au maximum à 693. Ce nombre est indépendant de N et il y a suffisamment de CLBs disponibles sur le FPGA pour mettre en œuvre cette application.

À la figure 4.9, on observe que le nombre de CLB requis pour mettre en œuvre le calcul d'une convolution de deux vecteurs et le calcul du produit de deux matrices est identique. En effet, dans les deux cas, on utilise les deux mêmes opérateurs arithmétiques, soient l'additionneur et le multiplicateur.

Comme on vient de le voir, cette métrique évalue la surface nécessaire pour mettre en œuvre chacune des applications. Soulignons que plus le nombre d'opérateurs sera grand pour mettre en œuvre une application donnée, plus le nombre de CLB nécessaires sera grand.

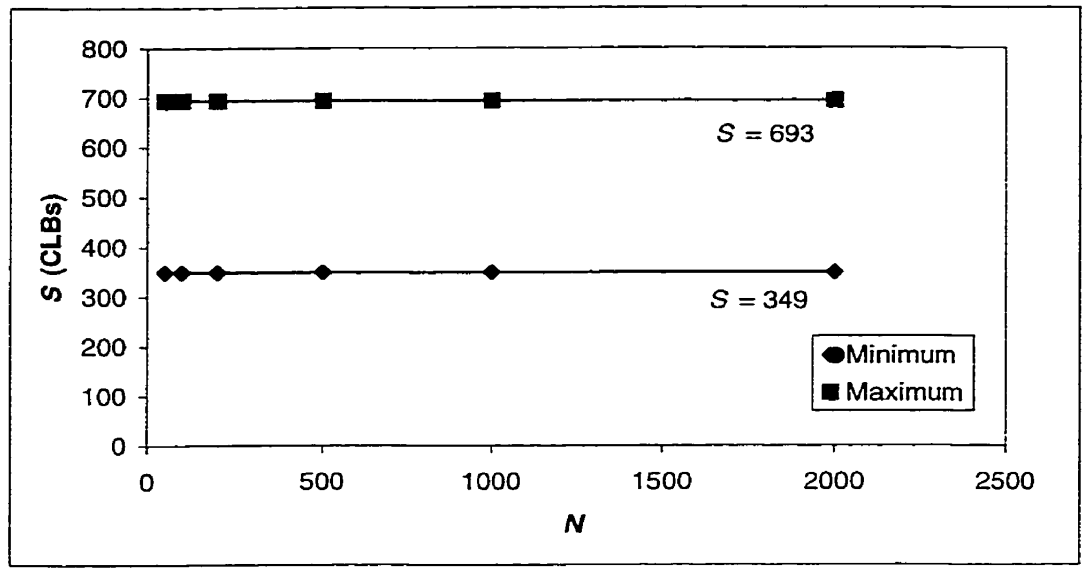


FIG. 4.9 Variation de S en fonction de la dimension N des deux vecteurs

4.2.4 Bande passante

Le tableau 4.4 présente des valeurs de la bande passante requise pour le transfert des données nécessaires afin d'effectuer chacune des applications sur le FPGA et ce, en fonction de la valeur de N. Dans ce tableau, BP représente la bande passante et est exprimée en étape de contrôle.

Rappelons que, d'après notre modèle de calcul de la bande passante (fig. 3.8), on force celle-ci à être égale à 3 étapes de contrôle. C'est ce qui explique que la bande passante entrante, sortante et totale, pour chacune des applications présentées au tableau 4.4, est fixe et égale à 2, 1 et 3 étapes de contrôle respectivement.

4.2.5 Partitionnement matériel/logiciel

Le tableau 4.5 identifie, en fonction des valeurs de métriques de temps d'exécution requis, d'espace mémoire requis, de bande passante requise et de surface matérielle utilisée, les applications les plus pertinentes à transposer et à exécuter sur le FPGA en fonction de la variable N. Dans ce tableau, *Ind_P* signifie indicateur de pertinence relié à

une éventuelle exécution matérielle sur le FPGA. Dans le champ *Remarque*, on retrouve le constat affectant l'état dudit indicateur de pertinence.

TABLEAU 4.4 Valeurs relatives à la métrique de bande passante requise pour le transfert des données

Application	***N***	***BP*** entrante	***BP*** sortante	***BP*** totale
$N^{ième}$ terme de la suite de Fibonacci	25	2	1	3
	50	2	1	3
	100	2	1	3
	1000	2	1	3
	5000	2	1	3
	10000	2	1	3
Produit de deux matrices carrées de dimension *N*	5	2	1	3
	10	2	1	3
	15	2	1	3
	20	2	1	3
	30	2	1	3
	50	2	1	3
Convolution de deux vecteurs de dimension *N*	50	2	1	3
	100	2	1	3
	200	2	1	3
	500	2	1	3
	1000	2	1	3
	2000	2	1	3

TABLEAU 4.5 Indicateur de pertinence relatif à chacun des cas analysés en fonction de la valeur de *N*

Application	*N*	*Ind_P*	Remarque
$N^{ième}$ terme de la suite de Fibonacci	25	NON	$T_{EXE\_FPGA} > T_{EXE\_PENT}$
	50	NON	$T_{EXE\_FPGA} > T_{EXE\_PENT}$
	100	NON	$T_{EXE\_FPGA} > T_{EXE\_PENT}$
	1000	OUI	Partitionnement M/L
	5000	OUI	Partitionnement M/L
	10000	OUI	Partitionnement M/L
Produit de deux matrices carrées de dimension *N*	5	OUI	Partitionnement M/L
	10	OUI	Partitionnement M/L
	15	OUI	Partitionnement M/L
	20	OUI	Partitionnement M/L
	30	OUI	Partitionnement M/L
	50	NON	Mémoire insuffisante
Convolution de deux vecteurs de dimension *N*	50	OUI	Partitionnement M/L
	100	OUI	Partitionnement M/L
	200	OUI	Partitionnement M/L
	500	OUI	Partitionnement M/L
	1000	OUI	Partitionnement M/L
	2000	NON	Mémoire insuffisante

La valeur de l'indicateur de pertinence découle directement des valeurs de chacune des métriques qui ont été insérées à l'intérieur de la fonction paramétrique. La mise en œuvre de cette fonction repose sur l'algorithme de partitionnement, tel que décrit à la section 3.4. Si le résultat de l'indicateur de pertinence est OUI, alors il y aura

transposition et exécution de l'application sur le FPGA, accentuant ainsi sa vitesse d'exécution. Dans le cas d'un NON, l'application restera en logiciel et sera exécutée sur le Pentium.

Analysons chacun des cas du tableau afin de bien comprendre le cheminement suivi affectant l'indicateur de pertinence. Tout d'abord, dès que le $N^{ième}$ terme est inférieur à ***1293*** dans le calcul de la suite de Fibonacci, on a démontré qu'il n'était pas profitable d'effectuer un partitionnement matériel/logiciel car le temps requis sur le FPGA pour exécuter cette tâche est supérieur au temps requis sur le Pentium pour exécuter cette même tâche.

On a démontré que pour calculer le produit de deux matrices carrées, leur dimension N ne doit pas être supérieure à ***44***. Sinon, la quantité de mémoire requise est supérieure à la quantité de mémoire disponible sur le FPGA, rendant du fait même, le partitionnement moins profitable car il faudrait transférer des données via une mémoire auxiliaire externe. Ceci impliquerait un coût additionnel en temps de communication, ce qui peut diminuer de façon importante la performance globale.

Pour terminer, on a montré que lors du calcul de la convolution, les deux vecteurs doivent avoir une dimension inférieure ou égale à ***1024***, car le même problème que précédemment survient, c'est-à-dire qu'il n'y aura pas suffisamment de mémoire disponible sur le FPGA pour effectuer tous les passages dans la boucle.

4.3 Discussion

Nous avons seulement présenté quelques cas dans le cadre de cette étude, mais une multitude d'autres pourraient y être analysés. Comme on vient de le voir, plus il y a de calculs à effectuer sur les données, plus il est pertinent d'exécuter ces applications en matériel. Prenons, le cas du produit de deux matrices carrées de dimension N. C'est sans aucun doute celui qui offre la meilleure accélération sur le FPGA. On constate que ce type de traitement exige beaucoup de calculs afin d'obtenir tous les éléments de la

matrice résultante. Ce type d'application est donc très intéressant pour une exécution sur le FPGA. À l'autre extrémité du spectre, la suite de Fibonacci effectue une simple sommation de deux termes. Ce genre d'application n'offre pas un bon gain de performance. Dans ce dernier cas, pour obtenir une meilleure accélération sur le FPGA, on pourrait en faire une reformulation parallèle de façon à exploiter le parallélisme inhérent à la boucle de calcul.

Mentionnons, en terminant, que plusieurs modules de notre outil sont automatisés. Le module d'estimation de la performance est enclenché dès le lancement de la phase de la compilation de l'application. Aucune intervention humaine est nécessaire. De plus, la partie partitionnement matériel/logiciel de chacun des blocs est présentement semi-automatique. En effet, l'émission du code, pour la partie logicielle, est réalisée directement par une primitive disponible avec le compilateur SUIF. Or, pour l'émission du code, pour la partie matérielle, un module traducteur du C ANSI au VHDL devra être mis en œuvre.

Conclusion

Dans ce mémoire, nous avons présenté différentes métriques permettant d'évaluer la pertinence de transposer et d'exécuter, sur un co-processeur FPGA, les segments de code les plus prometteurs accélérant ainsi la vitesse d'exécution d'une tâche. L'outil, tel que présenté, démontre bien que le compilateur SUIF est un environnement adéquat pour l'estimation de métriques de performance en vue d'automatiser le processus de partitionnement matériel/logiciel.

L'ensemble des résultats obtenus nous permet de conclure que certaines boucles d'une application donnée, contenant peu de traitements (c.-à-d. peu d'opérations), sont rarement pertinentes à transposer et à exécuter sur un FPGA. En effet, le temps de communication, pour acheminer les données au FPGA, devient très important comparativement au temps de traitement de ces boucles sur ce dernier. De plus, si on additionne au temps de communication, le temps de latence et de programmation du FPGA, alors le temps total d'exécution requis devient plus long que celui requis pour exécuter directement ces boucles sur un Pentium.

Un autre phénomène, celui-ci tout à fait à l'opposé, apparaît lorsqu'on traite des boucles qui ont beaucoup d'opérations à effectuer. On observe très souvent un fort gain de performance pour chacune de ces boucles. Mais, plus la boucle est grosse, plus celle-ci demande d'espace mémoire pour stocker ses données temporaires. Et comme la quantité

de mémoire est limitée sur un FPGA, il se peut que la mémoire disponible soit insuffisante et que le partitionnement soit moins profitable.

Finalement, un autre piège potentiel avec ce type de boucles, c'est qu'elles demandent un grand nombre de blocs logiques pour leur mise en œuvre. On peut se retrouver dans le cas où le nombre de CLB disponibles sur un FPGA est inférieur au nombre requis pour leurs mise en œuvre. Ceci rend une fois de plus le partitionnement difficile. Il est important de noter que l'hypothèse simplificatrice de départ consistait à utiliser qu'un seul opérateur (additionneur, soustracteur ou multiplicateur) sur le FPGA. Or, on observe qu'au maximum 10%, soit $^{693}/_{6912}$, de la surface matérielle (en CLB) est occupée pour l'ensemble des applications considérées dans cette étude. Cette faible proportion explique les faibles gains de performance observés au tableau 4.1.

En résumé, nous avons démontré que la pertinence d'un éventuel partitionnement matériel/logiciel repose sur plusieurs facteurs déterminant la qualité du partitionnement. Actuellement, nous sommes en mesure de quantifier l'impact qu'aura un éventuel partitionnement matériel/logiciel et de prédire les limites ou les contraintes physiques que celui-ci impose. Les diverses applications décrites dans cette étude présentent un éventail de contraintes que le système hybride, tel que définit précédemment, doit considérer. Les résultats obtenus jusqu'à maintenant démontrent que la détection et la caractérisation de chacun des blocs de base, d'une application logicielle, peuvent être fait à l'aide des quatre métriques de performance développées dans le cadre de ce projet.

Toutefois, un raffinement intéressant pourrait être apporté sur les divers modèles utilisés définissant les spécifications fonctionnelles à la base de l'outil, afin de le rendre plus robuste et ainsi obtenir des partitionnements matériel/logiciel optimaux. De toute évidence, le modèle utilisé à la section 3.2.4 pour le calcul de la bande passante pourrait être étendu afin de considérer la plupart des cas les plus fréquemment rencontrés dans les programmes scientifiques, garantissant du fait même de meilleures performances. Par exemple, ce modèle pourrait tenir compte, en plus des paramètres principaux liés à

l'application et à l'architecture cible, de certaines caractéristiques du processeur, telles que le nombre d'étages du pipeline. De plus, le modèle pourrait considérer le cas où trois données ou plus peuvent être acheminées au dispositif de calcul simultanément et en exploitant le parallélisme dans le segment de code analysé. Cette modification engendrera directement un gain important relié au temps de communication lors de l'envoi des données et au temps requis pour le calcul proprement dit. À cet effet, on disposerait non plus d'un seul opérateur, mais d'une banque d'opérateurs pouvant être ordonnancés de manière à représenter la structure logique du segment.

Dans le même sens, au lieu de n'être qu'une fonction binaire (0 ou 1), la fonction paramétrique pourrait être représentée par un espace vectoriel à quatre dimensions ($\Re^4$) où chacun des axes correspondrait à une métrique et la valeur générée serait la coordonnée vectorielle de ce vecteur. Ce concept permettrait d'avoir une meilleure représentation de l'ensemble des valeurs de métriques et, par le fait même, un moyen rapide de repérer les "boucles chaudes" les plus prometteuses en analysant le lieu des coordonnées.

Pour terminer, l'algorithme de partitionnement matériel/logiciel développé dans ce mémoire peut être approfondi et élargi afin d'atteindre un niveau supérieur de généralité permettant d'étendre le domaine des fonctionnalités des applications. On peut envisager d'utiliser conjointement des méthodes complémentaires comme le "software pipelining", le déroulage des boucles, le ré-ordonnancement du code, les circuits programmables additionnels, etc.

Références

[ADA96] ADAMS, J. K. et THOMAS, D. E. (1996). The Design of Mixed Hardware/Software Systems, Proceedings of the 33rd annual conference on Design Automation Conference (DAC-96), pp. 515-520.

[AHO86] AHO, A. V., SETHI, R. et ULLMAN, J. D. (1986). Compilers, Principles, Techniques and Tools, Addison-Wesley Publishing Company.

[ARE00] AREXSYS home page, http://www.arexsys.com.

[ARN00] ARNOUT, G. (2000). SystemC Standard, Asia-Pacific DAC, http://www.coware.com/conferencePapers.html

[ASH96] ASHENDEN, P. J. (1996). The Designer's Guide to VHDL, Morgan Kaufmann, San Fransco.

[ATH91] ATHANAS, P. M. (1991). An Adaptative Hardware Machine Architecture and Compiler for Dynamic Processor Reconfigurable, PhD thesis, Division of Engineering, Brown University, Providence, Rhode Island, USA.

[AUM96] AUMIAUX, M. (1996). Initiation au langage VHDL, Masson, Paris, 154 p.

[AXE97] AXELSSON, J. (1997). Analysis and Synthesis of Heterogeneous Real-Time Systems, Linkopoing Studies in Science and Technology, Suède, 190 p.

[BAL92] BALL, T. et LARUS, J. R. (1992). Optimally profiling and tracing programs, In Proceedings of the ACM SIGPLAN 92 Conference on Principles of Programming Languages, pp. 59-70.

[BAL94] BALL, T. et LARUS, J. R. (1994). Optimally profiling and tracing programs, ACM Transactions on Programming Languages and Systems. Vol. 16, No. 4, pp. 1319-1360.

[BAL96] BALL, T. et LARUS, J. R. (1996). Efficient Path Profiling, IEEE/ACM 29th International Symposium on Microarchitecture, Paris, France, pp. 45-57.

[BAR93] BARROS DA SILVA, E. (1993). Hardware/Software Partitioning using UNITY, PhD thesis, Tübingen.

[BER92] BERTIN, P., RONCIN, D. et VUILLEMIN, J. (1992). Programmable Active Memories : Performance Measurement, Proceedings of the First International Workshop on Field Programmable Gate Arrays, ACM SIGDA, pp. 57-59.

[BOL87] BOLOGNESI, T. et BRINKSMA, E. (1987). Introduction to the ISO Specification Language LOTOS, In Computer Networks and ISDN Systems 14, pp. 25-59.

[BOL97] BOLSENS, I., DE MAN, H. J., LIN, B., VAN ROMPAEY, K., VERCAUTEREN, S. et VERKEST, D. (1997). Hardware/Software Co-Design of Digital Telecommunication Systems, Proceedings of the IEEE - Special issue on Hardware/Software Co-design, Vol. 85, No. 3.

[BRA00] BRANT, K. (2000). Cycle Accurate C Modeling, C Level Design Inc., http://www.cleveldesign.com/press/index.html.

[BUC93] BUCHENRIEDER, K., SEDLMEIER, A. et VEITH, C. (1993). HW/SW Co-Design with PRAMs Using CODES, In Proceedings of the International Conference on Computer Hardware Description Languages (CHDL-93), Ottawa, Canada.

[CAR96] CARRERAS, C., LOPEZ, J. C., LOPEZ, M. L., DELGADO-KLOOS, C., MARTINEZ, N. et SANCHEZ, L. (1996). A Co-Design Methodology Based on Formal Specification and High-level Estimation, International Workshop on Hardware/Software Co-Design, Vol. 4, pp. 28-35.

[CHI93] CHIODO, M., GIUSTO, P., JURECSKA, A., LAVAGNO, L., HSIEH, H. et SANGIOVANNI-VINCENTELLI, A. (1993). A Formal Specification Model for Hardware/Software Codesign, In Handouts of the International Workshop on Hardware-Software Co-Design, Cambridge, Mass.

[CIN00] CINDERELLA home page, http://www.ee.princeton.edu/~yauli/cinderella.

[CLE00] C LEVEL DESIGN home page, http://www.cleveldesign.com/press/index.html.

[COM90] COMPOSANO, R. (1990). From Behavior to Structure : High-level Synthesis, IEEE Design & Test of Computers, pp. 8-19.

[CON96] CONTE, T. M., MENEZES, K. N. et HIRSCH, M. A. (1996). Accurate and Practice Profile-Driven Compilation Using the Profile Buffer, Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture, pp. 36-45.

[COW00] COWARE home page, http://www.coware.com.

[DAM94] D'AMBROSIO, J. G. et HU, X. (1994). Configuration-Level Hardware/Software Partitioning for Real-Time Embedded Systems, In Proc. of Int. Workshop on Hardware/Software Codesign, pp. 34-41.

[DEL95] DELGADO-KLOOS, C., MARIN, A., DE MIGUEL, T. et ROBLES, T. (1995). From LOTOS to VHDL, In Current Issues in Electronic Modeling, Kluwer Academic Publishers.

[DEM94] DE MICHELI, G. (1994). Synthesis and Optimisation of Digital Circuits, McGraw-Hill.

[DOU98] DOUCET, F. (1998). Définition de l'architecture générale d'un compilateur pour la synthèse de spécification de haut niveau et réalisation d'un module de génération de VHDL RTL, Rapport de projet de fin d'études, École Polytechnique de Montréal.

[DOU99] DOUCET, F. (1999). Rapport final de projet, Cours ELE6817 - Ré-ingénierie du logiciel, École Polytechnique de Montréal.

[DUT97] DUTRIEUX, L. et DEMIGNY, D. (1997). Logique programmable : architecture des FPGA et des CPLD, méthodes de conception, le langage VHDL, Eyrolles, Paris, 234 p.

[EDW93] EDWARDS, M. (1993). A Development System for Hardware/Software Co-Synthesis using FPGAs, Second IFIP International Workshop on Hardware-Software Co-Design.

[EDW94] EDWARDS, M. et FORREST, J. (1994). A Development for the Cosynthesis of Embedded Software/Hardware Systems, Department of Computation, UMIST, Los Alamitos, CA IEEE Computer Society Press.

[EDW97] EDWARDS, S., LAVAGNO, L., LEE, E. A. et SANGIOVANNI VINCENTELLI, A. (1997). Design of Embedded Systems : Formal Models, Validation and Synthesis, Proceedings of the IEEE - Special issue on Hardware/Software Co-design, Vol. 85, No. 3.

[EET00] EETIMES home page, http://www.eetimes.com.

[ELE94] ELES, P., PENG, Z. et DOBOLI, A. (1994). VHDL System-Level Specification and Partitioning in a Hardware/Software Co-Synthesis Environment, Proceedings of the Third International Workshop on Hardware/Software Co-Design, pp. 49-55.

[ERN92] ERNST, R. and HENKEL, J. (1992). Hardware-Software Codesign of Embedded Controllers Based on Hardware Extraction, Proceedings of the International Workshop on Hardware-Software Co-Design.

[ERN98] ERNST, R. (1998). Codesign of Embedded Systems : Status and Trends, IEEE Design & Test of Computers, pp. 45-54.

[FER87] FERRANTE, J., OTTENSTEIN, K. J. et WARREN, J. D. (1987). The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems, pp. 319-349.

[FRA95] FRASER, C. W. et HANSON, D. R. (1995). A Retargetable C Compiler : Design and Implementation, Addison-Wesley Publishing Company.

[GAJ94a] GAJSKI, D., VAHID, F. et NARAYAN, S. (1994). A Design Methodology for System Specification Refinement, In Proceedings of the Euro Design Automation Conference (EDAC-94), Paris, France.

[GAJ94b] GAJSKI, D. D., VAHID, F., NARAYAN, S. et Gong, J. (1994). Specifications and Design of Embedded Systems, PTR Prentice Hall, Englewood Cliffs, New Jersey 07632.

[GAJ94c] GAJSKI, D. D., VAHID, F., NARAYAN, S. et GONG, J. (1994). Specifications and design of Embedded Systems, PTR Prentice Hall, New Jersey, 450 p.

[GAJ95] GAJSKI, D. D. et VAHID, F. (1995). Specifications and Design of Embedded Hardware-Software Systems, IEEE Design & Test of Computers, VOL. 12, No. 1, pp. 53-67.

[GAJ00] GAJSKI, D. and ALLEN, R. (2000). The case for C/C++ hardware design, EEDesign, http://www.eedesign.com.

[GEN96] GENGLER, M., UBÉDA, S. et DESPREZ, F. (1996). Initiation au parallélisme, Masson, Paris.

[GON94] GONG, J., GAJSKI, D. D. et SANJIV, N. (1994). Software Estimation From Executable Specifications, Journal of Computer & Software Engineering, Vol. 2, No. 3, pp. 239-258.

[GUP92] GUPTA, R. K. et DE MICHELI, G. (1992). System-level Synthesis using Re-programmable Components, Proceedings of the European Design Automation Conference.

[GUP93a] GUPTA, R. K. (1993). Co-Synthesis of Hardware and Software for Digital Embedded Systems, PhD thesis, Stanford.

[GUP93b] GUPTA, R. et DE MICHELI, G. (1993). Hardware-Software Cosynthesis for Digital Systems, IEEE Design & Test of Computers, pp. 29-41.

[GUP94] GUPTA, R. K. et DE MICHELI, G. (1994). Constrained Software Generation for Hardware-Software Systems, In Proc. of Int. Workshop on Hardware/Software Codesign, pp. 56-63.

[HAR93] HARDT, W. et CAMPOSANO, R. (1993). Trade-Offs in HW/SW Codesign, In Proc. of Int. Workshop on Hardware/Software Codesign, oct.

[HAR96] HARDT, W. et ROSENSTIEL, W. (1996). Speed-Up Estimation for HW/SW Systems, Workshop on Hardware/Software Co-Design (4^{th} : 1996 : Pittsburgh) Vol. 4.

[HEN97] HENKEL, J. et ERNST, R. (1997). A Hardware/Software Partitioner Using a Dynamically Determined Granularity, Proceedings of the 34^{th} annual Conference on Design Automation Conference (DAC-97), pp. 691-696.

[INT99] INTEL home page, http://www.intel.nl/drg/pentiumII/appnotes/RDTSCPM1.HTM.

[ISM94] ISMAIL, T. B., ABID, M., O'BRIEN, K. et JERRAYA, A. (1994). An Approach for Hardware-Software Codesign, Proceedings of the Fifth International Workshop on Rapid System Prototyping, pp. 73-80.

[ISM95] ISMAIL, T. B. et JERRAYA, A. (1995). Synthesis Steps and Design Models for Co-Design, IEEE Computer,Vol. 28, No. 2, pp. 44-52.

[JAN94a] JANTSCH, A., ELLERVEE, P., ÖBERG, J. et HEMANI, A. (1994). A Case Study on Hardware/Software Partitioning, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, pp. 111-118.

[JAN94b] JANTSCH, A., ELLERVEE, P., ÖBERG, J., HEMANI, A. et TENHUNEN, H. (1994). Hardware/Software Partitioning and Minimizing Memory Interface Traffic, In Proceedings of EURO-DAC 94.

[KIE98] KIENLE, H. M. (1998). Introductory Manual : The SUIF 2.0 Compiler System, University of California, Santa Barbara.

[KNI96] KNIESER, M. J. et PAPACHRISTOU, C. A. (1996). COMET : A Hardware-Software Codesign Methodology, EURO-DAC 96.

[KOC93] KOCH, A. et GOLZE, U. (1993). An FPGA Based Co-Processor for SBus Workstations, 3rd International Workshop on Field-Programmable Logic and Applications.

[LAR95] LARUS, J. R. et SCHNARR, E. (1995). EEL : Machine-independent executable editing, In Proceedings of the SIGPLAN-95 Conference on Programming Language Design and Implementation (PLDI), pp. 291-300.

[LAR97] LARCHER, P. (1997). VHDL : introduction à la synthèse logique, Eyrolles, Paris, 183 p.

[LEW93] LEWIS, D. M., VAN IERSSEL, M. H. et WONG, D. H. (1993). A Field Programmable Accelerator for Compiled Code Applications, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines.

[LIE97] LIEM, C. (1997). Retargetable Compilers for Embedded Core Processors, Kluwer Academic Publishers.

[LUK93] LUK, W., LOK, V. et , I. (1993). Hardware acceleration of Divide and Conquer Paradigms : A Case Study, Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines.

[MAD98] MADSEN, J. et KNUDSEN, P. V. (1998). Communication Estimation for Hardware/Software Codesign, Codes/CASHE Workshop 98, Seattle, USA.

[MIC94] DE MICHELI, G. (1994). Synthesis and Optimisation of Digital Circuits, McGraw-Hill.

[ONI95] O'NILS, M., JANTSCH, A., HEMANI, A. et TENHUMEN, H. (1995). Interactive Hardware-Software Partitioning and Memory Allocation Based on Data Transfer Profiling, International Conference on Recent Advances in Mechatronics, ICRAM-95, Vol. 2, No. 1, pp. 447-452.

[PET95] PETERSON, J. B., O'CONNOR, R. B. et ATHANAS, P. M. (1995). Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures, Virginia Polytechnic Institute and State University, NSF MIP-9308390.

[RUN90] RUNDENSTEINER, E. A. et GAJSKI, D. (1990). A Design Representation Model for High-level-Synthesis, Technical report, University of California, Irvine.

[SAV96] SAVARIA, Y., BOIS, G., POPOVIC, P. et WAYNE, A. (1996). Computational Acceleration Methodologies : Advantages of Reconfigurable Acceleration Subsystems, SPIE's Photonics East, Boston, pp. 195-205.

[SMI91] SMITH, T. E. et SETLIFF, D. E. (1991). Towards an Automatic Synthesis System for Real-Time Software, In Proc. of 12th Real-Time Systems Symposium, pp. 34-42.

[SMI97] SMITH, M. J. S. (1997). Application-Specific Integrated Circuits, Addison-Wesley.

[SUI99] SUIF home page, http://suif.stanford.edu/suif.

[SUM00] SUMIT GUPTA'S home page, http://www.ics.uci.edu/~sumitg.

[SUZ96] SUZUKI, K. et SANGIOVANNI-VINCENTELLI, A. (1996). Efficient Software Estimation Methods for Hardware/Software Codesign, In Proc. of Vol. 33rd ACM IEEE Design Automation Conference, pp. 605-610.

[SYS00] SYSTEMC home page, http://www.systemc.org.

[THE99] THÉRIAULT, L. (1999). Prototype d'un outil d'analyse de performance : guide de l'utilisateur, http://www.grm94.polymtl.ca/~theriau/outil.

[THO93] THOMAS, D. E., ADAMS, J. K. et SCHMITT, H. (1993). A Model and Methodology for Hardware/Software Codesign, IEEE Design & Test of Computers, pp. 6-15.

[WIL98] WILSON, C. (1998). The SUIF Guide, Version 2.0, Stanford University.

[WOL93] WOLF, W. et MARTINEZ, J. C. (1993). C Program Performance Estimation for Embedded Systems Architecture Sizing, In Proc. of Int. Workshop on Hardware/Software Codesign.

[XIL99a] XILINX'S Data Book, Virtex 2.5 V Field Programmable Gata Arrays, http://www.xilinx.com/partinfo/virtex.pdf.

[XIL99b] XILINX home page, Core Generator, 1999.

[YE93] YE, W., ERNST, R., BENNER, T. et HENKEL, J. (1993). Fast Timing Analysis for Hardware/Software Co-synthesis, In Proc. of ICCD, pp. 452-457.

[YOU94] YOUNG, C. et SMITH, M. D. (1994). Improving the accuracy of static branch prediction using branch correlation, In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI), pp. 232-241.

Bibliographie

BALA, V. (1996). Low overhead path profiling, Technical Report, Hewlett Packard Labs.

BINH, N. N., IMAI, M., SHIOMI, A. et HIKICHI, N. (1996). Optimal Instruction Set Design through Adaptive database Generation, IEICE Trans. Fundamentals, Vol. E79-A, No. 3.

BOIS, G. (1997). Le codesign logiciel/matériel, Journal Industriel du Québec, en 3 volets de août à octobre 1997.

BOIS, G., BOSI, B. et SAVARIA, Y. (1997). A High Performance Reconfigurable Coprocessor for Digital Signal Processing, Proceedings of the 14th Annual Int. Conference Mentor Graphics Users' Group, Portland, Oregon.

BOSI, B., BOIS, G. et SAVARIA, Y. (1999). Reconfigurable Pipelined 2D Convolvers for Fast Signal Processing, IEEE VLSI Systems Transaction, pp. 1-33.

CALLAHAN, T. et WAWRZYNEK, J. (1996). Simple Profiling System for SUIF, Computer science Division, University of California, Berkeley.

CLAVEZ, J. P., HELLER, D. et PASQUIER, O. (1996). Uninterpreted Co-Simulation for Performance Evaluation of Hw/Sw Systems, International Workshop on Hardware/Software co-design (4th : 1996 : Pittsburgh) Vol. 4, pp. 132-139.

EDWARDS, M. D. et FORREST, J. (1996). Software acceleration using programmable hardware devices, IEE proceedings, Computers and digital techniques, Vol. 143, No. 1, pp. 55-63.

GOLDBERG, A. (1991). Reducing overhead in counter-based execution profiling, Technical Report CSL-TR-91-495, Computer System Laboratory, Standford University.

GONG, J., GAJSKI, D. D. et NARAYAN, S. (1994). Software Estimation from Executable Specifications, Journal of Computer & Software Engineering, Vol. 2, No. 3, pp. 239-258.

GRODE, J., KNUDSEN, P. V. et MADSEN, J. (1998). Hardware Resource Allocation for Hardware/Software Partitioning in the LYCOS System, Proceedings Design, Automation and Test in Europe, pp. 22-27.

HERRMANN, D., HENKEL, J. et ERNST, R. (1994). An Approach to the Adaptation of Estimated Cost Parameters in the COSYMA System, International Workshop on Hardware/Software codesign (3rd : 1994 : Grenoble) Vol. 3rd , pp. 100-107.

JAIN, S., BALAKRISHNAN, M., KUMAR, A. et KUMAR, S. (1997). Speeding Up Program Execution Using Reconfigurable Hardware and a Hardware Function Library, International Conference on VLSI on Design (11th : 1998 : Chennai, India), pp. 400-405.

KHALI, H. SAVARIA, Y. et HOULE, J. L. (1997). Computational Limits of Homogeneous Acceleration Using Lookup Tables, HPCS-97, Winnipeg, pp. 345-351.

KHALI, H. (1999). Algorithmes et architectures de calcul spécialisés pour un système optique autosynchronisé à précision accrue, Thèse de doctorat (chapitres 4 et 5), École Polytechnique de Montréal, pp. 85-158.

MOWRY, T. et HO, R. (1996). Feedback and Simulation Tools for Investigating D-Across Parallelism, First SUIF Compiler Workshop, Standford University.

NARAYAN, S. et GAJSKI, D. D. (1992). Area and Performance Estimation from System-level Specifications, Technical Report ICS-92-16.

OUDGHIRI, H., KAMINSKA, B. et RAJSKI, J. (1997). A Hardware/Software Partitioning Technique with Hierarchical Design Space Exploration, Custom Integrated Circuits Conference.

SAVARIA, Y., EL HASSAN, F., KHALI, H. ET SAWAN, M. (1998). An Effective Hardware/Software Implementation of a Viterbi Decoder Using an FPGA-based Reconfigurable Computing Platform, FPD'98, pp. 161-165.

SHADITALAB, M., BOIS, G. ET SAWAN, M. (1998). Self Sorting 1024 point FFT on Re-configurable Acceleration Subsystem with DSP Processor, Workshop on Field-Programmable Devices (FPD'98), Montréal.

SHADITALAB, M., BOIS, G. ET SAWAN, M. (1998). Self Sorting Radix-2 FFT on FPGA's using Parallel Pipelined Distributed Arithmetic Blocks, Symposium on FPGA Custom Computing Machines (FCCM'98), Napa, California.

Annexes

Annexe A

Proposition d'un diagramme détaillé de flux de données global

Un diagramme détaillé de flux de données global entre les principaux modules de l'outil d'analyse apparaît à la figure A.1. Le diagramme est constitué des modules suivants : le front-end[1]; le profilage, l'analyse (DFA/CFA/Slicing), le partitionnement, l'optimisation de boucle, le back-end[2] et l'estimation de performance; par ailleurs il met en lumière un certain de nombre de relations qui doivent interagir avec l'ensemble dont nous avons parlé à la section 2.3.1.

Les flèches montrent comment l'information circule entre les différents modules. Les lignes doubles horizontales représentent les bases de données où seront stockées les informations de flux, de profilage, etc.

---

[1] D'une façon simplifiée, on peut définir le *front-end* comme étant l'étape du "parsing" réalisant une conversion du langage C vers une description SUIF.
[2] D'une façon simplifiée, on peut définir le *back-end* comme étant l'étape d'émission du code cible (C ou VHDL) à partir de l'IR de SUIF.

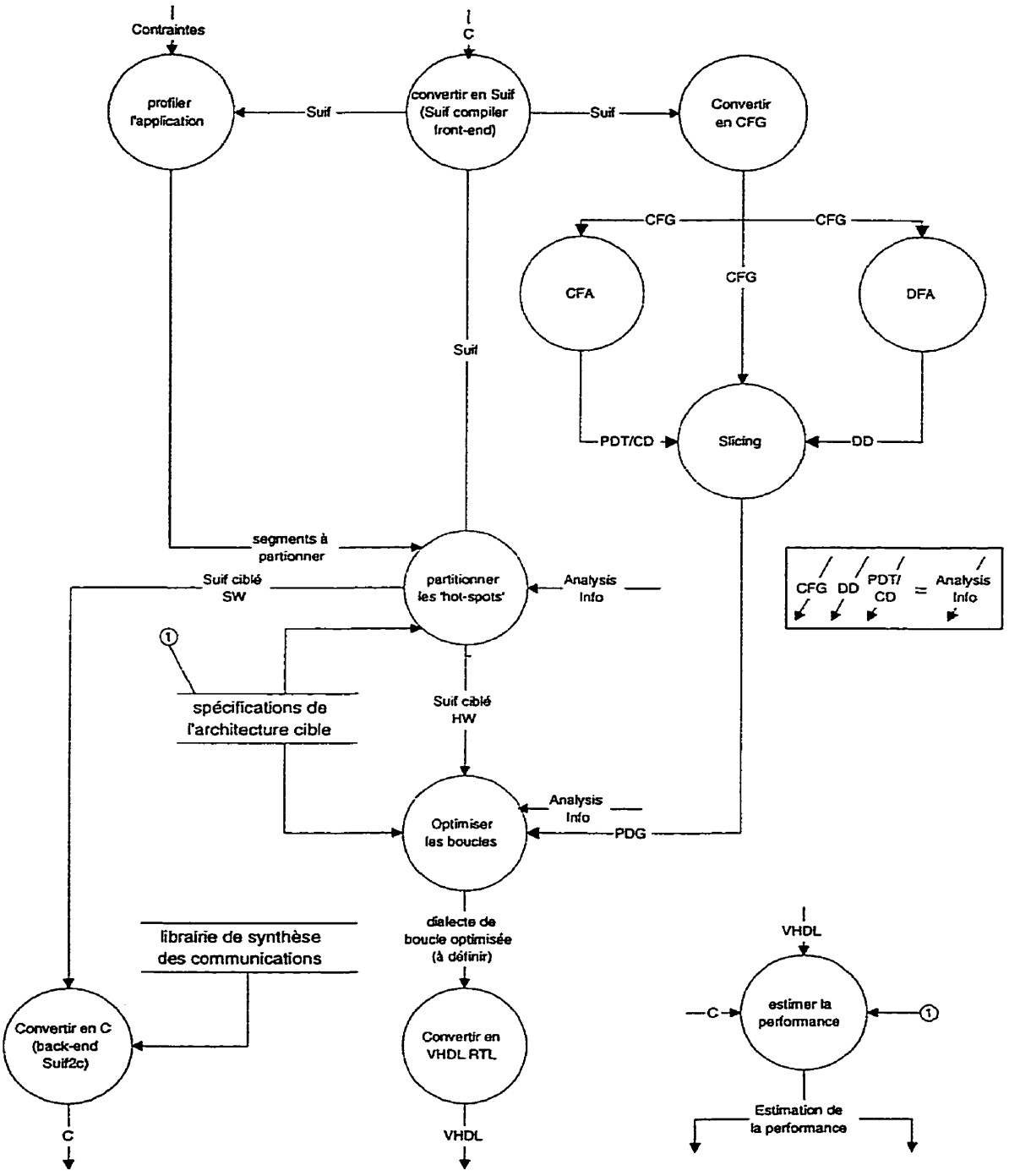


FIG. A.1 Diagramme détaillé de flux de données global (d'après [DOU98])

Annexe B

Prototype d'un outil d'analyse de performance : guide de l'utilisateur

B.1 Préambule

Ce guide est une introduction à l'utilisation du prototype d'outil d'analyse de performance supporté par l'environnement de compilation SUIF. Bien que la majorité des concepts s'appliquent en particulier à l'environnement SUIF2.0.0, l'essentiel reste aussi valable pour les versions antérieures (p. ex. SUIF1.1.2). Il ne cherche pas à expliquer en détail comment est mis en œuvre l'outil. La plus grande partie du guide suppose que vous n'avez aucune ou peu de notions préalables sur ces types de compilateur du domaine public. L'environnement SUIF fournit un excellent environnement pour l'analyse et la mesure de performance de programmes écrits en langage C ANSI. Le système de base comprend des compilateurs C et Fortran. L'environnement SUIF est très largement compatible avec les standards comme C ANSI, de même qu'avec son propre héritage, il est donc possible d'écrire des nouvelles structures qui compilent et s'exécutent sur une variété de plates-formes.

B.2 Objectifs

L'éventail des fonctionnalités disponibles et la lourdeur du logiciel peut, à prime abord, impressionner une personne non-familière avec les concepts de base liés à la compilation. Ce document se limite essentiellement à vous permettre d'extraire les données de flux et de profilage en vue de mesurer la performance de programmes en fonction de métriques pré-sélectionnées. Ces métriques de performance permettent de caractériser l'exécution de chacun des blocs de base et de détecter les blocs les plus prometteurs ("boucles chaudes"). Ces blocs seront éventuellement transposés et exécutés sur une architecture matérielle reconfigurable afin d'accentuer la vitesse d'exécution de ces blocs et ainsi d'accélérer l'exécution de toute l'application. La pertinence des partitionnements dépend évidemment de la qualité des métriques choisies. Ces métriques permettent à la fois d'estimer les ressources nécessaires et requises pour effectuer ces blocs sur le logiciel, en occurrence sur un Pentium, et sur une plate-forme matérielle reconfigurable de type FPGA, tout en tenant compte, évidemment, des contraintes physiques imposées par l'architecture.

B.3 Introduction

L'environnement de compilation qu'offre SUIF a été mis à contribution dans l'élaboration de ce prototype. Pour des raisons stratégiques, deux versions de ce même environnement a été utilisé pour mettre en œuvre l'ensemble des métriques. La première version de SUIF (SUIF1.1.2) offre une généreuse boîte à outils mais, en contre partie, sa mise en œuvre est médiocre. La seconde version (SUIF2.0.0), quant à elle, offre une boîte à outils plus restreinte, mais une philosophie de conception hautement supérieure. Pour ces raisons et pour des questions d'unification des efforts à l'intérieur du groupe CODE, on a opté pour l'utilisation de ces deux versions simultanément, en prenant bien soins de définir, à priori, un format standard utilisable à la fois par l'environnement SUIF1.1.2 et SUIF2.0.0. L'avantage majeur d'utiliser ces deux versions est d'exploiter les fonctionnalités respectives de chacune d'entre elles.

Voici comment chacune des versions a été mise à contribution dans le cadre de ce présent ouvrage. Tout d'abord, l'environnement SUIF1.1.2 a été utilisé pour mettre en œuvre la métrique de performance visant à estimer l'espace mémoire requis pour effectuer une application spécifique. Bien que, théoriquement, il était possible de mettre en œuvre l'ensemble des métriques avec cet environnement, cela impliquant la réécriture d'une grande partie des fonctionnalités (module d'analyse de flux, etc.) disponibles sous l'environnement SUIF2.0.0. Il était donc préférable d'utiliser SUIF2.0.0 pour mettre en œuvre les métriques nécessitant les informations relatives aux flux de données.

B.4 Installation

Avant tout, précisons que les fichiers sources originaux de la trousse de base de l'environnement SUIF1.1.2 et SUIF2.0.0 sont disponibles à [SUI99]. Toutefois, tous les fichiers sources de la structure logicielle du prototype résultant de notre recherche et des efforts des membres du Groupe de Recherche en Microélectronique de l'École Polytechnique de Montréal sont disponibles à [THE99].

Pour l'installer, il faut tout d'abord décompresser les deux arborescences de la trousse de base de l'environnement SUIF1.1.2 et SUIF2.0.0, intitulées *basesuif-1.1.2.tar.Z* et *basesuif-2.0.0.alpha.784.tar.Z* respectivement.

Par la suite, on lance le Makefile de chacune des arborescences. Pour de meilleurs résultats, lire le fichier README (présent à la racine de chacune des arborescences) afin de spécifier les bons chemins d'accès lors de l'installation. Une fois les deux environnements installés, décompresser les deux arborescences des répertoires de travail contenant les programmes sources et les programmes tests utilisés lors du développement, intitulées *métriques-SUIF1.1.2.tar.Z* et *métriques-SUIF2.0.0.tar.Z* respectivement. La première arborescence (*métriques-SUIF1.1.2*) permet d'estimer les métriques de performance relatives au temps d'exécution logiciel (sur un Pentium) et à l'espace mémoire. La seconde (*métriques-SUIF2.0.0*) permet d'estimer les métriques de

performance relatives au temps d'exécution matériel (sur un FPGA), à la surface matérielle et à la bande passante. La section suivante présente comment obtenir les informations relatives rattachées à chacune des métriques en fonction d'une application type.

B.5 Lancement

B.5.1 Temps d'exécution requis sur un Pentium

Premièrement, démarrer Microsoft Visual C++ 6.0 et ouvrir l'espace de travail ("workspace") intitulé *timer.dsw*. Celui-ci est localisé au répertoire suivant : ~/METRIQUE/timer/VisualC++/timer. Cet espace de travail contient le fichier *timer.c* permettant d'estimer le nombre de cycles requis pour effectuer un segment de code (boucle ou nid de boucles) d'une application. Pour obtenir ce nombre, il suffit que ce segment soit borné par les fonctions de comptage *startChrono()* et *stopChrono()* respectivement, tel qu'indiqué à la section 3.2.1.

Le calcul de la suite de Fibonacci sera utilisé comme exemple (fig. B.1) afin de mettre l'accent sur les sections cruciales avec lesquelles l'utilisateur devra interagir. Une fois les fonctions de comptage disposées, il suffit d'effectuer les trois étapes suivantes :

1) Sauvegarder les modifications apportées au fichier *timer.c*;
2) Compiler *timer.c*;
3) Exécuter *timer.c*.

Le résultat de l'exécution sera affiché dans une fenêtre MS-DOS. Les valeurs obtenues représentent le nombre de cycles requis pour exécuter cette boucle. Pour obtenir une valeur temporelle, en secondes, il suffit de diviser ce nombre de cycles par la fréquence d'horloge du Pentium sur lequel a été exécuté le segment de code.

```
...
#define Fibnum 50   /* ce paramètre spécifie que l'on calcul le 50ème terme de la suite */
...
/* ---------- placer le code de l'application à analysée ---------- */
void fib()
{
   int i;
   int F[Fibnum];
   F[0] = 0;
   F[1] = 1;
   startChrono(C1);   /* départ du compteur C1 */
   for(i = 2; i < Fibnum; i++)
      F[i] = F[i - 1] + F[i - 2];
   stopChrono(C1);   /* arrêt du compteur C1 */
}
...
int i;
float p;
double x, m;
void main()
{
   SetPriorityClass(GetCurrentProcess(), 31);   /* priorité maximale */
   init_timer();
   C1 = 0; m = 0;
   for(i = 1; i <= 100000; i++)   /* astuce pour obtenir une valeur de moyenne significative de C1 en
                                     exécutant 100000 fois le même segment */
   {
      fib();   /* appel de la fonction (application analysée) */
      p = (float)i;
      x = (double)C1;
      m = ((p - 1) / p) * m + (1 / p) * x;   /* calcul de la moyenne */
      C1 = 0;
   }
   printf("moy = %lf\n", m);   /* affichage du nombre de cycles requis à l'écran */
}
```

FIG. B.1 Exemple illustrant les fonctions de comptage à l'intérieur d'une application type

B.5.2 Métriques relatives à l'architecture matérielle du FPGA

Pour obtenir les valeurs des métriques relatives au temps d'exécution requis sur un FPGA et celles relatives à la surface matérielle utilisée sur ce même FPGA, on doit, premièrement, copier le code source de l'application (p. ex. *fib.c*) que l'on désire analyser dans le répertoire suivant : ~/projet_CODE/src/bins/tests. Notons que tous les fichiers sources analysés doivent avoir l'extension *.c* et respecter le standard d'écriture C ANSI.

Par la suite, on exécute les primitives[1] SUIF servant à estimer la performance de cette application (fig. B.2) de la façon suivante :

1) scc -.spd *fib.c*;
2) do_convertsuif1tosuif2 *fib.spd fib.s2*;
3) ../do_code_compiler *fib.s2* out > *out.dat*.

Dans le fichier *out.dat* se trouve toutes les informations relatives au temps d'exécution requis et à la surface matérielle requise (en CLB) en fonction d'un FPGA XVC300-200 MHz de la famille Virtex de XILINX. Les résultats obtenus sont, évidemment, en fonction de chacun des blocs de base (boucles ou nids de boucles) présents à l'intérieur de l'application analysée.

B.5.3 L'espace mémoire requis

Pour obtenir les valeurs de la métrique relative à la quantité de mémoire requise, on doit, premièrement, copier le fichier source de l'application (p. ex. *fib.c*) que l'on désire analyser dans le répertoire suivant : ~/METRIQUE/timer/VisualC++/qt-mem. Nous utiliserons, encore une fois, le code source calculant la suite de Fibonacci (réf. fig. B.2), pour mettre l'accent sur les sections cruciales avec lesquelles l'utilisateur devra interagir. Ainsi, pour obtenir la valeur de cette métrique, on procédera d'une façon très systématique. Tout d'abord, il faut modifier chacune des opérations de lecture des

---

[1] Les primitives SUIF utilisées sont : *scc*, *do_convertsuif1tosuif2* et *s2c*. Le lecteur intéressé à approfondir ce sujet peut consulter les ouvrages qui y sont consacrés [WIL98, SUI99].

variables vectorielles, en les remplaçant par des appels de fonction de type *AddAccess(...)*. Cette fonction permet d'enregistrer toutes les séquences d'accès de ces variables (sect. 3.2.2). Ainsi, pour chaque opération de lecture correspond un appel de fonction de type *AddAccess(1, 1, i - 1)* (fig. B.3) qui est l'équivalent d'une opération de lecture de la variable vectorielle *F[i - 1]* de la figure B.2.

```
#define Fibnum 50
void fib()
{
   int i;
   int F[Fibnum];
   F[0] = 0;
   F[1] = 1;
   for(i = 2; i < Fibnum; i++)
      F[i] = F[i - 1] + F[i - 2];
}
```

FIG. B.2 Code source calculant la suite de Fibonacci

```
#define Fibnum 50
void fib()
{
   int i;
   for(i = 2; i < Fibnum; i++)
   {
      AddAccess(1, 1, i - 1);   /* correspond à F[i - 1] */
      AddAccess(1, 1, i - 2);   /* correspond à F[i - 2] */
   }
}
```

FIG. B.3 Code source modifié de la figure B.2

Après quoi, il faut modifier le fichier *varsize_table.cc* (localisé au même répertoire) afin d'intégrer les appels de fonction de type *PrintAccess(...)* permettant de visualiser ces séquences d'accès et, ainsi, déterminer l'espace mémoire requis. La figure B.4 met

l'accent sur les sections importantes où l'utilisateur doit intervenir. On note que pour chaque appel de la forme *AddAccess(1, 1, i - 1)* et *AddAccess(1, 1, i - 2)* correspond un appel de la forme *PrintAccess(1, i)*. Une fois terminé, il ne reste qu'à exécuter les quatre étapes de la façon suivante :

1) scc -.spd *fib.c*;
2) s2c *fib.spd output.c*;
3) g++ *varsize_table.cc*;
4) a.out > *out.dat*.

Dans le fichier *out.dat* se trouve toutes les séquences d'accès aux variables vectorielles sélectionnés précédemment (à l'intérieur des segments de code analysés) ainsi que la quantité de mémoire requise, en bits.

```
#define Fibnum 50
...
#include "output.c"    /* inclut le fichier source de l'application en format SUIF */
...
main()
{
   int i, QM;
   fib();   /* appel de l'application analysée */
   for(i = 0; i < Fibnum; i++)
   {
      printf("F[%d]  ", i); PrintAccess(1, i);
   }
   QM = MemRequired();   /* appel de la fonction estimant l'espace mémoire
                            requis (en bloc-mémoire) */
   printf("QM (en bits) = %d\n", 32*QM);   /* affichage à l'écran de la quantité de
                                              mémoire requise pour effectuer la
                                              boucle ci-dessus (1 bloc = 32 bits) */
...
}
```

FIG. B.4 Procédure permettant l'affichage des moments d'accès du code source de la figure B.3

B.5.4 Bande passante

Dans le cadre de cette étude, une simplification a été imposée limitant les solutions matérielles à un seul FPGA, tel que mentionné à la section 3.1.3. Pour ces raisons, les paramètres de cette métrique sont fermes. Les valeurs de celle-ci ont été obtenues en utilisant le modèle de calcul de la bande passante explicité à la figure 3.8 (sect. 3.2.4).

B.6 Contact

Concepteur du présent ouvrage

Lévis Thériault (theriau@GRM94.PolyMtl.CA)
École Polytechnique de Montréal
Pavillon André Aisenstadt
2920 Chemin de la tour
Montréal (Québec)
H3T 1J4

Autres concepteurs du projet CODE

François-R Boyer (boyerf@IRO.UMontreal.CA)
Université de Montréal
Pavillon André Aisenstadt
2920 Chemin de la tour
Montréal (Québec)
H3T 1J4

Frédéric Doucet (doucet@GRM94.PolyMtl.CA)
École Polytechnique de Montréal
Pavillon André Aisenstadt
2920 Chemin de la tour
Montréal (Québec)
H3T 1J4