UNIVERSITY OF YAOUNDÉ I
DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITÉ DE YAOUNDÉ I
DÉPARTEMENT D'INFORMATIQUE

Vers un profiling adaptatif :

État de l'art des méthodes actuelles d'analyse de programmes et perspectives

MÉMOIRE DE DEA

présenté par

Bénédicte Ramelie KENMEI YOUTA

Année académique 2001/2002

Directeurs :
Emmanuel KAMGNIA
Basile LOUKA

Encadrement :
Philippe CLAUSS

Table des matières

Dédicaces

A Dave, Patrick, Gaddyel et Linda :

Puissiez-vous trouver en ce travail un puissant encouragement à persévérer sans relâche et avec foi, aussi bien sur le plan intellectuel qu'en tous les autres domaines de votre vie.

Remerciements

Grâce soit rendue à Dieu par Jésus-Christ. C'est Lui qui m'a accordé le privilège de mener à bien ce travail et qui, tout au long de cette année encore, a renouvelé mes forces à chaque fois que j'en avais besoin.

Je remercie les Dr. Kamgnia Emmanuel et Louka Basile pour avoir accepté de diriger ce mémoire.

Le stage de DEA dont ce document est le résultat n'aurait pu se dérouler sans la part active qu'y a prise mon encadreur, le Pr Philippe Clauss ; pour sa disponibilité, sa patience, sa cordialité qui ont en outre facilité mon intégration à l'équipe du LSIIT à Strasbourg, je lui adresse mes sincères remerciements.

A tous les membres du laboratoire sus mentionné, en particulier Vincent Loechner, Benoît Meister, Frédérique Wagner, je dis également merci.

Je voudrais remercier tout spécialement mon chef de département le Dr. Kamgnia Emmanuel pour son dynamisme et son ardeur qui nous ont permis, à mes camarades et moi, de recevoir des cours de la part de plusieurs enseignants et chercheurs venant de plusieurs universités de grand renom. De plus, il est l'initiateur de tout le processus qui m'a conduite à effectuer ce stage de DEA à l'université Louis Pasteur.

Merci également à tous les camarades avec qui j'ai eu la joie de travailler : Donfack Hervé, Chana Anne Marie, Nankep Pierre, Tchoupe Maurice, Talla Narcisse, Kouonou Annicet, Tagne.

A ceux qui m'ont soutenue par leur présence, leurs paroles réconfortantes et leur appui, j'exprime ici ma profonde reconnaissance ; il s'agit de mes parents Youta Bernard et Mambou Marthe, de Youdom Y. J. Valéry, Djiki N. Yves, Georges Mbeck, Nwouega Y. Jean et Dieffe Jules.

Je ne saurais oublier Youta K. Emmanuel, Youbissi Lisette, Fangseu B. Edwige et Maboundou Bertin dont l'aide a été pour moi inestimable durant toutes ces dernières semaines.

Que le Tout Puissant bénisse toutes ces personnes.

Résumé

Ce mémoire présente un état de l'art des méthodes actuelles d'analyse de programmes, en faisant la distinction entre l'analyse statique et l'analyse dynamique. Il s'inscrit aussi dans la liste assez brève de documents traitant en théorie du profiling : celui-ci étant essentiellement lié à l'exécution des programmes, nombreux sont ceux qui le manipulent de façon pratique et qui n'en ont par conséquent qu'une connaissance empirique. En établissant les avantages et inconvénients de chacune de ces méthodes d'analyse et en les confrontant, il ouvre des perspectives sur un nouveau type possible d'analyse, devant tenir compte à la fois des qualités relevées dans les deux méthodes, et des inconvénients existants pour les amoindrir : il s'agira d'un profiling adaptatif des programmes.

Abstract

In this work, we present a state of the art of methods encountered in the litterature, classified into static and dynamic approaches. By comparing the advantages and inconvenients of these methods, we bring out a new approach for program analysis witch take care of positive points mentionned in the previous comparison ; that is, an adaptated program profiling.

Chapitre 1

Introduction

Le visage de la société contemporaine a été profondément marqué par les progrès scientifiques et techniques, dans plusieurs domaines comme les transports, la communication, l'économie. Le présent siècle, souvent appelé *siècle de la vitesse*, semble pour sa part se distinguer par des mots-clés tels que **performance, efficacité, sécurité, portabilité** et bien d'autres encore. Un regard attentif mettrait en exergue le domaine de l'informatique, qui a ceci de particulier qu'il s'insinue, voire même qu'il s'impose dans tous les autres. Ce prosélytisme demande que sur le plan du matériel comme sur le plan du logiciel, de nouvelles approches soient conçues et développées, lesquelles approches doivent garantir une parfaite intégration des systèmes manipulés, en regard notamment aux mots-clés évoqués ci dessus.

Sur le plan du matériel, les progrès technologiques actuels permettent une plus grande fréquence de fonctionnement pour les processeurs.[1] Seulement, cette performance des processeurs est obtenue au prix d'un accroissement de leur complexité : les concepteurs ont dû ajouter un grand nombre de structures (pipeline, architecture superscalaire, caches, prédiction de branchement, exécution dans le désordre...).

Sur le plan logiciel, ces ajouts ont une certaine incidence, car ils induisent des cas où la performance maximale ne peut être atteinte (rupture du pipeline, défaut de cache, mauvaise prédiction, ...). En conséquence de tout ceci, l'écart entre la performance maximale et la performance réelle obtenue par les programmes s'accroît rapidement.

L'efficacité de l'exécution d'une application, quelque soit le processeur sur laquelle est déployée, dépend très fortement de la structure des programmes, structure qui est imposée par le programmeur ; cependant elle comporte des degrés de liberté que des procédés logiciels, appelés optimisations de code, peuvent exploiter pour augmenter la performance des applications. Ces procédés d'optimisation utilisés dans les compilateurs requièrent l'analyse des programmes et permettent aux programmes de s'abstraire de l'architecture cible des machines.

Ainsi le but de l'analyse de programmes se définit comme suit : partir d'un programme donné devant s'exécuter sur une architecture donnée, et trouver les endroits où l'on peut effectuer des transformations judicieuses afin de pouvoir exploiter au mieux les possibilités

[1] Il est aujourd'hui possible d'intégrer sur un même composant une dizaine d'unités fonctionnelles et une grande mémoire cache fonctionnant à une fréquence de l'ordre du Ghz.

offertes par le matériel (dans le cas d'environnements embarqués mobiles il s'agira par exemple de réduire la consommation en électricité).

Jusqu'à présent, l'analyse de programmes s'est faite soit de manière statique soit de manière dynamique. Dans ce mémoire, nous avons étudié ces deux types d'analyse qui se font indépendamment l'un de l'autre, pour en dégager les avantages et les inconvénients ; à l'issue de cette étude, il nous a paru intéressant de nous orienter vers une méthode d'analyse qui tiendrait compte aussi bien des résultats obtenus statiquement que des résultats obtenus dynamiquement, méthode qui donnerait ainsi lieu à un *profiling adaptatif* des programmes.

Ce mémoire de DEA est organisé comme suit. Le chapitre 2 introduit l'approche formelle de modélisation des structures de programmes qui est à la base de la suite de nos travaux, à savoir le modèle polyédrique ; il est suivi de la présentation de techniques d'analyse statique connues. Nous traitons des limitations de l'approche statique en fin de chapitre. La technique de prédiction de branchement peut relever à la fois d'une approche statique et dynamique, c'est pourquoi le chapitre 4 lui est consacré. Le " profiling" ou analyse dynamique suit immédiatement, avec les différentes techniques couramment utilisées et les limitations qui s'y rattachent actuellement. Ces limitations, considérées par rapport à l'importance signalée du profiling pour les programmes à structure de contrôle dynamique, peuvent être amoindries de manière significative par une collaboration du profiling avec l'analyse statique, ce qui est l'objet du dernier chapitre.

Chapitre 2

Le modèle polyédrique

2.1 Présentation du modèle

Beaucoup de programmes et en particulier les applications scientifiques et multimédia passent le plus clair de leur temps dans les boucles, c'est pourquoi elles font le plus souvent l'objet d'études. C'est ainsi que dans le modèle polyédrique, l'attention est portée en grande partie sur les nids de boucles.

Définition 2.1.1 *Un nid de boucles (voir figure 2.1) est dit parfait lorsque les conditions suivantes sont réunies :*

- *chaque boucle en contient au maximum une seule autre,*
- *toutes les instructions sont situées dans la boucle la plus interne,*
- *il n'existe aucun saut permettant d'entrer ou de sortir du nid de boucles,*
- *les bornes inférieures (l_k) et supérieures (h_k) des indices de la boucle de niveau k s'expriment comme des fonctions affines à coefficients rationnels des $k-1$ indices des boucles externes et d'un ensemble $\mathcal{N}$ de paramètres.*

for $i_1 = l_1(\mathcal{N}), h_1(\mathcal{N})$ ⟵ boucle de niveau 1
 for $i_2 = l_2(i_1, \mathcal{N}), h_2(i_1, \mathcal{N})$
  ⋮
  for $i_n = l_n(i_1, i_2, \ldots, i_{n-1}, \mathcal{N}), h_n(i_1, i_2, \ldots, i_{n-1}, \mathcal{N})$ ⟵ boucle niveau n
   instructions ...
  end for
  ⋮
 end for
end for

FIG. 2.1 – Nid de boucles parfait

Selon l'ordre d'imbrication des boucles, on parle de boucles externes ou de boucles internes. Chaque nid de boucles correspond de manière duale à une intersection finie et bornée de demi espaces appelée *polytope.*

Définition 2.1.2 *Soit $\mathcal{E}$ un espace de dimension m et $\overrightarrow{x}$ un vecteur de $\mathcal{E}$; on appelle demi espace l'ensemble des vecteurs $\overrightarrow{n}$ de $\mathcal{E}$ tels que $\overrightarrow{n}\overrightarrow{x} \geq 0$.*

2.1.1 Liens entre nids de boucles et polytopes

Définition 2.1.3 *Soit le système d'inéquations donné par*

$$Ax + a \geq 0 \tag{2.1}$$

où A est une matrice de constantes, x un vecteur de variables et a un vecteur de constantes. L'ensemble de tous les x qui satisfont le système (2.1) est un polyèdre (convexe). C'est une intersection finie de demi espaces.

Propriété 2.1.1 *Convexité :*
Si x_1 et x_2 sont deux points d'un polyèdre, alors toutes les combinaisons convexes données par $\lambda x_1 + (1-\lambda)x_2$, $0 \leq \lambda \leq 1$ sont également dans ce polyèdre.

Définition 2.1.4 *On appelle polytope un polyèdre borné.*

Les bornes de boucles l_k et h_k (avec $1 \leq k \leq n$, n étant le niveau le plus élevé d'imbrications) définissent un ensemble de contraintes linéaires sur les indices de boucles i_k, de la forme :

$$\sum_k a_k i_k + \sum_k b_k n_k + c \geq 0, \;\; a_k, b_k, c \in \mathbf{Q}, \;\; i_k \in \mathbf{Z}, \;\; n_k \in \mathcal{N} \tag{2.2}$$

Un tel ensemble de contraintes définit un polyèdre dans l'espace des indices de boucles, basé sur l'ensemble des paramètres $\mathcal{N} = [n_k]$. Chaque indice de boucle étant borné, ce polyèdre est borné, c'est donc un polytope. Il est noté $\mathcal{D}$: le domaine d'itérations associé au nid de boucles.

Exemple 2.1.1 *Soit le nid de boucles suivant :*
for $i = 1$ **to** n
  for $j = 2$ **to** $2i - 3$
    $A[i, j] = B[3 * i - 1, j]$
  end for
end for

L'ensemble des contraintes est donné par :

$$\begin{array}{rcrcrcrcl} i & + & 0j & + & 0n & - & 1 & \geq & 0 \\ -i & + & 0j & + & n & + & 0 & \geq & 0 \\ 0i & + & j & + & 0n & - & 2 & \geq & 0 \\ 2i & - & j & + & 0n & - & 3 & \geq & 0 \end{array} \quad \textit{i.e.} \quad \begin{array}{rcl} i & \geq & 1 \\ i & \leq & n \\ j & \geq & 2 \\ j & \leq & 2 * i - 3 \end{array}$$

ce qui correspond à

$$\begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 0 \\ 2 & -1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \\ -2 \\ -3 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Traçons sur un graphe l'ensemble des points correspondant aux valeurs de (i, j) *pour* $n = 7$*, i.e le polyèdre correspondant aux systèmes ci-dessus .*

FIG. 2.2 – Domaine d'itérations

L'ensemble des points en noir représente le domaine d'itérations $\mathcal{D}$ *des boucles. Une itération est un couple* (i, j) *avec* $(i, j) \in Z^2$ *; on voit alors clairement que l'ensemble des itérations du nid de boucles correspond à un triangle, qui est un polytope.*

Définition 2.1.5 *On appelle réseau-polytope l'intersection d'un réseau régulier de points avec un polytope.*

Les valeurs prises par les indices de boucle étant incrémentées d'une constante entière qui est le pas des boucles, elles appartiennent à un réseau régulier de points. $\mathcal{D}$ forme un réseau-polytope.

Par une opération appelée normalisation, on peut toujours effectuer un changement de variables sur les indices de boucles, de sorte que l'incrément sur chaque boucle soit de $+1$, et que les valeurs prises par les indices de boucle soient entières. Nous pouvons donc considérer des boucles normalisées sans nuire à la généralité. L'ensemble des valeurs prises par les indices de boucle est alors l'intersection de $\mathcal{D}$ avec Z^n

Définition 2.1.6 *Un Z-polytope de dimension n est l'intersection du réseau standard* Z^n *avec un polytope. C'est donc un réseau-polytope particulier.*

L'association d'un nid de boucles avec un polyèdre a donné naissance au modèle polyédrique développé et utilisé par de nombreux chercheurs, notamment dans le domaine de la parallélisation automatique [23].

Comme on part d'un nid de boucles pour aboutir à un polytope donné, il est possible qu'à partir d'un domaine d'itérations et donc d'un polytope, l'on retrouve les éléments de tableau accédés à chacune des itérations ; pour cela, on utilise des notations matricielles sur lesquelles sont faites quelques transformations.

2.1.2 Transformation et accès aux tableaux

Reprenons l'exemple (2.1.1). Pour une itération (i, j) donnée, on désire trouver l'élément du tableau B qui est accédé ; quand $i = 2$ et $j = 3$, on accède à l'élément de B donné par $B[5, 3] = B[3 * i - 1, j]$, i.e.

$$\begin{pmatrix} 5 \\ 3 \end{pmatrix} = \begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} + \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

ou

$$\begin{pmatrix} 5 \\ 3 \end{pmatrix} = \begin{pmatrix} 3 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}$$

ce qui est la représentation homogène de la même égalité ;

la matrice $R = \begin{pmatrix} 3 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}$ est appelée matrice d'accès ou de référence.

L'application de cette matrice à l'ensemble des itérations du domaine $\mathcal{D}$ permet d'obtenir l'ensemble des éléments de tableau accédés par la boucle.

Accès à des tableaux : Une référence à un tableau est une transformation affine de l'espace homogène des indices et des paramètres vers l'espace des données du tableau que l'on peut représenter par une matrice appelée matrice de référence ou matrice d'accès.[23][28].

Propriété 2.1.2 *Soit R une matrice d'accès, l'ensemble des indices des éléments de tableau référencées par R est l'image par R des points entiers du domaine d'itérations $\mathcal{D}$.*

Ainsi, travailler sur des nids de boucles accédant à des tableaux revient à travailler sur des polytopes et des fonctions affines.

2.2 Utilité du modèle polyédrique

Afin de pouvoir effectuer des optimisations correctes, le compilateur doit disposer d'un certain nombre d'informations sur le code source ; ces informations proviennent de l'analyse du flot de contrôle du programme ainsi que de l'analyse du flot de données.

Le modèle polyédrique permet de représenter, d'analyser et de transformer des programmes, qu'ils soient séquentiels ou parallèles. Il est possible d'analyser finement la circulation des informations au cours du déroulement du programme et d'associer chaque valeur lue aux opérations

qui y accèdent. On peut ainsi repérer les données non réutilisées, ou décrire la manière dont les opérations utilisant les mêmes données seront regroupées dans le temps (*localité temporelle*). La portée du modèle polyédrique est toutefois limitée, puisqu'il modélise des nids de boucles aux propriétés particulières. Alléger ces contraintes sans perdre la finesse d'analyse du modèle est un objectif actuel de recherche.

En établissant la correspondance nids de boucles-polytopes, on utilise les propriétés des polytopes pour effectuer des transformations de programmes, ce qui favorise l'optimisation des codes. Par exemple, pour connaître le nombre d'itérations contenues dans un nid de boucles ou alors le volume des données qui y sont accédées, on détermine le nombre de points à coordonnées entières contenus dans le polytope correspondant. Ce nombre, dépendant des paramètres du nid de boucles, est donné par un *polynôme d'Ehrhart*.

2.3 Polynômes d'Ehrhart

On peut aujourd'hui dénombrer de façon symbolique les points à coordonnées entières contenus dans un Z-polytope dépendant de paramètres à l'aide de polynômes particuliers, appelés pseudo-polynômes. Ce résultat porte le nom de *polynômes d'Ehrhart* et a été rendu possible grâce aux travaux de Eugène Ehrhart en mathématiques [32]. Les polynômes d'Ehrhart ont été repris et étendus par Philippe Clauss et Vincent Loechner au cas des polytopes définis par des contraintes affines à plusieurs paramètres [35][36], et un logiciel de calcul de ces polynômes a été développé à l'ICPS et intégré à la librairie *polylib* [37].

2.3.1 Nombres périodiques et pseudo-polynômes

Un nombre périodique s'exprime sous forme d'un tableau de valeurs rationnelles où :

• la dimension est le nombre de paramètres

• la valeur sélectionnée est fonction du modulo de chaque paramètre par rapport au nombre de valeurs possibles dans la dimension du paramètre.

De manière formelle, on a la définition qui suit :

Définition 2.3.1 ***Nombre périodique***
Soit le vecteur $N = (n_j)_{j=1...p}$ où p est un entier naturel. Un nombre périodique U_N est défini par un tableau U de dimension p et de taille $s_1 \times s_2 \times \ldots \times s_p$ de la manière suivante : $u_N = U[i_1, i_2, \ldots, i_p]$ si pour tout j de 1 à p, $i_j = n_j \bmod s_j$. Le vecteur $s = (s_j)$ est appelé la période de u_N.

Remarque : u_N est la valeur du nombre périodique U_N

Comme corrollaire à cette définition introduite par Clauss on a :

Corollaire 2.3.1 *Soit le vecteur $N = (n_j)_{j=1...p}$ où p est un entier naturel. Un nombre périodique U sur un corps K, de période $S = (s_j)_{j=1...p}$ est une fonction de $Z^p \longrightarrow K$ qui à N associe u_N telle que : $u_{N+t_j} = u_N \quad \forall j \in [1, p]$, avec $t_j = \begin{pmatrix} 0 & \ldots & s_j \ldots 0 \end{pmatrix}^T$*

Exemple 2.3.1 *Le nombre périodique* $[2, 1, 5]_N$ *est de dimension* 1 *(il y a un seul paramètre* N*) et de période* 3 *en* N *; il vaut :*

$$\begin{array}{cccccc} 2 & si & N & mod & 3 & = & 0 \\ 1 & si & N & mod & 3 & = & 1 \\ 5 & si & N & mod & 3 & = & 2 \end{array}$$

Le nombre périodique suivant

$$\begin{bmatrix} 3 & 1 \\ 2 & 0 \\ 1 & 5 \\ 3 & 2 \end{bmatrix}_{N,M}$$

est de dimension 2 *et de période* 4×2.
Il est égal à $[3, 1]_M$ *lorsque* $N \bmod 4 = 0$*, à* $[2, 0]_M$ *lorsque* $N \bmod 4 = 1$*, à* $[1, 5]_M$ *lorsque* $N \bmod 4 = 2$*, à* $[3, 2]_M$ *lorsque* $N \bmod 4 = 3$*. Il vaut donc :*

$$\left\{ \begin{array}{ccccccccccc} 1 & si & N & mod & 4 & = & 0 & et & M & mod & 2 & = & 1 \\ 5 & si & N & mod & 4 & = & 2 & et & M & mod & 2 & = & 1 \\ 1 & si & N & mod & 4 & = & 2 & et & M & mod & 2 & = & 0 \\ 3 & si & N & mod & 4 & = & 3 & et & M & mod & 2 & = & 0 \\ 0 & si & N & mod & 4 & = & 1 & et & M & mod & 2 & = & 1 \end{array} \right.$$

Définition 2.3.2 *Un pseudo-polynôme est un polynôme dont les coefficients sont des nombres périodiques.*

2.3.2 Polynômes d'Ehrhart pour un polytope bordé

Définition 2.3.3 *Un* ***polytope paramétré*** *est* ***bordé*** *dans* Q^n *s'il est défini par un ensemble de contraintes linéaires à coefficients entiers et si chaque coordonnée de ses sommets s'exprime comme une fonction affine de ses paramètres.*

Soit $\mathcal{D}$ un polytope paramétré bordé dans un espace de dimension n.

Théoreme 1 *Le nombre de points de* Z^n *contenus dans un tel polytope, exprimé en fonction des paramètres, est un pseudo-polynôme, appelé* ***polynôme d'Ehrhart****, dont les variables sont les paramètres.*

Ehrhart et Clauss ont mis en évidence des relations entre certaines caractéristiques d'un polytope et celles du polynôme d'Ehrhart correspondant.

Les bornes des boucles, i.e. les contraintes définissant les polytopes que nous traitons ici, sont des fonctions affines à coefficients rationnels (pouvant être entiers) des indices des boucles. Ceci entraîne que les sommets du polytope ainsi décrit ont des coordonnées rationnelles.

Définition 2.3.4 *On appelle* ***dénominateur d'un polytope*** *le plus petit commun multiple des dénominateurs des coordonnées rationnelles des sommets du polytope.*

Théoreme 2 *La période maximale d'un coefficient périodique en N du polynôme d'Ehrhart associé à un polytope bordé est égale au plus petit commun multiple des dénominateurs des coefficients en N des coordonnées des sommets du polytope. De manière plus générale, la période des coefficients d'un polynôme d'Ehrhart est bornée par le dénominateur du polytope.*

Théoreme 3 *Le degré du polynôme d'Ehrhart est borné par la dimension du polytope.*

Ainsi, le nombre de points entiers d'un polytope dépendant de paramètres $n_1, n_2, \ldots, n_p$ noté $np(n_1, n_2, \ldots, n_p)$ est un pseudo-polynôme dont les variables sont les paramètres et dont le degré est la dimension du polytope. Nous pouvons l'illustrer par l'exemple suivant :

Exemple 2.3.2 *On considère le polytope paramétré de la figure suivante :*

FIG. 2.3 – Polytope de dimension 2

Ce polytope se dénombre par un polynôme d'Ehrhart dépendant du seul paramètre n ; la dimension du plan étant égale à 2, le pseudo-polynôme pour ce carré est np(n), de degré 2 en n. Le ppcm des dénominateurs des coordonnées des sommets en n est de 2, les coefficients du polynôme d'Ehrhart seront donc de période 2 au maximum.

Ce pseudo-polynôme est donné par :

$$np(n) = [c_1, c_2]_n.n^2 + [c_3, c_4]_n.n + [c_5, c_6]_n \tag{2.3}$$

où $c_1, \ldots, c_6$ sont des constantes ; après avoir instancié n à des valeurs initiales, on trouve

$$np(n) = \frac{1}{4}.n^2 + [1, \frac{1}{2}]_n.n + [1, \frac{1}{4}]_n \tag{2.4}$$

$[1, \frac{1}{2}]_n$ et $[1, \frac{1}{4}]_n$ sont des nombres périodiques.

Si $n = 10$ alors le carré de la figure 2.3 contient $100/4 + 10 + 1 = 36$ points à coordonnées entières.

2.3.3 Cas d'un polytope quelconque

Propriété 2.3.1 *Tout polytope paramétré défini par un ensemble fini de contraintes linéaires à coefficients entiers est décomposable en un ensemble de polytopes bordés.*

Les sommets d'un tel polytope s'expriment comme des fonctions affines définies sur des domaines de valeurs adjacents des paramètres. Loechner [44] a montré que les valeurs des paramètres pour lesquelles le polytope est un polytope bordé sont définies par des contraintes affines à coefficients entiers sur ces paramètres, que l'on appelle **domaine de validité** du polytope bordé. L'ensemble des domaines de validité forme une partition de l'espace des paramètres. Un polytope paramétré se décompose alors en un ensemble de polytopes bordés associés à leur domaine de validité. A chaque polytope bordé (et donc à chaque domaine de validité) correspond un polynôme d'Ehrhart.

La définition des polynômes d'Ehrhart est étendue à un cas plus général ainsi qu'il suit : le résultat de toute opération ensembliste sur des polytopes définis par des contraintes linéaires à coefficients entiers possède un polynôme d'Ehrhart, calculé par décomposition selon cette opération. Ceci est permis parce que l'ensemble des polynômes d'Ehrhart muni de l'opération d'addition est un groupe inversible.

Le modèle polyédrique tel que décrit en ce chapitre constitue l'un des cadres les plus utilisés pour l'analyse de programmes ; ici comme ailleurs, il est question d'étudier le comportement des programmes et selon le moment où s'effectue cette étude, on distinguera l'analyse statique de l'analyse dynamique.

Chapitre 3

L'analyse statique

Définition 3.0.5 *L'analyse statique est l'ensemble des techniques qui permettent de déduire automatiquement des propriétés de programmes à partir de leur code source.*

Effectuée au moment de la compilation, l'analyse statique s'appuie sur des bases théoriques de sémantique de programme ; ce type d'analyse consiste à examiner le code source et à étudier ses schémas d'accès aux données, en vue d'aider à la production de code optimisé. En effet, sur les régions de code qui sont identifiées comme lieux de perte possible de performance, le travail d'optimisation est susceptible de donner le plus de bénéfices.
Un analyseur statique (outil permettant de faire de l'analyse statique) donne le comportement d'un programme par l'examen de son code source. Cette donnée permet alors de :

- réduire la quantité de calculs à effectuer dans certains programmes,
- réduire le temps d'exécution,
- paralléliser automatiquement des programmes,
- etc ...

L'analyse statique peut opérer sur plusieurs types de programmes et de plusieurs manières ; parmi ces types de programme, on a :

- les programmes logiques (ex : Prolog) : ici il s'agit entre autres d'analyse de mode ;
- les programmes fonctionnels paresseux (ex : Ada) : l'analyse de nécessité ;
- les programmes impératifs (ex : C) : on mentionne à ce niveau l'analyse de flots de données et l'analyse de flots de contrôle.

L'analyse statique par interprétation abstraite (pour les programmes logiques et les programmes fonctionnels paresseux) est très utile pour les tests et vérification de logiciels. En procédant par interprétation abstraite, P. et R. Cousot [24] ont donné une formalisation des méthodes d'analyse de flots de données itératives, lesquelles méthodes feront en particulier l'objet de notre étude. L'interprétation abstraite est une méthode de conception d'analyse statique basée sur une description rigoureuse de la sémantique des langages de programmation : en résumé, on part de la sémantique standard du langage et on en déduit une sémantique approchée (dite abstraite) qui est calculable et qui exprime les propriétés recherchées des programmes. Tout comme la sémantique standard manipule un domaine de valeurs (domaine concret), la sémantique abstraite manipule son propre domaine (domaine abstrait).

Il existe également des méthodes géométriques d'analyse statique de programmes et le

modèle qui sous tend cette approche à été présenté au précédent chapitre. Ainsi, tout au long de ce mémoire, nous nous intéresserons aux langages impératifs en présentant quelques techniques d'analyse pour la modélisation des accès mémoire.

3.1 Analyse de dépendances

Au cours de l'exécution d'un programme, une instruction du texte source prescrit une modification de l'état mémoire de la machine. Plusieurs instructions composent généralement une exécution, et il est possible que la même instruction soit exécutée un grand nombre de fois. Il est aussi possible que des instructions différentes accèdent à la même cellule mémoire et ce, dans un ordre bien prescrit par la sémantique du programme ; cet ordre dans lequel s'effectuent les accès est sensible et doit donc être impérativement respecté, ce qui est l'objet de l'analyse de dépendances.

Définition 3.1.1 *L'analyse de dépendances rassemble des informations sur les instructions d'un programme qui, au cours d'une exécution, accèdent à une même cellule mémoire.*

3.1.1 Différents types de dépendances

Il existe plusieurs sortes de dépendances : les dépendances de sortie (WAW), les dépendances de flot (RAW) et les anti-dépendances (WAR).

• **Les anti-dépendances ou écriture après lecture (WAR) :** ce type de conflit se produit lorsqu'une instruction cherche à écrire à un emplacement mémoire avant que celui-ci n'ait pu être lu par une instruction antérieure. Avec l'avènement de l'exécution dans le désordre[1] ce conflit tend à devenir fréquent.

• **Les dépendances de sortie ou écriture après écriture (WAW) :** ce conflit se produit lorsqu'une instruction essaie d'écrire au même emplacement qu'une instruction précédente.

• **Les dépendances de flot ou lecture après écriture (RAW) :** ce conflit se produit lorsqu'une instruction cherche à lire une donnée avant que cette dernière ait pu être écrite par une instruction lancée antérieurement.

Remarque : une lecture après une lecture n'est pas une dépendance de données.

Pour deux instructions S_1 et S_2 d'un programme, les conditions de Bernstein [42] permettent de déterminer s'il existe des dépendances de l'une des trois sortes mentionnées ci-dessus.

Définition 3.1.2 *Conditions de Bernstein :*
Soient S un programme, $L(S)$ l'ensemble des variables lues dans S, $M(S)$ l'ensemble des variables mises à jour dans le programme S ; deux parties de programme notées S_1 et S_2 peuvent

[1] le processeur change l'ordre des opérations à effectuer ou tente d'utiliser des registres différents

être exécutées dans un ordre quelconque (i.e sont indépendantes) si les trois conditions suivantes sont simultanément vérifiées :

$$M(S_1) \cap L(S_2) = \emptyset \tag{3.1}$$

$$L(S_1) \cap M(S_2) = \emptyset \tag{3.2}$$

$$M(S_1) \cap M(S_2) = \emptyset \tag{3.3}$$

la condition (3.1) correspond à une dépendance de flots, (3.2) correspond à une anti-dépendance et (3.3) correspond à une dépendance de sortie.

La dépendance de flot est appelée vraie dépendance parce qu'elle ne peut être supprimée, contrairement aux autres dépendances qui peuvent l'être, par des renommages de variables ou des duplications de code. Le paragraphe suivant décrit comment les dépendances peuvent être formulées mathématiquement.

3.1.2 Formulation mathématique des dépendances

Définition 3.1.3 *(Ordre lexicographique)*
Soient deux itérations

$$I = \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_n \end{pmatrix} \quad et \quad J = \begin{pmatrix} j_1 \\ j_2 \\ \vdots \\ j_n \end{pmatrix}$$

I est lexicographiquement inférieur à J (on note $I \prec J$) si $\exists \quad k \in \{1, 2, \ldots, n\}$ tel que $i_1 = j_1,\ i_2 = j_2,\ \ldots,\ i_{k-1} = j_{k-1},\ i_k < j_k$.

Exemple 3.1.1

$$\begin{pmatrix} 5 \\ 3 \\ 2 \end{pmatrix} \quad \prec \quad \begin{pmatrix} 5 \\ 4 \\ 0 \end{pmatrix}$$

J est dépendant de I si et seulement si il existe un élément affecté par I et référencé par J avec $I \prec J$ et il n'y a pas d'autre itération K avec $I \prec K \prec J$ telle que J soit dépendante de K pour le même élément.

Les dépendances entre les instructions peuvent dégrader les performances des programmes et des microprocesseurs. Il importe donc de bien les étudier afin de pouvoir aider de manière efficace à l'optimisation des codes. Cette analyse peut être faite aussi bien par usage des variables d'induction [45] que par utilisation du modèle polyédrique [43].

Outre les dépendances, la localité des données est un facteur important dont doit tenir compte toute analyse.

3.2 Analyse de localité

3.2.1 Goulets d'étranglement architecturaux

La mémoire d'un ordinateur est organisée en hiérarchie, avec au sommet des unités de petites tailles et de faible temps d'accès. Plus on descend dans la hiérarchie, plus la taille augmente, ainsi que le temps nécessaire au processeur pour y accéder.

Avec le développement du matériel, la vitesse de travail du processeur s'est accrue considérablement, ce qui n'a pas été le cas pour les mémoires. C'est ainsi qu'on note une grande disparité entre les vitesses des processeurs, de la mémoire vive, des disques durs et des CD-ROM. Ce déséquilibre entraîne un ralentissement dans l'exécution des programmes, le processeur étant le plus souvent obligé d'attendre sans rien faire. Dans le but de palier à ces différentes latences mémoire, on a développé des mécanismes matériels tels que les mémoires caches et les TLB (Translation Lookaside Buffer) ; ces derniers sont en général internes au processeur.

• **La mémoire cache** constitue un niveau intermédiaire entre deux niveaux hiérarchiques de mémoire de temps d'accès trop différents. Pour notre part, nous nous intéresserons au cache entre la mémoire vive et le processeur. Lorsque le processeur tente d'accéder à une donnée et la trouve dans le cache, on dit qu'il y a *cache hit* ; dans le cas contraire, il y a défaut de cache, ce qui est coûteux en termes de cycles processeur (entre 8 et 32 cycles selon l'architecture).

• **Le TLB (Translation Lookaside Buffer)** comme son nom l'indique est un registre, utilisé pour la traduction d'adresse virtuelle en adresse mémoire. La phase de traduction étant assez coûteuse (elle l'est d'autant plus lorsque le jeu d'instructions comporte plusieurs formats d'adresses), l'utilisation du TLB a pour but de réduire ce temps de traduction. Lorsque l'on accède à des pages mémoire dont l'adresse virtuelle se trouve encore dans le TLB (i.e. des pages mémoire ayant été très récemment accédées), il n'y a plus besoin de traduction d'adresse, mais dans le cas contraire, il y a défaut de TLB, ce qui coûte dans l'ordre de 100 cycles processeur.

Les mécanismes de cache et de TLB sont ainsi utilisés pour réduire les latences mémoires et donc pour réduire le temps d'exécution des programmes ; cependant, on se rend compte que l'amélioration apportée par ces mécanismes dépend fortement de la manière dont les programmes accèdent à leurs données. Si l'on n'y prend garde, le temps d'exécution de ces programmes peut s'avérer plus long que dans des cas où ces mécanismes ne sont pas mis en place, ce qui est regrettable. De là provient alors l'intérêt de l'analyse de la localité des données. Cette localité se décompose en deux, la localité spatiale et la localité temporelle.

– *La localité spatiale* est observée lorsque l'on accède à plusieurs données proches en mémoire, dans un court intervalle de temps.

– *La localité temporelle* est observée lorsque l'on accède de manière répétée à une même donnée en mémoire, dans un court intervalle de temps.

Au niveau du processeur, la localité des données se manifeste au travers du nombre de

cache hits ; au niveau du cache, elle se manifeste par la présence des données cherchées dans une même ligne du cache ; au niveau du TLB, la localité s'observe par la présence des données cherchées dans la même page mémoire.

Il devient donc clair qu'en portant une attention particulière à la localité des données, on réduit le temps d'exécution des programmes, par la baisse des défauts de cache et de TLB. En analysant la réutilisation et la durée de vie des variables, on améliore l'exploitation de la localité temporelle.

3.2.2 Durée de vie des variables

La durée de vie d'une variable est le temps écoulé entre sa première et sa dernière utilisation. L'analyse de durée de vie est intéressante en ceci qu'elle permet d'optimiser l'allocation des registres. En fait, une des propriétés importantes de l'allocation des registres dans les boucles sans branchement est qu'il faut et il suffit de contrôler le nombre V de variables en vie à chaque instant pour assurer à posteriori l'allocation de ces variables sur V registres. On sait donc qu'il n'est pas nécessaire d'utiliser plus de registres que le nombre maximal de variables simultanément en vie à chaque pas de temps.
Une variable est vivante en mémoire entre sa définition et son utilisation.

Exemple 3.2.1 *Soit le bout de code Fortran suivant :*

$$
\begin{array}{llllllll}
1 & & x := & 0 & & & & \\
2 & 11: & y := & x+1 & & & & \\
3 & & s := & s+y & & & & \\
4 & & x := & 2*y & & & & \\
5 & & if & x<N & then & goto & 11 & \\
6 & & return & s & & & &
\end{array}
$$

x vit de la fin de 1 *au début de* 2*, et de la fin de* 4 *à l'entrée de* 2*,*
y vit de la sortie de 2 *à l'entrée dans* 4
s vit pendant tout le programme.

Définition 3.2.1 *Pour un nid de boucles, la durée de vie d'une variable est le nombre d'itérations pendant lesquelles elle est utile en mémoire.*

On peut calculer la durée de vie des variables grâce aux polynômes d'Ehrhart []. Une fois ce calcul effectué pour toutes les variables, on construit un graphe d'interférence : c'est un graphe non dirigé dont les noeuds sont les variables et les dont arcs relient les variables vivant simultanément. Intuitivement, deux variables qui interfèrent ne peuvent être allouées au même endroit [38].

3.2.3 Réutilisation des variables

L'analyse de la réutilisation d'une variable permet d'améliorer sa localité temporelle, en regroupant par exemple les itérations qui l'utilisent.

Définition 3.2.2 *Lorsqu'une donnée est accédée par deux itérations I et J telles que $I \prec J$, on dit qu'il y a réutilisation de cette variable. Le vecteur $J - I$ est appelé vecteur de réutilisation ; on dit encore qu'il y a réutilisation dans la direction de $J - I$.*

Exemple 3.2.2 *Soit la boucle suivante :*

for $i = 2$ **to** 7
for $j = 1$ **to** $2i - 3$
$A(6i + 9j - 5) = A(6i + 9j - 5) + 3$
end for
end for
l'itération $(i + 3, j - 2)$ accède à la donnée $A(6i + 9j - 5)$ après que (i, j) y ait accédé ; il y a donc réutilisation de $A(6i + 9j - 5)$ et le vecteur de réutilisation est $(3, -2)$

Une donnée peut être réutilisée lors d'itérations différentes à travers une même référence (matrice d'accès) ou à travers des références différentes ; dans le premier cas on parlera d'*auto-réutilisation* et dans le second cas, on de *réutilisation de groupe*. On parle de réutilisation *spatiale* dans le cas d'accès à un même bloc mémoire (voir le paragraphe ci-dessous).

Les réutilisations sont considérées comme des cas particuliers de dépendances et [28][29][33][34] présentent de nouveaux résultats sur les optimisations qui peuvent en être déduites, partant toujours du modèle polyédrique.

Dépendances et localité des données constituent un point focal particulier dans la modélisation des accès mémoire, et c'est toujours en rapport avec ces éléments que se déploient les différentes techniques de modélisation.

3.3 Modélisation des accès mémoire

3.3.1 Les mémoires cache

Une mémoire cache est constituée de plusieurs lignes ou blocs, chaque ligne étant un ensemble de mots mémoire contigus. Tous les blocs du cache sont de même taille, i.e. tous les mots sont de taille fixe. La mémoire aussi est divisée en lignes et lorsqu'une donnée est référencée, tout au moins pour la première fois, la ligne mémoire qui la contient est chargée dans une ligne du cache. Si pour une référence ultérieure à cette donnée, elle est absente du cache, le processeur la cherchera en mémoire et ensuite remplacera une des lignes du cache par la ligne mémoire la contenant. La manière dont les blocs sont chargés et remplacés dans le cache dépend de son organisation. On distingue à cet égard trois types de cache, les caches à placement direct, les caches associatifs, et les caches associatifs par ensembles.

- Cache à placement direct : un bloc $\overline{B_j}$ de la mémoire sera placé dans un bloc $\overline{B_i}$ du cache si et seulement si $j \bmod n = i$ où n est le nombre de lignes du cache.
- Cache associatif : les blocs mémoire peuvent être placés à n'importe quelle ligne du cache, et lors du chargement d'un bloc, on écrase une ligne de cache en utilisant une stratégie de remplacement comme LRU (*Least recently Used*) par exemple.
- Cache associatif par ensemble S_i : un bloc $\overline{B_j}$ de la mémoire sera placé dans un ensemble S_i si et seulement si $j \bmod m = i$ où m est le nombre d'ensemble de lignes du cache ;

le placement à l'intérieur d'un ensemble se fait par la stratégie LRU. L'associativité du cache est le nombre de lignes par ensemble.

Lorsqu'on recherche une donnée qui n'est pas présente dans le cache, cela cause un défaut de cache. Il y a plusieurs sortes de défaut de cache, 3 en particulier, à savoir les défauts compulsifs, les défauts de capacité et les défauts de conflits :

-les défauts compulsifs sont incompressibles, parce qu'ils ont lieu lors de la première référence à la donnée ; ce type de défaut ne peut être évité que par un préchargement (prefetching) de la donnée ;

-les défauts de capacité sont liés à la taille limitée du cache ;

-les défauts de conflits résultent de nombreuses réutilisations du même bloc dans la mémoire cache, à travers des accès proches dans le temps de données contenues dans des blocs mémoire différents mais placés sur la même ligne de cache.

3.3.2 Equations de défauts de cache (Cache Miss Equations)

Tandis que la plupart des analyses essaient de limiter le nombre de défauts de cache pouvant survenir au cours de l'exécution d'un programme, Ghosh, Martonosi et Malik [26] introduisent une méthode formelle de calculs des défauts de cache, laquelle permet de connaître précisément le nombre de défauts de cache générés par chaque référence à un tableau et de cibler exactement la cause de ces défauts. Il s'agit des équations de défauts de caches (Cache Miss Equations). L'idée de base est la suivante.

Il y a utilisation optimale du cache lorsque l'on accède continuellement à des données qui y sont présentes ; on fait donc de la réutilisation de lignes de cache. Si une donnée est absente du cache, c'est soit qu'elle n'y avait pas été chargée auparavant, soit qu'elle en a été délogée et dans ce cas la réutilisation ne peut être effective. Il faut donc s'intéresser aux cas de réutilisation non effectives. Etant donnée l'associativité m d'une mémoire cache, m lignes seulement peuvent être placées dans un ensemble de ce cache. On s'intéresse alors à toutes les itérations qui se sont produites entres 2 utilisations d'un bloc mémoire et pour chacune on regarde si les blocs accédés ont été alloués sur le même ensemble que le bloc réutilisé. Cela conduit à un système d'équations et d'inéquations pour chaque itération considérée. Si on trouve plus de m solutions à ce système, c'est qu'un conflit a lieu. Chacune des solutions de ce système correspond à une situation plausible de défaut de cache.

La résolution des équations de défauts de cache est à la fois coûteuse et difficile à mettre en oeuvre (problème NP-complet) [40] ; les travaux mentionnés en [26] ne résolvent pas directement le système, mais utilisent des techniques mathématiques de manipulation des équations Diophantiennes pour donner soit une approximation des solutions, soit une estimation de la borne supérieure sur le nombre de solutions. D'autres méthodes de résolution de ce système, basée sur le modèle polyédrique, ont été proposées dans [40][27] et continuent à faire l'objet de recherches.

3.3.3 Fenêtres de références généralisées

Les fenêtres de référence ont été introduites pour la première fois par Gannon, Jalby et Gallivan [39] afin de définir un modèle pour l'évaluation des propriétés de localité d'un programme, et aussi pour conditionner des transformations de programme qui soient optimisantes. La taille d'une fenêtre de référence représente la taille minimale de la mémoire locale qui est requise pour éviter d'avoir à recharger des données en cours d'exécution. En effet, à chaque point du programme, la fenêtre de référence représente l'ensemble des mots mémoire (les variables) qui sont en vie, dans le sens où ces variables ont été accédées dans le passé et seront à nouveau accédées dans le futur. La fenêtre de référence évalue ainsi le nombre de défauts de cache dûs aux capacités de taille.

Cette notion a été étendue afin de prendre en compte les défauts de cache dûs aux conflits. L'idée principale est de considérer chaque ligne dans la mémoire cache comme étant de la mémoire locale et d'évaluer la fenêtre de référence associée à cette ligne. Typiquement, dans les mémoires caches à accès direct, si la taille de la fenêtre de référence généralisée (GRW) est plus grande que 1 à un moment de l'exécution du programme et pour une ligne donnée dans la mémoire cache, des défauts de cache vont se produire après cet instant.

Pour définir la GRW, on part de la définition initiale d'une fenêtre de référence.

Définition 3.3.1 *Etant donné un point p dans la trace de l'exécution d'un programme, la fenêtre de référence est l'ensemble des emplacements mémoire accédés dans le passé avant et au point p et qui seront reaccédés dans le futur après et au point p.*

Si la taille —GRW— de la fenêtre de référence n'est jamais plus grande que la taille de la mémoire locale, toutes les réutilisations possibles seront exploitées et aucun accès mémoire ne sera utile pour recharger les données une fois qu'elles ont été chargées dans la mémoire cache. Dans le cas d'une mémoire cache associative par ensembles, cette définition est étendue en deux temps :

- Dans un premier temps, les fenêtres de référence sont étendues pour prendre en compte la localité spatiale. Cela revient fondamentalement à considérer les blocs mémoires au lieu des emplacements mémoires.
- Dans un second temps, l'associativité est prise en compte en ne considérant que les blocs mémoires qui vont se placer sur un bloc fixé de la mémoire cache.

La fenêtre de référence généralisée se formule alors ainsi :

- pour un nids de boucles à bornes paramétrées affines où chaque instruction contient une référence paramétrée affine à un tableau,
- pour un bloc s fixé dans la mémoire cache
- pour une instruction S_0 et un vecteur d'itération $\overrightarrow{i_0}$,

$$GRW^{s}_{S_0(\overrightarrow{i_0})} = \{B \in M_s, \exists S, \exists \overrightarrow{i}, S(\overrightarrow{i}) \leq S_0(\overrightarrow{i_0}), B = BN(A_S(\overrightarrow{i})), s = CN(A_S(\overrightarrow{i}))\}$$

$$\bigcap$$

$$\{B \in M_s, \exists T, \exists \overrightarrow{j}, S_0(\overrightarrow{i_0}) \leq T(\overrightarrow{j}), B = BN(A_T(\overrightarrow{j})), s = CN(A_T(\overrightarrow{j}))\}$$

avec

- M_s l'ensemble des blocs de la mémoire principale qui vont se charger dans le bloc s de la mémoire cache,
- S et T deux instructions dans le code source,
- $\overrightarrow{i}$, $\overrightarrow{j}$ deux itérations dans le nid de boucles considéré,
- pour une instruction S, $S(\overrightarrow{i})$ représente la position de l'instruction S à l'itération $\overrightarrow{i}$ selon l'ordre lexicographique d'exécution de toutes les itérations, et $A_S(\overrightarrow{i})$ l'adresse absolue accédée par l'instance de S à l'itération $\overrightarrow{i}$,
- les fonctions BN et CN associent à une adresse absolue dans la mémoire principale, le numéro de bloc dans la mémoire principale et dans la mémoire cache.

La taille de la fenêtre $|GRW|$ est le nombre de points entiers de l'ensemble GRW. La détermination de cette taille nécessite une reformulation équivalente. Pour simplifier, on suppose que deux tableaux ne partagent pas le même bloc dans la mémoire principale : deux références n'accèdent pas à un même bloc sauf dans le cas où elles accèdent au même tableau. Pour deux instructions S et T qui accèdent au même tableau, la GRW associée est :

$$M_S \cap BN\{A_S(\overrightarrow{i}), S(\overrightarrow{i}) \leq S_0(\overrightarrow{i_0})\} \cap BN\{A_T(\overrightarrow{j}), S_0(\overrightarrow{i_0}) \leq T(\overrightarrow{j})\} \tag{3.4}$$

La fenêtre de référence généralisée est l'union des ensembles des fenêtres de références généralisées de telles paires d'instructions. Cette approche vise donc à établir des équations linéaires et des inéquations, paramétrées par des facteurs tels que le temps, le nombre d'ensembles du cache, les paramètres du programmes, le processeur, l'adresse de base du tableau. Les fenêtres de référence généralisées s'exprimant comme un ensemble d'inéquations linéaires dont chacune est considérée comme un polyèdre, le nombre de points entiers de ces polyèdres (polynôme d'Ehrhart) doit être calculé.

Le nombre de solutions de ce système est un indicateur pour la transformation de programme en vue d'améliorer la localité des données. Le calcul de ce nombre a été fait avec la librairie *polylib* dans [40], mais non sans difficulté, car les ensembles élémentaires ne sont pas en général des polyèdres mais des projections de polyèdres, ce qui les rend plus difficiles à manipuler, spécialement quand on énumère des points entiers. Toujours dans [40] une autre manière de calculer les fenêtres est proposée. Il s'agit d'un calcul dynamique de GRW.

En comparaison avec le formalisme des *Cache Miss Equations* introduit par *Ghosh et al.* [26], qui s'intéresse aux nombre de défauts de cache et à ce qui en a été la cause, les GRW s'intéressent aux endroits où se produisent ces défauts et leur formalisme entraîne une complexité de calcul inférieure.

3.4 Analyse statique et optimisation : avantages et limites

Les résultats d'une analyse statique permettent de guider le processus d'optimisation du programme analysé. Ce thème a fait l'objet, presque depuis les débuts de l'informatique, de nombreux travaux qu'il ne serait pas possible de tous évoquer ici. Une forte tendance est d'intégrer le processus d'optimisation dans l'analyse statique, qui du reste s'y prête favorablement selon les cas.

3.4.1 Les avantages

Les méthodes d'analyse de flots de données itératives, tout comme leur formalisation en interprétation abstraite, sont universellement applicables, i.e, elles sont indépendantes de l'architecture. L'analyse statique est utile pour les tests et vérification formelle et automatique de logiciels. Grâce à l'analyse statique on est parvenu à développer et à maîtriser certaines méthodes d'optimisation, lesquelles donnent des résultats intéressants ; nous pouvons citer le déroulage de boucle, le pavage de domaines d'itération, etc. ... Cependant, ces optimisations ne sont possibles que pour des cas bien particuliers de programme et cela découle du caractère statique même de cette analyse, certaines données ne pouvant être connues qu'en cours d'exécution des programmes.

3.4.2 Les limites

D'une manière générale, l'analyse statique peut estimer avec imprécision le nombre d'exécutions : les sorties d'instructions conditionnelles, le comptage d'itérations d'une boucle, la profondeur de récursion sont des choses difficilement prévisibles par des techniques statiques. Par exemple, un nid de boucles dans une instruction conditionnelle n'influence pas le temps d'exécution si la condition est toujours fausse ; optimiser une telle boucle jouerait défavorablement sur la performance du programme si le temps d'exécution des autres parties s'en trouve accru.

Les méthodes géométriques, fruits de la collaboration des communautés françaises de la parallélisation automatique et des réseaux systoliques, s'appliquent difficilement hors de certains cas favorables bien que donnant alors des résultats exacts.

Les limites de cette méthode d'analyse proviennent aussi des observations suivantes :

- il est possible d'induire de fausses dépendances, à cause du manque d'informations pour les analyser,
- il y a un manque d'informations pour mettre en évidence les constantes ou les variables jamais utilisées,
- on ne dispose pas d'informations suffisantes pour le traitement des variables à la frontière des procédures,
- les connaissances du programmeur sont forcément requises,
- l' ordre des calculs est à respecter et les temps d'exécution sont ignorés.

De plus, l'analyse statique du comportement du programme se fait souvent pour une architecture donnée. Mais à cause de la complexité des architectures, elle repose souvent sur des modèles simplistes ou qui ne considèrent qu'un des composants de l'architecture, ce qui ne permet pas toujours d'obtenir des gains de temps d'exécution élevés. Un autre inconvénient non négligeable est que les résultats dépendent forcément de la structure des programmes. Dans ce cas tout changement dans le code entraîne une réévaluation de la performance.

L'analyse statique des programmes regroupe toutes les techniques permettant de vérifier ou de "deviner" des propriétés "intéressantes" des programmes avant leur exécution (lors de la compilation, par exemple), à partir des codes sources. Cette analyse étudie en particulier les flots de données et les flots de contrôle pour guider des optimisations efficaces.

L'analyse du flot de données répond à la question de savoir si une variable donnée sera utilisée ultérieurement dans le programme, et ce faisant, donne lieu à de nombreuses opti-

misations basées sur la localité des données, l'optimisation des accès mémoire. L'analyse de flots de contrôle quand à elle, répond à la question de savoir si un point donné sera atteint durant l'exécution du programme ; ce faisant, elle fournit des indications sur le "chemin" qu'empruntera le programme au cours de son exécution, et donc sur sur les endroits réels où l'optimisation est susceptible d'avoir de l'effet.

L'analyse statique a été longuement étudiée et continue à faire l'objet de nombreuses recherches. Nous présentons dans le chapitre suivant un des élément principaux qui la caractérisent, à savoir la prédiction de branchement.

Chapitre 4

La prédiction de branchement

4.1 Définitions et généralités

Les processeurs actuels sont capables d'effectuer plusieurs instructions par cycle, et les pipelines d'exécution sont de plus en plus grands (4 étages pour Intel pentium Pro (1995) et 20 étages pour l'Intel Pentium 4 (2000))[1]. Les instructions sont ainsi chargées en séquence, afin d'exploiter au mieux la vitesse de travail du processeur. Il arrive bien souvent qu'une instruction de saut conditionnel ou non (**un branchement**) soit rencontrée dans le pipeline et il se produit alors une rupture de charge. A ce moment, l'on doit attendre que le calcul effectif de la condition et/ou de la cible soit effectué, et cela donne lieu à un goulet d'étranglement.

Définition 4.1.1 *Un branchement est un saut conditionnel ou inconditionnel ; c'est l'arrêt de l'exécution linéaire et séquentielle d'un programme, pour la continuer en un autre endroit.*

Etant donné que sur beaucoup d'applications, plus d'une instruction sur 5 ou 6 sont des branchements, il s'avère nécessaire de mettre en oeuvre des mécanismes appropriés pour réduire le temps d'attente. Par ailleurs, les branchements sont un problème pour le processeur, dans la mesure où ils entravent l'exécution dans le désordre.

Définition 4.1.2 *Lorsqu'au cours de l'exécution d'un programme, le processeur change l'ordre des opérations à effectuer (dans le but d'éloigner les instructions avec des dépendances de flots RAW) ou alors qu'il tente d'utiliser des registres différents (pour enlever les dépendances WAW et WAR), on dit qu'il fait de l'exécution dans le désordre.*

Cela étant, le processeur ne peut aller chercher une instruction après un branchement sans savoir au préalable le résultat de l'exécution de ce branchement. Ainsi il faut trouver une solution aux problèmes que posent les branchements.

Pour amoindrir les effets de la rupture du pipeline, Cliff Young et al. proposent une solution basée sur l'alignement des branchements [19], qui est un cas particulier du placement de code.

Une autre solution parmi beaucoup d'autres (suspension du pipeline, branchement différé...) est de prédire le branchement, pour pouvoir continuer le séquencement spéculatif des instructions après un branchement, sans en attendre la résolution.

[1] Le CPU est découpé en sous-circuits séquentiels, chacun pouvant à un instant donné traiter une partie d'une instruction. De cette manière, le CPU réalise un travail à la chaîne. Chaque maillon de la chaîne est alors un étage du pipeline.

4.2 La prédiction

Un processeur peut rencontrer, au cours de son travail, une instruction qu'il ne peut exécuter en un cycle d'horloge à cause de l'absence des données ; il doit alors attendre et gaspille ainsi du temps. Cette attente peut être comblée en déterminant quelles sont les instructions susceptibles d'être traitées aux prochains cycles. On analyse pour ce faire la structure algorithmique du code et on essaie de déterminer les embranchements qui vont suivre : on fait de la prédiction de branchement.

Une fois le branchement prédit, on exécute les instructions qui y correspondent, sans être déjà assuré que ce branchement sera effectivement effectué : on parle ici d'*exécution spéculative*, i.e. qu'on exécute certaines instructions sans savoir avec certitude si elles doivent être exécutées. A ce niveau, deux issues seulement sont possibles : soit la prédiction a été correcte, soit elle ne l'a pas été.

- Si on a bien prédit, on a gagné du temps, on a fait un travail utile.

- Si on a mal prédit, il faut revenir "sur ses pas", et donc éliminer les effets des instructions exécutées i.e. inverser leurs effets pour revenir à l'état précédent, et le travail fait a été inutile.

Dans ce cas néanmoins, on n'a pas perdu plus de temps que si on avait attendu. Toutes les instructions séquencées (et parfois même déjà exécutées) doivent être annulées et le séquencement doit être repris sur le chemin effectif : cela est une lourde pénalité pour le processeur en termes de cycles, car tout le pipeline doit être vidé. C'est pourquoi la prédiction de branchement doit être la plus correcte possible. C'est d'ailleurs à ce mécanisme qu'est liée en partie la performance des processeurs actuels.

Pour inverser l'effet des instructions qui ne devaient pas être exécutées, une méthode possible est de ne pas modifier les registres visibles par le programmeur avant d'être certain que l'instruction doit bien être exécutée. On place les résultats temporaires spéculatifs dans d'autres registres, qui seront copiés dans les registres visibles si les instructions exécutées correspondent bien à celles qui auraient effectivement dû l'être ; le pentium Pro utilise cette méthode.

Ainsi, des prédictions pertinentes doivent être faites et des schémas de prédiction de plus en plus sophistiqués sont mis en oeuvre dans les processeurs : il s'agit de prédiction dynamique. La prédiction de branchement peut également se faire de manière statique et nous verrons de quelles manières.

4.3 Prédiction statique

La prédiction statique de branchement peut se faire selon les manières suivantes :

- *Bit dans l'instruction :* à la compilation, on place un 1 ou un 0 pour dire si la condition doit être prédite comme vraie ou comme fausse ; par exemple, on peut prédire que le branchement ne sera pas effectué (on met le bit à 0) et dans ce cas, on continuera d'exécuter les instructions qui suivent ce branchement.

– *Adresse de destination :* on prédit les branchements vers l'arrière comme étant pris et ceux vers l'avant comme non pris. On part du fait que, en général dans une boucle, le branchement de la fin se fait vers le début.

La prédiction statique de branchement ne s'effectue donc pas au moment de l'exécution du programme ; elle permet de définir quelle branche doit ou ne doit pas être prise, mais elle ne permet pas de dire où aller. En effet, il faudrait alors connaître l'adresse mémoire des instructions de la branche qu'il faut exécuter. Un programme ne se chargeant pas à un endroit précis en mémoire, cette adresse n'est jamais connue de manière statique.

La prédiction statique de branchement a été étudiée par plusieurs chercheurs, notamment Thomas Ball, James Larus et Youfeng Wu [21, 22]. La plupart utilise des heuristiques pour prédire le branchement, ou encore utilisent des informations issues de l'analyse dynamique ; d'autres comme Jason R. C. Patterson proposent une autre approche, la prédiction par propagation de domaine de valeurs [18]. Cette approche a l'avantage d'être plus efficace que la précédente (en termes de correction de la prédiction) et de plus elle décharge davantage le programmeur.

4.3.1 Prédiction statique par propagation de domaines de valeurs

Description de la stratégie

La stratégie utilise des techniques de propagation de constante [13] et d'analyse de domaines [14]. Par propagation de constante on entend l'identification des expressions constantes pour toutes les exécutions possibles d'un programme pour pouvoir les évaluer à la compilation plutôt qu'à l'exécution ; ces valeurs sont ensuite propagées simplement le long du graphe de flots de données jusqu'à atteindre un point fixe.

Des travaux récents [13] ont permis que la propagation des constantes puisse être étendue au cas des branchements conditionnels comme dans l'exemple suivant :

Exemple 4.3.1 *on a :*

$$
\begin{array}{l}
i_1 \leftarrow 1 \\
j_1 \leftarrow i_1 * 4 \quad \Rightarrow \quad j_1 \leftarrow 4 \\
\textbf{\textit{if}}\ (j_1 + 1 > 2) \\
\quad i_2 \leftarrow 2 \\
\textbf{\textit{else}} \\
\quad i_3 \leftarrow 3 \\
i_4 \leftarrow \phi(i_2, i_3) \quad \Rightarrow \quad i_4 \leftarrow \phi(i_2) \\
k_1 \leftarrow 3 + i_4 \quad \Rightarrow \quad k_1 \leftarrow 5
\end{array}
$$

Au lieu de déterminer les expressions de valeur constante pour toutes les exécutions possibles, la méthode de Jason R. C. Patterson s'attelle à déterminer des pondérations de domaines de valeurs prises par les variables :

• Chaque variable peut prendre un certain nombre de valeurs contenues dans un domaine ou alors constituant un ensemble.

De manière formelle, on détermine pour chaque variable les affectations qui en donnent la valeur en sortie. Ces valeurs de sortie représentent la connaissance qu'on a des valeurs de la variable et sont normalement sous forme d'un lattice de plusieurs niveaux. Le plus haut point de ce lattice représente une valeur maximale indéterminée, les points en dessous représentent des propriétés connues de la variable et enfin le point le plus bas indique que la valeur contenue dans la variable ne peut être prédite à la compilation. Par des opérations ensemblistes sur ce lattice, les domaines de valeurs sont déterminés [18].

• A chacun de ces domaines est associée une pondération ou un coefficient.

Le coeur de la méthode est la propagation des différentes pondérations associées aux variables, propagation qui est effectuée le long du graphe de contrôle de flots (CFG). Les branchements qui sont contrôlés par une variable donnée pourront être efficacement prédits en tenant simplement compte de la pondération qui y correspond. De fait, la propagation de domaine de valeurs produit les probabilités selon lesquelles les branchements s'effectuent : les arcs sortant de chaque branchement conditionnel sont marqués par leur probabilité et cette information est très intéressante pour la prédiction.

Voici, à travers un exemple simple, comment sont déterminées les probabilités de branchement.

Exemple 4.3.2 *on a le bout de code suivant :*

```
for (x = 0; x > 10; ++x)
{
    if (x > 7){y = 1;}
    else {y = x;}
   ...
    if (y == 1) { ...Bloc A ...}
   ...
 }
```

On désire calculer la probabilité que le bloc A soit exécuté ou encore la fréquence d'exécution du bloc A, et on procède en faisant les observations et les calculs ci dessous :

- les valeurs de x sont comprises entre 0 et 9
- le test $x > 7$ sera positif pour $x = 8$ et $x = 9$ i.e. 2 fois sur 10, donc 20% du temps
- si le test est positif alors y vaudra 1
- si le test est négatif y prend ses valeurs entre 0 et 7 et donc on a 12.5% de chance que y soit égal à 1
- à la fin y a $(0.2*1+0.8*0.125) = 30\%$ de chances de valoir 1, ce qui donne la probabilité que le bloc A soit exécuté

Avantages et limites

Cette approche a l'avantage de baser la prédiction sur des probabilités calculées, au lieu de se baser sur de trop simples heuristiques qui décrètent une branche prise (OUI) ou non prise

(NON) ; chaque sortie d'arc est donc marquée par la probabilité qu'elle soit prise ou non (entre 0 et 1). De plus, elle fournit des résultats très proches de ceux qui résultent d'une analyse dynamique préalable, en enlevant toutefois les contraintes imposées aux programmeurs.

Cependant, bien que la prédiction soit faite de manière linéaire, elle nécessite beaucoup d'efforts d'implémentation et des calculs complexes [18] ; par ailleurs, dans les cas où le domaine de valeurs ne peut être déterminé de manière statique, les heuristiques qui sont présentées par Jason R. C. Patterson comme non pertinentes doivent bel et bien être utilisées.

4.3.2 Prédiction statique basée sur des heuristiques

Nous nous sommes intéressés particulièrement aux travaux de Youfeng Wu et James R. Larus [22], lesquels travaux reprennent et complètent de nombreux autres que nous nous contenterons simplement de mentionner.

Description de la stratégie

A cause des exécutions spéculatives, le compilateur doit déterminer le chemin susceptible d'être emprunté par le programme. Le but principal de la méthode est de déterminer, parmi les nombreux chemins possibles pour une exécution spéculative, ceux qui sont les plus sûrs en terme de précision dans la prédiction. La stratégie part du constat selon lequel, si on dispose d'une évaluation de la fréquence d'exécution de chaque portion de code, on peut s'en servir pour étudier les coûts et bénéfices des différents choix de chemins possibles, pour enfin en prendre un.

Les auteurs donnent un algorithme qui évalue statiquement ces informations en trois étapes :

• Evaluation des probabilités des branchements.

Une probabilité de branchement est une estimation de la probabilité qu'une branche sera prise.

Le *hit ratio* d'une heuristique (fréquence à laquelle une heuristique est vérifiée) étant une bonne estimation de la probabilité qu'une branche prédite sera prise au moment de l'exécution, on se sert des heuristiques présentées par Thomas Ball et James R. Larus dans [21]. Chacune de ces heuristiques est perçue comme une expérience binaire et *on assimile la probabilité qu'une branche soit prise à la fréquence à laquelle cette branche est correctement prédite.*

Or, plusieurs heuristiques peuvent s'appliquer simultanément à une branche et à chacun des *hit ratio* correspondants équivaut une certaine probabilité. Pour avoir une meilleure estimation de la probabilité de cette branche, on combine toutes les probabilités possibles grâce aux outils mathématiques fournis par la théorie de Dempster-Shafer [15].

• Obtention des fréquences locales de blocs de base et des arcs.

Une fréquence de bloc ou de branche est une mesure du nombre de fois qu'un bloc est exécuté ou qu'une branche est choisie.

On propage les probabilités des branchements obtenues ci- dessus le long du graphe de contrôle de flots d'une procédure. La fréquence d'un bloc b_i (notée $bfreq(b_i)$)est la somme des

fréquences des arcs y afférant. La fréquence d'un arc $(b_i \rightarrow b_j)$(notée $freq(b_i \rightarrow b_j)$) est le produit de la fréquence d'exécution de b_i par la probabilité de l'arc $(b_i \rightarrow b_j)$.

• Calcul des fréquences globales des blocs et des arcs.

On commence par déterminer les fréquences des fonctions et des appels de fonctions, vu que :

fréquence globale d'un bloc = fréquence locale du bloc
× fréquence d'exécution de la fonction qui la contient

Il en va de même pour la fréquence globale d'un arc.
La fréquence globale d'invocations, elle, s'obtient en propageant les fréquences locales le long du graphe des appels des fonctions.

Avantages et limites

Les expériences effectuées sur les jeux d'essai en calcul entier SPEC92 et les applications Unix montrent que les blocs les plus fréquemment exécutés, les arcs et les fonctions identifiés par ces techniques correspondent assez à ceux qui seraient donnés par profiling.

Seulement, on est parti de la présomption selon laquelle plusieurs branchements de programme ne dépendent pas des données en entrée, ce qui est discutable ; en effet, une analyse statique pourrait déterminer que les données d'entrée n'influenceront pas les branchements qui suivront, mais on peut avoir des cas ou statiquement on n'en sait rien.

En outre, les heuristiques utilisées reposent sur des branchements à deux cas possibles, et on ne sait quel serait leur comportement pour des boucles avec des structures de type "switch".

Enfin, les appels indirects de fonctions sont difficiles à analyser statiquement, et donc l'usage des pointeurs nécessitera que de nouvelles techniques d'analyse soient proposées.

De manière générale, la prédiction statique de branchement est peu coûteuse parce qu'elle ne requière pas de matériel supplémentaire, ce qui n'est pas le cas de la prédiction dynamique.

4.4 Prédiction dynamique

Le processeur dispose du BPU pour la prédiction dynamique de branchement. La BPU est l'unité de prédiction de branchement.
A l'exécution, le processeur garde une table (mémoire cache) des derniers branchements effectués (table avec les adresses des branchements et leurs destinations) et une information aidant la prédiction ; cette information peut être par exemple : " *est ce que le branchement a été pris la dernière fois ?*". Ainsi le matériel conserve durant l'exécution des informations sur le comportement passé des branchements, de façon à prédire leur comportement futur ; cela nécessite du matériel supplémentaire et s'avère donc coûteux.
Le processeur dispose également d'un algorithme mathématique basé essentiellement sur des statistiques et deux tableaux contenant les informations sur les parties de programme déjà exécutées.

4.4.1 Description des tables

Branch Target Buffer (tampon des branches cibles)

C'est une table qui contient les adresses mémoires des branches d'un programme. C'est une sorte de mémoire cache mais qui contient plutôt des adresses mémoires au lieu des données ou des instructions comme les caches L1 ou L2.

Branch History Table (table de l'historique des branchements)

Cette table sert à retracer les décisions de choix de tel ou tel branchement. Son organisation peut varier en fonction de l'algorithme mathématique de prédiction de branchement. Le cas le plus simple consiste en un codage sur 2 bits pour chaque branche de programme, chaque valeur informant sur la qualité de la prédiction de branchement.

Exemple 4.4.1 *On peut avoir les correspondances suivantes :*
00 : prédiction non connue
01 : Prédiction faible
10 : prédiction forte
11 : Prédiction sûre

Ainsi dans la prédiction dynamique de branchement, le processeur tente de prédire le branchement à faire au moment même où il traite une instruction de test. C'est parce que les possibilités de cette prédiction sont bien plus importantes que celles de la prédiction statique qu'on a intégré cette technique dans les processeurs. Les recherches cependant continuent dans d'autres directions, et nous signalons à cet égard la réduction des interférences sur les tables de prédiction de branchement [16] et la prédiction des branchements indirects [17].

La prédiction de branchement joue un rôle très important dans la production de codes optimisés, et de plus en plus de techniques sont développées pour assurer qu'elle soit le plus pertinente possible. Pour atteindre cet objectif, de nombreux travaux de recherche se sont intéressés au comportement effectif (par opposition au comportement prédit) des programmes en cours d'exécution ; c'est la raison pour laquelle, dans le foisonnement des techniques rencontrées, force a été de constater qu'une autre méthode d'analyse de programmes était utilisée au préalable, à savoir l'analyse dynamique ou *profiling*. Le chapitre qui suit est consacré à ce type d'analyse qui par ailleurs s'est avérée utile dans des domaines infiniment variés, bien au delà de la seule prédiction de branchement.

Chapitre 5

L'analyse dynamique

Un des objectifs majeurs de l'optimisation des programmes est de gagner du temps CPU, et les optimisations classiques (copie et propagation) réduisent le temps d'exécution en éliminant les parties de code redondantes ; les autres types d'optimisation (élimination de variables inductives dans les boucles, invariant de boucles) déplacent les instructions des zones les plus fréquemment exécutées vers les zones les moins fréquemment exécutées. Il faut donc trouver les points chauds (zones de code utilisant beaucoup de temps CPU) et éliminer les goulets d'étranglement (zones de code utilisant inégalement les ressources).

Afin d'identifier les zones dont il est question, de reconnaître les parties de code redondantes ou les parties qui ne sont que très rarement exécutées, on effectue une analyse dynamique des programmes, toujours en vue de la production de code optimisé. Indépendamment de l'analyse statique qui part du code source du programme pour déterminer des optimisations à effectuer, l'analyse dynamique étudie le programme alors qu'il est en cours d'exécution.

Définition 5.0.1 *L'analyse dynamique consiste en l'observation du comportement du programme au cours de son exécution, afin de déterminer les portions de code les plus susceptibles d'être optimisées.*

Ainsi, il est question d'étudier le comportement effectif des programmes. Les outils d'analyse dynamique, en donnant des informations sur ce que font les programmes, facilitent le travail d'optimisation à réaliser sur ceux-ci.

5.1 Le profiling

5.1.1 Définition et généralités

Le *profiling* (l'analyse dynamique) des programmes est une technique d'analyse permettant de déterminer, à l'exécution, les causes de mauvaises performances de programmes et donc les zones du code où l'optimisation serait le plus rentable. Le but est d'obtenir, à partir d'un programme donné et par des transformations adéquates, une version susceptible d'être exécutée de manière optimale, et ce pour une architecture précise. Ici, les transformations à effectuer sont déduites de l'observation du programme en cours d'exécution et cette observation consiste en un comptage dynamique des événements tout au long de cette exécution. De

nombreuses caractéristiques peuvent faire l'objet d'une analyse dynamique, parmi lesquelles

- le temps CPU
- les accès mémoire
- le graphe des appels
- les blocs de base
- les entrées/sorties
- le nombre de cycles CPU pour chaque ligne d'instructions dans un ou plusieurs sous programmes.

En identifiant les portions de programmes les plus fréquemment exécutées, le profiling indique aux programmeurs et aux compilateurs les lieux les plus propices à des optimisations. Il s'agit donc par cette analyse de générer une vue globale montrant les relations entre les blocs de base des programmes, puis plus tard, lors d'un autre profilage ou d'une compilation, de se concentrer sur les parties nécessitant un travail supplémentaire, dans une perspective optimisante. C'est un processus itératif, qui présente plusieurs phases généralement regroupées en trois : l'instrumentation du code, la compilation et l'exécution du code instrumenté.

5.1.2 Phases du profilage

L'instrumentation du code

Il est question ici d'"équiper" les codes sources pour le profilage, i.e. y ajouter des informations particulières devant guider le profilage en lui-même.

Exemple 5.1.1 *on a le bout de code :*

```
if (b > c)
  t = 1 ;
  else
  b = 3 ;
```

le code instrumenté est le suivant :

```
if (b > c) {
 bb[0]++ ;
  t = 1 ;
   }
  else{
 bb[1]++ ;
  b = 3 ;
   }
```

Une application instrumentée paraît inchangée pour l'utilisateur, et après son exécution, des données sont recueillies. La version instrumentée du programme doit donc être exécutée

sur une ou plusieurs entrées "significatives" du programme, afin de collecter en sortie des informations de profilage.

L'instrumentation de code peut se faire grâce aux compilateurs et aux éditeurs d'exécutable :

- les compilateurs procèdent par insertion d'opérations extra dans le code ; ils requièrent pour cela les sources du programme et l'instrumentation produite peut donner lieu à une recompilation ;

- les éditeurs d'exécutable insèrent les informations d'instrumentation après l'édition des liens : ils ne nécessitent donc pas de connaître les sources du programme et l'instrumenté produit est difficilement rattachable au code source.
  Exemple d'outils d'instrumentation (basés sur l'exécutable) : hiprof, pixie, 3rd.

L'instrumentation peut être utile non seulement pour le profiling, mais aussi pour la modélisation de nouvelles architectures (déterminer le miss ratio d'un nouveau cache, déterminer ce que fera un nouveau prédicteur de branchement) et pour l'optimisation des programmes.

La compilation et l'exécution du code instrumenté

L'exécution s'effectue sur une ou plusieurs données d'entrées correctement choisies, et il y a, en sortie, génération de plusieurs fichiers de statistiques.

La visualisation et l'analyse des statistiques

L'arbre des appels des fonctions est analysé, le pourcentage de temps CPU utilisé est calculé, le nombre d'appels à chaque fonction également, ...
C'est ainsi qu'on obtient les données de profilage.

Recompilation du programme

Grâce aux données de profilage précédemment collectées, on procède à une nouvelle compilation du programme pour produire du code optimisé.
Tout ceci peut être résumé comme suit :

FIG. 5.1 – Les phases du profiling

Il est à noter que l'on peut être amené à reprendre toutes les étapes ci-dessus avant de pouvoir obtenir un code optimisé satisfaisant.

Les différents éléments observés lors du profiling le sont de plusieurs manières, en fonction de la technique de profilage utilisée, et les données collectées sont également de plusieurs types.

5.1.3 Techniques d'instrumentation et données de profilage

Par l'observation du programme en cours d'exécution, le profiling permet d'effectuer plusieurs sortes d'optimisations, lesquelles dépendent fortement du type d'informations recueillies ; pour cela il existe plusieurs types de données de profilage, à savoir les données issues du profiling du graphe de contrôle de flots, les données issues du profiling des valeurs et les données issues du profiling des adresses mémoires. Chacune de ces données peut être recueillie par l'une des deux techniques que sont l'échantillonnage et le traçage.

L'échantillonnage

Le programme (non modifié) est interrompu fréquemment, et le compteur de programme (PC) ou la pile d'appels sont enregistrés lors de ces interruptions, avec pour but de déterminer dans quelles procédures, dans quelles lignes de code le programme passe le plus de temps, et les régions du code qui sont exécutées le plus fréquemment.

Le traçage

Ici, une copie de l'exécutable est modifiée afin d'y insérer des instructions de traçage, en général à la fin de chaque bloc élémentaire d'instructions.
Durant l'exécution, le nombre exact d'utilisations de chaque bloc de base est comptabilisé, et ainsi on obtient un profil fidèle du comportement du programme. A partir des mesures effectuées, divers compte rendus peuvent être générés. C'est par cette technique que s'effectue le profiling du graphe de contrôle de flots.

Définition 5.1.1 *Un bloc d'instructions de base est un ensemble d'instructions à une entrée et une sortie sans instruction de branchement inconditionnel (goto, return,...)*

Données issues du profiling

Grâce à l'échantillonnage on peut profiler des valeurs et des adresses mémoires :

- **Données issues du profiling des valeurs**
  Il est question ici d'identifier les valeurs spécifiques rencontrées comme opérandes d'instructions, ainsi que les fréquences auxquelles ces valeurs sont rencontrées. Grâce à ces informations, l'on peut déterminer les opérandes dont la valeur est toujours constante, identifier les codes invariants dans les boucles, ce qui permet des optimisations telles que la propagation de constantes, la réduction forte, ...

– **Données issues du profiling des adresses mémoires**
Ces données sont sous forme d'ensembles d'adresses mémoires référencées par un programme. Elles sont utiles pour effectuer la disposition des données en mémoire et les transformations par placement de code, toutes choses qui améliorent de manière sensible l'exploitation de la hiérarchie des mémoires. En effet, les programmeurs organisent leurs structures de données de manière logique et mettent ensemble des objets qui sont logiquement liés entre eux ; cependant, cette relation logique peut être différente de l'ordre dans lequel on accède effectivement aux structures de données au moment de l'exécution du programme. Grâce aux informations du profiling d'adresses mémoires, on peut réorganiser convenablement le placement des objets ou le placement des champs à l'intérieur d'une donnée structurée, de telle manière que les données auxquelles l'on accède le plus souvent soient placées côte à côte. c'est cet aspect que nous exploitons dans le chapitre 6 de ce mémoire.

– **Données issues du profiling du graphe de contrôle de flots (CFG)**
On capture ici la trace du chemin d'exécution que prend le programme. Cette trace représente l'ordre dans lequel les noeuds (correspondant aux blocs de base dans le graphe de contrôle de flots du programme) sont visités. Ce profiling se note CFT, i.e. trace du contrôle de flots. Par l'examen d'un CFT, on peut calculer la fréquence d'exécution d'un sous chemin donné du programme.

Un exemple de CFG et de CFT est donné à la figure 5.2.

FIG. 5.2 – Exemple de CFG et CFT

Exemples de profiling du CFG
Vue la taille excessive que nécessiterait le stockage des CFT en machine, l'on se contente des approximations de CFT parmi lesquelles on retrouve :

– Le profiling des noeuds.

Ceci donne la fréquence d'exécution des blocs de base dans le CFG.
- Le profiling des arcs.
  Ceci donne la fréquence d'exécution de chacun des arcs du CFG.
- Le profiling de deux arcs.
  Ceci donne la fréquence d'exécution de chaque paire d'arcs consécutifs dans le CFG.
- Le profiling des chemins.
  Celui-ci enfin donne la fréquence d'exécution des sous chemins acycliques [1] dans le CFG et ces sous chemins sont intraprocéduraux.

5.2 Le profiling dans quelques optimisations classiques de code

Les algorithmes simples d'optimisation opèrent sur des instructions qui ont été déterminées comme optimisables lors d'une analyse statique ; d'autres algorithmes plus performants opèrent sur des instructions conditionnellement optimisables, et qui ont été ainsi déterminées soit au moment de l'exécution, soit par analyse statique. Ce faisant, ces algorithmes introduisent du nouveau code dans les programmes (ils font de la réplication de code ; on peut citer en exemple l'inlining qui consiste à remplacer un appel de procédure par tout le corps de la procédure à cet endroit) et créent des copies optimisées et des copies non optimisées de ces codes. En fonction des conditions dans lesquelles on se trouve, les copies appropriées de code sont exécutées ; ce procédé est appelé *spécialisation de code.*

Définition 5.2.1 *La spécialisation de programmes à l'exécution est une technique d'optimisation qui permet d'exploiter des informations connues uniquement au moment de l'exécution.*

Il est généralement question ensuite d'effectuer une élimination des copies (élimination des redondances, élimination des codes "morts") ou encore de produire du code plus simple et plus efficace (réduction forte, propagation de constante).
Pour permettre que la spécialisation de code soit effectuée dans le sens d'obtenir des programmes optimisés et donc pour éliminer l'effet de la réplication de codes, deux classes de transformations sont utilisées : le déplacement de code *(code motion)* et la restructuration du graphe de contrôle de flots.

5.2.1 Techniques de transformation de code

Le déplacement de code *(code motion)*

Le déplacement de code, en plus d'assurer le respect des dépendances contenues dans le programme, garantit qu'à chaque instruction exécutée dans le code optimisé corresponde une instruction dans le code non optimisé. Par conséquent, si une exception survient lors de l'exécution du code optimisée, c'est qu'elle serait pareillement survenue lors de l'exécution du code non optimisé.

[1] Un sous chemin acyclique ne comporte pas d'arcs retour à l'intérieur d'une boucle.

Il existe deux sortes de déplacement de code : le déplacement spéculatif de code *(spéculative code motion)* et le déplacement prédicaté de code *(predicated code motion).*

- Le déplacement spéculatif de code permet au compilateur d'introduire des instructions à exécuter dans le code optimisé alors que ces instructions n'étaient pas présentes dans le code non optimisé. Pour que ceci soit bénéfique, le coût des instructions supplémentaires doit être très inférieur au gain apporté par la transformation. C'est pourquoi on a besoin à ce niveau d'une analyse de coût/bénéfice basée sur le profiling.

- Le déplacement prédicaté de code permet d'effectuer bien plus librement les transformations : les instructions peuvent être déplacées des structures de contrôle et être exécutées à des endroits différents sous les conditions de départ, en faisant une exécution prédicatée avec des expressions construites de manière appropriée. L'on se servira à ce niveau du profiling pour déterminer si le bénéfice obtenu par la transformation est supérieur au coût de l'évaluation de l'expression prédicatée.

La restructuration du graphe de contrôle de flot

Si les conditions sous lesquelles une instruction peut être optimisée sont satisfaites seulement le long de certains chemins d'exécution et non suivant les autres chemins, alors la restructuration de CFG peut être effectuée pour rendre possible cette optimisation. Grâce à la restructuration on crée un programme tel que lorsque l'on atteint l'instruction concernée, et en fonction de l'arc par lequel on a abouti à cette instruction, on peut déterminer si l'instruction peut être optimisée ou non. On peut faire des copies optimisées et des copies non optimisées de cette instruction et les placer le long de l'arc entrant sur le bloc de base correspondant à l'instruction. Le coût majeur de cette technique vient de l'accroissement de la taille du code. Là encore on a besoin d'une analyse de coût/bénéfice basée sur le profiling pour déterminer s'il y a lieu ou non d'effectuer cette transformation.

Il apparaît donc que le guidage de l'optimisation avec un profil permet la réduction additionnelle de croissance de code en choisissant des duplications dont le coût est justifié par des gains suffisants en temps d' exécution. Pour illustrer ce qui précède nous verrons le cas de quelques optimisations effectuées grâce au déplacement de code mais aussi grâce à la restructuration du CFG.

5.2.2 Quelques optimisations de code avec illustration des techniques de transformation.

Élimination partielle de redondances (*PRE*)

Cette optimisation est effectuée généralement par le déplacement de code. Il est question de supprimer des instructions qui calculent des expressions dont on dispose déjà la valeur. Un exemple de redondance partielle est donné à la figure 5.3. On s'aperçoit que l'évaluation de l'expression $x + y$ dans le noeud 7 est partiellement redondante car si le noeud 2 est visité

FIG. 5.3 – (a) Redondance partielle

avant d'atteindre le noeud 7, alors l'expression a déjà été évaluée et n'a plus besoin de l'être à nouveau.
En utilisant le déplacement spéculatif de code comme à la figure 5.4, la redondance partielle de l'expression $x+y$ est nettement réduite, à condition que la fréquence d'exécution du noeud 3 soit inférieure à celle du noeud 7.
En utilisant la restructuration du CFG comme à la figure 5.5, la redondance est entièrement éliminée. Seulement, le code s'est sensiblement accru et on a besoin d'une analyse de coût/bénéfice basée sur profiling pour choisir quelle optimisation effectuer en définitive.

FIG. 5.4 – PRE avec déplacement spéculatif de code

FIG. 5.5 – PRE avec restructuration de CFG

Élimination de code mort partiel (*PDE*)

On obtient la *PDE* d'une instruction en retardant l'exécution de cette instruction jusqu'à un point où son exécution est réellement nécessaire. Sur la figure 5.6, l'affectation du noeud 2 est partiellement morte car si le contrôle passe par les noeuds 6 ou 7, alors la valeur calculée en 2 ne sera jamais utilisée.
En utilisant le déplacement prédicaté de code comme à la figure 5.7 on peut déplacer cette expression au noeud 8 et atteindre ainsi la *PDE*. Ceci n'est intéressant que si la fréquence d'exécution du noeud 8 est plus petite que celle du noeud 2.
En utilisant la restructuration du CFG comme à la figure 5.8 on réalise une bonne *PDE* et

à nouveau une analyse de coût/bénéfice basée sur le profiling est nécessaire pour déterminer laquelle des précédentes optimisations appliquer.

FIG. 5.6 – Code partiellement mort

FIG. 5.7 – PDE avec déplacement prédicaté de code

FIG. 5.8 – PDE avec restructuration de CFG

L'élimination de branches conditionnelles, des analyses plus exhaustives de ces optimisations, ainsi qu'un algorithme de *PRE* basé sur le profiling peuvent être consultés dans [3].

5.3 Autres optimisations déduites du profiling

Grâce aux informations obtenues du profiling, il est possible de se concentrer sur des portions précises du code en vue de les optimiser. A la base, le profiling donne un comptage de blocs de base, des arcs, des arcs d'appel, et de manière avancée, il permet un profil des chemins, le développement de nouveaux prédicteurs de branchements et une réduction des défauts de cache.

5.3.1 Identification des parties coûteuses d'un programme

Pour améliorer les performances d'un programme, il faut en déterminer la consommation en ressources machine. A chaque point, la ressource limitante peut être différente et limite ainsi la vitesse de l'exécution ; ces ressources peuvent être la vitesse du processeur et sa disponibilité, les entrées/sorties, la taille mémoire et sa capacité, les bugs. Les programmes ont des

comportements différents pendant différentes phases de leur exécution : il faut donc identifier la ressource limitante de chaque phase (les lectures, les calculs, les écritures ...). Après avoir identifié la ressource limitante pour chaque phase, on peut faire une analyse profonde pour trouver le problème ; ensuite on peut s'intéresser à de nouveaux problèmes dans la même phase ou ailleurs...

- Une exécution limitée par le CPU passe le plus clair de son temps dans le CPU et en est limitée par la vitesse ou la capacité ; les améliorations possibles consisteront alors à modifier l'algorithme, réordonner les codes et éviter les blocages, supprimer les boucles superflues, appliquer le blocking pour conserver les données en cache ou dans les registres ou même changer d'algorithme.
- Une exécution limitée par les entrées/sorties est telle que le programme attend des entrées/sorties pour se terminer et peut être limité par la vitesse des accès disques ou les mémoires cache ; les améliorations possibles sont : optimisation de l'usage des données pour minimiser les accès disques, compression de données, ...
- Une exécution limitée par la mémoire est telle que le programme fait de fréquents swap out des pages mémoires (défaut de TLB, de cache) ; on peut améliorer les accès mémoire en améliorant la localité des références. On peut aussi diminuer, le cas échéant, la taille mémoire utilisée par le programme.
- Une exécution peut être limitée par des bugs : le programme lit incessamment la même valeur dans le même fichier ou alors il y a un "floating point exception" qui ralentit le programme. Il peut également subsister un ancien code non enlevé complètement (mises à jour inachevée) ou encore un manque de mémoire pour le programme causé par des *malloc()* non suivis de *free()*

Telles sont les optimisations de base auxquelles peut donner lieu un profiling. De manière plus avancée, il est possible de faire du profilage de chemin pour la prédiction de branchement.

Les différentes autres optimisations déduites du profiling sont celles qui sont effectuées dans le cadre des observations des accès mémoire.

L'analyse de la localité des programmes étant essentielle pour obtenir de bonnes performances sur les machines dotées d'une hiérarchie mémoire, il faut détecter les réutilisations potentielles de données et caractériser les parties de structures de données sujettes à des réutilisations. Ces informations sont ensuite utilisées dans la transformation des programmes afin d'améliorer l'utilisation de la hiérarchie mémoire. Ce type de transformation est essentiel pour une utilisation efficace des processeurs actuels dont le goulet d'étranglement principal est constitué par les accès à la mémoire.
L'espace d'adressage d'un programme peut être divisé en trois parties : la pile, le tas et l'espace global. Les données qui utilisent les piles font une utilisation optimale du cache ; dans les autres cas, on se sert d'une allocation statique des données globales (temps de compilation) et d'une allocation dynamique des données structurées (temps de compilation + temps d'exécution), afin de pouvoir améliorer l'utilisation des caches. Parmi les techniques générales [3] d'optimisation du cache, l'on recense :

- Le placement des données *(Object placement).*

Cette technique utilise les données issues du profiling des adresses mémoires ; elle permet de déterminer le placement des données en relations les unes avec les autres, pour améliorer les comportements de la mémoire cache. La migration des données [2] et le clustering sont des exemples de technique de placement de données.
- La disposition des données en mémoire *(Object layout.)* cette technique détermine la disposition des champs d'une structure de donnée, de manière à ce que les champs les plus fréquemment utilisées se retrouvent les uns à côté des autres.
- Disposition et placement des données. C'est une combinaison des deux techniques précédentes. Exemple : la coloration.
- La compression des données.

Nous nous pencherons sur l'analyse de la localité des accès à la mémoire dans les programmes irréguliers. Les structures de données dont les accès sont inefficaces sont donc recherchées. A partir de ces informations, des méthodes d'amélioration de rangement des données en mémoire sont développées.

5.3.2 Observation des accès mémoire pour un meilleur placement

Les pointeurs sont utilisés pour stocker les données dans de nombreux langages comme C, C++, Simula, Pascal et dans les systèmes d'exploitation ; l'accès y est moins régulier que dans le cas des tableaux et les zones mémoires sont allouées en tas. C'est ainsi que les techniques visant à réduire les latences pour l'utilisation des tableaux (accès aléatoire et uniforme aux données, analyse de dépendances possible) sont inefficaces dans le cas des pointeurs (transparence de localité, toute transformation pour réordonner le code peut sensiblement modifier le sens du programme). Les travaux de T. Chilimbi et al. [2][1] sont un début de présentent une solution à la réduction des latences pour les programmes manipulant les pointeurs.

L'idée principale est d'optimiser l'utilisation des mémoires caches en effectuant dessus un meilleur placement des données pointées et ce placement s'obtient après une observation des différents accès à la mémoire. Il est ainsi question de réorganiser la disposition en mémoire des structures de données dynamiques, ce qui est possible parce qu'on opère sur des pointeurs : les données peuvent être placées de façon arbitraire dans la mémoire et donc dans le cache. Dans un premier temps, on établit que des techniques telles que le groupage, la compression et la coloration permettent d'améliorer manuellement les performances du cache et ce sont ces techniques que nous présenterons ; dans un second temps on développe des outils semi-automatiques et automatiques pour effectuer ce même travail.

•Amélioration des performances du cache

Les trois types de défaut de cache peuvent être réduits par des placements judicieux des données dans le cache ; les *cold miss* par exemple peuvent être évités par des techniques de préchargement et lorsqu'une ligne de cache est chargée par des données accédées dans le même temps, on a fait sur ces données un préchargement implicite.

[2]Lors de l'allocation des objets structurés, on utilise les outils mémoire standards (*malloc*) ; au moment de l'exécution, on effectue une migration de ces objets de leur localisation de départ vers emplacements qui améliorent la localité

FIG. 5.9 – préchargement implicite

Les défauts liés à la capacité du cache peuvent résulter d'une allocation semblable à celle de la figure 5.10, et en regroupant les mots dans les lignes du cache, on peut réduire les défauts de capacité.

FIG. 5.10 – Chargement aléatoire

FIG. 5.11 – Utilisation d'un set

Le dernier type de défaut de cache est celui qui est lié à l'associativité du cache : une faible associativité induit de nombreux défauts de ce type ; en allouant les données accédées dans le temps sur des lignes différentes dans le cache, ce défaut se trouve réduit.

FIG. 5.12 – résolution de conflit de cache

Pour améliorer les performances du cache, les techniques qui donnent des résultats intéressants notamment dans le cas de placement en mémoire de données structurées, de leur disposition mémoire utilisent en général le clustering, la coloration et la compression des données.

- Clustering
  Il s'agit d'une technique visant à regrouper dans une ligne de cache les éléments qui sont susceptibles d'être accédés dans le même temps. Cette technique améliore la localité spatiale et la localité temporelle et permet aussi d'effectuer un préchargement implicite.

Sur un arbre, par exemple, on peut réaliser le clustering en regroupant tous les éléments d'un sous arbre dans une ligne du cache.
Le clustering nécessite une intervention partielle du programmeur dans la gestion de la mémoire, ce qui alourdit sa tâche ; par exemple pour l'allocateur d'espace mémoire *cc-malloc* proposé par [1, 2], le programmeur doit fournir la taille des données à regrouper et un pointeur sur une donnée existante qu'il sait être susceptible d'être utilisée dans le même temps.

– Coloring
Cette technique vise à associer aux éléments accédés dans le même temps une région du cache ; cette région est constituée d'un ensemble de lignes pour lesquelles les défauts ne se produisent pas. Ce mappage assure donc que les les éléments les plus souvent accédés ne sont pas en conflit et qu'ils ne sont pas remplacés par ceux qui sont le moins souvent accédés.

– Compression
Compresser les éléments de la structure de données permet que plus d'éléments puissent être regroupés dans une ligne de cache. Ce faisant, on réduit aussi bien les défauts de capacité que les défauts de conflit. La compression nécessite cependant que le processeur intervienne dans le décodage de l'information compressée, mais, vu les coûts d'accès à la mémoire, cela est préférable à de nouvelles références mémoire. Les techniques de compression des structures de données dynamiques incluent à la fois l'encodage des données et les techniques d'encodage des structures telles que l'élimination des pointeurs.

Les améliorations apportées par la conception manuelle des structures de données efficaces pour les pointeurs ne masquent pas la difficulté de cette approche. En effet, l'application de ces principes nécessite une maîtrise parfaite du code des applications et de leurs structures de données ; elle nécessite également des connaissances de l'architecture de la machine sur laquelle on travaille, chose avec laquelle plusieurs programmeurs sont peu familiarisés. Il faut de plus passer à une réécriture du code et pour des milliards de ligne de logiciels, cela n'est pas envisageable. En guise de solutions à ces difficultés, T. Chilimbi et al ont conçu, mis en application et évalué des stratégies semi-automatiques et parfois automatiques [1, 2] pour produire les mêmes dispositions obtenues avec les techniques manuelles.

5.4 Quelques exemples d'outils de profiling

Il y en a plusieurs. Ils permettent d'étudier les performances des codes en donnant des statistiques sur les parties du code consommatrices de temps CPU, et sur les accès mémoire. On peut même avoir un diagnostic sur les pertes de performances, toujours dans le but d'optimiser le code.

5.4.1 SpeedShop

SpeedShop est un ensemble d'outils permettant d'effectuer le profilage d'applications. Ces outils d'analyse de performance, après des expérimentations sur les codes exécutables, examinent les résultats obtenus et génèrent des information de profiling.
SpeedShop fonctionne sous IRIX 6.2, et suivants, ou sur des exécutables compilés par par IRIX 6.2 (o32, n32 et 64), ou avec les compilateurs MIPSPro 7.0 (n32 et 64). SpeedShop fonctionne avec des programmes écrits en C, C++, FORTRAN, ADA. Ces programmes doivent utiliser les librairies (DSOs).

SpeedShop propose trois techniques différentes d'analyse :

- 1. Sampling (échantillonnage) : le programme est interrompu fréquemment, le compteur et la pile du programme sont enregistrés lors de ces interruptions. SpeedShop peut utiliser l' horloge du calculateur ou n'importe quel compteur matériel du processeur R10000 comme base de mesure temporelle. Différentes bases fournissent différentes informations à propos du comportement du programme.

- 2. Ideal time : une copie de l'exécutable (fichier binaire) est modifiée afin d'y insérer des instructions de traçage à la fin de chaque bloc élémentaire d'instructions. Durant l'exécution, le nombre exact d'utilisations de chaque bloc élémentaire est comptabilisé. On obtient ainsi un profil fidèle du comportement du programme. Divers comptes-rendus peuvent être générés à partir des mesures effectuées lors d'une telle exécution.

- 3. Exception trace : ce n'est pas exactement une méthode de profilage ; elle enregistre les exceptions flottantes qui surviennent ainsi que leur localisation.

Les deux techniques de profilage, sampling et ideal time peuvent être appliquées à des programmes parallèles aussi facilement qu'à des programmes séquentiels. Chaque thread d'une application possède sa propre information et les histogrammes peuvent être vus individuellement ou bien être fusionnés et donner une vue d'ensemble.

SpeedShop est composé de trois parties : ssrun, ssapi et prof.

ssrun

Les expérimentations de Speedshop sont réalisées par la commande
ssrun ; on établit un environnement pour capter les données de performance pour un exécutable donné, et lorsque ces données sont collectées, elles sont placées dans un fichier qui sera analysé plus tard par la commande *prof*. Ainsi, *ssrun* réalise les expérimentations (exécutions échantillonnées) et collecte les informations. On a :

ssrun -[option_ssrun] program [option_prog]

Quelques exemples d'option :

- usertime : trace le temps (inclusif et exclusif) du programme interrompu toutes les 30 millisecondes. Les données de usertime sont des statistiques et peuvent varier d'une exécution à une autre. Ici on fait du profiling sur la pile et on utilise le temps de l'hor-

loge interne

– [f]pcsamp[x] : fournit des statistiques sur l'échantillonnage de PC effectué sur le code sur une base de temps de 10 millisecondes. Le [f] permet l'échantillonnage sur une base de temps de 1 milliseconde au lieu de 10. Le [x] fait un codage des informations sur 32 bits au lieu de 16. Les données de pcsamp sont des statistiques et peuvent varier d'une exécution à une autre. Ici on échantillonne le PC et on utilise le temps utilisateur et le temps système

– ideal : fournit des statistiques de comptage de blocs basiques exécutés. On a aussi un résumé sur les appels de fonctions et à partir de toutes ces informations, et on peut calculer le temps réel passé dans chaque fonction, voire même le temps inclusif grâce à la commande *prof*.
Sur les machines avec compteurs matériels pour la performance, (R10000 machines), on a aussi

– fpe : trace toutes les exceptions flottantes.

– gi_hwc : fournit des informations statistiques supplémentaires basées sur les compteurs matériels de performances.

ssapi

L'interface de la librairie *ssapi* permet à l'utilisateur de placer des points de marquage qui permettent de profiler des sections spécifiques du programme ou des phases de l'exécution [7].

prof

prof analyse les données recueillies et établit un rapport[7]. On a
prof -[option_prof] ¡output file¿
Les options sont les mêmes que celles de *ssrun*.

Pour plus d'informations, on peut se référer à [5][6][7].

Un autre outil d'analyse : *pixie*

pixie est un outil d'instrumentation qui mesure aussi la fréquence d'exécution des codes dans un programme ; *pixie* lit un exécutable, le partitionne en blocs de base et produit un programme équivalent avec du code additionnel qui effectue le comptage de chaque bloc de base. Il est invoqué à partir de *ssrun* et génère des fichiers qui seront analysés par *prof*.

5.4.2 Perfex

L'outil de profilage le plus simple est **perfex**, il est documenté dans [4]. **Perfex** est un logiciel de comptage d'évènements matériels. Il s'exécute sur une application et génère des

résultats concernant l'exécution ; on a :
L'application *commande* et ses *arguments* sont fournis par l'utilisateur. **perfex** initialise les compteurs événementiels du processeur R10000 et commence l'exécution. Lorsque celle-ci prend fin, **perfex** génère un fichier contenant les valeurs des compteurs matériels activés. Selon l'option choisie, **perfex** fournit soit la valeur exacte de un ou deux évènements soit une valeur approximative des 32 évènements existants. **perfex** effectue son étude sans modification du comportement de l'application et avec un effet limité sur son temps d'exécution.
Ainsi, **perfex** exécute un programme en fournissant un rapport d'événement des compteurs R10000. Il permet d'identifier le type de problèmes lié au manque de performance du programme. perfex $-[option\_perfex]$ program $[option\_prog]$ Les principales options sont :

- $-e$ $e0$: spécifie l'évènement (e0) à considérer .
- $-a$: prend en compte tous les événements (32 au total).
- $-x$: complète l'option -a avec des informations sur les exceptions.
- $-y$: fournit des statistiques sur les valeurs obtenues par les compteurs.
- $-mp$: fournit un rapport par thread aussi bien que pour l'ensemble des threads du programme parallèle.

Dans la pratique, on utilise souvent les 3 options $-a$, $-x$, $-y$; ce qui s'écrit sous la forme : perfex $-a-x-y$ program $[option\_prog]$
Pour analyser une partie du programme, il faut insérer un appel dans le programme pour initialiser les compteurs, un autre appel pour arrêter ces mêmes compteurs et un dernier pour afficher les différents compteurs.

5.5 Profiling et optimisation : avantages et limites

5.5.1 Avantages

Les outils d'analyse dynamique sont des aides précieuses pour le diagnostic d'erreur, la génération d'explications de points délicats d'un langage ou d'un programme, l'optimisation de code ou le "reverse-engineering".
A l'exécution du code, l'on dispose de beaucoup plus d'informations pour pouvoir guider des optimisations de manière efficace, étant donné que l'on sait exactement quels sont les endroits sur lesquels travailler et quelles sont les causes du manque de performance. Cependant, à cause même du fait que le programme soit exécuté à l'avance, les avantages du profiling sont atténués de plusieurs manières.

5.5.2 Limites

- Le profiling nécessite une exécution préalable (de l'instrumenté) qui peut être très longue, et ceci d'autant plus que la version initiale de l'exécutable n'est pas, ou très peu, optimisée ;

- Pour cette raison d'ailleurs, la taille du problème traité par le programme est souvent petite lors de l'exécution du profiling et pour une taille de problème plus importante, les transformations d'optimisation déduites peuvent provoquer une baisse de performance ; ainsi, les informations recueillies par profiling risquent parfois d'être inadéquates, voire même néfastes. De façon plus générale, les expérimentations se font sur des données de petites tailles, et les résultats obtenus ne garantissent pas ceux qui seront obtenus sur des cas concrets.
- L'application d'une optimisation suite à l'observation d'un comportement d'une partie de programme peut avoir une influence importante sur les comportements d'autres parties du programmes ; en effet, l'optimisation d'exécution est un processus expérimental, et beaucoup de techniques qui sont utiles dans une circonstance peuvent avoir l'effet de réduire l'exécution dans d'autres circonstances. Ces différentes relations n'étant pas prises en compte lors de la recompilation, certaines transformations dédiées à des optimisations lors du profiling peuvent avoir un effet contraire et néfaste sur la suite du programme ; c'est pourquoi une répétition des phases de profiling et de recompilation devrait être effectuée, le nombre de répétitions utiles étant indéfini.
- Très liées à l'architecture, les optimisations obtenues pour un processeur donné ne sont plus de mise si le processeur vient à changer.

A ce jour, le profiling se limite à une observation de l'exécution, sans utilisation d'une analyse statique initiale. Pourtant, l'analyse dynamique est un complément significatif de l'analyse statique ; en effet, les informations prédites statiquement peuvent être vérifiées sur la trace des programmes. En outre, ces informations recueillies statiquement au préalable pourraient guider l'observation du profiling afin d'éliminer toute observation inutile, et même au contraire affiner certaines observations utiles à des optimisations pressenties lors de l'analyse statique. Ainsi, il y a lieu de s'interroger sur les intérêts d'une collaboration effective entre analyse statique et analyse dynamique, et c'est d'ailleurs ce qui fait l'objet de la thèse que nous entamerons en décembre prochain. Nous indiquons dans le chapitre suivant quelques perspectives dans lesquelles nous envisageons la suite de ce travail.

Chapitre 6

Collaboration profiling-analyse statique : application au comportement mémoire

Dans cette partie nous désirons présenter quelques idées concernant la collaboration de l'analyse statique avec l'analyse dynamique.

6.1 Le profiling comme complément de l'analyse statique

L'analyse statique à elle seule ne suffit pas pour déterminer toutes les opportunités d'optimisation possibles, notamment celles qui n'apparaissent qu'au moment de l'exécution du programme. Considérons les exemples ci-dessous :

Exemple 6.1.1 *Soit l'instruction suivante :*

$i \leftarrow i + 1$;
De manière statique, il est clair que la valeur de i *est incrémentée de* 1*, mais cette valeur est inconnue ici, car elle dépend des données d'exécution : les données d'entrée, le nombre d'itérations effectuées pour cette instruction, ...*

Exemple 6.1.2 *Soit la boucle suivante :*

for $i = 1$ ***to*** n ***do***
  $A[i] \leftarrow *p[i].val + 1;$

Une analyse statique pourrait détecter cette boucle comme lieu intéressant d'optimisation, mais la valeur de l'objet pointé ne pouvant être définie statiquement, il faudra faire intervenir le profiling pour compléter l'analyse et pouvoir mener à bien l'optimisation.

C'est pourquoi l'on peut se servir du profiling pour détecter de nouvelles opportunités d'optimisation de codes.

Une autre illustration du profiling comme complément de l'analyse statique est donnée à travers des transformations de programmes.

Exemple 6.1.3 *Considérons les trois schémas de programmes ci après :*

1. if () then
2. $z \leftarrow x \times y$
3. else
4. ...
5. endif
6. if () then
7. $w \leftarrow x \times y$
8. else
9. ...
10. endif

FIG. 6.1 – Code original

1. if () then
2. $t \leftarrow z \leftarrow x \times y$
3. else
4. $t \leftarrow x \times y$
5. endif
6. if () then
7. $w \leftarrow t$
8. else
9. ...
10. endif

FIG. 6.2 – Suppression de la redondance

1. if () then
2. $t \leftarrow z \leftarrow x \times y$
3. else
4. $t \leftarrow (y = 1?)\ x : x \times y$
5. endif
6. if () then
7. $w \leftarrow t$
8. else
9. ...
10. endif

FIG. 6.3 – Réduction forte

Le code de la figure 6.1 est susceptible d'être amélioré, car si les deux conditions de test sont évaluées à *true*, alors l'expression $x \times y$ sera calculée deux fois. L'optimisation possible est ainsi déterminée lors de l'analyse statique et le code tranformé est donné à la figure 6.2.

A présent, l'on exécute ce code et on se rend compte par profiling que la variable y de l'instruction 4 prend régulièrement la valeur 1 ; on a ainsi trouvé par profiling une possibilité d'optimisation non déductible statiquement. Le programme optimisé est donné à la figure 6.3, où l'instruction $x \times y$ est effectuée dans une instruction conditionnelle.

6.2 Analyse statique comme préalable au profiling

Le profiling est indispensable à l'optimisation de parties de programme ne pouvant être analysées statiquement, mais il comporte une phase primordiale qui est l'instrumentation du code. Or, la version instruméntée d'un code peut être très longue et est en général très peu optimisée : c'est ainsi que l'exécution préalable du programme peut être excessivement longue, pour ne donner par la suite que de piètres résultats. En effet, ces résultats sont fortement couplés aux choix des données d'instrumentation en entrée du profilage, choix qui peuvent conduire ou non à des optimisations. Cependant, les résultats d'une analyse statique

préalable pourraient fort bien abaisser de manière considérable le taux d'aléa inhérent à l'instrumentation.

Afin d'éliminer toute observation inutile lors de l'analyse dynamique et même d'affiner certaines observations, les informations recueillies statiquement au préalable peuvent être utilisées. Cela est possible grâce au formalisme de l'analyse statique, qui peut ainsi mettre en évidence les interactions entre comportements à l'exécution, ainsi que les parties de code déterminantes pour la performance. Ce faisant, l'analyse statique génèrerait des informations d'entrée de profiling permettant de l'orienter.

6.3 Vers une collaboration des analyses statique et dynamique

Comme nous l'avons vu dans les chapitres précédents, l'analyse statique et l'analyse dynamique, losqu'elles sont utilisées séparément, ont des portées limitées impossible à étendre. Le travail auquel nous désirons nous consacrer dans les prochaines années consiste à élaborer et à exploiter une utilisation collaborative des deux approches, permettant d'étendre la portée des techniques sous-jacentes.

Le scénario de fonctionnement d'un système collaboratif complet pourra être le suivant et tel que représenté à la figure 6.4.

6.3.1 Premières Analyses

L'analyse statique du code source et un premier traçage d'exécution pourront servir à identifier les parties les plus coûteuses d'un code : identification de structures de contrôle statique englobant des structures dynamiques et engendrant un volume important d'opérations et/ou d'accès mémoire ; dénombrement des appels à des blocs de code afin d'identifier les blocs les plus souvent exécutés.

Ces premières analyses serviront principalement à conditionner l'instrumentation du code afin d'affiner les informations obtenues.

6.3.2 Instrumentation du code

L'instrumentation devra pouvoir être automatique ou quasi-automatique. Une phase de pré-compilation consistera à insérer par exemple des instructions de dénombrement de blocs exécutés, de mémorisation de durée de vie de variables, de mémorisation d'adresses mémoire accédées, de fréquences de branchement, de mesures de temps d'exécution, ...On pourra par exemple proposer un langage de directives de compilation entrainant telle ou telle instrumentation : un tel langage donnera également au programmeur la possibilité de construire lui-même aisément sa propre instrumentation. Le comportement à l'exécution du code ainsi obtenu ne devra pas être trop différent du code original, afin d'assurer un degré de fiabilité acceptable des mesures.

Le résultat de l'instrumentation pourra provoquer la création d'un, ou de plusieurs, fichiers des informations recueillies lors d'une, ou de plusieurs, exécutions.

6.3.3 Exploitation des informations de profiling

Les informations recueillies pourront soit conditionner une nouvelle instrumentation, soit directement servir à compléter les premières analyses. L'ensemble final des résultats des analyses devra pouvoir être suffisant à l'application de transformations du programme et d'allocations mémoire. Celles-ci devront être automatiques : le fichier d'informations suite aux analyses servira à produire des instructions de compilation données en entrée du compilateur.

De nombreuses transformations pourront être développées à partir de techniques aujourd'hui connues, mais à ce jour applicables uniquement dans le cadre de l'analyse statique. D'autres nouvelles techniques pourront être proposées.

Dans le cadre d'une réalisation pratique de ce qui précède, nous avonn commencé par nous intéresser à l'optimisation de programmes à structures de données dynamique et plus particulièrement à l'observation de leurs accès mémoires. Nous présentons à la section suivante le problème et les solutions mises en oeuvre, ainsi que les premiers résultats obtenus.

6.4 Début de mise en oeuvre de la collaboration

Tout au long de la partie pratique réalisée, le but principal était le suivant : améliorer les accès mémoire effectués par un code faisant grand usage de structures dynamiques. Ces accès étant assez irréguliers, ils peuvent donner lieu à de nombreux défauts de cache. On est parti de l'idée suivante.

• Les structures dynamiques accèdent à la mémoire de manière aléatoire à priori, mais en étudiant ces accès au moment de l'exécution des programmes, il est possible d'y découvrir une certaine régularité .

• Si on parvient ensuite à modéliser ces accès, i.e., à trouver une fonction qui donne de manière la plus proche possible les adresses des emplacements accédés, on pourra alors reorganiser ces différents accès de telle manière que les données accédées successivement au cours d'une autre exécution du programme soient stockées de manière proche en mémoire.

• Le nombre de défauts de cache étant ainsi sensiblement réduit, on aura accru la localité des données et donc optimisé le programme.

Les différentes phases de cette réalisation ont consisté à trouver des codes qui manipulent des données structurées et nous avons utilisé pour cela les *Numerical Recipes Books* [48] Notre étude se constitue des étapes suivantes :

- analyser statiquement le code pour détecter la ou les structures les plus accédées, i.e les endroits les plus intéressants pour un profiling ;
- instrumenter le code en s'appuyant sur les résulats de l'analyse statique ;
- recueillir dans un fichier les adresses mémoire accédées au cours du profiling ;
- trouver une fonction qui génère ces différentes adresses et donner une courbe comparative des accès par cette fonction et des données du profiling ; cette fonction peut être obtenue

FIG. 6.4 – Schéma général

par interpolation (polynômiale, par des splines, ...) ou par fonction génératrice [46][47]
- en fonction du résultat, réorganiser les accès ;
- comparer le comportement du nouveau code au code initial.

6.4.1 Analyse, instrumentation du code et création du fichier d'adresses mémoire

Les codes choisis, en C, on été repris et instrumentés sur un pentium III 800 Mhz. La compilation a été faite avec une version non optimisée de *gcc*, afin de pouvoir observer tout simplement les accès non optimisés. Une analyse préalable a permis d'identifier les zones les plus exécutées et les variables les plus utilisées, et l'instrumentation a été focalisée à ces endroits. Un extrait de code instrumenté est présenté à la figure 6.5 et il montre également le format dans lequel ont été produites les différentes adresses mémoires. Le deuxième code instrumenté (algorithme multigrid pour la résolution d'équations différentielles partielles) que nous ne présentons pas ici a été traité de manière identique au premier.

6.4.2 Fonctions génératrices et courbes

L'exécution des codes instrumentés ayant permis d'obtenir des fichiers d'adresses mémoires accédées par les programmes, l'on a essayé de déterminer la fonction génératrice associée à ces adresses.

D'après les travaux présentés en [46][47], on peut calculer des fonctions génératrices à partir des premiers termes d'une suite de nombres rationnels, en utilisant le calcul formel. L'idée est d'essayer de trouver une forme explicite pour la série génératrice associée aux termes de la suite, à partir des premiers. Ainsi, pour une suite de nombres rationnels en entrée, la fonction génératrice permet de trouver les nombres devant logiquement suivre. Une fois la fonction génératrice obtenue, un développement de Taylor peut en donner n'importe quel terme, et on peut également obtenir un développement asymptotique du n-ième terme, n quelconque.

Nous avons utilisé les packages *gfun*, *genfunc* utilisables dans Maple V et versions suivantes.

La suite en entrée a été constituée de quelques adresses mémoires relevées dans le fichier obtenu précédemment, et à partir de cette liste, la série à étudier a été obtenue grâce à la procédure *listtoseries*. Par des approximations de Padé efectuées dans *convert(", ratpoly)*, la fraction rationnelle correspondant à la série a pu être calculée. Cette fonction a permis de déterminer, de manière approchée, les adresses mémoires suivant celles de la liste en entrée.

Cependant, on remarque une bonne corrélation entre les adresses obtenues par la fonction génératrice et les adresses effectivement accédées, comme l'illustrent les courbes des figures 6.6 et 6.7. Ainsi, pour les deux programmes que nous avons instrumentés, la modélisation par fonction génératrice est intéressante pour la prédiction des accès mémoire.

Afin d'être convaincu de l'intérêt de cette approche, d'autres modélisations pour d'autres programmes doivent être testées. De plus, une caractérisation des programmes propices au modèle de fonctions génératrices devra être étudiée. Mais nous pouvons d'ores et déjà dire que ces premiers résultats sont encourageants.

```
void tri_bulle_lst(adr_comp *prem)
/* le premier ne sera peut-être plus le même donc passage par adresse */
{
 int ok,base;
 adr_comp prec, actu, suiv, anc;
 anc = *prem;
 base = (int)anc/10000;
 base = base * 10000;
 printf("base = %d \n ",base);
 do{
  ok = 1; /* vrai */
  prec=NULL;
  actu= *prem;
  suiv= actu→ suiv;
  printf("%d, ",(int)actu−(int)base);
  anc=actu;
  while(suiv!=NULL){
   printf("adresse 2 = %d, saut = % d \n",(int)actu −
    (int)base,(actu − anc));
   anc=actu;
   printf("%d, ",(int)suiv−(int)base);
   anc=suiv;
   if(actu → val > suiv→val){
    ok= 0;
    if(prec!=NULL) {
     prec→suiv=suiv;
     printf("adresse 4 = %d, sau t= %d \n",(int)prec −
      (int)base,(prec − anc));
     anc=prec;}
    else *prem=suiv;
    actu→suiv=suiv→suiv;
    printf("%d, ",(int)actu−(int)base);
    anc=actu;
     printf("%d, ",(int)suiv−(int)base);
    anc=suiv;
    suiv→suiv=actu;}
    prec=actu;
    actu=suiv;
    suiv=actu→suiv;
    printf("adresse7 =%d, saut=%d\n",actu,(actu−anc));
    anc=actu;}}
 while( !ok);
 }
```

FIG. 6.5 – Extrait de code instrumenté

FIG. 6.6 – Courbes comparatives des accès mémoires effectifs et des accès de la fonction génératrice : cas 1

FIG. 6.7 – Courbes comparatives des accès mémoires effectifs et des accès de la fonction génératrice : cas 2

6.4.3 Reorganisation des accès mémoire et comparaison au code original

Cette partie n'a pu être réalisée pour le moment et va constituer une des prochaines étapes de ce travail. Elle permettra de tester directement à l'exécution des programmes l'efficacité de la reorganisation des accès mémoire, reorganisation basée sur les fonctions génératrices.

Chapitre 7

Conclusion

Dans la recherche de performances pour les machines leurs programmes, de nombreuses études ont établi que les performances pour les processeurs ne sont pas linéairement suivies de performances pour les programmes ; cela provient de ce que beaucoup de programmes ne tiennent pas compte de la structure des processeurs sur lesquels ils vont être exécutés. Afin de déterminer dans les programmes les endroits les plus favorables à des optimisations de code et de masquer aux utilisateurs la complexité du matériel avec lesquels ils travaillent, des méthodes d'analyse de programme ont été développées : les méthodes d'analyse statique et les méthodes d'analyse dynamique.

L'analyse statique est l'ensemble des techniques qui permettent de déduire automatiquement des propriétés de programmes à partir de leur code source. Elle s'effectue au moment de la compilation et s'appuie sur des bases théoriques de sémantique de programme. Par l'examen du code source et l'étude de ses schémas d'accès aux données, cette méthode d'analyse est une aide significative à la production de code optimisé. Seulement, les approches statiques manquent de précision. En effet, lors de l'analyse du code source, les outils sont confrontés à une multitude de valeurs inconnues, ce qui oblige à faire des suppositions parfois restrictives. Un autre inconvénient non négligeable est que bien souvent, les optimisations à la compilation ne peuvent être développées suffisamment rapidement pour arriver à maturation lors de la production des nouveaux processeurs.

Tandis qu'un analyseur statique donne le comportement d'un programme par l'examen de son code source, un analyseur dynamique oeuvre durant l'exécution du programme, donnant ainsi des informations bien plus précises et détaillées.

L'analyse dynamique consiste en l'observation du comportement du programme au cours de son exécution, afin de déterminer les portions de code les plus susceptibles d'être optimisées. Le profiling des programmes est donc une technique d'analyse permettant de déterminer, à l'exécution, les causes de mauvaises performances de programmes et donc les zones du code où l'optimisation serait le plus rentable. Le profiling consiste essentiellement en trois phases : l'instrumentation du code, la compilation et l'exécution. L'instrumentation de code est un vieux procédé : à chaque point important du programme, on insère des routines spéciales pour capturer les informations qui nous intéressent. L'inconvénient de cette méthode est que les résultats obtenus ne sont applicables parfois qu'à certaines entrées. Tout dépend si le com-

portement du programme est fortement couplé ou non aux valeurs de ses entrées. Les deux techniques d'analyse dynamique recensées sont l'échantillonnage et le traçage. Cette méthode consomme beaucoup de ressources et de temps vu que les fichiers de trace générés sont souvent très grands.

L'analyse dynamique est un complément significatif de l'analyse statique car les informations prédites statiquement peuvent être vérifiées sur la trace des programmes. De plus, ces informations recueillies statiquement au préalable peuvent guider l'observation du profiling et affiner plutôt certaines observations utiles à des optimisations pressenties lors de l'analyse statique. Nous pensons qu'une méthode d'analyse tenant compte de toutes ces observations serait susceptible de produire de bien meilleurs résultats que ceux que nous avons mentionnés tout au long de ce mémoire. Nous envisageons de poursuivre, dans le cadre d'une thèse, des recherches sur cette nouvelle approche pour la compilation et l'optimisation de programmes, approche caractérisée par la collaboration de l'analyse statique et de l'analyse dynamique.

Bibliographie

[1] T. Chilimbi, Mark D. Hill, James R. Larus. *Making Pointer-Based Data Structures Cache Conscious.* IEEE Computer, Vol. 33, Num. 12, pp. 67-74, Décembre 2000.

[2] T. Chilimbi, Mark D. Hill, James R. Larus. *Improving pointer-based Codes Through Cache-Conscious Data Placement.* Technical Report CS-TR-98-1365, University of Wisconsin—Madison, Mar., 1998.

[3] Rajiv Gupta, Eduard Mehofer, Youtao Zhang. *Profile Guided Compiler Optimizations.*, to appear, *The Compiler Design Handbook : Optimizations and Machine Code Generation*, 2002.

[4] Les outils d'analyse : perfex. http ://www.crihan.fr/CRIHAN/calcul/par /doc/Doc/Descript_log/Outil_analyse/perfex.html

[5] Les outils d'analyse. http ://www.crihan.fr/CRIHAN/calcul/par/doc/ Doc/Descript_log/Outil_analyse/outil_analyse.html#debut

[6] Outils d'analyse : Speedshop. http ://www.crihan.fr/CRIHAN/calcul/par /doc/Doc/Descript_log/Outil_analyse/outil_analyse.html#speedshop

[7] Guide utilisateur. http ://biology.ncsa.uiuc.edu/library/SGI_bookshelves /SGIindex/SGI_Developer_SShop_UG.html

[8] Jason R. C. Patterson. *Accurate Static Branch Prediction by Value Range Propagation.* ACM SIGPLAN '95 Conference on Programming Language Design And implementation pp 67-78 , June 1995.

[9] Cliff Young, David S. Johnson, David R. Karger and Michael D. Smith. *Near-Optimal Intraprocedural Branch Alignment.* ACM SIGPLAN'97 Conference on Programming Language Design And implementation, June 1997.

[10] Paul Feautrier. *Dataflow Analysis of array and scalar references.* Int. J. Parallel programming, 20(1) : 23-51, 1991

[11] Thomas Ball and James R. Larus. *Branch Prediction for Free.* Proceedings of ACM SIGPLAN'93 Conference on Programming Language Design and Implementation, pp. 300-313, Juin 1993.

[12] Youfeng Wu and James R. Larus. *Static Branch frequency and program profile Analysis.* 27th IEEE/ACM Inter'l Symposium on Microarchitecture (MICRO-27), Nov. 1994.

[13] Mark N. Wegmann and F. Kenneth Zadeck. *Constant Propagation with Conditionnal Branches.* ACM transactions on Programming Languages and Systems 13(2), April 1991, pages 181-210.

[14] William H. Harrison. *Compiler Analysis of the Value Ranges for Variables.* IEEE Transactions on Software Engineering 3(3), May 1977, pages 243-250.

[15] Shafer, G. *Mathematical Theory of Evidence.* Princeton University Press, 1976.

[16] P. Michaud, A. Seznec, R. Uhlig. *Trading conflict and capacity aliasing in conditional branch predictors.* In : Proceedings of the 24th International Symposium on Computer Architecture, IEEE-ACM (éditeur), Denver, juin 1997.

[17] P. Chang, E. Hao, Y. Patt. *Target prediction for indirect jumps.* In : Proceedings of the 24th Annual International Symposium on Computer Architecture, 1997.

[18] Jason R. C. Patterson. *Accurate Static Branch Prediction by Value Range Propagation.* ACM SIGPLAN '95 Conference on Programming Language Design And implementation pp 67-78 , June 1995

[19] Cliff Young, David S. Johnson, David R. Karger and Michael D. Smith. *Near-Optimal Intraprocedural Branch Alignment.* ACM SIGPLAN'97 Conference on Programming Language Design And implementation, June 1997

[20] Paul Fautrier. *Dataflow Analysis of array and scalar references* Int. J. Parallel programming pp 23-51, 1994.

[21] Thomas Ball and James R. Larus. *Branch Prediction for Free.* Proceedings of ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (June, 1993) pp. 300-313.

[22] Youfeng Wu and James R. Larus. *Static Branch frequency and program profile Analysis.* 27th IEEE/ACM Inter'l Symposium on Microarchitecture (MICRO-27), Novembre 1994.

[23] Paul Feautrier. *Automatic parallelization in the Polytope Model.* The data Parallel Programming model : Foundations, HPF Realization and Scientific Applications, 1996.

[24] Patrick Cousot and Radhia Cousot. *Systematic Design of Program Analysis Frameworks.* In Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages pp 269–282, 1979.

[25] Paul Feautrier. *Dataflow Analysis of array and scalar references.* Int. J. Parallel programming pp 23-51, 1991.

[26] Somnath Ghost, Margaret Martonosi and Sharak Malik. *Cache Miss Equations : A compiler Framework for Analysing and Tuning Memory Behavior.* ACM Transaction on Programming Languages and Systems, 1999.

[27] Wagner Fréderic. *Compilation pour les processeurs EPIC.* Mémoire de DEA, Université Louis Pasteur, Strasbourg, 2002.

[28] Benoît Meister. *Méthodes Mathématiques d'Optimisation des Accès Mémoire pour les Logiciels Enfouis.* Mémoire de DEA, Université Louis Pasteur, Strasbourg, 2001

[29] Benoît Meister. *Localité de données dans les opérations stencil.* In RenPar'13, pp 37-42, ASTI, 2001.

[30] Eugène Ehrhart. *Sur un problème de géométrie diophantienne linéaire i.* Reine Angew math., 226 : 1-29, 1967.

[31] Eugène Ehrhart. *Sur un problème de géométrie diophantienne linéaire ii.* Reine Angew math., 227 : 27-49, 1967.

[32] Eugène Ehrhart. *Polynômes arithmétiques et Méthodes des polyèdres en Combinatoire.* volume 35 of International Series of Numerical Mathmatics. Birkhaäusser Verlag, Basel/Stuttgart, 1977.

[33] V. Loechner, B. Meister, and P. Clauss. *Precise Data Locality optimization of Nested Loops.* The Journal of supercomputing, 21, 37-76, 2002.

[34] V. Loechner, B. Meister, and P. Clauss. *Data sequence locality : a generalization of temporal locality.* In EuroPar, European Conference on Parallel Computing Manchester, 2001.

[35] Philippe Clauss. *Counting solutions to linear and nonlinear constraints through Ehrhart polynomials : Applications to analyse and transform scientific programs.* In 10th ACM International Conference on Supercomputing Philadelphia, 1996.

[36] Philippe Clauss and Vincent Loechner. *Parametric analysis of polyhedra iteration spaces.* Journal of VLSI Signal Processing 19(2) : 179-194, 1998. http ://icps.u-strasbg.fr/pub-98

[37] La librairie Polylib. *http ://icps.u-strasbg.fr/ loechner/polylib/*

[38] Durée de vie, allocation de registres, allocation mémoire. *http ://pauillac.inria.fr/ levy/x/m2/7/*

[39] C. Eisenbeis, W. Jalby, D. Windheiser et F. Bodin *: A strategy for array management in local memeory.* Rapport de recherche INRIA RR-1262, jullet 1990.

[40] J. Abella, N. Bermudo, C. Ciuraneta, J. M. Codina, C. Eisenbeis, A. Gonzalez, J. Llosa, A. Randrianatoavina, F. Thomasset, Sid Ahmed Ali Touati, X. Vera. *MHAOTEU : Memory Hierarchy Analysis and Optimizations Tools for The End-User.* 30 Novembre 1990.

[41] D. Gannon, W. Jalby et K. Gallivan. *Strategies for Cache and local Memory Management by Global Program Transformation.* Proceedings of the International Conference on Supercomputing, Springer Verlag, New york, 1987 and Journal of parallel and Distributed Computing, Oct. 1988.

[42] Bernstein, A. J. *Analysis of programs for Parallel Processing.* IEEE Transactions on Electronic Computers, October, 15(5), 1996.

[43] P. Feautrier. Lncs 1132. IN *The Data Parallel Programming Model* chapter Automatic parallelization in the polytope Model, pages 79-100, 1996.

[44] V. Loechner. *Contribution à l'étude des polyèdres paramétrés et applications en parallélisation automatique*, Thèse de l' Université Louis Pasteur, Strasbourg, 1997. http ://icps.u-strasbg.fr/pub-97/

[45] Pierre Aminaroff, Albert Cohen, Paul Feautrier. *Variables d'inductions généralisées pour l'analyse par instances de programmes récursifs.* Projet A3, rapport de recherche n°4252-13, septembre 2001.

[46] Simon Plouffe. *Une méthode pour obtenir la fonction génératrice algébrique d'une série.* LACIM, Université du Qubec à Montréal, mars 1993.

[47] Bruno Salvy, Paul Zimmermann. *GFUN : a Maple package for the manipulation of generating and holonomic functions in one variable.* Rapports techniques N° 143, Inria rocquencourt, Octobre 1992.

[48] Numerical Recipes Books : *http ://www.ulib.org/webRoot/Books/Numerical_Recipes /bookc.html*