
THÈSE de DOCTORAT de l'UNIVERSITÉ PARIS 6

Spécialité : Systèmes Informatiques

présentée par

**Cédric Bastoul**

pour obtenir le grade de DOCTEUR de l'UNIVERSITÉ PARIS 6

Sujet de la thèse :

# Amélioration de la localité dans les programmes à contrôle statique

## *Improving Data Locality in Static Control Programs*

Soutenue le *7 décembre 2004*

Devant le jury composé de :

| | | |
|---|---|---|
| Pr. Claude | Girault | Président |
| Pr. Philippe | Clauss | Rapporteur |
| Pr. Christian | Lengauer | Rapporteur |
| Pr. Paul | Feautrier | Directeur |
| Pr. Nathalie | Drach-Temam | Examinateur |
| Pr. Patrice | Quinton | Examinateur |

## Résumé

Les mémoires cache ont été introduites pour réduire l'influence de la lenteur des mémoires principales sur les performances des processeurs. Elles n'apportent cependant qu'une solution partielle, et de nombreuses recherches ont été menées dans le but de transformer les programmes automatiquement afin d'optimiser leur utilisation. Les enjeux sont considérables puisque les performances ainsi que la consommation d'énergie dépendent du trafic entre les niveaux de mémoire.

Nous revisitons dans cette thèse le processus de transformation de code basé sur le modèle polyédrique afin d'en améliorer le spectre d'application comme la qualité des résultats. Pour cela nous avons redéfini le concept de parties à contrôle statique (SCoP) dont nous montrons expérimentalement la grande présence dans les applications réelles. Nous présentons une politique de transformation libérée des limitations classiques aux fonctions unimodulaires ou inversibles. Nous étendons enfin l'algorithme de Quilleré et al. pour être capables de générer en des temps raisonnables des codes particulièrement efficaces, levant ainsi une des principales idées reçues sur le modèle polyédrique. Nous avons proposé au dessus de ce schéma de transformation une méthode d'amélioration de la localité s'appuyant sur un modèle d'exécution singulier qui permet une évaluation du trafic en mémoire. Cette méthode considère chaque type de localité de même que la légalité comme autant de contraintes composant un système dont la transformation recherchée est une solution. Nous montrons expérimentalement que cette technique permet l'amélioration à la fois de la localité et des performances dans des cas traditionnellement embarrassants comme les structures de programmes complexes, les dépendances de données complexes ou les références non uniformément générées.



## Abstract

Cache memories were invented to decouple fast processors from slow memories. However, this decoupling is only partial, and many researchers have attempted to improve cache use by program optimization. Potential benefits are significant since both energy dissipation and performance highly depend on the traffic between memory levels.

This thesis will visit most of the steps of high level transformation frameworks in the polyhedral model in order to improve both applicability and target code quality. To achieve this goal, we refine the concept of static control parts (SCoP) and we show experimentally that this program model is relevant to real-life applications. We present a transformation policy freed of classical limitations like considering only unimodular or invertible functions. Lastly, we extend the Quilleré et al. algorithm to be able to generate very efficient codes in a reasonable amount of time. To exploit this transformation framework, we propose a data locality improvement method based on a singular execution scheme where an asymptotic evaluation of the memory traffic is possible. This information is used in our optimization algorithm to find the best reordering of the program operations, at least in an asymptotic sense. This method considers legality and each type of data locality as constraints whose solution is an appropriate transformation. The optimizer has been prototyped and tested with non-trivial programs. Experimental evidence shows that our framework can improve both data locality and performance in traditionally challenging programs with e.g. non-perfectly nested loops, complex dependences or non-uniformly generated references.

# Remerciements

Un chercheur, par définition, ne fait pas comme les autres me dirait Claude Girault. C'est pourtant bien volontiers et même avec un certain plaisir que j'écris ces quelques lignes rituelles pour témoigner de ma profonde gratitude envers ceux qui m'ont permis d'en être là. Je suis sûr qu'il me pardonnera cet écart de conduite !

Mes remerciements vont tout d'abord à Paul Feautrier pour avoir dirigé mes travaux, pour sa disponibilité malgré un emploi du temps surhumain, pour sa confiance et la liberté qu'il m'a accordée. J'ai eu un plaisir toujours renouvelé à apprendre avec lui le métier de chercheur et à sortir de nos réunions avec des idées et du travail pour dix ans. Son optimisme et sa passion entre autres qualités ont fait que j'ai appris à son contact bien plus que de la science.

Je tiens ensuite à remercier les membres de mon jury, notamment Claude Girault pour m'avoir fait le plaisir et l'honneur de présider ce jury, mais aussi pour m'avoir fait partager son enthousiasme et son dynamisme à chaque fois que nous avons pu nous rencontrer. Mes remerciements vont aussi à Philippe Clauss et Christian Lengauer qui m'ont accordé le réel privilège d'être les rapporteurs de ma thèse. Leurs suggestions et commentaires forcément éclairés, leur intérêt et leur regard sur mon travail ont été, sont et seront une aide considérable. Merci enfin à Nathalie Drach-Temam et à Patrice Quinton pour avoir accepté d'être mes examinateurs, pour leur curiosité quant à mes travaux et leur ouverture par leurs remarques et leurs questions à de nouvelles problématiques.

Je remercie sincèrement Christine Eisenbeis qui m'a accueilli à l'INRIA et toujours considéré comme membre de son équipe. Merci à mes aînés du PRiSM comme de l'INRIA, Denis Barthou, Albert Cohen, François Thomasset et Sid *complexité linéaire* Touati pour leurs précieux conseils, leur aide, leur soutien et toutes les discussions passionnantes que nous avons pu avoir. De même, merci à Martin Griebl de l'université de Passau pour toutes nos discussions mais aussi pour sa gentillesse et son hospitalité qui resteront pour moi des modèles. Un sincère merci à tous mes compagnons de route du PRiSM, Christophe Alias, Karine Brifault, Patrick Carribault, Ivan Djelic, Christophe Lemuet, Ani Sedrakian et tous les autres et en particulier les habitants du $4^{ème}$ ainsi qu'à l'INRIA Pierre Amiranoff, Sylvain Girbal et à Yosr Slama de l'université de Tunis pour l'ambiance de travail amicale, décontractée et même parfois délirante (hein captain'web ?). Merci à eux et je leur souhaite toute la réussite possible pour la suite des évènements.

Puisque c'est l'endroit où jamais, je tiens aussi à remercier chaleureusement ceux qui m'ont attiré vers l'informatique et la recherche, en particulier mes vieux compagnons Clément Delamare et Sébastien Salva (c'est surtout lui le vieux). Mais aussi Hacène Fouchal et Patrice Moreaux de l'université de Reims pour m'avoir poussé dans cette voie étrange et exaltante qu'est la recherche. Enfin je n'oublierai pas Jérôme *binôme* Gaspard, pour tous les grands moments passés autour d'une machine et ailleurs !

Je remercie ma famille pour son soutien constant, et en particulier mes parents pour leur confiance et pour m'avoir permis et laissé libre d'aller là où je voulais aller. Merci enfin et surtout à ma compagne, Agnès, pour sa patience, son aide et son amour, merci du fond du coeur.

# Contents

# List of Figures

# Notations

Here is a compendium of the main notations used throughout this thesis. In order to avoid too many notations, some of them are context-dependent. This means that we use the same symbol for neighboring concepts when this is possible in an intuitive way; for instance $\rho(x)$ is the number of subscripts if $x$ is an array but the number of surrounding loops if $x$ is a statement.

| Array context | |
|---|---|
| $\mathcal{A}_P$ | array set of a program $P$ |
| $\rho(A)$ | rank of the array $A \in \mathcal{A}_P$, i.e. its number of subscripts |
| $\mathcal{D}_A$ | subscript domain of the array $A \in \mathcal{A}_P$ |
| $A[\vec{\imath}]$ | memory cell of $A$ with subscripts $\vec{\imath} \in \mathcal{D}_A$ |
| $\mathcal{M}_P$ | data set of the program $P$, i.e. the disjoint union of all $\mathcal{D}_A$ |
| $\langle A, f \rangle$ | reference to the array $A \in \mathcal{A}_P$ with the subscript function $f$ |

| Statement context | |
|---|---|
| $\mathcal{S}_P$ | statement set of the program $P$ |
| $\rho(S)$ | rank of the statement $S \in \mathcal{S}_P$, i.e. the number of surrounding loops |
| $\mathcal{D}_S$ | iteration domain of the statement $S \in \mathcal{S}_P$ |
| $S(\vec{x})$ | instance of the statement $S \in \mathcal{S}_P$ with iteration vector $\vec{x} \in \mathcal{D}_S$ |
| $\mathcal{O}_P$ | operations set of the program $P$, i.e. the disjoint union of all $\mathcal{D}_S$ |
| $\mathcal{D}_{S(\vec{x})}$ | set of the memory cells accessed by the operation $S(\vec{x}) \in \mathcal{O}_P$ |
| $r_S$ | number of references in the statement $S \in \mathcal{S}_P$ |

| Program context | |
|---|---|
| $<_P$ | sequential execution order of the program $P$ |
| $\delta_P$ | dependence relation of the program $P$ |

| Chunking context | |
|---|---|
| $\mathcal{T}$ | traffic |
| $\mathcal{F}(\vec{c})$ | set of the memory cells accessed by the operation set number $\vec{c}$ (footprint) |

| Misceleanous | |
|---|---|
| $\lceil a \rceil$ | is the lowest integral number greater than $a \in \mathbb{R}$ (ceil) |
| $\lfloor a \rfloor$ | is the greatest integral number lower than $a \in \mathbb{R}$ (floor) |
| $A_{i,\bullet}$ | $i^{th}$ row vector of the matrix $A$ |
| $A_{\bullet,i}$ | $i^{th}$ column vector of the matrix $A$ |

# Résumé

Cette partie présente un résumé en français de la thèse écrite en anglais. Son organisation respecte celle du document original et chacune de ses sections correspond à un chapitre ou une partie dont les développements, algorithmes, preuves et exemples ont été simplifiés ou retirés. Le lecteur s'intéressant à un sujet particulier est donc naturellement invité à se reporter aux chapitres correspondants pour y trouver un niveau de détail satisfaisant.

## Table des matières

# I Introduction

Le rapport *First Draft of a Report on the EDVAC* présenté par John von Neumann en 1945 décrivait l'organisation des ordinateurs connue à présent comme *architecture de von Neumann* où la mémoire et les unités de contrôle et d'arithmétique sont séparées. Depuis lors, les systèmes informatiques utilisent cette organisation qui n'a pas encore de réelle alternative. La performance globale des ordinateurs dépend donc de la capacité qu'a la mémoire de fournir les données aux unités de calcul à la vitesse désirée. Malheureusement la puissance de traitement des microprocesseurs a augmenté et augmente encore bien plus rapidement que la vitesse des mémoires. On estime que l'écart de performance entre les processeurs et les mémoires principales grandit d'environ 50% par an [58]. Ce phénomène est appelé le *trou de la mémoire* (*memory gap*). Il a conduit dès les années 60 à plusieurs évolutions majeures des architectures. Tout d'abord l'utilisation de jeux d'instructions complexes (CISC : Complex Instruction Set Computers) a permis de diminuer la taille des programmes et le nombre d'accès à la mémoire. Ensuite, les *mémoires cache*, des mémoires intermédiaires plus petites et plus rapides que la mémoire principale, ont été introduites. Leur principe est de donner l'illusion d'avoir à disposition une mémoire ayant la vitesse de la plus rapide des mémoires cache (dite cache de niveau 1) et la taille de la mémoire principale, en plaçant dans le cache les données les plus utiles au processeur à tout moment. Dans les architectures récentes, le trou de la mémoire est si large que plusieurs niveaux de caches sont nécessaires pour le combler, créant ainsi une hiérarchie de mémoires complexe.

L'utilisation efficace de cette hiérarchie de mémoires est une des clés pour obtenir de hauts niveaux de performance sur les architectures modernes. La solution la plus fréquemment utilisée est de réorganiser les programmes de manière à ce que le nombre d'échanges entre le cache et la mémoire principale soit minimisé. D'une manière générale, lorsqu'une donnée est placée dans le cache, le programme final devrait exécuter autant d'instructions nécessitant cette donnée que possible. Les premières tentatives pour atteindre ce but étaient des astuces de programmation menant à des optimisations non portables, des programmes illisibles et où corriger les erreurs était donc particulièrement difficile. Pour éviter ces problèmes, de nombreux travaux ont proposé de placer le travail d'optimisation sous la responsabilité du compilateur. Plusieurs directions ont été explorées. Premièrement, les techniques classiques de transformation de programmes utilisées pour la parallélisation automatique comme les transformations unimodulaires [10] ou le *pavage* [107] on été adaptées. Ensuite, des transformations de données ont été créées pour minimiser le besoin de mémoire des programmes [95] ou pour réorganiser les données en mémoire en fonction des accès [77]. Malheureusement, la plupart des techniques d'optimisation de l'utilisation des mémoires cache souffrent de sévères limitations. Ainsi il est très difficile de choisir quelles transformations sont utiles et dans quel ordre elles doivent être appliquées. De plus, elles sont très sensibles aux dépendances entre les données, ce qui a conduit à limiter leur action à des programmes très simples où aucune dépendance n'existe ou ont une forme suffisamment simple. Les transformations de données manquent quant à elles de flexibilité et ne sont en général pas capables de trouver une solution optimale pour des programmes complets dans des cas réalistes.

Aucune des techniques actuelles ne prend en considération l'existence d'outils permettant au logiciel de participer à la gestion du cache. Nous proposons dans cette thèse une nouvelle méthode d'amélioration de la localité basée sur l'utilisation de tels outils. Son principe est de réorganiser les opérations d'un programme pour former des sous-ensembles dont toutes les données accédées tiennent ensemble dans le cache. Cette technique, appelée *chunking*, exploite les mémoires cache tout en étant libérée de la plupart des limitation inhérentes aux méthodes existantes. De plus,

contrôler le contenu du cache à tout moment nous permet d'envisager d'offrir des garanties de type temps-réel.

Cette thèse va aborder la plupart des aspects d'un optimiseur de haut niveau, depuis l'extraction des informations utiles dans le code source jusqu'à la génération du programme final. Elle est divisée en trois parties qui reflètent les principales étapes du processus de transformation de code. La première partie introduit les concepts de base. En section II nous présentons notre modèle de programme et des résultats expérimentaux sur son importance dans diverses applications réelles. Nous y verrons que contrairement à une idée reçue, une grande partie des codes provenant de programmes bien connus entre dans le cadre du modèle polyédrique. La section III décrit le principe de transformation dans ce modèle. Nous y libérerons les contraintes usuelles d'unimodularité ou d'inversibilité des transformations et y ouvrirons la voie à des transformations complexes, sans pour autant compliquer ou interdire la génération du code final. La seconde partie s'attache aux problèmes d'amélioration de la localité. Dans la section IV nous introduisons une manière d'évaluer le trafic en mémoire grâce à un schéma d'exécution particulier. Nous verrons qu'il permet de déduire une estimation du trafic en fonction de la transformation elle-même. Nous montrerons en section V comment tirer parti de cette information pour construire des transformations optimisantes. La méthode proposée présente la propriété de s'appliquer dans des cas aussi compliqués que courants de dépendances complexes, de structures de programmes complexes ou de références non uniformément générées. La dernière partie est consacrée à l'ultime étape : la génération de code. Nous montrerons que la qualité du programme cible dépend des fonctions de transformation elles-mêmes et non uniquement de l'algorithme de génération de code. Nous proposerons alors des méthodes pour choisir ou modifier ces fonctions de manière à assurer la génération d'un code performant. Nous présenterons plusieurs améliorations à un algorithme connu permettant de produire un code particulièrement efficace. Nos améliorations comprendront le support des pas de boucles, la réduction de l'explosion de la taille du code généré ainsi que du temps de génération. Enfin nous concluons et discutons des travaux à venir en section VII.

# II Modèle de programme

Le modèle de programme permet et limite à la fois les opportunités d'exposer la réutilisation des données, de proposer des solutions efficaces pour exploiter cette réutilisation et d'appliquer ces solutions. D'un côté on pourrait choisir de traiter des programmes très généraux, mais leur analyse pourrait ne pas être possible même pour les compilateurs les plus récents ou être trop difficile pour être effectuée en un temps raisonnable. D'un autre côté il est possible de ne considérer qu'un ensemble de programmes très restreint tel que l'analyse soit aisée, mais de tels cas pourraient ne pas refléter les problèmes réalistes. Il est donc nécessaire de trouver le meilleur compromis entre la représentativité et la puissance d'analyse.

Un candidat adéquat semble être l'ensemble des programmes à contrôle statique. Tout d'abord, il inclut des programmes avec des schémas d'accès particulièrement réguliers qui sont mal supportés par les mécanismes des mémoires cache. Il est donc une cible particulièrement intéressante dans le cadre de l'optimisation de la localité. Ensuite, il supporte un large éventail de programmes dont les nids de boucles qui sont parmi les noeuds de calculs les plus intensifs dans les applications. Enfin, il permet d'entrer dans le *modèle polyédrique* et de profiter des outils mathématiques qui en découlent et qui permettent, par exemple, l'analyse exacte des dépendances. Les propriétés

des programmes à contrôle statique sont définies dans [43] et peuvent être grossièrement résumées ainsi : (1) les structures de contrôle sont les boucles **do** avec des bornes affines et les conditionnelles **if** avec des conditions affines ; (2) les tableaux sont les seules structures de données et leurs indices sont affines ; (3) les bornes comme les conditions ne dépendent que des itérateurs des boucles englobantes et de paramètres de structure ou de taille ; (4) les appels à des fonctions ou à des routines ont été remplacés par le corps de ces fonctions ou routines.

Dans la suite, nous allons étendre le concept de contrôle statique. Nous présentons d'abord les notions élémentaires en section II.1, puis nous étudions la partie contrôle du modèle en section II.1 et enfin la partie structure de données en section II.3.

## II.1 Notions élémentaires

Dans la suite nous distinguons différents niveaux d'abstraction concernant les commandes d'un programme. Une *instruction* est une structure de programme qui ordonne à l'ordinateur d'exécuter une action spécifique. Cette action peut être évaluer une expression arithmétique, instancier une variable ou appeler une procédure. L'ensemble des instructions d'un programme $P$ est appelé $\mathcal{S}_P$. Une instruction peut être exécutée plusieurs fois (par exemple lorsqu'elle appartient à une fonction ou quand elle est incluse dans une boucle) ; chaque instance d'une instruction est appelée une *opération*. L'ensemble des opérations d'un programme $P$ est appelé $\mathcal{O}_P$.

Un *nid de boucles* est un ensemble fini de structures itératives imbriquées, comme il en existe dans tout langage impératif comme C ou FORTRAN, et ayant la forme suivante :

```
do x1 = L1, U1, S1
|  ...
|  do x2 = L2, U2, S2
|  |  ...
|  |  do xn = Ln, Un, Sn
|  |  |  Corps
|  |  ...
|  ...
```

où `Corps` peut être une liste de commandes ou un nid de boucles. Un nid de boucles est dit *parfait* quand toutes les commandes sont dans `Corps`. Pour la $k^{ème}$ boucle, $i_k$ est une variable entière appelée *itérateur* ou *compteur*. La valeur du compteur de boucle est mise à jour à la fin de chaque itération de la boucle par l'ajout de $S_k$, une variable entière non nulle appelée le *pas* de la boucle. $L_k$ et $U_k$ sont des expressions correspondant aux *bornes* de la boucle : la valeur de départ du compteur est $L_k$, et la boucle termine lorsqu'il devient plus grand que $U_k$.

Un nid de boucle peut être représenté par un vecteur colonne de taille $n$ appelé le *vecteur d'itération* :

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix},$$

où $x_k$ est le $k^{ème}$ itérateur, $n$ est appelé la *profondeur* de la boucle et $\mathbb{Z}^n$ est l'*espace d'itération*. L'ensemble des valeurs possibles du vecteur d'itération pour une instruction donnée est appelé le *domaine d'itération* de l'instruction. L'instruction est exécutée pour chaque élément appartenant

au domaine d'itération. Ainsi, chaque point du domaine d'itération correspond à une opération. On note $S(\vec{x})$ l'opération instance de l'instruction $S$ pour le vecteur d'itération $\vec{x}$. La figure 1 montre un exemple de nid de boucle et de la représentation du domaine d'itération de son unique instruction où chaque axe correspond à une boucle et chaque point à une itération du nid de boucles dirigeant l'exécution du l'instruction $S$.

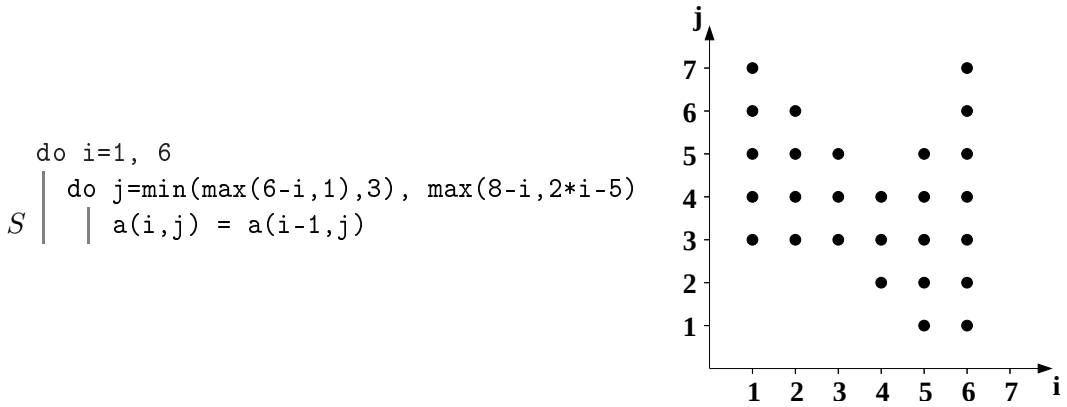



Fig. 1 – Domaine d'itération d'un nid de boucles

## II.2 Contrôle statique

Lorsque les bornes et les conditions qui entourent une instruction sont des fonctions affines qui ne dépendent que des itérateurs des boucles englobantes, de paramètres formels et de constantes, le domaine d'itération peut toujours être spécifié par un ensemble d'inéquations qui définissent un polyèdre [68]. Nous utiliserons abusivement le terme *polyèdre* pour définir un *ensemble de points entiers dans un treillis* (aussi appelé $\mathbb{Z}$-polyèdre ou polyèdre-treillis), c'est à dire un ensemble de points dans un $\mathbb{Z}$-espace vectoriel borné par des inéquations affines [97] :

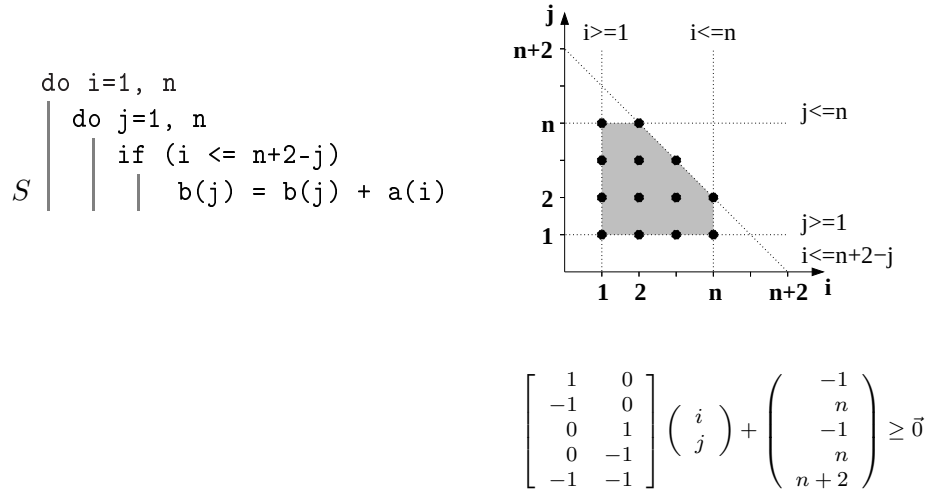
$$\mathcal{D} = \left\{ \vec{x} \mid \vec{x} \in \mathbb{Z}^n, A\vec{x} + \vec{a} \geq \vec{0} \right\},$$

où $\vec{x}$ est le vecteur d'itération ; $A$ est une matrice constante et $\vec{a}$ est un vecteur constant éventuellement paramétrique. Le domaine d'itération est un sous-ensemble de l'espace d'itération : $\mathcal{D} \subseteq \mathbb{Z}^n$. La figure 2 illustre la correspondance entre le contrôle englobant et les domaines polyédriques : le domaine d'itération présenté en figure 2(b) peut être défini par des inéquations affines qui peuvent être extraites directement à partir du programme en figure 2(a). L'ensemble de contraintes affines peut être représenté en utilisant la notation matricielle telle que présentée en figure 2(b). Nous pouvons en déduire aisément que toute instruction avec un contrôle englobant de la forme suivante pour chaque niveau de boucle $x_k$ est à contrôle statique :

```
...
do x_k = MAX_{i=1}^{m_l} ⌈(L_i(x_1,...,x_{k-1}) + l_i)/c_k⌉, MIN_{i=1}^{m_u} ⌊(U_i(x_1,...,x_{k-1}) + u_i)/c_k⌋
  if (AND_{i=1}^{m_g} G_i(x_1,...,x_k) + g_i ≥ 0⃗ )
    ...
      Instruction
```

Le programme en figure 1 n'a pas un contrôle statique car le domaine d'itération n'est pas convexe. Cependant, il peut être séparé en une union d'ensembles convexes qui peuvent être

considérés séparément. Xue a montré comment il était possible de trouver automatiquement les composantes convexes de tels programmes où les bornes de boucles sont composées de minima et de maxima d'expressions affines [111].




(a) contrôle englobant $S_1$ (b) domaine d'itération de $S_1$

FIG. 2 – Contrôle statique et domaine d'itération correspondant

**Definition 0.1 (SCoP)** *Un ensemble maximum d'instructions consécutives avec des domaines polyédriques convexes est appelé une partie à contrôle statique, ou un SCoP.*

La définition de SCoP est une légère extension des nids à *contrôle statique* [43] introduite dans [16]. Dans un corps de fonction, une partie à contrôle statique est un ensemble maximal d'instructions consécutives sans boucle `while` et où les bornes de boucles et les conditionnelles ne peuvent dépendre linéairement que d'invariants pour cet ensemble d'instructions. Ces invariants incluent les constantes symboliques, les paramètres des fonctions et les itérateurs des boucles englobantes ainsi que tout invariant apparaissant dans les indices des tableaux : ils sont appelés les *paramètres globaux* du SCoP. Ainsi défini, un SCoP peut inclure des accès mémoire quelconques aussi bien que des appels à des fonctions ; un SCoP est donc plus permissif qu'un nid de boucles à contrôle statique [43]. On dit qu'une partie à contrôle statique est *riche* lorsqu'elle inclut au moins une boucle non vide ; les SCoPs riches sont alors une cible naturelle pour les transformations polyédriques. Un exemple d'extraction de SCoPs dans un programme général est présenté en figure 3. L'extraction d'informations depuis la représentation intermédiaire du compilateur Open64/ORC après sa première étape de pré-traitements grâce à l'outil WRAP-IT [16] nous a permis de mettre en évidence un taux d'instructions appartenant à des SCoPs supérieur à 80% à partir de benchmarks issus des SPEC2000fp et PerfectClub.

## II.3 Références statiques

Chaque langage offre plusieurs manières d'accéder à des valeurs ou des données dans les instructions, par exemple les scalaires, tableaux, pointeurs ou fonctions. Dans notre travail, nous

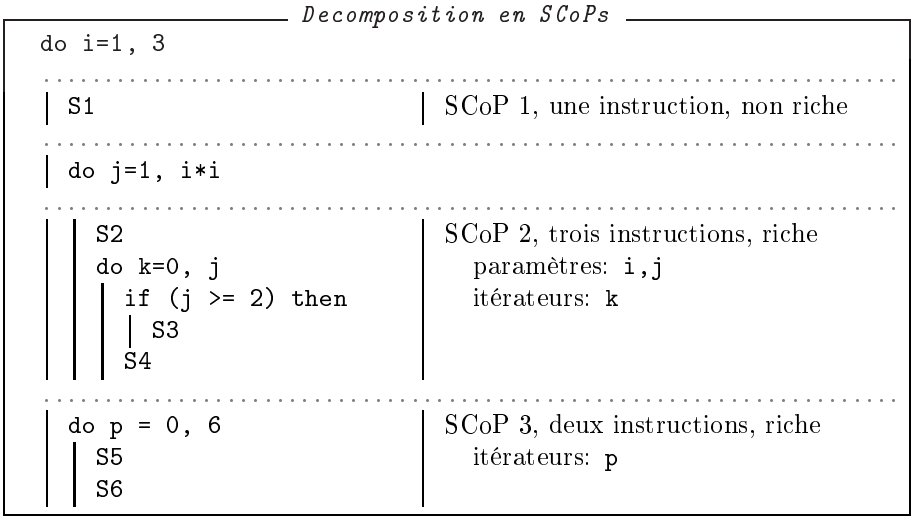



Fig. 3 – Exemple de décomposition en parties à contrôle statique

allons considérer uniquement un sous-ensemble de ces références appelé *références statiques*. Dans cette classe particulière sont incluses uniquement les références à des tableaux (ou des variables qui sont un cas particulier de tableaux). L'ensemble des tableaux d'un programme $P$ est appelé $\mathcal{A}_P$. Une référence à un tableau $B \in \mathcal{A}_P$ dans une instruction $S \in \mathcal{S}_P$ est écrite $\langle B, f \rangle$, où $f$ est la *fonction d'index*. La cellule mémoire de $B$ accédée à l'itération $\vec{x}$ est écrite $B[f(\vec{x})]$. Quand la fonction d'index d'une référence est affine, on peut l'écrire $f(\vec{x}) = F\vec{x} + \vec{f}$ où $F$ est la matrice d'index de dimension $\rho(A) * \rho(S)$ avec $\rho(A)$ le nombre d'indices du tableau $B$ et $\rho(S)$ le nombre de boucles englobantes de $S$. $\vec{f}$ est un vecteur constant. La figure 4 illustre la correspondance entre référence dans un programme source et fonction d'index affine : dans la figure 4(a), le tableau $B$ a deux dimensions, la fonction d'index est un vecteur bidimensionnel comme montré par la figure 4(b). De la même manière que pour le contrôle statique, nous avons pu mettre en évidence avec le compilateur Open64/ORC un taux de références statiques supérieur à 75% à partir de benchmarks issus des SPECfp2000 et PerfectClub.

```
do i=1, n
S | do j=1, n
  |   ... B(i+j,2*i+1) ...
```

(a) Référence à un tableau $B$

$$f\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

(b) Fonction d'index de la référence

Fig. 4 – Notation matricielle d'un exemple de fonction d'index affine

**Definition 0.2 (Référence statique)** *Une référence est dite statique quand elle fait référence à une cellule d'un tableau grâce à une fonction d'index affine qui ne dépend que des itérateurs des boucles englobantes et de paramètres formels.*

# III Transformations de programme

Appliquer la transformation de Fourier à un signal qui varie dans le temps rend possible des traitements qui seraient trop difficiles à réaliser sinon. Une fois que le traitement est achevé,

on peut alors utiliser la transformation inverse pour revenir du domaine des fréquences vers celui du temps. La situation est similaire pour les transformations de programme. Quand les étapes d'analyses lexicale et syntaxique sont terminées, les programmes sont généralement mis sous la forme d'arbres de syntaxe abstraite. Des optimisations simples comme la propagation de constantes ou l'élimination de code mort peuvent leur être appliquées sous cette structure de données particulièrement rigide. Mais les transformations les plus intéressantes, comme la permutation de boucles, impliquent la modification de l'ordre des opérations d'un programme, et cela n'a rien à voir avec la syntaxe. Elles imposent donc de briser la structure de l'arbre et sont particulièrement difficiles à appliquer. à l'opposé, nous allons voir dans cette section que raisonner dans le modèle polyédrique au lieu d'utiliser un arbre de syntaxe abstraite nous permet d'appliquer facilement des transformations complexes ainsi que des compositions de ces transformations.

La majorité des techniques de transformation ont été étudiées séparément, comme la permutation de boucles, la torsion, l'inversion, la fusion, le pavage etc. (on peut se référer à [108] pour une description des transformations connues). Ces techniques ont pour la plupart leurs propres propriétés (par exemple leur test pour vérifier si elles sont *légales* ou non, ou leur modèle de coût pour statuer si les appliquer aboutit à un certain bénéfice). Par conséquent il est particulièrement difficile de choisir quelles transformations devraient être utilisées mais aussi dans quel ordre on devrait les appliquer pour obtenir les meilleurs résultats. *A contrario*, l'utilisation de fonctions d'ordonnancement dans le modèle polyédrique par exemple, permet de décrire des compositions de permutations de boucles, torsions, inversions à un unique nid de boucles de manière complètement uniforme [10]. Cette pratique a évolué naturellement en associant à chaque instruction sa propre fonction d'ordonnancement [44, 45] puis par ajouter à cette méthode de nouvelles transformations telles que l'éclatement ou la fusion de boucles.

Bien que le modèle polyédrique offre de réelles facilités pour restructurer un programme, l'utilisation qui en est faite ne permet que de couvrir un ensemble très restreint parmi les transformations possibles, appelé transformations *unimodulaires*, ou dans le meilleur des cas *inversibles*. Dans cette section, nous abordons une méthode de transformation générale capable de traiter les transformations non-unimodulaires, non-inversibles, non-entières et même non uniformes, en fait l'unique contrainte à respecter est l'affinité des fonctions. Nous montrons tout d'abord en section III.1 comment l'ordre d'exécution peut être exprimé grâce à des fonctions d'ordonnancement, puis en section III.2 comment appliquer des transformations affines dans le modèle polyédrique.

## III.1 Description de l'ordre d'exécution

Une méthode pratique pour exprimer l'ordre d'exécution est de donner à chaque opération une date d'exécution unique. Dans le modèle polyédrique, chaque instance d'instruction est caractérisée par ses coordonnées dans le domaine d'itération correspondant (voir section II.1). La plupart du temps les domaines d'itération ont plusieurs dimensions ; les différentes composantes des coordonnées peuvent être vues comme les jours, heures, minutes etc. Quand chaque pas de boucle d'un programme est une variable positive, nous appelons cet ordre, l'ordre *lexicographique*. De manière formelle, cela signifie que dans un espace à $n$ dimensions, l'opération correspondant au point entier défini par les coordonnées $(a_1...a_n)$ est exécuté avant celui correspondant aux coordonnées $(b_1...b_n)$ si et seulement si il existe un entier $i$ tel que la $i^{ème}$ composante de $(a_1...a_n)$

est plus petite que la $i^{ème}$ composante de $(b_1...b_n)$, et toutes les composantes précédentes sont égales entre elles. Nous notons cet ordre $(a_1...a_n) \ll (b_1...b_n)$ :

$$(a_1...a_n) \ll (b_1...b_n) \Leftrightarrow \exists i, 1 \leq i < n, (a_1...a_i) = (b_1...b_i) \wedge a_{i+1} < b_{i+1}. \quad (1)$$

La figure 5 illustre l'exécution des opérations d'un nid de boucles donné dans l'ordre lexicographique de leurs vecteurs d'itération. Elle montre que le domaine d'itération est borné par des inéquations affines dans un espace à deux dimensions. Les arcs y montrent l'ordre d'exécution des 13 itérations que compte le nid de boucles.

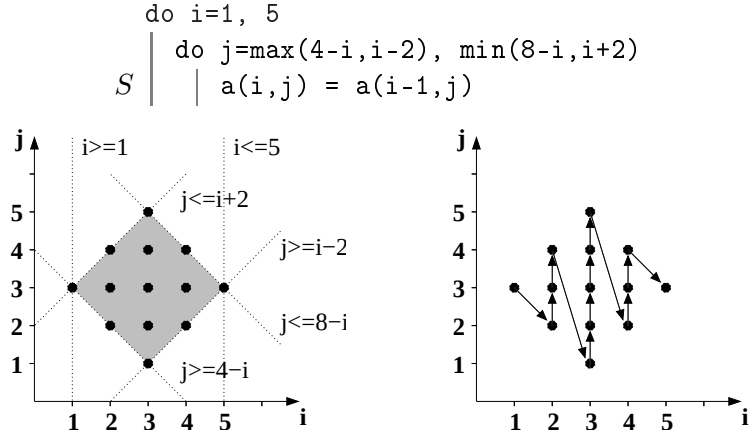



Fig. 5 – Domaine d'itération et ordre d'exécution d'un nid de boucles

L'ordre lexicographique et les vecteurs d'itération ne sont pas suffisants pour décrire l'ordre d'exécution des instances de différentes instructions. Par exemple, considérons le nid de boucles imparfait de la figure 6. Les instructions $S_1$ et $S_3$ ont la même profondeur, mais l'ordre lexico-

```
      do i=1, n
S1    | x = a(i,i)
      | do j=1, i-1
S2    | | x = x - a(i,j)**2
S3    | p(i) = 1.0/sqrt(x)
      | do j=i+1, n
S4    | | x = a(i,j)
      | | do k=1, i-1
S5    | | | x = x - a(j,k)*a(i,k)
S6    | | a(j,i) = x*p(i)
```

Fig. 6 – Factorisation de Cholesky

graphique ne suffit pas pour décider laquelle des opérations $S_1(1)$ et $S_3(1)$ est exécutée avant l'autre, car les vecteurs d'itération sont identiques. De la même manière, l'ordre lexicographique n'est d'aucun secours lorsque les vecteurs d'itération n'ont pas le même nombre de dimensions, par exemple il n'est pas possible de comparer $S_1(1)$ à $S_2(1,2)$ car $S_1(1)$ n'a pas de seconde composante. Le raffinement a été apporté par Feautrier en prenant en compte l'ordre des instructions tel qu'il est décrit par le texte du programme [43]. Soit $s$ le nombre de boucles communes à deux instructions $S_1$ et $S_2$, alors $S_1(a_1...a_n)$ est exécutée avant $S_2(b_1...b_m)$ si $(a_1...a_s) \ll (b_1...b_s)$ ou si les vecteurs d'itération sont identiques pour les dimensions communes et $S_1$ précède $S_2$ dans

le texte du programme. Nous notons cet ordre $S_1(a_1...a_n) \prec S_2(b_1...b_m)$ :

$$S_1(a_1...a_n) \prec S_2(b_1...b_m) \Leftrightarrow \begin{array}{l}(a_1...a_s) \ll (b_1...b_s) \\ \vee((a_1...a_s) = (b_1...b_s) \wedge S_1 \text{ textuellement avant } S_2).\end{array} \quad (2)$$

Définir toutes les dates d'exécution pour toutes les opérations d'un programme séparément peut aboutir à un système d'ordonnancement particulièrement énorme. De plus dans le cas où les domaines d'itération sont paramétrés (par exemple quand une boucle est bornée par une constante inconnue), il n'est pas possible de connaître le nombre exact des instances d'une instruction donnée. Ainsi une solution adéquate est de construire les ordonnancements au niveau de l'instruction plutôt que celui de l'opération en définissant une fonction qui associe à chaque instance de l'instruction correspondante un ordre d'exécution. Ces fonctions sont typiquement choisies affines pour de multiples raisons : c'est le seul cas à l'heure actuelle où on peut vérifier exactement la légalité de la transformation et où on sait générer le code final. Les fonctions d'ordonnancement ont donc la forme suivante :

$$\theta_S(\vec{x}_S) = T_S\vec{x}_S + \vec{t}_S, \quad (3)$$

où $\vec{x}_S$ est le vecteur d'itération ; $T_S$ est une matrice constante dite de transformation, et $\vec{t}_S$ est un vecteur constant (qui peut inclure des expressions affines utilisant les paramètres globaux, c'est à dire les constantes symboliques, principalement les tailles de tableaux ou les bornes des boucles).

Par exemple on peut facilement capturer l'ordre d'exécution séquentiel de tout programme à contrôle statique avec des fonctions d'ordonnancement en utilisant l'arbre de syntaxe abstraite (AST) de ce programme [45]. L'idée principale est d'inclure l'ordre textuel entre chaque composante du vecteur d'itération. Par exemple on peut lire directement les fonctions d'ordonnancement du programme en figure 6 sur son AST présenté en figure 7, c'est à dire : $\theta_{S1}(\vec{x}_{S1}) = (0, i, 0)$, $\theta_{S2}(\vec{x}_{S2}) = (0, i, 1, j, 0)$, $\theta_{S3}(\vec{x}_{S3}) = (0, i, 2)$ etc. L'ordre lexicographique est alors suffisant pour connaître l'ordre d'exécution d'instances d'instructions ayant de telles fonctions.

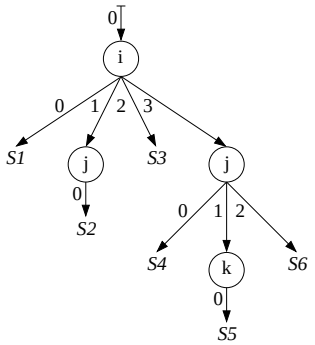



FIG. 7 – AST du programme en figure 3.2

## III.2 Transformations affines

Les transformations de programme dans le modèle polyédrique peuvent être exprimées par des fonctions d'ordonnancement bien choisies. Elles modifient les polyèdres sources en des polyèdres cibles contenant les mêmes points mais dans un nouveau système de coordonnées, donc avec

un nouvel ordre lexicographique. Il a été largement montré que les fonctions linéaires pouvaient exprimer la plupart des transformations utiles. En particulier, les transformations de boucles (comme le retournement, la permutation ou la torsion) peuvent être modélisées par un cas particulier simple appelé *transformations unimodulaires* (la matrice $T_S$ doit être carrée et avoir un déterminant de $\pm 1$) [10, 106]. Des transformations complexes comme le pavage [107] peuvent aussi être appliquées grâce à des ordonnancements linéaires [112]. La puissante théorie mathématique sous-jacente (par exemple pour calculer exactement les dépendances [44]) et sa très intuitive représentation géométrique ont contribué à rendre ce modèle très populaire. De nombreux travaux ont eu pour objet de trouver de bonnes fonctions d'ordonnancement dans un but particulier (par exemple pour le parallélisme ou la localité des données) [17, 25, 33, 40, 44, 45, 110]. Mais la plupart des précédentes méthodes de transformation polyédriques imposaient de sévères limitations sur les fonctions elles mêmes, i.e. être unimodulaires [5, 69] ou tout du moins être inversibles [78, 110, 92, 25]. La raison était que, considérant un polyèdre original défini par le système de contraintes $A\vec{x} + \vec{a} \geq \vec{0}$ et la fonction de transformation $\theta$ menant aux nouvelles coordonnées $\vec{y} = T\vec{x}$, on pouvait déduire que le polyèdre transformé dans le nouveau système de coordonnées était défini par $(AT^{-1})\vec{y} + \vec{a} \geq \vec{0}$, un changement de base. Un autre point qui mena à la popularité des transformations unimodulaires était leurs propriétés face aux compositions. Appliquer une transformation $\vec{y_2} = T_2\vec{y_1}$ après $\vec{y_1} = T_1\vec{x}$ est équivalent à utiliser $\vec{y_2} = T_2T_1\vec{x}$. Et parce que le produit de matrices unimodulaires est une matrice unimodulaire, la composition est encore une transformation unimodulaire.

Dans cette thèse nous n'imposons aucune contrainte sur les fonctions de transformation car nous ne cherchons pas à effectuer un changement de base d'un polyèdre original vers son système de coordonnées final. à la place, nous imposons un nouvel ordre lexicographique au polyèdre en ajoutant de nouvelles dimensions en première position. Ainsi, pour chaque polyèdre $\mathcal{D}$ et fonction d'ordonnancement $\theta$, il est possible de construire un nouveau polyèdre $\mathcal{T}$ ayant l'ordre lexicographique approprié :

$$\mathcal{T} = \left\{ \left(\begin{array}{c} \vec{y} \\ \hline \vec{x} \end{array}\right) \;\middle|\; \left[\begin{array}{c|c} I & -T \\ \hline 0 & A \end{array}\right] \left(\begin{array}{c} \vec{y} \\ \hline \vec{x} \end{array}\right) + \left(\begin{array}{c} \vec{-t} \\ \hline \vec{a} \end{array}\right) \begin{array}{c} = \\ \hline \geq \end{array} \vec{0} \right\},$$

où $I$ est la matrice identité et par définition, $(\vec{y}, \vec{x}) \in \mathcal{T}$ si et seulement si $\vec{y} = \theta(\vec{x})$. Les points à l'intérieur du nouveau polyèdre sont ordonnés lexicographiquement jusqu'à la dernière dimension de $\vec{y}$. Ensuite, il n'y a pas d'ordre particulier à respecter pour les dimensions restantes.

En utilisant cette politique de transformation, les données des domaines d'itération originaux ainsi que celles de la transformation sont inclues dans le nouveau polyèdre. Pour illustrer cela, considérons le polyèdre $\mathcal{D}_{S_2}$ en figure 8(a) et l'ordonnancement $\theta_{S_2}(i, j) = 2i + j$. La matrice de transformation correspondante $T = [2\ 1]$ n'est pas inversible, mais elle peut être étendue en $T = \left[\begin{smallmatrix} 2 & 1 \\ 0 & 1 \end{smallmatrix}\right]$ comme proposé par Griebl et al. [55]. Le polyèdre qui en résulte typiquement est présenté en figure 8(b). Notre politique, par contre, mène directement au polyèdre en figure 8(d), attendu que l'on choisisse l'ordre lexicographique pour les dimensions sans ordonnancement. Une projection sur les dimensions $i'$ et $i$ mènerait au même résultat qu'obtenu en figure 3.5(b). Les dimension additionnelles portent les informations de la transformation, c'est à dire dans ce cas $j = i' - 2i$. Cette propriété est très utile durant la phase de génération de code où nous devons mettre à jour les références aux itérateurs dans le corps des boucles, et est *indispensable* quand la transformation n'est pas inversible.

Une autre propriété de cette politique de transformation est de ne jamais produire un sys-

tème de contraintes rationnel. Cela posait un grave problème à la plupart des précédentes techniques, qui survenait lorsque les fonctions de transformation n'étaient pas unimodulaires. On peut observer ce phénomène en figure 3.5(b). Les points entiers sans marques n'ont pas d'images dans le polyèdre original. En effet, les coordonnées originales peuvent être déterminées à partir des coordonnées cibles par $\overrightarrow{original} = T^{-1}\overrightarrow{target}$. Mais lorsque $T$ est non-unimodulaire, $T^{-1}$ a des élément rationnels. Donc des points entiers dans le domaine cible ont une image rationnelle dans le domaine source ; ils sont appelés les *trous*. Pour éviter de considérer ces trous, les *strides* (les pas entre chaque point entier à considérer) doivent être trouvées. Plusieurs travaux ont proposé d'utiliser la Forme Normale de Hermite [97] de différentes manières pour résoudre ce problème [78, 110, 40, 92]. L'idée principale est de trouver une matrice unimodulaire $U$ et une matrice inversible, non-négative, triangulaire-inférieure $H$ dans laquelle chaque ligne a un unique maximum qui se trouve être l'élément diagonal, et telle que $T = HU$. Alors on peut considérer la transformation unimodulaire intermédiaire $U$ et trouver les strides et les bornes inférieures grâce aux éléments diagonaux de $H$. La décomposition correspondant à notre exemple, $T = \left[\begin{smallmatrix}1 & 0\\ 1 & 2\end{smallmatrix}\right]\left[\begin{smallmatrix}2 & 1\\ -1 & 0\end{smallmatrix}\right]$, la transformation intermédiaire avec $U = \left[\begin{smallmatrix}2 & 1\\ -1 & 0\end{smallmatrix}\right]$, est présentée en figure 3.5(c). Contrairement à cette approche, nous ne changeons pas la base du polyèdre source, mais nous appliquons seulement un ordre lexicographique approprié. Il en résulte que nos systèmes de contraintes cibles sont toujours entiers et qu'il n'y a pas de trous dans les polyèdres correspondants. L'information de stride est explicitement inclue dans le système de contraintes sous la forme d'équations.

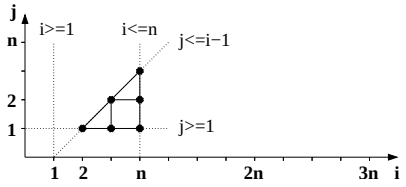
Le coût de cette méthode est d'ajouter de nouvelles dimensions au système de contraintes. Cela peut se révéler un important problème car tout d'abord cela augmente la complexité de l'étape de parcours pour la génération de code. Ensuite, cela augmente la taille du système de contraintes alors même que la génération de code de haut niveau est un problème qui nécessite typiquement beaucoup de mémoire. En pratique, le traitement des dimensions additionnelles est souvent trivial avec les méthodes que nous présenterons en section VI. En définitive notre prototype est plus efficace et a un besoin de mémoire réduit par rapport aux méthodes existantes (voir chapitre 7).

## IV Un modèle pour évaluer le trafic en mémoire

L'amélioration de la finesse de gravure des circuits intégrés se traduit en augmentation de la vitesse pour les processeurs, mais en augmentation de la capacité pour les mémoires. Si rien n'était fait, les gains de puissance des processeurs ne pourraient être exploités, faute de mémoire suffisamment rapide pour les alimenter. Les mémoires hiérarchiques sont une solution très répandue, peu coûteuse et souvent efficace. Pour autant, elles ne sont exemptes ni de limitations ni de défauts. D'une part, on sait que leur mécanisme convient mal aux programmes de calcul numérique qui utilisent de grandes structures de données de façon régulière. D'autre part, leur comportement difficilement prévisible les rend inadaptées aux systèmes temps réel. Enfin, la forte consommation en énergie que requièrent les transferts de données entre les différents niveaux de mémoire freine leur exploitation au sein des systèmes embarqués.

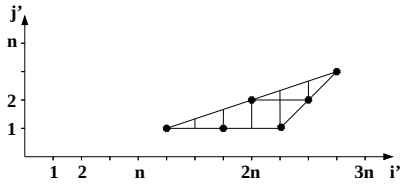
Les compilateurs optimiseurs ont parmi leurs principaux objectifs de transformer les programmes afin d'utiliser au mieux la hiérarchie des mémoires. Il s'agit pour eux d'exploiter au maximum la réutilisation des données. Pour cela, le principe classique est d'appliquer des transformations de boucles [106, 77] guidées par un modèle de coût [85] puis d'effectuer un pavage [107]

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



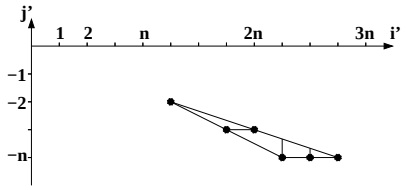

(a) polyèdre original $A\vec{x} + \vec{a} \geq \vec{0}$

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



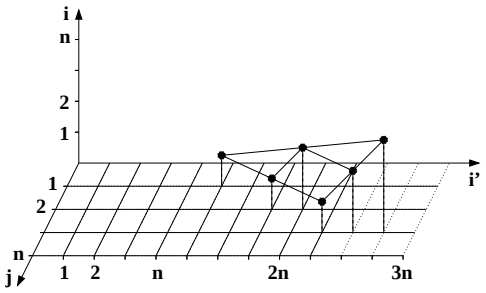

(b) transformation usuelle $(AT^{-1})\vec{y} + \vec{a} \geq \vec{0}$

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




(c) transformation intermédiaire avec la Forme Normale de Hermite $(AU^{-1})\vec{y} + \vec{a} \geq \vec{0}$

$$\left[\begin{array}{c|cc} 1 & -2 & -1 \\ \hline 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -1 \end{array}\right] \left(\begin{array}{c} i' \\ \hline \\ i \\ j \\ \\ \end{array}\right) + \left(\begin{array}{c} 0 \\ \hline -1 \\ n \\ -1 \\ -1 \end{array}\right) \begin{array}{c} = \\ \hline \\ \geq \\ \\ \end{array} \vec{0}$$




(d) notre transformation $\mathcal{T}$

Fig. 8 – Politiques de transformation pour $\mathcal{D}_{S2}$ en figure 3.2 avec $\theta_{S2}(i,j) = 2i + j$

de taille convenablement choisie [37]. Cette démarche a en particulier pour défaut de ne pouvoir s'appliquer que sur des nids de boucles parfaits. On est ainsi souvent amené à convertir des nids de boucles. Du résultat de cette conversion dépend la qualité du pavage final ; or il n'y a pas de méthode systématique pour l'effectuer au mieux [65]. Alternativement à cette approche centrée sur le contrôle, le *data-shackling* propose de raisonner sur les données [66]. Son principe est d'agir directement sur le transfert des données plutôt que par le biais de manipulations des structures de contrôle.

Tous ces algorithmes reposent en général sur des approches heuristiques. Par exemple, considérons deux accès à la même cellule mémoire. Il paraît vraisemblable que plus ces accès seront proches dans le temps, plus le second aura de chances d'être un succès. On cherchera donc à transformer les boucles pour que ces deux accès soient les plus proches possible. Nous voulons au contraire nous appuyer sur une évaluation au moins approximative du trafic entre cache et mémoire principale, et trouver le programme qui minimise ce trafic. La méthode proposée consiste

en un découpage du programme en *chunks*. Nous en présentons le principe en section IV.1. La section V est consacrée à la construction de chunks respectant les dépendances et minimisant le trafic. La section VI montre comment on génère le programme objet quand le système de chunks est donné. Nous terminerons en présentant quelques résultats de notre approche et en indiquant les problèmes qui restent en suspens.

## IV.1 Chunking

Il y a plusieurs raisons aux limitations des précédentes techniques. La première vient du fait que la gestion du cache est laissée à un mécanisme purement matériel. Ainsi il est particulièrement difficile de trouver une solution précise au problème de l'évaluation du trafic pour un type de cache donné. Pourtant, il existe déjà des possibilités, bien que souvent très limitées, pour contrôler le contenu du cache (par exemple la primitive *Write Back Invalidate* WBINVD du jeu d'instructions i386 vide le cache en propageant les modifications apportées aux cellules qu'il contient). Ce contrôle se renforce peu à peu dans les nouvelles architectures (comme IA64) et mieux encore, la plupart des systèmes embarqués récents ont un cache totalement géré par le logiciel de manière à offrir des garanties de type temps-réel. Cette section présente le *chunking*, une technique de transformation des programmes qui s'appuie sur l'existence de tels outils. L'idée principale est de combiner une architecture cible aux propriétés particulières pour éviter les défauts de conflit et un modèle d'exécution singulier pour éviter les défauts de capacité. Nous verrons alors qu'une évaluation du trafic devient possible.

Des hypothèses ont été posées sur le programme source comme sur l'architecture cible. Le programme source doit être à contrôle statique, et l'architecture cible doit disposer quant à elle :

- d'une mémoire principale et d'un unique niveau de cache entre elle et le processeur, la mémoire centrale étant assez grande pour contenir l'ensemble des données nécessaires à l'exécution du programme ;
- d'une mémoire cache idéalement gérée par le logiciel ou sans défauts de conflit i.e. *complètement associative*, mais utiliser un cache moderne à haut degré d'associativité ou considérer un cache plus petit qu'il ne l'est réellement permet d'approcher cette hypothèse de manière satisfaisante ;
- éventuellement d'un jeu d'instructions dédié au contrôle du cache :

| Instruction | Description |
|---|---|
| LOAD CACHED | charge une ligne dans le cache et dans les registres du processeur |
| LOAD NOT CACHED | charge une ligne dans les registres du processeur sans qu'elle soit chargée dans le cache |
| STORE CACHED | force l'actualisation d'une ligne du cache par le processeur |
| STORE NOT CACHED | force l'actualisation d'une ligne de la mémoire par le processeur |
| INVALIDATE | retire une ligne du cache en actualisant ses données en mémoire si elles ont été modifiées |
| FLUSH | généralise INVALIDATE à tout le contenu du cache |

Le chunking est une nouvelle manière de réorganiser les opérations d'un programme. Son

principe est de partitionner l'ensemble des opérations en sous ensembles plus petits dont les données utilisées peuvent tenir dans le cache : les *chunks*. Ces sous ensembles doivent être tels que leur exécution suivant un ordre déterminé soit équivalente à l'exécution du programme original $P$. Les opérations appartenant à un même chunk sont exécutées en respectant $<_P$. En pratique, on numérotera les chunks de manière croissante, dans l'ordre dans lequel ils devront être exécutés. Un numéro de chunk sera assigné à chaque opération : nous cherchons pour chaque instruction $S$ une *fonction de chunking* $\theta_S$ associant à chaque vecteur d'itération $\vec{x}_S$ un numéro de chunk $\theta_S(\vec{x}_S)$. Les opérations ayant le même numéro de chunk seront alors exécutées dans l'ordre séquentiel original. à la fin de chaque chunk, nous devons exécuter le code de gestion de la mémoire cache. Cela consiste à écrire des instructions INVALIDATE pour toutes les cellules mémoires accédées par le chunk. Si cette solution s'avère trop compliquée, il est possible de vider entièrement le cache par un appel à FLUSH. L'ensemble des cellules mémoire accédées par les opérations d'un même chunk $\vec{c}$ est appelée l'*empreinte* de ce chunk, $\mathcal{F}(\vec{c})$ :

$$\mathcal{F}(\vec{c}) = \cup_{S \in \mathcal{S}_P} \cup_{<A,f> \in S} \left\{ A[f(\vec{x}_S)] \mid \vec{x}_S \in \mathcal{D}_S, \theta_S(\vec{x}_S) = \vec{c} \right\}. \tag{4}$$

Ainsi, le déroulement d'un programme ayant subit une transformation de chunking peut être résumé par le pseudo-code suivant :

```
FLUSH
do c=1, N
  | exécution du chunk c suivant <_P
  | INVALIDATE F(c)
```

Où $N$ est le nombre total de chunks et $<_P$ est l'ordre séquentiel original du programme $P$. Chaque chunk doit satisfaire à la *condition de taille* : chaque empreinte doit tenir dans le cache,

$$\forall \vec{c},\ \text{Card}\ \mathcal{F}(\vec{c}) \leq C \tag{5}$$

où $C$ est la taille du cache. Un ensemble de chunks qui satisfont la contrainte de taille et les dépendances est dit un *système de chunks* pour $P$.

La qualité d'un système de chunks peut être quantifiée grâce à deux grandeurs. Tout d'abord, la *taille d'empreinte* qui est le nombre de cellules mémoire accédées par les opérations d'un chunk. Ensuite, le *trafic* qui est le nombre de mouvements de données entre le cache et la mémoire principale. Nous cherchons à construire un système de chunks optimal, c'est à dire tel que chaque empreinte tient dans le cache et où le trafic est minimal. Alors durant l'exécution du programme transformé, les seuls défauts de cache sont ceux obligatoires au début de chaque chunk. De manière à réaliser une étude des dépendances précise et à être en mesure de générer le code final, nous recherchons des fonctions de chunking affines. Donc pour une opération $S(\vec{x}_S)$, instance de l'instruction $S$ avec le vecteur d'itération $\vec{x}_S$ dans le domaine $D_S$, le numéro de chunk peut être écrit :

$$\theta_S(\vec{x}_S) = T_S \vec{x}_S + \vec{t}_S.$$

$T_S$ est une matrice appelée *matrice de chunking*, et $\vec{t}_S$ un vecteur constant.

Le chunking permet de court-circuiter le mécanisme de remplacement des caches gérés par matériel car son principe est de ne jamais l'utiliser. En combinant les propriétés d'un matériel limitant (ou annulant) le nombres de défauts de conflit et le modèle d'exécution du chunking, il devient possible de calculer une évaluation du trafic comme expliqué dans la section suivante. De

plus, le choix d'une méthode contrôlant le contenu du cache permet des garanties de type temps réel. En effet, prédire le contenu du cache et sa stabilité au même point de différentes exécutions devient possible en particulier avec l'utilisation de mémoires locales où les défauts de conflit sont inexistants. Il est tout aussi possible d'utiliser cette méthode sur des systèmes disposant d'un cache conventionnel pour améliorer les performances ou réduire la dissipation d'énergie, et si le temps réel n'est pas important, il n'y a aucun besoin d'insérer des instructions FLUSH pour vider le cache entre chaque chunk puisque le mécanisme de remplacement sélectionne toujours une donnée du chunk précédent pour être remplacée. C'est vrai pour LRU, approché pour FIFO et faux pour la politique RANDOM. Ainsi on évite la surcharge que provoque les instructions FLUSH et on bénéficie de la réutilisation inter-chunk bien qu'elle ne soit pas considérée dans notre modèle.

### IV.2 Calcul du trafic en mémoire

Dans notre modèle, il est possible d'estimer les grandeurs des empreintes et du trafic. Pour une instruction $S$, un tableau $A$ et une fonction d'index $f$, l'empreinte d'un chunk $\vec{c}$ est l'ensemble des données que l'on peut accéder durant l'exécution de ce chunk :

$$\mathcal{F}_{S,A,f}(\vec{c}) = \left\{ f(\vec{x}_S) \mid \vec{x}_S \in \mathcal{D}_S, \theta_S(\vec{x}_S) = \vec{c} \right\}.$$

On considère que le cache est vide avant l'exécution de chaque chunk : une même donnée accédée dans différents chunks comptera donc pour autant de fois dans le calcul du trafic. On peut alors voir le trafic comme le nombre de couples ⟨donnée, numéro de chunk⟩ possibles :

$$\mathcal{T}_{S,A,f} = \text{Card } \left\{ \left\langle f(\vec{x}_S), \theta_S(\vec{x}_S) \right\rangle \mid \vec{x}_S \in \mathcal{D}_S \right\}.$$

Nous nous plaçons dans l'hypothèse classique où le programme original est à contrôle statique, la fonction d'index est donc affine et s'écrit : $f(\vec{x}_S) = F\vec{x}_S + \vec{f}$, où $F$ est la matrice d'index de dimension $\rho(A) \times \rho(S)$, avec $\rho(A)$ le nombre de dimensions du tableau $A$, et $\vec{f}$ un vecteur constant qu'on ignorera. Les ordres de grandeur des cardinaux des ensembles décrivant empreinte et trafic sont alors connus. On sait que si chaque élément de $\vec{x}_S$ est un entier appartenant à un segment de longueur $m$, alors :

$$\begin{aligned} \text{Card } \mathcal{F}_{S,A,f}(\vec{c}) &= O\left(m^l\right), \text{ avec } l = \text{rank} \begin{bmatrix} T_S \\ F \end{bmatrix} - \text{rank } T_S, \\ \mathcal{T}_{S,A,f} &= O\left(m^k\right), \text{ avec } k = \text{rank} \begin{bmatrix} T_S \\ F \end{bmatrix}, \end{aligned}$$

où $\begin{bmatrix} T_S \\ F \end{bmatrix}$ est une matrice constituée pour ses premières lignes de la matrice $T_S$ et pour les lignes suivantes de la matrice $F$. La matrice $F$ peut être obtenue par analyse du code source, $T_S$ quant à elle est l'inconnue qu'il s'agit de découvrir.

## V Chunking, algorithmes et expériences

L'objectif est à présent de savoir comment construire de manière totalement automatique un système de chunks pour tout programme à contrôle statique. Cette construction doit fournir

un système optimal, ce qui sera le cas lorsque chaque cellule apparaîtra dans le moins d'empreintes différentes possibles. Cette condition est suffisante pour garantir que nous avons trouvé le meilleur système possible et pour limiter le nombre de chunks. Les fonctions de chunking sont calculées grâce à des contraintes particulières qui correspondent à autant de propriétés qu'elles doivent respecter. Les section suivantes décrivent chacune d'entre elles. La section V.1 définit les contraintes liées à l'optimisation de la localité temporelle propre. En section V.2 sont présentées les contraintes relatives à la légalité d'une transformation. Les sections V.3 et V.4 montrent les contraintes permettant d'exploiter localité de groupe et localité spatiale respectivement. Enfin nous présentons des résultats expérimentaux en section V.5.

## V.1 Localité temporelle propre

On dispose grâce aux évaluations d'un moyen de quantifier la qualité d'un chunking. Dans le cas d'une instruction comportant $n$ références, les matrices d'index $F_i$ pour $1 \leq i \leq n$ sont connues, on cherche alors à construire la matrice de chunking $T$ ayant les meilleures propriétés. Cette construction est guidée par les évaluations. Nous effectuons tout d'abord une énumération dans l'ordre du trafic croissant des tuples $\left\langle \text{rank } T, \text{rank} \begin{bmatrix} T \\ F_i \end{bmatrix} \text{ pour } 1 \leq i \leq n \right\rangle$ tels que chaque empreinte générée puisse tenir dans le cache. Nous tentons ensuite de construire $T$ sous les conditions de rang du meilleur tuple possible.

Pour une référence isolée, construire une matrice $T$ de rang $v$ telle que $\text{rank} \begin{bmatrix} T \\ F \end{bmatrix} = w$ est toujours possible pourvu que $v$ et $w$ soient des valeurs acceptables et compatibles entre elles. On constitue pour cela une matrice génératrice avec en particulier $\rho(S) - w$ vecteurs d'une base de $\ker F$, puis on en calcule l'inverse. $T$ sera alors composée de $v$ lignes convenablement choisies de la matrice résultat puis complétée de lignes nulles si nécessaire.

Pour généraliser à $n$ références, on doit combiner les $n$ exigences $\text{rank} \begin{bmatrix} T \\ F_i \end{bmatrix} = w_i$ pour $1 \leq i \leq n$. La matrice génératrice doit posséder pour chaque référence exactement $\rho(S) - w_i$ vecteurs d'une base de $\ker F_i$ pour un total d'au plus $v$ vecteurs. Une telle matrice n'existe pas toujours. Le choix des vecteurs à inclure dans la matrice génératrice est déterminant. On peut le guider en préférant ajouter pour chaque référence le plus possible de vecteurs déjà présents dans la matrice. S'il n'existe pas de solution pour un tuple, alors il faut tenter d'en trouver une pour le prochain tuple le plus intéressant.

Il existe toujours une matrice de chunking possible telle que les empreintes tiennent dans le cache. En effet, la contrainte la plus dure pour les empreintes est d'avoir une taille en $O(m^0)$, et la dernière possibilité tentée sera le tuple $\langle \rho(S), w_i = \rho(S) \text{ pour } 1 \leq i \leq n \rangle$. Le chunking correspondant génère pour chaque référence une empreinte en $O\left(m_i^0\right)$ et le trafic maximum en $O\left(m_i^{\rho(S)}\right)$. Sa solution $T = I$ existe toujours et correspond au chunking trivial où chaque opération est dans un chunk.

Considérons le code source en figure 9 où on suppose que `a` est un tableau de `n` éléments capable de tenir dans le cache et que `b` est un tableau de `m` éléments ne pouvant pas tenir dans le cache : Les ordres de grandeur acceptables pour la taille des empreintes sont donc $O\left(n^1\right)$ et $O\left(m^0\right)$. Le programme est constitué de deux instructions :

– l'instruction S1 a une seule référence au tableau `a` avec pour matrice d'index $F_{S1,1} = \begin{bmatrix} 1 \end{bmatrix}$.

```
   do i=1, n
S1 | a(i) = i
   | do j=1, m
S2 | | b(j) = (b(j) + a(i))/2
```

FIG. 9 – Exemple suivi

La matrice $T_{S1}$ ayant les meilleures propriétés correspond au tuple $\langle 1, 1\rangle$, elle génèrera des empreintes de tailles en $O\left(n^1\right)$ et un trafic en $O\left(n^1\right)$. On construit $T_{S1} = \begin{bmatrix} 1 \end{bmatrix}$ ;

– l'instruction S2 possède deux références, l'une au tableau `a` avec pour matrice d'index $F_{S2,1} = \begin{bmatrix} 1 & 0 \end{bmatrix}$ et l'autre au tableau `b` avec pour matrice d'index $F_{S2,2} = \begin{bmatrix} 0 & 1 \end{bmatrix}$. La matrice $T_{S2}$ ayant les meilleures propriétés correspondrait au tuple $\langle 1, 2, 1\rangle$, elle générerait des empreintes de tailles en $O\left(m^0 + n^1\right)$ et un trafic en $O\left(m^1 + n^2\right)$. La construction est possible et donne $T_{S2} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$.

## V.2 Légalité

Puisque le chunking est une technique de réordonnancement des opérations, on doit s'assurer qu'il ne viole aucune dépendance. Rappelons que les chunks sont numérotés dans l'ordre dans lequel ils doivent être exécutés. à l'intérieur de chacun d'eux, les opérations respectent l'ordre séquentiel original. Considérons $I_P$ l'ensemble des instructions du programme $P$, et $\delta_P$ la relation de dépendance sur $P$, un système de chunks est alors valide si et seulement si :

$$\forall S, R \in I_P, \quad S[x]\, \delta_P R[y] \Rightarrow \theta(S[x]) \leq \theta(R[y]).$$

La construction des matrices de chunking est réalisée pour chaque instruction indépendamment les unes des autres. à ce stade, rien n'interdit un entrelacement illégal des opérations. Il est possible de corriger les fonctions de chunking afin qu'elles respectent les dépendances. On dispose en effet sur ces fonctions de certains degrés de liberté que l'on peut exploiter. D'une part, on peut appliquer aux matrices de chunking toutes les transformations qui ne changent pas leurs propriétés de rang. Et d'autre part, on peut renseigner les éléments des vecteurs constants $\vec{t_S}$ des fonctions de chunking. Pour cela, nous calculons l'espace des transformations possible, dit espace de Farkas [44], puis nous résolvons un problème de programmation linéaire en nombres entiers dans cet espace. Ces opérations sont répétées autant de fois qu'il y a de lignes dans la plus grande matrice de chunking. La solution si elle existe donne pour chaque instruction les composantes du vecteur $d$ et les transformations à effectuer sur chaque matrice de chunking. Si aucune solution n'existe, nous devons retourner à la construction des matrices de chunking et tenter la proposition suivante.

Un chunking valide tel que les empreintes tiennent dans le cache existe toujours. Il correspond à la dernière proposition tentée, dans laquelle toutes les matrices de chunking sont des matrices identité. On retrouve alors le programme original, avec un chunk par opération et un trafic maximal.

Reprenons l'exemple commencé en section V.1 correspondant au code en figure 9. Si on se contentait d'utiliser les matrices trouvées, nous obtiendrions les fonctions de chunking suivantes :

- $\theta_{S1}\begin{pmatrix} i \end{pmatrix} = \begin{bmatrix} 1 \end{bmatrix}\begin{pmatrix} i \end{pmatrix} + \begin{pmatrix} 0 \end{pmatrix} = \begin{pmatrix} i \end{pmatrix}$,
- $\theta_{S2}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} j \\ 0 \end{pmatrix}$.

Ces fonctions ne décrivent pas un chunking valide. En effet, la dépendance de $S1$ vers $S2$ est violée. Par exemple, l'opération $S2\begin{pmatrix} 2 \\ 1 \end{pmatrix}$ est exécutée dans le chunk numéro 1 alors que l'opération $S1\begin{pmatrix} 2 \end{pmatrix}$ dont elle dépend est exécutée après, dans le chunk numéro 2. Notre méthode permet de corriger ce chunking afin que toutes les dépendances soient respectées et les propriétés du chunking conservées. La correction proposée par notre prototype est la suivante :

- $\theta_{S1}\begin{pmatrix} i \end{pmatrix} = \begin{bmatrix} 1 \end{bmatrix}\begin{pmatrix} i \end{pmatrix} + \begin{pmatrix} 0 \end{pmatrix} = \begin{pmatrix} i \end{pmatrix}$,
- $\theta_{S2}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} n \\ 0 \end{pmatrix} = \begin{pmatrix} j+n \\ 0 \end{pmatrix}$.

Pour homogénéiser les fonctions de chunking, on peut ajouter des dimensions nulles ou en enlever si elles sont nulles pour toutes les fonctions, puisque cela ne change pas les rangs. On a finalement $\theta_{S1}\begin{pmatrix} i \end{pmatrix} = \begin{pmatrix} i \end{pmatrix}$ et $\theta_{S2}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j+n \end{pmatrix}$. Le code résultant de cette transformation est présenté en figure 10, l'ordre des opérations a été bouleversé pour une exploitation maximale de la localité temporelle compte tenu des hypothèses de départ. Dans le programme résultat, la variable `c` donne à tout instant le numéro de chunk dans lequel on se trouve.

```
   do c=1, n
S1 | a(c) = c
   do c=n+1, n+m
   | do i=1, n
S2 | | b(c-n) = (b(c-n) + a(i))/2
```

Fig. 10 – Exemple suivi après transformation

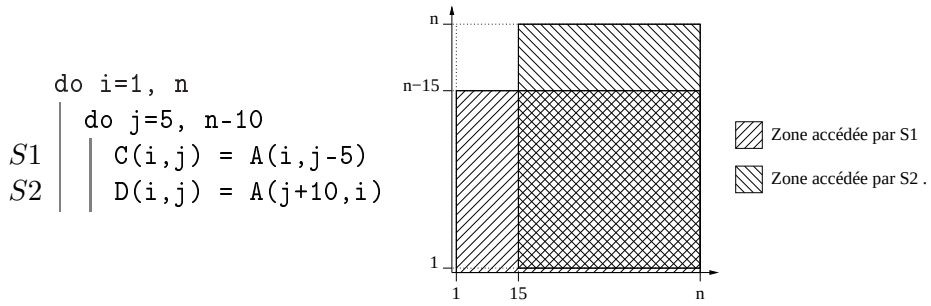
## V.3 Réutilisation de groupe

Il y a réutilisation de groupe lorsque deux instructions $S1$ et $S2$, accèdent le même tableau `A` avec des matrices d'index $F_1$ et $F_2$ (nous utilisons dans cette section des coordonnées homogènes, ce qui signifie que nous incluons les informations sur les paramètres et le scalaire dans les matrices d'index et le vecteur d'itération). Il y a effectivement réutilisation s'il existe deux vecteurs $\vec{x}_1$ et $\vec{x}_2$ tels que $F_2\vec{x}_2 = F_1\vec{x}_1$, et cette réutilisation est exploitée si les deux opérations appartiennent à un même chunk :

$$\forall\vec{x}_1\forall\vec{x}_2, F_2\vec{x}_2 - F_1\vec{x}_1 = \vec{0} \Rightarrow T_2\vec{x}_2 - T_1\vec{x}_1 = \vec{0}. \quad (6)$$

Nous pouvons observer d'une part que cette contrainte est de la même forme qu'une contrainte de dépendance et que nous n'imposons pas que les instructions appartiennent à un même nid de boucles. Nous avons montré que (6) est vraie si et seulement si $\left[T_2\ ,\ -T_1\right] = N\left[F_2\ ,\ -F_1\right]$ où $N$ est une matrice de rang plein (voir section 5.3). Nous résolvons cette contrainte en même temps que nous corrigeons la transformation pour s'assurer qu'elle soit légale.

Par exemple, considérons le code en figure 11(a). Toutes les méthodes centrées sur le contrôle vont estimer qu'il n'y a pas de réutilisation propre ni de réutilisation de groupe exploitable. La raison est qu'elles ne sont pas capables de considérer les références non uniformément générées




(a) code original (b) Zones accédées de `A`

FIG. 11 – Exemple de réutilisation de groupe

(c'est à dire telles que les fonctions d'index ne diffèrent que par le terme constant [49]). En réalité il y a une excellente réutilisation entre les deux instructions sur un partie du tableau `A` comme montré par la figure 11(b). Pour cet exemple il n'y a aucune dépendance, nous pouvons alors utiliser la solution triviale $\left[T_2\ ,\ -T_1\right] = N\left[F_2\ ,\ -F_1\right]$, c'est à dire $T_1 = F_1$ et $T_2 = F_2$. Cela conduit aux fonctions de chunking suivantes :

$$\theta_{S1}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ j-5 \end{pmatrix}; \theta_{S2}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j+10 \\ i \end{pmatrix}.$$

Cette transformation conduit au code final ci-dessous. La localité de groupe est alors maximale : dans la partie partagée de `A`, les deux instructions accèdent la même cellule mémoire durant la même itération.

```
   do c1=1, 14
   |  do c2=0, n-15
S1 |  |  C(c1,c2+5) = A(c1,c2)
   do c1=15, n
S1 |  C(c1,5) = A(c1,0)
   |  do c2=1, n-15
S1 |  |  C(c1,c2+5)  = A(c1,c2)
S2 |  |  D(c2,c1-10) = A(c1,c2)
   |  do c2=n-14, n
S2 |  |  D(c2,c1-10) = A(c1,c2)
```

## V.4 Localité spatiale

Il y a réutilisation spatiale lorsqu'une ou plusieurs références accèdent des données appartenant à une même ligne de cache. Dans notre modèle, la localité spatiale est obtenue lorsque les opérations accédant à cette même ligne appartiennent à un même chunk. Considérons un tableau $A$ et sa matrice d'index $F$. Soit $i$ le numéro de la dimension majeure du tableau $A$, c'est à dire la dimension telle que les données sont rangées successivement en mémoire. $i$ dépend du langage de programmation, par exemple il vaut 1 pour C mais $\rho(A)$ pour FORTRAN. Alors la localité spatiale est obtenue pour $A$ si les opérations accédant les cellules mémoire de la dimension majeure sont dans le même chunk. En d'autre termes nous obtenons la localité spatiale si $F_{i,\bullet} \in \ker T$.

Cette contrainte est ajoutée à l'algorithme de construction de $T$. Si elle empêche la construction de $T$, il est possible de proposer une autre ligne de la matrice d'index (au lieu de $i$) et de proposer la transformation de disposition de données correspondante. Ce résultat peut être comparé à celui de Kandemir et al. [61], où transformations de boucles et de disposition des données sont utilisés conjointement pour améliorer la localité spatiale. Le chunking n'impose pas l'utilisation de matrices de transformation non-singulières, mais ne peut obtenir la localité spatiale que pour un niveau de boucle donné. Cependant les résultats sont souvent comparables.

## V.5 Implantation et résultats

De la recherche des fonctions de chunking à la génération de code, notre méthode a été complètement automatisée. Le prototype *Chunky* implémente l'ensemble du processus en langage C à l'exception du calcul des dépendances et des espaces de Farkas où il utilise encore un code Maple. La résolution des dépendances et la génération de code font une utilisation intensive d'opérations sur les polyèdres. Nous avons pour cela utilisé la PolyLib [105] et PIP [42].

Cette automatisation nous a permis d'effectuer des tests sur différents problèmes non triviaux et ainsi d'évaluer l'intérêt de notre méthode. Nous avons choisi une évaluation aussi précise que possible en utilisant les compteurs matériels pour comparer les nombres de défauts de cache. La machine de test utilisée possède un cache de niveau 1 de 16Ko et un cache de niveau 2 de 256Ko. Nous présentons en figure 12 l'évolution du nombre de défauts de cache du programme utilisé en exemple suivi avant (figure 9) et après transformation (figure 10), en fonction de $m$. Nous avons fixé le rapport $m/n$ à 64 de manière à mettre les phénomènes en évidence. On observe

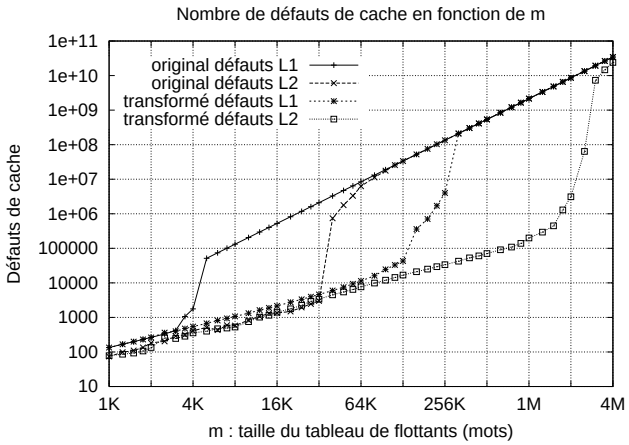



Fig. 12 – Comparaison en défauts de cache de l'exemple

que dans le cas du programme original, dès que le tableau `b` devient plus grand qu'un niveau de cache, le nombre de défauts sur ce niveau grandit brusquement. Le programme transformé a un meilleur comportement, puisqu'il réagit plus tard, quand c'est au tour du tableau `a` de ne plus tenir dans un niveau de cache. Nous avons pu observer le même type de comportement sur la plupart des programmes que nous avons testés. Quelques résultats sur des problèmes bien connus sont résumés en figure 5.10. On y montre le nombre de défauts de cache et les performances pour des tailles de tableaux $m \times m$ tels qu'ils ne tiennent plus en cache de niveau 1 ou 2. Pour

les comparaisons, l'option de compilation est O3 pour les programmes originaux, et O1 pour les programmes transformés, afin d'éviter que le compilateur ne perturbe le chunking. Comme précédemment, les défauts de cache sont prévenus jusqu'au delà d'un ordre de grandeur. On peut observer la répercussion positive sur les performances, bien qu'elle ne soit pas garantie. En effet, malgré le soin apporté à la génération de code, il est parfois difficile d'éviter la présence de structures de contrôle très lourdes ; c'est le cas du programme Gauss - Jordan pour $m = 70$. Cependant, la pénalité pour un défaut de cache L2 étant de l'ordre de 10 fois plus grande que celle de L1, la réduction des défauts de cache L2 conduit presque systématiquement à une amélioration des performances.

## VI Génération de code

Reprenons l'analogie commencée en section III entre modèle polyédrique pour la compilation et transformation de Fourier pour le traitement du signal. Une fois que les transformations utiles ont été appliquées dans l'espace intermédiaire, nous devons revenir à l'espace original en utilisant la transformée de Fourier inverse. Dans le modèle polyédrique, cette transformation inverse qui part de la représentation polyédrique d'un SCoP pour retourner à un arbre de syntaxe abstraite (ou directement au code source final) est appelée *génération de code*. Utiliser des méthodes de génération de code basiques va tendre à produire un programme inefficace qui masquera les effets des transformations optimisantes qui auront été appliquées. Il faut en particulier éviter qu'une mauvaise gestion des structures de contrôle ne détériore les performances.

Le programme original étant à contrôle statique, le domaine d'exécution de chaque instruction peut être décrit par un polyèdre [68]. Dans le cas d'un chunking, on ajoutera à ce polyèdre autant de dimensions et de contraintes que le numéro de chunk en possède. La génération de code est ensuite un problème bien connu de parcours de polyèdres [5] dont la solution la plus aboutie est celle de Quilleré et al. [91]. Cette technique génère chaque niveau de boucle par séparation des polyèdres de manière à ce qu'ils soient disjoints sur la dimension courante, puis récursion sur chacun d'eux pour générer le niveau supérieur, et enfin triage afin de respecter l'ordre lexicographique. Nous avons amélioré cette méthode dans plusieurs directions :

- Nous l'avons complétée de manière à tirer profit des ordonnancements faibles où seul un ordre partiel est imposé au parcours de certains polyèdres. Nous avons proposé de trouver les translations (et éventuellement rotations) des polyèdres sur les axes libres de tout ordonnancement de manière à maximiser le nombre de points entiers dans l'intersection des polyèdres. Nous calculons ce nombre en utilisant les polynômes d'Ehrhart [32]. Cette méthode force le partage du contrôle lorsque plusieurs polyèdres doivent être parcourus par un même code.
- Nous avons montré comment il était possible d'utiliser les pas des boucles pour remplacer de coûteuses conditions comprenant des calculs de modulos générées par l'algorithme de Quilleré et al. Une nouvelle étape dans cet algorithme permet en effet grâce à notre politique de transformation de déterminer le *plus grand pas commun* imposé par les contraintes définissant les polyèdres.
- Nous avons proposé une méthode de réduction de la taille du code généré sans compromis sur l'efficacité du code généré. En effet un des inconvénients de la méthode de Quilleré et al. est d'introduire une explosion inutile du code due à la séparation de certains sommets hors des polyèdres. Nous avons montré que les propriétés de cette méthode nous permettaient

de repérer et de rassembler de manière sûre et efficace les parties inutilement séparées. Ainsi la taille du code est réduite au prix de l'ajout d'itérations dans certaines boucles existantes, ce qui est le minimum en terme de contrôle. Cette méthode nous a permis de réduire la taille des code générés de plus de 30% en moyenne.

– La méthode de Quilleré et al. est une technique basée sur la séparation d'un groupe de polyèdres en polyèdres disjoints et qui a donc une complexité dans le pire des cas de $3^n$ (où $n$ est le nombre de polyèdres) en opérations polyédriques elles-mêmes de complexité exponentielle (pour deux polyèdres $\mathcal{D}_1$ et $\mathcal{D}_2$ nous devons calculer $\mathcal{D}_1 - \mathcal{D}_2$, $\mathcal{D}_2 - \mathcal{D}_1$ et l'intersection entre $\mathcal{D}_1$ et $\mathcal{D}_2$ ; s'il y a un troisième polyèdre nous devons calculer ces trois opérations pour chaque polyèdre résultat et ainsi de suite). Cette propriété rend rapidement la méthode inapplicable à de grands problèmes impliquant plusieurs dizaines voire centaines de polyèdres (et nous avons montré que cela arrivait fréquemment dans les codes réels). Par l'étude de programmes bien connus (SPEC2000fp et PerfectClub) nous avons mis en évidence que les domaines étaient souvent identiques ou disjoints. Nous avons dont mis au point des méthodes rapides pour s'assurer de ces propriétés et ainsi pouvoir donner la réponse triviale à une opération polyédrique coûteuse immédiatement (par exemple la différence entre deux polyèdres identiques est le polyèdre vide). Ainsi de nombreuses opérations n'ont plus à être effectuées et nous obtenons un gain de vitesse d'un facteur supérieur à 4 par rapport au générateur de code le plus fréquemment utilisé.

Reprenons l'exemple en section V.1 dont le code est en figure 9. Les polyèdres décrivant les domaines d'exécution de $S1$ et $S2$ se déduisent de l'étude du code original. On les complète avec l'unique dimension du chunking $c$ et les contraintes qu'elle porte. Les systèmes de contraintes décrivant les polyèdres sont alors :

Système de contraintes pour $S1$ Système de contraintes pour $S2$

$$\begin{cases} c - i = 0 \\ -i + n \geq 0 \\ i - 1 \geq 0 \end{cases} \qquad \begin{cases} c - j - n = 0 \\ -i + n \geq 0 \\ i - 1 \geq 0 \\ -j + m \geq 0 \\ j - 1 \geq 0 \end{cases}$$

Sur la première dimension, $c$, les deux polyèdres sont disjoints : le premier décrit $1 \leq c \leq n$ et le second $n + 1 \leq c \leq n + m$. Il y aura donc un nid de boucle pour chaque instruction. La récursion sur ces nids est ensuite triviale puisqu'ils ne contiennent qu'une instruction chacun. Il s'agit enfin d'ordonner les nids de boucles pour respecter l'ordre lexicographique. On peut facilement remarquer que le premier polyèdre doit précéder le second. Le code produit est celui du programme transformé en section V.2 présenté en figure 10.

# VII Conclusion et travaux futurs

Exploiter la localité est une des clefs pour obtenir de hauts niveaux de performance sur la plupart des ordinateurs et est donc l'un des principaux challenges pour l'optimisation à la compilation. Les transformations de programme sont une des techniques les plus efficaces pour résoudre ce type de problème. Les approches classiques couvrent seulement un ensemble très limité de programmes. Par exemple, seuls les nids de boucles parfaits ou les nids de boucles complètement permutables sont majoritairement considérés. De la même manière, l'ensemble

des transformations acceptées est lui aussi souvent très réduit, par exemple aux transformations unimodulaires. Dans cette thèse, nous avons présenté un schéma de transformation pour améliorer la localité comme les performances et qui peut automatiquement appliquer des transformations complexes à un large panel de programmes. Tous les aspects d'une méthode d'amélioration de la localité de haut niveau ont été explorés, de l'extraction des données utiles à partir du programme source à la génération du programme cible optimisé.

## VII.1 Contributions

**Un schéma de transformation amélioré** Dans la section II nous avons défini notre modèle de programme, une légère extension des boucles à contrôles statiques bien connues. Ce modèle a pour but d'absorber la grande majorité des noyaux de calcul intensif qui s'inscrivent dans le modèle polyédrique dans les programmes généraux en séparant contrôle statique et références statiques. Nous avons pu montrer expérimentalement que les parties de programmes entrant dans ce modèle étaient très présentes dans les applications réelles mais aussi qu'il existe de nombreuses manières d'en augmenter automatiquement le nombre comme la taille.

Les parties de programmes à contrôle statique peuvent être manipulées en utilisant la représentation polyédrique de chaque instruction. Les précédentes méthodes n'étaient pour la plupart pas capables d'utiliser des fonctions d'ordonnancement affines générales pour transformer le programme car elles considéraient chaque transformation comme un changement de base de la représentation polyédrique. En conséquence, elles nécessitaient des traitements laborieux à plusieurs étapes de la transformation, comme étendre les fonctions non inversibles ou calculer leurs inverses voire la Forme Normale de Hermite de ces inverses si elles n'étaient pas unimodulaires. Au lieu de cela nous avons proposé en section III une méthode simple qui permet de composer avec les transformations non-unimodulaires, non-inversibles, non-entières et même non-uniformes en modifiant l'ordre lexicographique grâce à de nouvelles dimensions. Nous sommes alors capables de traiter simplement des transformations complexes.

**Amélioration de la localité par le chunking** Nous avons présenté en section IV une nouvelle méthode permettant d'améliorer la localité inspirée de l'existence d'outils pour gérer le cache depuis le logiciel. Nous avons suggéré de restructurer le programme en des ensembles d'opérations appelés *chunks*. Au commencement de chaque ensemble, le cache est vide. La taille des chunks est choisie de manière à ce que les données accédées tiennent dans le cache. Le mécanisme de remplacement qui est particulièrement difficile à modéliser n'intervient donc pas. à la fin de chaque chunk, le cache est vidé et on commence l'exécution du suivant. Nous avons aussi montré qu'il n'était pas utile de vider le cache quand le mécanisme de remplacement était LRU ou FIFO.

Dans ce modèle, il est possible de fournir des estimations asymptotiques du trafic en mémoire et d'utiliser cette information pour trouver la meilleure transformation. Nous avons montré en section V que les propriétés du programme final, incluant la légalité et tous les types de localité, pouvaient être exprimées sous forme de contraintes sur la fonction de transformation. Nous avons défini les algorithmes qui construisent les fonctions de transformation de manière à respecter ces contraintes. Notre méthode possède de certains avantages puisque son domaine d'application est l'ensemble des programmes à contrôle statique, sans contraintes de structure ou de forme des dépendances. Des fonctions de transformation affine sont automatiquement construites par nos algorithmes et notre prototype. Nous avons montré expérimentalement que notre méthode était

capable d'améliorer la localité comme les performances dans des cas traditionnellement difficiles, par exemple les nids de boucles non-parfaits, les situations de dépendances complexes ou les références non-uniformément générées. Cette méthode ne nécessite rien d'autre que le code initial et les grandeurs relatives du cache et des données, elle peut ainsi fournir des solutions s'adaptant à différentes tailles du jeu de données.

**Une méthode pour corriger les transformations** Dans la section V, nous exposons une méthode générale pour corriger les transformations de programme de manière à ce qu'elles soient légales, et ce, sans conséquence sur leurs propriété de localité. Cette méthode a été implantée dans notre prototype, remplaçant avantageusement certains pré-traitements et évitant à un nombre conséquent de transformations d'être simplement ignorées. Elle peut être utilisée combinée avec de nombreuses méthodes d'amélioration de la localité existantes, dans le cas mono-processeur comme dans celui des systèmes parallèles qui utilisent des placements spatio-temporels [73]. Nous pensons que cette méthode peut être étendue à la correction de transformations dédiées à d'autres fins telle que la parallélisation automatique, mais la démonstration est laissée à de futurs travaux.

**Génération de code pour les optimisations fragiles** La pratique courante dans l'optimisation de programme est de découpler la sélection de la transformation optimisante et son application au code source original. La plupart des transformations sont des réordonnancements, suivis optionnellement par des modifications des instructions elles-mêmes. L'outil ou le programmeur qui va effectivement transformer le programme doit être informé de la réorganisation souhaitée. cela est fait classiquement par le biais de directives comme *pave*, *fusionne* ou *tords*. Il est difficile de savoir si un ensemble de directives est complet ou de comprendre de quelle manière elle peuvent interagir. Nous pensons que donner une fonction d'ordonnancement est une autre manière de spécifier une transformation, et qu'elle a plusieurs avantages sur celle utilisant des directives. Elle est plus précise, a de meilleurs propriétés de composition et dans de nombreux cas on peut calculer les fonctions d'ordonnancement automatiquement. Le principal désavantage est que construire un programme à partir d'un ordonnancement prend du temps et peut introduire des surcharges de calcul à l'exécution. Par exemple il n'est pas acceptable de sauver un défaut de cache coûtant 10 cycles processeur en générant un contrôle qui en coûte 11.

Nous avons montré en section VI plusieurs méthodes pour éviter la surcharge de contrôle depuis la sélection des transformations de programme jusqu'à la génération de code elle-même. Nous avons profondément amélioré un des algorithmes les plus performants pour générer des codes particulièrement efficaces. Nos contributions incluent l'amélioration du partage de contrôle entre instructions, la suppression du contrôle complexe et la limitation de l'explosion de la taille du code. Par ailleurs nous avons aussi proposé des solutions afin de réduire la complexité de cette méthode. Nous pensons que des outils comme CLooG ont éliminé la difficulté que représentait la génération d'un code efficace en un temps acceptable. La transformation source-à-polyèdre-à-source a été appliquée avec succès à 12 benchmarks parmi les SPEC2000fp et PerfectClub avec un gain de vitesse d'un facteur $4,05$ par rapport au plus utilisé des générateurs de code, pour les parties des benchmarks que ce dernier était capable de traiter.

## VII.2 Futurs

Notre travail laisse de nombreux problèmes, questions palpitantes et opportunités d'application ouverts. Le schéma de transformation présenté dans cette thèse n'est pas adapté à tous les types de programmes. Par exemple, appliquer le chunking à un programme à contrôle statique sans réutilisation aboutira sans doute seulement à augmenter la charge de contrôle. De même nous avons vu que lorsque les jeux de données sont extrêmement volumineux, le chunking ne produit qu'une solution triviale ou un pavage ayant des formes de pavés simples. Pourtant dans les autres cas, il se révèle particulièrement efficace et peut optimiser des programmes là où les techniques existantes seraient inapplicables ou n'auraient aucun effet. Il serait donc intéressant de disposer d'un critère pour savoir quand utiliser le chunking. Des travaux plus poussés d'implantation sont de plus nécessaires pour supporter des benchmarks complets dans notre prototype et pour disposer de plus de résultats statistiques, en particulier sur les transformations corrigées. De plus la question du passage à l'échelle est laissée ouverte car avec plusieurs dizaines d'instructions profondément imbriquées, le nombre d'inconnues dans les systèmes de contraintes peuvent devenir excessivement grands. Séparer le problème en fonction du graphe de dépendances est une solution à l'étude.

Les travaux en cours concernant la génération de code sont de trouver de nouvelles améliorations à la qualité du code généré. Notre attention se porte en particulier sur de meilleures solutions au recouvrement de polyèdres de même qu'à la solution générale au problème des pas de boucles. Malgré nos contributions, il existe encore des cas réels menant à une explosion en taille ou en temps. Utiliser le *pattern matching*, c'est à dire éviter des calculs polyédriques lourds dans des cas simples (e.g. domaines rectangulaires), semble être une piste prometteuse pour réduire le temps de génération de code. Nous avons montré dans cette thèse que la principale cause de l'explosion de la taille du code provenait du nombre de paramètres libres, car leurs interactions menaient à une augmentation exponentielle de code généré. En amont de la génération de code, il est possible pour les compilateurs de réduire à la fois en temps et en taille la génération de code en trouvant des contraintes affines sur les paramètres des parties à contrôle statique [38]. Une autre voie est de repérer les noyaux de calcul intensif et de diriger le générateur de code afin qu'il n'optimise que la partie concernée.

Un de nos buts quand nous avons mis au point le chunking était la compilation pour des systèmes disposant de mémoires locales. Cette mémoire remplace souvent les caches classiques sur les systèmes embarqués. Elle n'a ni mécanisme de remplacement ni système d'adressage associatif et peut être gérée par le compilateur. Puisque les applications embarquées n'ont typiquement pas de paramètres (par exemple les tailles d'images pour les traitements vidéo sont fixes), nous pensons que nos méthodes peuvent être étendues et dérivées dans ce but. De la même manière elles pourraient être utilisées pour les calculs *out-of-core*, c'est à dire pour améliorer l'utilisation de la mémoire virtuelle pour les programmes utilisant des jeux de données extrêmement grands. Parce que ce type de mémoire est complètement associatif, il est une cible particulièrement appropriée pour notre politique de transformation.

# Chapter 1

# Introduction

The *First Draft of a Report on the EDVAC* by John von Neumann (1945) presented the computer infrastructure known as the *von Neumann Architecture* where the memory and the arithmetic and control units are separated. Today and near future systems are still using this organization without any real alternative. As a consequence the overall performance of a computer highly depends on the ability of the memory to provide data to the processing unit at the required speed. Unfortunately the arithmetic effectiveness of microprocessors increased and is still increasing far much faster than memory speed. We estimate that the performance gap between processor and main memories grows by 50% per year [58], this phenomenon is known as the *memory gap*. This led in the 60s to major design evolutions. First, the use of CISC (Complex Instruction Set Computers) instruction set architectures allowed to produce smaller program sizes and fewer calls to memory. Second, *cache memories*, some intermediate memories smaller and faster than the main one, were introduced. Purpose was to take advantage of the speed of the fastest memory (level 1 cache) combined to the size and to the price of the slowest (the RAM), by putting in the cache the most useful data for the processor at any time. In present day designs, the memory gap is so large that we need several caches to bridge it, thus creating a complex memory hierarchy.

The effective use of this memory hierarchy is one of the keys to achieve the best performance. The most widely used solution is to reorganize the input programs in such a way that the number of exchanges of data between the cache and main memory is minimized. Thus once a datum is brought into the cache, the target program should execute as many instructions referring to this datum as possible. The first attempts to achieve this goal were programming crafts, leading to non-portable, nearly illegible, and error-prone codes. To bypass these problems, many works proposed to put the optimization work under the responsibility of the compiler. Several directions were investigated. First, to adapt well known program restructuring techniques already used for automatic parallelization as unimodular transformations [10] or tiling [107]. Second, to apply data transformations in order to use the less possible amount of data as with array restructuring [95] or to map the array data layout to the access pattern [77]. Unfortunately, most cache optimization techniques suffer big limitations. It is difficult to choose the useful program transformations, as well as the order in which to apply them. Furthermore, these transformations are very sensitive to data dependences. To bypass the dependence problem, most of the existing methods apply only to very simple programs in which dependences are non-existent or have a special form. Data layout transformations lack flexibility and do not find an optimal layout for the whole program in real-life cases.

None of the current techniques takes into account the existence of tools allowing the software to participate in cache management. We propose in this thesis a new method of locality optimization, based on the use of these tools. The principle is to reorganize the operations of a program to form subsets for which all the referenced data hold in the cache. This technique, called *chunking* takes advantage of the cache memory by being freed from inherent limitations of already existing methods. Furthermore, controlling the cache contents at any time allows us to offer guarantees of real-time type.

For a clear understanding of nowadays memory designs, we will first recall in section 1.1 the different ways to implement memory at the hardware level. Then in section 1.2 we define the principles of locality, and show how computer designers took advantage of these opportunities to build cheap and efficient memory systems. Nevertheless we will also show the limits of their solution, and present in section 1.3 related work on methods to bypass these limitations by exploiting the locality at compile time. Lastly we discuss in section 1.4 the weakness of the existing data locality improvement methods and present in section 1.5 an overview of our attempt to overcome these restrictions.

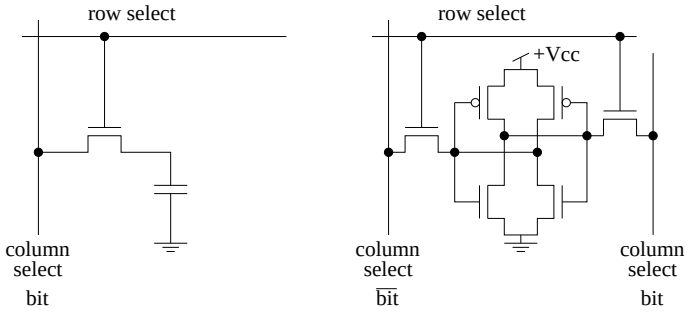
## 1.1 What is memory ?

Basically, memory is a two-dimensional array of memory cells, each datum being defined by its coordinates i.e. the row and column numbers. The random access memory (RAM) family includes two major memory devices: dynamic RAM (DRAM) and static RAM (SRAM). They differ from each other by the lifetime of the stored data, the latency, the bandwidth and the cost-per-byte.

On one hand, the cheapest way to make a memory cell is to use a transistor to drive a capacitor whose status determines whether the memory cell holds a logical 1 (filled capacitor) or 0 (empty capacitor). This is the basic principle of all DRAM memories, Figure 1.1(a) gives an illustration of such a memory cell. Unfortunately, using capacitors leads to some limitations. First they leak their charge due to faulty insulation, hence memory cells need to be refreshed periodically. This task is done by the DRAM controller and requires clock cycles during which the memory cells can not be accessed. Nevertheless this is not a major issue since a given DRAM cell is unavailable for about 1% of the time because of refresh. The main problem with DRAM is that the readout is destructive since reading a cell discharges the capacitor. When a memory cell is accessed, the whole row is first copied to the *sense amplifiers* before choosing the right element thanks to the column number, then each time a row is accessed it has to be rewritten. Moreover the address bus is typically multiplexed between row and column components, introducing delay between row and column address strobe signals, once again limiting performance.

On the other hand SRAM is only composed of transistors, with the data being stored by switching transistors in the right state, as in a processing unit. Figure 1.1(b) illustrates the so-called *flip-flop* organization of such memory cells. Therefore both modifying and reading the memory are very fast operations. Moreover, SRAM keep its content as long as electrical power is supplied to the chip, hence there is no need of refresh operations. Eventually, SRAM can run at higher clock speeds than DRAM, with a much lower latency. But as shown by Figure 1.1(b), we can see that SRAM needs more transistors than DRAM and therefore is far more expensive to produce. As a consequence, most computers use both memory devices. SRAM is typically used

along a critical data path where access speed is the most important (e.g. for cache memories), while DRAM is used for everything else.




(a) Dynamic RAM cell (b) Static RAM cell

Figure 1.1: RAM memory families

## 1.2 Principle of Data Locality

Memory accesses are known to be partly predictable since a long time. During a given period, most programs refer preferably to a small fraction of their address space. This hypothesis on the memory access behavior is called the *data locality principle*. We may divide it in two hypothesis reflecting the dimensions of the principle:

**Principle 1.1 (temporal locality)** *Once a given data is accessed, it will tend to be accessed again during the next clock cycles.*

**Principle 1.2 (spatial locality)** *Once a given data is accessed, nearby data in the address space will tend to be accessed during the next clock cycles.*

### 1.2.1 Exploitation

Computer designers took advantage of the principles of locality to offer low-cost fast memory systems by introducing hierarchical memories. The basic idea is quite natural: since at a given time only a small part of the memory is useful, it may be stored in a small but efficient SRAM device, while the remaining data may reside inside a large, cheap but slow DRAM. Once a datum is considered to be useful for the processing, it has to be brought inside the fastest memory by replacing another datum considered to be less useful.

The organization and the functioning of hierarchical memories systems are simple: the highest level of the hierarchy is the processor registers, the lower levels are slower but bigger (and less expensive) memories called *cache* memories. The largest but also the slowest (and the least expensive), is main memory. We may even consider swap disks as the lowest level of hierarchy. If a datum is present at a level of the hierarchy, it will be present also in every lower levels. Basically, when a processor needs a datum, it looks for it in the memory hierarchy from the upper to the lower level. If the datum is found in level $n$, we say that a level $n$ *hit* occurred and

that a *miss* occurred for every upper levels. Typically, the hit time for level 1 cache is 1 cycle, then we have to multiply by a factor 10 for each additional level (10 cycles for level 2, 100 for level 3 etc.). The additional time to reach a datum onto a lower level is called the *miss penalty*. To check quickly whether a data is in the cache or not, we typically use an *associative* (or *content addressable*) memory that may be accessed by content rather than by address. Such memory device includes comparison logic. A data key is basically broadcast to all words of storage and compared with the corresponding keys to find the value associated to that key.

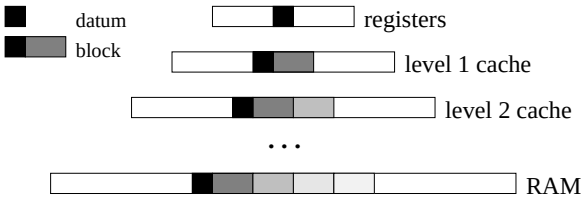



Figure 1.2: Memory hierarchy

When the datum is found, it is copied up to the upper level (temporal locality: if the processor wants to reference it again a short time later, it will be in the highest hierarchy level, thus referenced in the fastest way). Furthermore, some nearby data in the address space are copied at the same time (spatial locality: if the processor wants to reference one or some of these data a short time later, they will be in the top of the hierarchy, thus referenced in the fastest way). The data set simultaneously copied is called a *line* or a *block*. Supplementary data to the one that was referenced are its neighbors in the address space. A typical size for the cache line is a few bytes (16 to 64) for the level 1 cache, and this size grows with the cache level (basically by a factor of 2).

Since each level is smaller than the previous one, a block having to take place into a given level will necessarily replace another block (at least once this level is full). Choosing a block for eviction is a critical step since we may want to avoid *cache interferences*, i.e. to replace a block when it is still useful. Depending on the mapping strategy, the new block may only be placed in a restricted set of slots. Three mapping policies may be considered: if a block can be stored only into a given place in the cache, it is said to be *direct mapped*. In this case, the cache works like a hash-table by mapping a block thanks to a hash-function. For instance the block 3 at level $n$ in Figure 1.3 may be stored into the place (3 modulo 4) = 3 when the level $n+1$ has 4 slots. If the cache is divided into sets of size $m$ such that a given block can only take place into a given set thanks to a hash-function, it is said to be *m-way set associative*. For instance, the block 3 at level $n$ in Figure 1.3 may be stored anywhere in the set (3 modulo 2) = 1 when the level $n+1$ is two-way set associative. Lastly, when a given block can be placed anywhere in the cache, it is said to be *fully associative*.

Let us describe more precisely how a cache memory works with an illustration of a 2-way set associative cache as shown in Figure 1.4. The address of a datum we are looking for in the cache is shown in Figure 1.4(1), it is divided onto three parts: a block offset that identifies a datum within a block and the *key* of the associative memory which is composed of an address tag and a cache index. First, the index selects the set to be checked in Figure 1.4(2), since we are considering a 2-way set associative cache, the index is sent to 2 banks. Next, the tag of the datum we are looking for and the tags of the entries of both banks that correspond to the

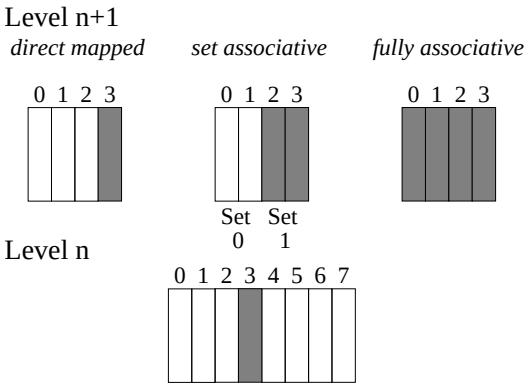



Figure 1.3: Cache associativity policies

index value are compared in Figure 1.4(3). We have to take care of the validity bit to ensure that the tag refers a valid block. Lastly, in Figure 1.4(4) a multiplexer will send the block that corresponds to the tag if it exists. From this example we can derive the organization of other cache types: increasing the associativity will increase the number of banks, comparators and multiplexers, up to $N$ and no index for $N$ blocks in the fully associative case, down to 1 and a heavy index in the direct mapped case.

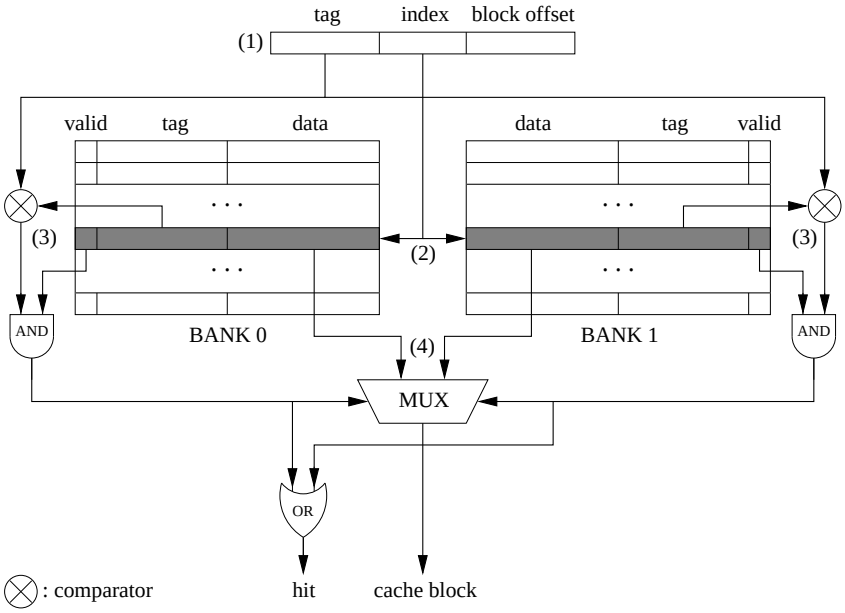



Figure 1.4: A 2-way set associative cache example

There exists several *replacement mechanisms* to choose which block has to be replaced when a choice is possible (i.e. when the cache is not direct mapped). The closer to the temporal locality principle is the *least-recently used* (LRU) policy: the block chosen for eviction is the one that has not been accessed during the longest time. Unfortunately this strategy needs to remember the access order and is costly to handle for large sets or fully associative caches. Thus there exists simpler and less effective strategies such as *first-in first-out* (FIFO) or RANDOM policies that

maintain a queue or select a block for eviction randomly respectively. In practice, associative caches are the most widely used, and the associativity degree grows with the level number. The replacement policy is very often a *pseudo-LRU* trying to emulate the LRU mechanism in a cheaper (and not well defined) way.

As a conclusion, there exist three reasons for a datum to be out of the cache:

- *Compulsory miss*: the datum belongs to a block that has never been referenced earlier, thus that has never been brought to the cache.
- *Capacity miss*: the data set of a program cannot fit in the cache, thus there must be exchanges between memory levels and a datum may be evicted from the cache many times.
- *Conflict miss*: if the cache is not fully associative, a block may be evicted if others block have to be stored in the same slot.

More precise information about the functioning of cache memories and more generally about hierarchical memories systems may be found in [58, 100].

### 1.2.2 Limits

For most of the general purpose programs, cache memories actually allow to benefit from data locality and to hide the main memory latency from the processor. But in situations where a large amount of data is accessed uniformly, e.g. for signal processing or scientific computing, this mechanism may be no more suitable. When processing large regular data structures as matrices or vectors, data may be evicted from the first cache levels between two accesses although they may be reused many times. Dramatic performance degradation may result from such temporal locality loss. In the same way, accessing non-consecutive data in memory does not allow to take advantage of spatial locality and results in poor performance as well.

Let us illustrate how it is possible to easily challenge the hierarchical memory systems. Experiments were made on an i386 machine providing a unique 32KB cache level and using a 32 bytes length line. In Figure 1.5 a program repeats $n$ times a processing on $m$ consecutive data

```
do i=1, n
| do j=1, m
| | v(j) = v(j) + constant
```

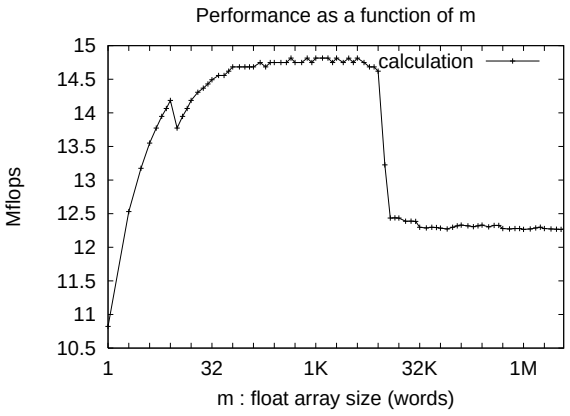



Figure 1.5: Temporal locality loss

of floating type (4 bytes) of an array $v$. Once $m$ becomes too large for all data to fit in the cache together, many $v$ elements have to be accessed from the main memory in a slower way. The performance degradation is about 20%. We may notice that the value $m$ to observe this phenomenon is not very high, for instance even a picture of 128x128 pixels for image processing is too big to be held in a 32KB cache. Hence, poor performance because of a bad exploitation of the memory hierarchy is a common problem in real life programs.

Figure 1.6 shows a similar example except that the processing of the innermost loop is no more on consecutive data in the address space. The program makes jumps in the array $v$ instead of processing cells one after the other. Thus it do not take advantage of the spatial locality and the performance deterioration increases with the length of the jump. This deterioration stops when the jump length becomes equal to the block length. At this moment, every data can only be found in the main memory and the performance degradation is about 80%. We may argue that such programming is not natural. However, it is found either explicitly e.g. in cyclic reduction algorithms or implicitly when we use many dimensional arrays. For instance, programmers may not be conscious that when they work with arrays, the data of the lines are consecutive in memory when they use the C language (*row-major* data layout) while the data of the columns are consecutive in memory in FORTRAN (*column-major* data layout). Thus depending on the way they access the arrays they may benefit from a lot of spatial locality or not.

```
do i=1, n
| do j=1, m, jump
| | v(j) = v(j) + constant
```

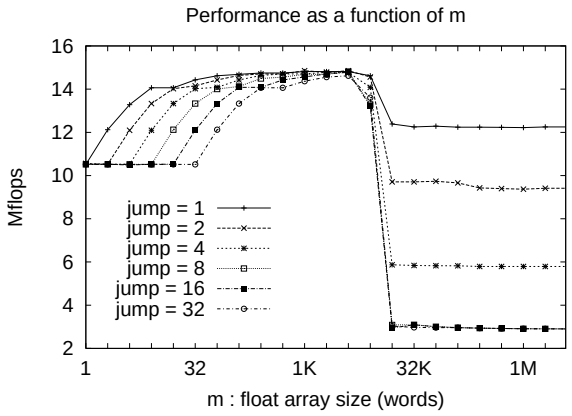



Figure 1.6: Temporal and spatial locality loss

These limits were known since the cache memories introduction. Since then, a lot of work focused at improving this situation.

## 1.3 Classical Data Locality Optimizing Approaches

We can consider two main families of techniques that aim at improving data locality: *program transformations* and *data transformations*. These techniques try to bring the processing on some memory cells closer by reordering the statement instances in the time space or the memory cells in the address space respectively. Thus data should be still in the cache when the program will access them again. The first family aims at finding a better execution order for operations by modifying control statements while the second one tries to adapt the memory data layout. In

the following sections, we will outline these techniques.

### 1.3.1 Program Transformations

Program transformations were the first proposed solutions for improving data locality. The basic idea is to modify the statement instance execution order by restructuring the input program in order to benefit from data reuse. They consist at first on various well known methods that have been shown to improve data locality and whose application was the responsibility of the programmer [2]. These techniques may have effects in various levels of the memory hierarchy: on top with scalar replacement and unroll-and-jam [39, 27]), on some levels with tiling [107, 106] or on all levels with loop transformations [85, 108]. Many loop transformations such as loop permutation, splitting, merging, skewing, reversal etc. may be considered for that purpose [108]. The resulting code was difficult to understand and non portable. As a result, many researches aimed at doing the optimization at compile time.

The natural solution was to drive the application of well known loop transformations during the compilation process [11, 64, 108, 85]. The problem was that each transformation technique has its own properties, e.g. how it may modify the program semantic or how we can decide if applying the transformation is a benefit or not. Furthermore it is difficult to understand their interactions. Hence it is not only hard to decide which transformation may be used, but also in which order we should apply them. As a result, some works proposed to use only a restrictive set of transformations and a definite order known to be useful for some typical problems. For instance, Gannon et al.[49] consider only loop permutation by evaluating data locality according to a given permutation. Thus they have to consider $n!$ possible transformations where $n$ in the depth of the loop nest in order to find the best transformation. McKinley et al. [85] proposed a technique based on a detailed cost model that drives the use of loop permutation, fusion and distribution. First, their algorithm use loop distribution to divide the loop nests to achieve the finest granularity i.e. the minimum number of statements inside each innermost level. Next, it determines for each loop nest the loop that may be placed innermost to achieve the best locality, and applies loop permutation for that purpose. If permutation is proved to change the program semantic, it tries to apply fusion to enable it. Once as many permutations as possible have been achieved, it applies loop fusion to increase the granularity and benefit from reuse between statements. Unfortunately such rigid frameworks may not be adapted to every input programs.

On the opposite to restructuring frameworks that select reordering by way of directives like *fuse*, *permute* or *skew*, another solution is to specify *scheduling functions* [62]. This framework that allows to handle many transformations in the same model has its roots in systolic array design for mapping computation to processors, a *space* mapping. It may be used in the same way to map computation in the time space, a *time* mapping. The power of such unified model is to handle sequences of directives as a single transformation. Thus finding the sequence boils down to finding the best *scheduling*. For instance the *unimodular transformation* framework is one of the most widely used model [10, 106]. It can represent any combination of loop permutation, reversal and skewing that may be applied to a given loop nest as a single transformation with unified properties (e.g. for dependence testing). Wolf and Lam [106] use this framework in combination with *tiling* [107] in their data locality optimizing algorithm. They use estimates of both spatial and temporal reuse to drive the transformation computation and to avoid loops that either do not carry reuse or cannot be permuted without modifying the program semantic. Loechner et al.

build unimodular transformations automatically according to the properties of data references and data dependences [82]. Li [77] generalizes the framework of unimodular transformations [10] by using linear, non-unimodular transformations to change the execution order.

Alternatively to these control centric techniques, Kodukula et al. [66] propose a data centric approach that plans to act on data movement directly, rather than as a side-effect of control flow manipulations. Their method aims at partitioning the code in pieces which are (almost) free of cache misses for a given data set: a given array is divided into blocks then they execute sequentially the program slice that access each block.

Because program transformations may modify the program semantics, they have to be chosen carefully in such a way that they respect *data dependences*. A typical solution is to focus only on programs where dependences are simple enough to enable any transformations (fully permutable loop). In compensation of the need for very simple dependences, first works on compiler techniques for improving data locality discuss *enabling transformations* to modify the program in such a way that the proposed method can apply. McKinley et al. [85] use *fusion* and *distribution* mainly to enable loop *permutation*. Wolf and Lam [106] proposed in their algorithm to use *skewing* and *reversal* to achieve a fully permutable loop and enable *tiling* as in previous works on automatic parallelization. A significant step on preprocessing techniques to produce fully permutable loop nests has been achieved by Ahmed et al. [1]. They use Farkas Lemma to find a valid *code sinking*-like transformation [108] if it exists. More sophisticated methods build transformations directly according to data dependences. For instance Loechner et al. propose to dedicate and deduce a part of the transformation function for that purpose [82]. Exact methods designed for automatic parallelization restrict the search space to the legal transformation space [44, 79]. The method of Griebl et al. [53] propose to use space-time mappings (i.e. a schedule and a placement) to provide a fully permutable loop nest well adapted to further tiling transformation. Their aim is to minimize the amount of communication in a distributed program, which is indeed a kind of locality optimization. They first take care of dependences by finding a legal space-time transformation and then tile in space-time to achieve the optimal granularity. Adapting these ideas to cache optimization seems by no mean obvious, although it is an interesting subject for further research.

### 1.3.2 Data Transformations

Another way to optimize locality is to act on data directly. We can distinguish between two main direction: *data layout* transformations that aim at reordering data in the address space in order to benefit from spatial locality and *memory contraction* to reduce the memory needed to execute a given program.

Optimizing data locality by using data layout is a more recent approach than program transformations. Li put its foundations in his thesis [77], expanding Wolf's theory [106]. The basic idea comes from a simple observation: the spatial locality depends on the data layout. Let us recall for instance that in the C language, the data rows of a matrix are ordered successively in memory (*row-major*), while this is the case of columns in FORTRAN (*column-major*). To improve locality for a given code with program transformations would lead to a different result for each of these cases. Hence instead of adapting program to a given memory layout, this approach adapts the layout to the program. Contrary to the program transformations, that can achieve an optimized code equivalent to input code, the data transformation take place more profoundly

within the compiler and may leave the code intact. So, they are not limited by dependences. Every data structure of the program can have a different layout in memory. Leung and Zahorjan showed how data layout transformations may improve locality while program transformations could not by using unimodular data transformation matrices for uniprocessor case [75]. O'Boyle and Knijnenburg achieve the same conclusion by using a more general class of transformation, i.e. non-singular, and by applying it in the multiprocessor case [87]. Before them, Anderson, Amarasinghe and Lam proposed an algorithm based on strip-mining and unimodular transformations to allow the processors of shared memory machines to reference consecutive data [7]. One of the most precise method for improving spatial locality is provided by Loechner et al. [82] by ensuring that the order of data in the address space corresponds exactly to the access ordering. The limitations of data layout transformations are on one hand that they can improve only spatial locality, and on the other hand, that we can guarantee their efficiency only for a given reference in a given loop nest. Indeed, an array layout transformation may affect all the reference to the considered array in the whole program. But the best data layout may not be the same for each reference to a given array, thus the best tradeoff has to be found.

Another data transformation aims at minimizing the need for memory of a given program by using *array contraction*. The basic idea is that if an array is referenced in an loop, and generates only loop independent dependences (between two statement instances in the same loop iteration) and is no more referenced after the loop, we may reduce the array dimensionality and even replace it by privatized scalars [108, 76, 79]. As a consequence, the array do not pollute the data cache. This improvement is usually enabled by using some preprocessing as applying fusion to loops that read and write onto the same array [50]. More accurate methods use loop reversal, permutation and fusion to expose array contraction opportunities [95, 76]. Lim improved the range of applicability of the method by generalizing array contraction to converting an array to a lower-dimensional array instead of a scalar [79].

### 1.3.3 Mixed approach

A recent direction is to propose a combination of program and data layout transformations. Such an approach is natural considering that the limits and possibilities of each separate method are somewhat complementary. Program transformations are limited by dependences but can be applied to program parts while data layout transformations can ignore dependences but should affect the whole program. Cierniak and Li have been the first to use such an approach [31], and were able to achieve in most of cases better performances than separate approaches. They use both non-singular transformations to restructure a given loop nest and an affine data mapping function to modify the memory layout of a given array. They introduce *stride vectors* to decide whether applying a mapping and transformation is a benefit or not. Due to the complexity of finding an optimal solution, they restrict the search space (transformation function components may only be 0 or 1) and rely on heuristics. Kandemir, Ramanujam and Choudhary propose to use an unified framework for both loop and data transformations and do not restrict the search space of possible program transformations [60]. Kandemir et al. [61] extend the research space by considering every data layout that can be expressed by hyperplanes and extend the method to several loop nests but the method still suffers some deficiencies (for instance it does not take advantage of tiling).

## 1.4 Some Weakness of Existing Methods

The classical data locality improvement framework often suffers from well known limitations. It mostly addresses a quite restricted set of transformations (i.e. unimodular or in the best case invertible transformations, see chapter 3) and a quite restricted set of input programs (e.g. perfectly nested, fully permutable loop nests, see chapter 2). We will propose solutions to widely extend both possible transformations and input programs. Furthermore we believe that the classical framework is not able to improve data locality even in many cases it is supposed to handle.

Classic optimization techniques typically deals with the following question: *which loops carry the most reuse* [49, 64, 11, 85, 106, 108] ? A loop is said to *carry reuse* if a data location is accessed in different iterations of this loop while outer loop iterations are fixed. For instance let us consider the following matrix by vector multiply code:

```
do i=1, n
| do j=1, m
| | x(i) = x(i) + A(i,j) * y(j)
```

Loops $i$ and $j$ carry reuse since all their iterations access $y[j]$ and $x[i]$ respectively. Basically, once the reuse for each loop is quantified thanks to some cost model, then transformations are applied in order to move the loops carrying the most reuse at the innermost levels. The underlying idea is to bring accesses on same memory cells closer and consequently to reduce the probability of a reference to be evicted from the cache before being reused. After that reordering, if more than one inner loop carry reuse, and if it is possible, a *tiling* is applied to these loops [107]. By reducing the number of iterations in the innermost loops, this transformation allows to exploit reuse in more than one dimension. We define this classical framework that rely on control to carry reuse and to drive transformations as being *control driven*.

Next sections will describe why control driven frameworks fail to achieve data locality in the general case. Through these demonstrations, we will describe our way to consider the problem and to bypass the preceding limitations: a *data driven* framework.

### Unexploited reuse

Although the control driven framework has shown a certain efficiency, and in spite of the huge amount of work based on it (e.g. see [108] and the references therein), it is clearly unadapted to some kind of problems. Let us illustrate this fact thanks to a first simple example, a two polynomials multiply code:

```
do i=1, n
| do j=1, m
| | z(i+j) = z(i+j) + x(i) * y(j)
```

As for the matrix by vector multiply case, the two loops carry self-temporal reuse because of the references $x[i]$ and $y[j]$. Formally, a loop $k$ carry self-temporal reuse of a reference if the index variable for the $k^{th}$ loop does not appear in the subscript of this reference [108]. In this way, it

is neither possible to detect, quantify nor exploit the reuse generated by the reference $z(i+j)$, while it produce the same amount of reuse as the other references. Each memory cell $z(l)$ being actually accessed as many times as there exist pairs $\langle i, j \rangle$ such as $i + j = l$, *i.e.* up to $m + n$ times in our example.

Because there are some reuse sources that are not carried by a specific loop, control driven methods cannot exploit them. Instead of asking which loops carry the most reuse, we will propose a method where the main question is *which references generate the most reuse ?*

Next, let us consider the following two matrix by vector multiplies, where one matrix is the transposition of the other one:

```
do i=1, n
| do j=1, n
| | x1(i) = x1(i) + A(i,j) * y1(j)
| | x2(i) = x2(i) + A(j,i) * y2(j)
```

By using a control driven approach, we can easily see that the loops carry self-temporal reuse because of the references to all the vectors. It is clear that there is no loop carrying the reuse generated by the two references to the matrix $A$, while they access exactly the same data but not in the same order. Basically, the group-reuse study is always made between uniformly generated references, *i.e.* such that their index functions differ in at most the constant term [49] (for instance references `A(i,j)` and `A(i,j+1)` are uniformly generated, `A(i,j)` and `A(j,i)` are not). The reason is that it is supposed that the data reuse is exploitable only in these cases. On one hand, this makes sense when all the references belong to the same statement, because all the transformations applied to the whole program have the same impact on each of them (as long as we do not split the statements). In this case, if the references are not uniformly generated, the actually exploitable reuse is very low. On the other hand, considering only uniformly generated references makes no sense when they are not in the same statement, because we can apply a different transformation for each of them (in particular with our methods). Thus loops can be transformed in such a way that data accessed in their bodies will correspond at least in part.

By considering a loop body as a block and by applying transformations only to this block, there are some sources of reuse that cannot be exploited. Instead of trying to find a transformation for the whole loop body, we will propose a method where *each statement has a specific transformation function.*

### Non-adaptable solutions

Previous works on data locality improvement make one of the following assumptions: there is no parameters in loop bounds or their values are supposed to be infinite (*i.e.* we have always to consider the worst case). The first hypothesis is not realistic, and the second one do not always give the best results since most of the time, the right transformation to apply depend on the parameter values. For instance, let us consider the following code and the cache size $C$:

```
do i=1, n
| do j=1, m
| | invReq(i) = invReq(i) + nbR(i) / R(j)
```

If according to the cache size, $2n$ is very small while $m$ is very large, then it should be more interesting to put the j-loop outermost in order to benefit for the high self-locality of the reference $R(j)$. In the same way if $m$ is very small while $2n$ is very large regarding $C$ then we should leave the i-loop outermost to benefit for the self-locality of both references $invReq(i)$ and $nbR(i)$. In contrast, without any knowledge on the relative sizes of the cache and data, a classical method using tiling would only suggest to tile both loops. This is not the best solution except when both $2n$ and $m$ are bigger than the cache size, and this is likely to introduce some control overhead.

Trying to find the perfect code versioning that will enumerate the best codes for every case of program parameters or system features is not realistic. We will propose a neat tradeoff by considering the orders of magnitude of the number of accessed data for each reference. If parametric values are found, we can either *produce a restricted code versioning*, possibly being helped by interaction with the programmer.

## 1.5 Thesis Overview

This thesis will address all aspects of locality optimization, from extracting the useful informations in the source code to final code generation. Is is divided into three parts. The first part will introduce the basic concepts. In chapter 2 we present our program model and experimental results on real-life applications. We will see in this chapter that contrary to a common belief, a large part of the program code in popular benchmarks fit the polyhedral model. Chapter 3 describes the transformation framework in this model. This framework will remove usual limitations to only unimodular or invertible transformations and will open the way to complex code restructuring without challenging the code generation process. The second part of the thesis focuses on data locality improvement. In chapter 4 we discuss a model for evaluating the memory traffic. Thanks to a singular execution scheme, we will be able to compute an estimate of the memory traffic as a function of the transformation. We show in chapter 5 how we can benefit from such an information to build transformations to optimize data locality. We will see that the method can apply onto traditionally challenging problems as complex data dependences, complex program structures or non uniformly generated references. The last part is dedicated to the final step of our framework: code generation. Chapter 6 presents the problem of generating code in our model. We will show that the quality of the target code highly depend on the transformation functions and how they can be constructed or post-processed to avoid control overhead. Chapter 7 presents our improvements to an existing algorithm to generate a very efficient code and preserve performance. These improvements include support of loop steps, avoiding code size explosion and reducing code generation complexity. Finally, we conclude in chapter 8 and suggest future work.

# Part I

# Basic Concepts

# Chapter 2

# Program Model

The program model allows and constrains at the same time opportunities to expose data reuse, to propose effective solutions for exploiting this reuse and to apply these solutions. On one hand we could handle very general programs, but their analysis may not be possible for state-of-the-art compilers or may be too difficult to be achieved in a reasonable amount of time. On the other hand it is possible to work on a very restricted set of programs such that the analysis is easy, but such cases may not reflect real-life programs. Thus it is necessary to find a good tradeoff between representativeness and analysis power.

The well known static control program class [43] appears to be a convenient candidate. First of all it addresses programs with highly regular access patterns that challenge cache memories, thus it is an interesting target for data locality improvement purpose. Next, it embeds a large range of programs including loop nests which are known to be the most compute intensive kernels in general applications. Lastly it opens the way to the *polyhedral model* and the subsequent mathematical tools allowing to e.g. achieve an exact dependence analysis. The properties of static control programs as defined in [43] can be roughly summarized in this way: (1) control statements are **do** loops with affine bounds and **if** conditionals with affine conditions; (2) arrays are the only data structures, and their subscripts are affine; (3) affine bounds, conditions and subscripts depend only on outer loop counters and structure (or size) parameters; (4) subroutine and function calls have been inlined.

In the following, we will extend the concept of static control programs and study the relevance of such model in real-life programs. In section 2.1 we present the background of this work. In section 2.2 we focus on the pure control side of the model while in 2.3 we discuss data structures and references.

## 2.1 Background

In the following, we will distinguish between several abstraction levels of program commands. A *statement* is a program structure directing the computer to perform a specified action. This action may be evaluating an arithmetic expression, instantiating a variable or calling a procedure. The set of statements in a program $P$ is called $\mathcal{S}_P$. A statement can be executed several times (for instance when it belongs to a function or when it is enclosed inside a loop); each instance of a statement is called an *operation*. The set of operations in a program $P$ is called $\mathcal{O}_P$. A

statement instance in a high-level language can represent several machine-language *instructions*. These abstraction levels are illustrated in Figure 2.1: in Figure 2.1(a) the statement $S$ is enclosed in a loop and will be executed 100 times. Next, in Figure 2.1(b) is shown an operation, instance of the statement $S$ for $i = 4$, we write it $S(4)$. Lastly, the machine-language instructions corresponding to the operation $S(4)$ are listed in Figure 2.1(c).

```
  do i=1, 100
S | a(i) = a(i-1) + 1
```

(a) Statement level

```
a(4) = a(3) + 1
```

(b) Operation level

```
...
movl -60(%ebp), %eax
incl %eax
movl %eax, -56(%ebp)
```

(c) Instruction level

Figure 2.1: Program command abstraction levels

A *loop nest* is a finite set of nested iterative program structures, as they exist some in every imperative languages like C or FORTRAN, with the following form:

```
do x1 = L1, U1, S1
| ...
| do x2 = L2, U2, S2
| | ...
| | do xn = Ln, Un, Sn
| | | Body
| | ...
| ...
```

where `Body` can be a list of program commands or a loop nest. A loop nest is said to be *perfect* when every program commands belong to `Body`. Considering the $k^{th}$ loop, $i_k$ is an integer variable called the *loop iterator* or *loop counter*. The value of the loop counter is updated at the end of each iteration of the loop by adding $S_k$, an integer, non-zero variable called the *step* of the loop. $L_k$ and $U_k$ are expressions corresponding to the *loop bounds*: the starting value of the iterator is $L_k$, and the loop ends when it becomes larger than $U_k$.

A loop nest can be represented using a $n$-entry column vector called its *iteration vector*:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix},$$

where $x_k$ is the $k^{th}$ loop index, $n$ is called the *loop depth* and $\mathbb{Z}^n$ is the *iteration space*. The set of possible iteration vector values for a given statement is called the *iteration domain* of the statement. The statement is executed once for each element of the iteration domain. Thus, each integral point of the iteration domain corresponds to an operation. The notation $S(\vec{x})$ refers to the operation instance of the statement $S$ with the iteration vector $\vec{x}$. Figure 2.2 shows an example of a loop nest and its iteration domain representation where each axis corresponds to a loop, and each dot is an iteration of the loop nest driving the execution of the statement $S$.

```
do i=1, 6
| do j=min(max(6-i,1),3), max(8-i,2*i-5)
S | | a(i,j) = a(i-1,j)
```

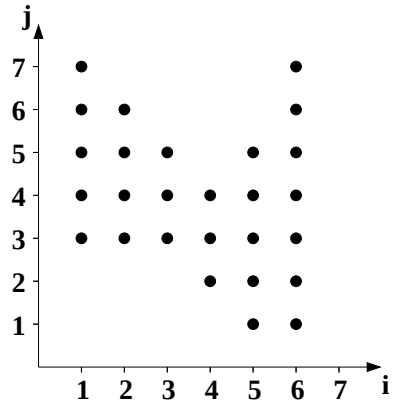



Figure 2.2: Iteration domain of a loop nest

## 2.2 Static Control Parts

In this work we do not consider loop nests as general as discussed in section 2.1. Instead, we study a program class called *static control parts* (SCoPs). This program type is defined in section 2.2.1, in section 2.2.2 we present a brief survey on automatic methods to improve the static control ratio in general programs. In section 2.2.3 we outline the static control parts extraction algorithm then we discuss their significance in real applications in section 2.2.4.

### 2.2.1 Definition

When loop bounds and conditionals enclosing a statement are affine functions that only depend on surrounding loop counters, formal parameters and constants, the iteration domain can always be specified by a set of linear inequalities defining a polyhedron [68]. The term *polyhedron* will be used in a broad sense to denote a *convex set of points in a lattice* (also called $\mathbb{Z}$-polyhedron or lattice-polyhedron), i.e. a set of points in a $\mathbb{Z}$ vector space bounded by affine inequalities [97]:

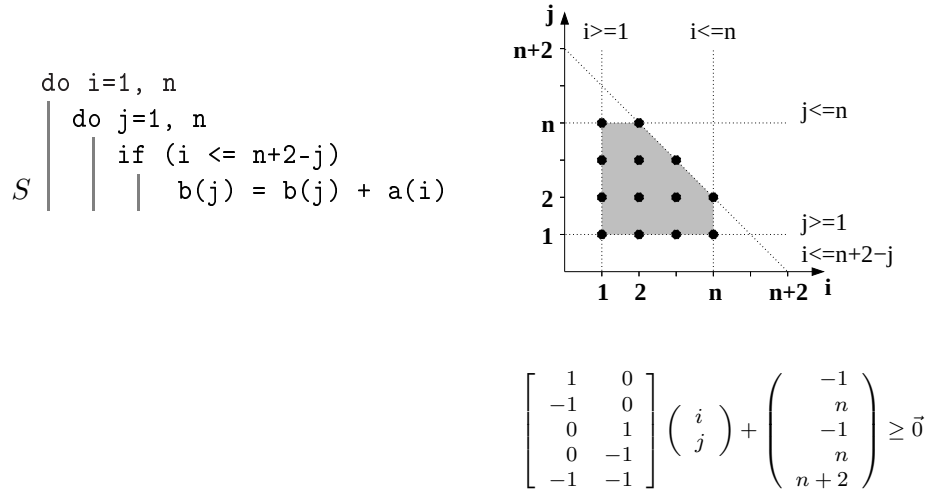
$$\mathcal{D} = \left\{ \vec{x} \mid \vec{x} \in \mathbb{Z}^n, A\vec{x} + \vec{a} \geq \vec{0} \right\},$$

where $\vec{x}$ is the iteration vector, $A$ is a constant matrix and $\vec{a}$ is a constant vector, possibly parametric. The iteration domain is a subset of the iteration space: $\mathcal{D} \subseteq \mathbb{Z}^n$. Figure 2.3 illustrates the correspondence between surrounding control and polyhedral domains: the iteration domain in Figure 2.3(b) can be defined using affine inequalities that are extracted directly from the program in Figure 2.3(a). The set of affine constraints may be represented with the matrix notation as shown in Figure 2.3(b). We can easily derive that any statement with a surrounding control of the following form for each loop level $x_k$ has static control:

```
...
do x_k = MAX_{i=1}^{m_l} ⌈(L_i(x_1,...,x_{k-1}) + l_i)/c_k⌉, MIN_{i=1}^{m_u} ⌊(U_i(x_1,...,x_{k-1}) + u_i)/c_k⌋
| if (AND_{i=1}^{m_g} G_i(x_1,...,x_k) + g_i ≥ 0⃗ )
| | ...
| | | Statement
```

The program in Figure 2.2 is not a SCoP since its iteration domain is not convex. However, it can be split into a union of several convex sets that may be considered separately. Xue showed

how is is possible to find automatically the convex components of such programs where the loop bounds are expressed using any composition of minima and maxima of affine expressions [111].




(a) surrounding control of $S_1$ (b) iteration domain of $S_1$

Figure 2.3: Static control and corresponding iteration domain

**Definition 2.1 (SCoP)** *A maximal set of consecutive statements in a program with convex polyhedral iteration domains is called a static control part, or SCoP for short.*

The SCoP definition is a slight extension of *static control* nests [43] introduced in [16]. Within a function body, a static control part is a maximal set of consecutive statements without `while` loops, where loop bounds and conditionals may only linearly depend on invariants within this set of statements. These invariants include symbolic constants, formal function parameters and surrounding loop counters: they are called the *global parameters* of the SCoP, as well as any invariant appearing in some array subscript within the SCoP. As such, a SCoP may have arbitrary memory accesses and function calls; a SCoP is thus larger than a static control loop nest [43]. A static control part is called *rich* when it holds at least one non-empty loop; *rich* SCoPs are the natural candidates for polyhedral loop transformations. An example is shown in Figure 2.4. We will only consider *rich* SCoPs in the following.

### 2.2.2 Preprocessing For Static Control

We can consider two main stream in the automatic parallelization and optimization field. The first one try to exit from (or to extend) the polyhedral model. For instance, considering extensions to *sparsely-irregular* codes (with a few unpredictable conditionals and `while` loops) is possible [36, 109], although a *fully* polyhedral representation is not possible anymore. Unifying hybrid static-dynamic optimization [93] with the polyhedral model would also be interesting. On the other hand, the second stream try to put more programs into the polyhedral model by using some preprocessing.

Many preprocessing methods are possible. *Constant propagation* may be useful for instance to put non-linear expressions to affine form as shown in Figure 2.5(a). Some `while` loops may

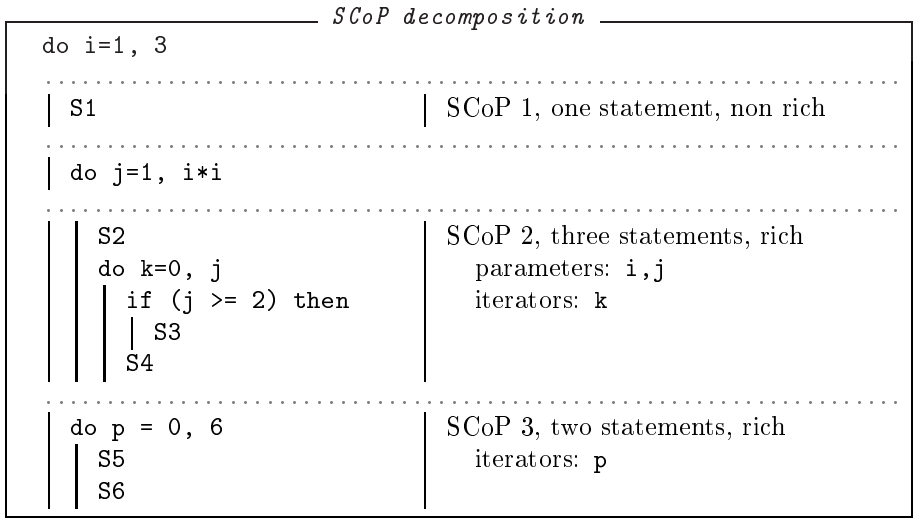



Figure 2.4: Example of decomposition into static control parts

be converted to `do` loops as long as it is possible to determine the number of loop iterations as illustrated in Figure 2.5(b). We may also refer in the same way to *goto elimination* to convert a `goto`-based control to `do` loops. *Loop normalization* is often used to simplify a loop and to avoid non-unit loop steps, it usually replaces a loop `do i=L, U, S` with a new loop `do ii=1, (U-L+S)/S` and changes each reference to the original loop counter `i` with `ii*S-S+L`. A final assignment `i=ii*S-S+L` should also be inserted after the end of the normalized loop. An example of such transformation is shown in Figure 2.5(c). Since it is not possible to check directly whether a call to a function leaves a static loop, we may rely on *inlining* as shown in Figure 2.5(d). Parameters in affine expressions are supposed to be constant, if they refer to induction variables that are not outer loop counters, we have to eliminate these induction variables as illustrated in Figure 2.5(e). More details on these transformations and other methods may be found in [86].

### 2.2.3 Automatic Discovery of SCoPs

SCoP extraction is greatly simplified when implemented within a modern compiler infrastructure such as Open64/ORC. This compiler accept multiple front ends: C, C++, FORTRAN 77 and 90. Previous phases include function inlining, constant propagation, loop normalization, integer comparison normalization, dead-code and `goto` elimination, and induction variable substitution, along with language-specific preprocessing: pointer arithmetic is replaced with indexed arrays, pointer analysis information is available (but not yet used in our tool), etc.

The algorithm for SCoP extraction proceeds through the following steps:

**Gather useful information:** traverse the syntax tree of a given function (after inlining) and store `do` loop counters, bounds and strides, conditional predicates, array references, and parameters involved in these expressions.

**Recognize affine information:**

- among `do` loops, select the static control ones by checking the bound expressions and the invariance of the loop counter and parameters within the loop body;

```
n = 10
do i=1, m
| do j=n*i, n*m
| | S1
```

original non-static loop

```
do i=1, m
| do j=10*i, 10*m
| | S1
```

target static loop

(a) Constant propagation example

```
i = 1
while (i<=m)
| S1
| i = i + 1
```

original non-static loop

```
do i=1, m
| S1
```

target static loop

(b) While-loop to do-loop conversion

```
do i=1, m
| do j=i, n, 2
| | S1
```

original static loop

```
do i=1, m
| do jj=1, (n-i+2)/2
| | S1(j = 2*jj - 2 + i)
```

target static loop

(c) Loop normalization example

```
do i=1, m
| function(i,m)
```

original non-static loop

```
do i=1, m
| do j=1, n
| | S1
```

target static loop

(d) Inlining example

```
ind = 0
do i=1, 100
| do j=1, 100
| | ind = ind + 2
| | a(ind) = a(ind) + b(j)
| c(i) = a(ind)
```

original non-static loop

```
do i=1, 100
| do j=1, 100
| | a(200*i+2*j-200) =
| |      a(200*i+2*j-200) + b(j)
| c(i) = a(200*i)
```

target static loop

(e) Induction variable substitution example

Figure 2.5: Preprocessing for static control examples

- select the static control conditionals whose predicate is an affine expression of parameters and loop counters;
- restrict previous static control loops to those enclosing only static control loops and static control conditionals;
- select the array references whose subscript is an affine expression of parameters and loop counters.

**Build static control parts:** traverse the syntax tree, starting with the creation of a new SCoP. Let $N$ be the first operational or control node in the loop body:

- if $N$ is a static control loop (this has to be checked recursively for each loop), add this loop and its enclosed statements to the SCoP;
- if $N$ is a static control conditional, add this conditional with its branches to the SCoP;
- if $N$ is not a conditional or loop node, add it to the SCoP;
- otherwise, close the current SCoP and create a new one;
- drop the SCoP if it does not contain at least one loop.

Then, set $N$ to the next node and continue the traversal.

**Identify the global parameters:** for each SCoP, iterate over the loop bounds, conditional expressions and array subscripts to gather the global parameters.

This algorithm outputs a list of SCoPs associated with any function in the syntax tree.

### 2.2.4 Significance Within Real Applications

Thanks to an implementation of the previous algorithm into Open64, we studied the applicability of our polyhedral framework to several benchmarks[1].

Figure 7.7 summarizes the results for the SpecFP 2000 and PerfectClub benchmarks handled by our tool (single-file programs only, at the time being). All codes are written in Fortran77, except for art and equake in C, and lucas in Fortran90. The first column shows the number of functions, outlining that code modularity does not seem to impact the applicability of polyhedral transformations (inlining was not applied in these experiments). The *SCoP* section gives some general informations about SCoPs. The first column count the total number of SCoPs. The next column is very encouraging since a majority of SCoPs happen to be *rich*: they enclose at least one loop; this means that a majority of loops will be captured by our representation. The next two columns count the number of *rich* SCoPs with at least one global parameter and the maximum number of parameters inside a given SCoP. These columns advocate for parametric analysis and transformation techniques. The last column of the section shows how many *rich* SCoPs enclose at least one conditional; it shows that only a few conditional expressions are affine, leaving room for techniques that do not handle static-control conditionals. The next two columns in the *Statements* section shows that rich SCoPs hold a large majority of statements which reinforces the coverage of static control parts, and also illustrates the computationally intensive nature of the benchmarks (many statements are enclosed in loops).

[1] The only criterion for selecting a given benchmark is the ability of ORC/Open64 2.0 to compile it with the -keep option, i.e. to allow us to extract the WHIRL, the intermediate representation of ORC/Open64.

In accordance with earlier results using Polaris [41], the coverage of regular loop nests is strongly influenced by the quality of the constant propagation, loop normalization and induction variable detection.

| | | SCoPs | | | | | Statements | |
|---|---|---|---|---|---|---|---|---|
| | Functions | All | Rich | Parametric | Max parameters | Affine ifs | All | Rich |
| applu | 16 | 25 | 19 | 15 | 6 | 1 | 757 | 633 |
| apsi | 97 | 109 | 80 | 80 | 14 | 25 | 2192 | 1839 |
| art | 26 | 62 | 28 | 27 | 8 | 4 | 499 | 343 |
| lucas | 4 | 11 | 4 | 4 | 13 | 2 | 2070 | 2050 |
| mgrid | 12 | 12 | 12 | 12 | 4 | 2 | 369 | 369 |
| equake | 27 | 40 | 20 | 14 | 7 | 4 | 639 | 489 |
| swim | 6 | 6 | 6 | 6 | 3 | 1 | 123 | 123 |
| adm | 97 | 109 | 80 | 80 | 14 | 25 | 2260 | 1899 |
| dyfesm | 78 | 112 | 75 | 70 | 4 | 3 | 1497 | 1280 |
| mdg | 16 | 33 | 17 | 17 | 6 | 5 | 530 | 464 |
| mg3d | 28 | 63 | 39 | 39 | 11 | 6 | 1442 | 1242 |
| qcd | 35 | 74 | 30 | 23 | 8 | 6 | 819 | 641 |

Figure 2.6: Coverage of static control parts in high-performance applications

Our tool also gathers detailed statistics about the number of parameters and statements per SCoP, and about statement depth (within a SCoP, not counting non-static enclosing loops). Figure 2.7 shows that almost all SCoPs are smaller than 100 statements, with a few exceptions, and that loop depth is rarely greater than 3. Moreover, Figure 2.8 shows that deep loops also tend to be very small, except for applu, adm and mg3d which contain depth-3 loop nests with tenths of statements. This means that most polyhedral analysis and transformations will succeed and require only a reasonable amount of time and resources. It also gives an estimate of the scalability required for worst-case exponential algorithms, like the code generation phase to convert the polyhedral representation back to source code.

### 2.2.5 Going Further

As shown by our experimental results, a typical problem that can be extracted by state of the art compilers is a SCoP including about ten statements with a loop depth of 3. Such SCoPS may include many optimization opportunities and they are small enough to be supported by high complexity methods. Nevertheless it may be advantageous to consider bigger problems e.g. to expose more group locality. This can be done by improving or activating the useful preprocessing as shown in section 2.2.2, but another solution is to *merge* some SCoPs. Two SCoPs may be trivially merged if (1) they belong to the same loop and are at the same loop level, (2) there are no data dependences between the SCoPs and the non-static program part between them and (3) no parameter of a SCoP is updated in the other SCoP. Then the non-static part can be moved after the two SCoPs and we can consider the new bigger static part.

On the other hand, when some problems are proved to be challenging for a given optimizing method (for instance because of a worst case exponential complexity), it may be interesting to *split* a SCoP into several parts. There exists many ways to split a SCoP, from considering the different loop nests separately up to applying any kind of affine partitioning [80].

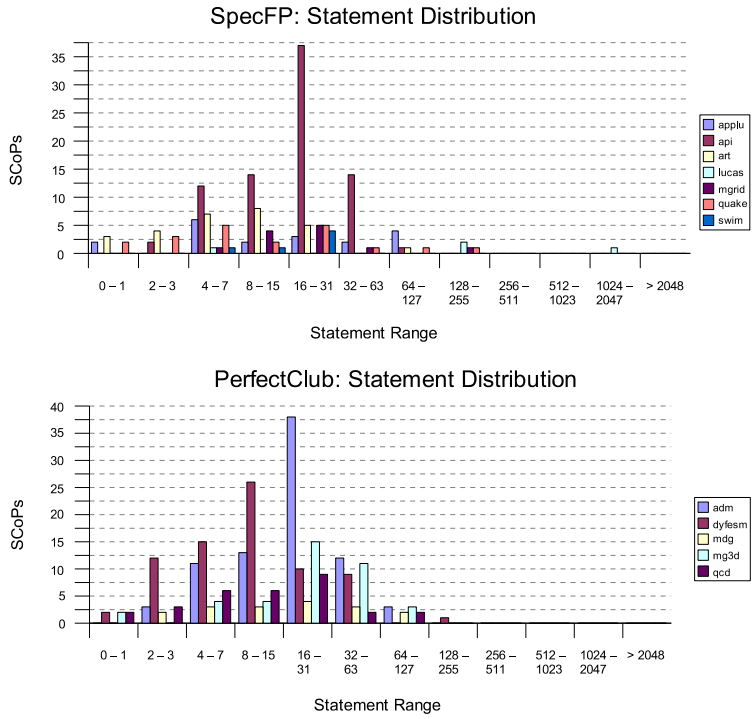



Figure 2.7: Distribution of SCoP size

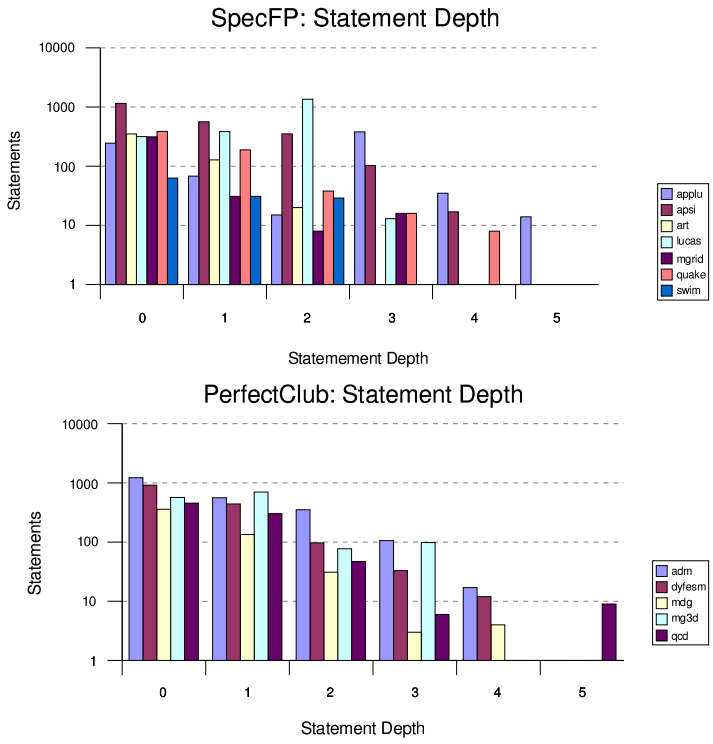



Figure 2.8: Distribution of statement depths

## 2.3 Static References

Each language provides several ways to refer to values or data in program statements, e.g. scalars, arrays, pointers or functions. In this work we will consider only a subset of the possible references called *static references*.

### 2.3.1 Definition

Each statement may include one or several references to arrays (or scalars which are particular cases of arrays). The set of arrays of the program $P$ is $\mathcal{A}_P$. A reference to an array $B \in \mathcal{A}_P$ in a statement $S \in \mathcal{S}_P$ is written $\langle B, f \rangle$, where $f$ is the *subscript function*. The memory cell of $B$ accessed at iteration $\vec{x}$ is written $B[f(\vec{x})]$. When the subscript function of a reference is affine, we can write it $f(\vec{x}) = F\vec{x} + \vec{f}$ where $F$ is the *subscript matrix* of dimension $\rho(A) * \rho(S)$ with $\rho(A)$ the number of subscripts of the array $B$ and $\rho(S)$ the number of surrounding loops of $S$. $\vec{f}$ is a constant vector. Figure 2.9 illustrates the correspondence between reference in source program and affine subscript function: in Figure 2.9(a), the array $B$ is two-dimensional, thus the corresponding subscript function is a two-dimensional vector as shown in Figure 2.9(b).

```
do i=1, n
  do j=1, n
S   ... B(i+j,2*i+1) ...
```

$$f\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

(a) Reference to an array $B$ (b) Subscript function of the reference

Figure 2.9: Matrix notation of affine subscript function example

**Definition 2.2 (Static Reference)** *A reference is said to be static when it refers to an array cell by using an affine subscript function that depends only on outer loop counters and formal parameters.*

### 2.3.2 Preprocessing For Static References

In the same way as static control, we can distinguish between two ways for handling non-static references. On one hand by reasoning directly in a non-static model. Interprocedural and pointer analysis are necessary for precise dependence analysis of program parts involving function calls or pointer references. Several models are dedicated to these problems. The most relevant reference type is array reference since it often incur a large data volume. Dedicated solutions were proposed to handle non-linear subscript functions [34] or subscripts of subscripts (e.g. `A(B(i))`) [81]. On the other hand several analysis are possible to convert non-static references into static ones. *Delinearization* as well as *array reshaping* [57] may avoid complex and often non-linear subscripts generated by *linearizing* multi-dimensional arrays, i.e. converting them into one-dimensional arrays to e.g. provide long vectors for a vector architecture. *Array recovery* may translate pointer-based arrays that are common when using C to explicit array accesses [48]. Lastly, using conservative static approximations of array accesses (for instance considering an array as a variable) is always possible for array analysis (e.g. for dependence analysis, see section 3.3).

### 2.3.3 Significance Within Real Applications

We checked the same set of benchmarks from SpecFP2000 and PerfectClub suites as in section 2.2.4 for static references. The results are shown in Figure 7.7. They are very promising for dependence and locality analysis: most subscripts are affine except for lucas and mg3d; moreover, the ratio is over 99% in 7 benchmarks, and the overall is 76%, thus approximate array dependence analyses will be required for a good coverage of the 5 other benchmarks. Additional experimental study on reference subscripts may be found in [98].

| | Array References | |
|---|---|---|
| | All | Affine |
| applu | 1245 | 100% |
| apsi | 977 | 78% |
| art | 52 | 100% |
| lucas | 411 | 40% |
| mgrid | 176 | 99% |
| equake | 218 | 100% |
| swim | 192 | 100% |

(a) SpecFP2000

| | Array References | |
|---|---|---|
| | All | Affine |
| adm | 147 | 95% |
| dyfesm | 507 | 99% |
| mdg | 355 | 84% |
| mg3d | 1274 | 19% |
| qcd | 943 | 100% |

(b) PerfectClub

Figure 2.10: Coverage of affine array subscripts in high-performance applications

## 2.4 Conclusion

This chapter presented our program model, a slight extension of the well known *static control programs*. We divided control and data structure analysis in order to handle as many statements as possible in SCoPs, since a conservative approximation of general references is always possible for further analysis as data dependence calculation. The goal of this model is to handle most of the loop nests in a program. The advantages of such a choice are clear: in scientific or signal processing programs, most of the overall execution time is spent in few loops and the volume of accessed data is potentially very high. Moreover this program class is of a particular interest for data locality improvement since addressing the memory regularly challenges the memory hierarchy mechanisms that have been designed for random accesses. We showed experimentally that most programs are not static as a whole but are made of several SCoPs which cover a large proportion of the program, which can be optimized separately, and which (hopefully) represent a large proportion of the running time. The limitation to affine expressions in loop bounds, conditionals or subscript functions may be relaxed by preprocessing. Hence the interest of local transformations.

# Chapter 3

# Program Transformations

Applying the Fourier Transformation to a signal varying in time makes it possible to achieve processing that may be too difficult to express in the time domain. Once the processing is done, we can apply the inverse transformation from the frequency to the original time domain. The situation is similar with restructuring compilation. Once the parsing step is achieved, an input program is typically described using an abstract syntax tree. Simple optimizations, like constant folding, can be directly applied on such a rigid data structure. But the most interesting ones (e.g. loop interchange) incur modifying the execution order of the program, and this has nothing to do with syntax. On the opposite, as it will be shown in this chapter, reasoning in the polyhedral model instead of the abstract syntax tree gives us the opportunity to easily apply complex transformations and composition of transformations.

Many transformation techniques have been studied independently, as loop interchange, skewing, reversal, fusion, tiling etc. (see [108] for an exhaustive survey on known transformations). Most of the time they have their own properties (eg. legality test or cost model to decide whether applying the transformation is a benefit). Hence it is very hard to decide which transformation should be used and in which order we have to apply them to achieve the best results. Using scheduling functions in the polyhedral model allow us to handle many transformations in a single framework [90]. Well known unimodular transformations for instance may describe a composition of loop interchange, skewing, reversal on a single loop nest [10]. The framework evolved naturally by associating to each statement in the program a scheduling function [44, 45] and to include new transformations in the framework as loop splitting or fusion.

Although the polyhedral model offers many program restructuring facilities, current frameworks only cover a restrictive set of possible transformations, known as *unimodular* or in the best case *invertible* transformations. In this chapter we discuss a general transformation framework able to deal with non-unimodular, non-invertible, non-integral or even non-uniform transformations. The chapter is organized as follows. In section 3.1 we show that the program execution can be expressed using scheduling functions. Section 3.2 shows how to consider very general scheduling functions as transformation on the initial polyhedral representation of the program. Lastly section 3.3 deals with the legality of such transformations. It will show how to build the legal transformation space where each point is a legal solution and then how to use it for checking whether a transformation is legal or if it can be corrected under some constraints in order to be legal.

## 3.1 Describing the Execution Order

A convenient way to express the execution order is to give for each operation of a program a unique execution date. In the polyhedral model, every statement instance is characterized by its coordinates in the iteration domain of the corresponding statement (see section 2.1). Most of the time the iteration domains have several dimensions; the components of the operation coordinates can be seen as days, hours, minutes etc. When each loop step of the input program is a positive variable, we call the subsequent order the *lexicographic order*. Formally, this means that in a $n$-dimensional space, the operation corresponding to the integral point defined by the coordinates $(a_1...a_n)$ is executed before those corresponding to the coordinates $(b_1...b_n)$ iff there is an integer $i$ such that the $i^{th}$ entry of $(a_1...a_n)$ is lower of the $i^{th}$ entry of $(b_1...b_n)$, and all previous entries are equal. We denote this order $(a_1...a_n) \ll (b_1...b_n)$:

$$(a_1...a_n) \ll (b_1...b_n) \Leftrightarrow \exists i, 1 \leq i < n, (a_1...a_i) = (b_1...b_i) \wedge a_{i+1} < b_{i+1}. \quad (1)$$

Figure 3.1 illustrates the execution of a given loop nest operations in lexicographic order of their iteration vectors. It shows the iteration domain bounded by affine inequalities in a 2-dimensional space. The arcs show the execution order of the 13 iterations within the loop nest.

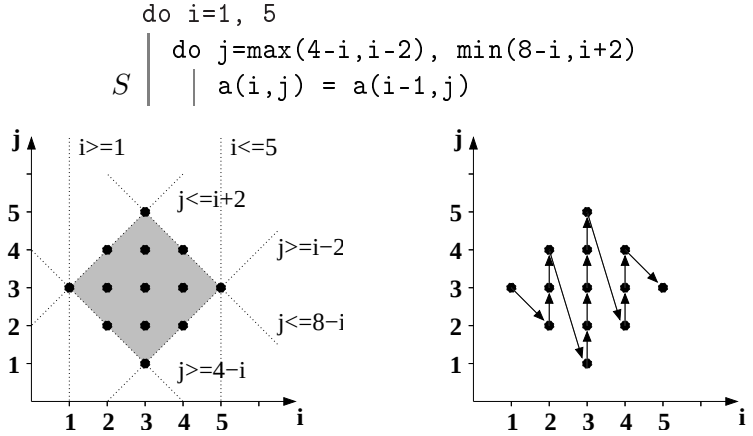



Figure 3.1: Iteration domain and execution order of a loop nest

The lexicographic ordering and iteration vectors are not sufficient to specify the execution order of instances of distinct statements. For instance let us consider the imperfectly nested loop in Figure 3.2. Statements $S_1$ and $S_3$ have the same depth but lexicographic ordering is not sufficient to state if the operation $S_1(1)$ is executed before $S_3(1)$ since the iteration vectors are the same. In the same way, lexicographic ordering cannot help if the iteration vectors do not have the same dimension, e.g. with $S_1(1)$ and $S_2(1,2)$ it is not possible to compare the second dimension since $S_1(1)$ do not have a second component. The refinement was first defined by Feautrier by taking care of the top-down order in the program text [43]. Let $s$ be the number of shared loops by two statements $S_1$ and $S_2$ then $S_1(a_1...a_n)$ is executed before $S_2(b_1...b_m)$ if $(a_1...a_s) \ll (b_1...b_s)$ or if iteration vectors are equal on the first common dimensions and $S_1$ precedes $S_2$ in the program text. We denote the execution order $S_1(a_1...a_n) \prec S_2(b_1...b_m)$:

$$S_1(a_1...a_n) \prec S_2(b_1...b_m) \Leftrightarrow \begin{array}{l}(a_1...a_s) \ll (b_1...b_s) \\ \vee((a_1...a_s) = (b_1...b_s) \wedge S_1 \text{ textually before } S_2).\end{array} \quad (2)$$

```
    do i=1, n
S1  | x = a(i,i)
    | do j=1, i-1
S2  | | x = x - a(i,j)**2
S3  | p(i) = 1.0/sqrt(x)
    | do j=i+1, n
S4  | | x = a(i,j)
    | | do k=1, i-1
S5  | | | x = x - a(j,k)*a(i,k)
S6  | | a(j,i) = x*p(i)
```

Figure 3.2: A Cholesky factorization kernel

Defining all the execution dates of every operation in the program separately would usually require very large scheduling systems. Moreover in the case of parametric iteration domains (e.g. when a bound of a loop is an unknown constant) it is not possible to state the exact number of instances of a given statement. Then a convenient solution is to build schedules at the statement level by finding a function that specifies the execution time for each instance of the corresponding statement. These are typically chosen affine for multiple reasons: this is the only case where we are able to decide exactly the transformation legality and where we know how to generate the target code. Thus, scheduling functions have the following shape:

$$\theta_S(\vec{x}_S) = T_S\vec{x}_S + \vec{t}_S, \tag{3}$$

where $\vec{x}_S$ is the iteration vector, $T_S$ is a constant transformation matrix and $\vec{t}_S$ is a constant vector (possibly including affine parametric expressions using the global parameters of the program i.e. the symbolic constants, mostly array sizes or iteration bounds).

For instance we can easily capture the sequential execution order of any static control program with scheduling functions by using the abstract syntax tree of this program [45]. The basic idea is to interleave the textual order informations between every component of the iteration vector. E.g. we can read directly the scheduling functions for the program in Figure 3.2 on the AST shown in Figure 3.3, i.e. $\theta_{S1}(\vec{x}_{S1}) = (0, i, 0)$, $\theta_{S2}(\vec{x}_{S2}) = (0, i, 1, j, 0)$, $\theta_{S3}(\vec{x}_{S3}) = (0, i, 2)$ etc. The lexicographic order is enough to state the execution order of the statement instances with such functions.

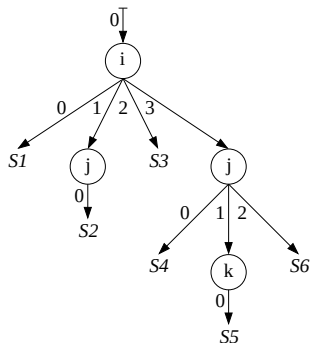



Figure 3.3: AST of the program in Figure 3.2

## 3.2 Transformation Functions

Program transformations in the polyhedral model can be specified by well chosen scheduling functions. They modify the source polyhedra into target polyhedra containing the same points but in a new coordinate system, thus with a new lexicographic order. Implementing these transformations is the central part of the polyhedral framework. The current polyhedral code generation algorithms lack flexibility by addressing only a subset of the possible functions. How to use general affine scheduling functions to apply a new lexicographic order to the original polyhedra is explained in section 3.2.1. Section 3.2.2 and section 3.2.3 respectively deal with the special case of non-integral and non-uniform transformations and shows how it is possible to handle them in this framework.

### 3.2.1 Affine Transformations

It has been extensively shown that linear transformations can express most of the useful transformations. In particular, loop transformations (such as loop reversal, permutation or skewing) can be modeled as a simple particular case called unimodular transformations (the $T_S$ matrix has to be square and has determinant $\pm 1$) [10, 106]. Complex transformations such as tiling [107] can be achieved using linear schedules as well [112]. The powerful underlying mathematical theory (e.g. for precise data dependence calculation [44]) and its intuitive geometric interpretation contributed to make this model quite popular. A lot of works are devoted to finding good scheduling functions for a particular purpose (e.g. parallelism or data locality) [17, 25, 33, 40, 44, 45, 110]. But previous polyhedral transformation frameworks required severe limitations on the scheduling functions, e.g. to be unimodular [5, 69] or at least to be invertible [78, 110, 92, 25]. The underlying reason was, considering an original polyhedron defined by the system of affine constraints $A\vec{x}+\vec{a} \geq \vec{0}$ and the transformation function $\theta$ leading to the target index $\vec{y} = T\vec{x}$, we deduce that the transformed polyhedron in the new coordinate system can be defined by $(AT^{-1})\vec{y} + \vec{a} \geq \vec{0}$, a change of basis. Figure 3.4 illustrates this transformation scheme for well known unimodular transformations. Another point that led to the popularity of unimodular transformations is their compositional properties. Applying a transformation $\vec{y_2} = T_2\vec{y_1}$ after $\vec{y_1} = T_1\vec{x}$ is equivalent to use $\vec{y_2} = T_2T_1\vec{x}$. And because a product of unimodular matrices is a unimodular matrix, the composition is still a unimodular transformation.

In this document we do not impose any constraint on the transformation functions because we do not try to perform a change of basis of the original polyhedron to the target index. Instead, we apply a new lexicographic order to the polyhedra by adding new dimensions in leading positions. Thus, from each polyhedron $\mathcal{D}$ and scheduling function $\theta$, it is possible to build another polyhedron $\mathcal{T}$ with the appropriate lexicographic order:

$$\mathcal{T} = \left\{ \begin{pmatrix} \vec{y} \\ \vec{x} \end{pmatrix} \;\middle|\; \left[\begin{array}{c|c} I & -T \\ \hline 0 & A \end{array}\right] \begin{pmatrix} \vec{y} \\ \vec{x} \end{pmatrix} + \begin{pmatrix} \vec{-t} \\ \vec{a} \end{pmatrix} \begin{matrix} = \\ \geq \end{matrix} \vec{0} \right\},$$

where $I$ is the identity matrix and by definition, $(\vec{y}, \vec{x}) \in \mathcal{T}$ if and only if $\vec{y} = \theta(\vec{x})$. The points inside the new polyhedron are ordered lexicographically until the last dimension of $\vec{y}$. Then there is no particular order for the remaining dimensions.

By using such a transformation policy, the data of both original iteration domains and transformations are included in the new polyhedra. As an illustration, let us consider the polyhedron

**Interchange Transformation**

The transformation matrix is the identity with a permutation of two rows.


$$\Longrightarrow$$

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0} \qquad \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \qquad \begin{bmatrix} 0 & 1 \\ 0 & -1 \\ 1 & 0 \\ -1 & 0 \end{bmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} -1 \\ 2 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0}$$

(a) original polyhedron $A\vec{x} + \vec{a} \geq \vec{0}$ (b) transformation function $\vec{y} = T\vec{x}$ (c) target polyhedron $(AT^{-1})\vec{y} + \vec{a} \geq \vec{0}$

**Reversal Transformation**

The transformation matrix is the identity with one diagonal element replaced by $-1$.


$$\Longrightarrow$$

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ 3 \\ -1 \\ 4 \end{pmatrix} \geq \vec{0} \qquad \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \qquad \begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} -1 \\ 3 \\ -1 \\ 4 \end{pmatrix} \geq \vec{0}$$

(a) original polyhedron $A\vec{x} + \vec{a} \geq \vec{0}$ (b) transformation function $\vec{y} = T\vec{x}$ (c) target polyhedron $(AT^{-1})\vec{y} + \vec{a} \geq \vec{0}$

**Skewing Transformation**

The transformation matrix is the identity with one 0 element replaced with a non null element.


$$\Longrightarrow$$

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ 3 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0} \qquad \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \qquad \begin{bmatrix} 1 & -1 \\ -1 & 1 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} -1 \\ 3 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0}$$

(a) original polyhedron $A\vec{x} + \vec{a} \geq \vec{0}$ (b) transformation function $\vec{y} = T\vec{x}$ (c) target polyhedron $(AT^{-1})\vec{y} + \vec{a} \geq \vec{0}$

Figure 3.4: Classical unimodular transformations

$\mathcal{D}_{S_2}$ in Figure 3.5(a) and the schedule $\theta_{S_2}(i,j) = 2i + j$. The corresponding scheduling matrix $T = [\,2\ 1\,]$ is not invertible, but it can be extended to $T = \left[\begin{smallmatrix} 2 & 1 \\ 0 & 1 \end{smallmatrix}\right]$ as suggested by Griebl et al. [55]. The usual resulting polyhedron is shown in Figure 3.5(b). Our policy leads directly to the polyhedron in Figure 3.5(d), provided we choose the lexicographic order for the free dimensions. A projection onto $i'$ and $i$ would lead to the result in Figure 3.5(b). The additional dimensions carry the transformation data, i.e. in this case $j = i' - 2i$. This is helpful since during code generation we have to update the references to the iterators in the loop body, and *necessary* when the transformation is not invertible.

Another property of this transformation policy is never to build rational target constraint systems. Most previous works were challenged by this problem, which occurs when the transformation function is non-unimodular. We can observe the phenomenon in Figure 3.5(b). The integer points without heavy dots have no images in the original polyhedron. The original coordinates can be determined from the target ones by $\overrightarrow{original} = T^{-1}\overrightarrow{target}$. Because $T$ is non-unimodular, $T^{-1}$ has rational elements. Thus some integer target points have a rational image in the original space; they are called *holes*. To avoid visiting the holes, the strides (the steps between the integral points to consider) have to be found. Many works proposed to use the Hermite Normal Form [97] in different ways to solve the problem [78, 110, 40, 92]. The basic idea is to find a unimodular matrix $U$ and a non-singular, non-negative, lower-triangular matrix $H$ in which each row has a unique maximum being the diagonal entry such as $T = HU$. Then we can consider the intermediate unimodular transformation matrix $U$ and find the strides and the lower bounds thanks to the diagonal elements of $H$. The corresponding decomposition for our example is $T = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 2 \end{smallmatrix}\right] \left[\begin{smallmatrix} 2 & 1 \\ -1 & 0 \end{smallmatrix}\right]$, the intermediate transformation with $U = \left[\begin{smallmatrix} 2 & 1 \\ -1 & 0 \end{smallmatrix}\right]$ is shown in Figure 3.5(c). In the opposite, we do not change the basis of the original polyhedra, but we only apply an appropriate lexicographic order. As a consequence, our target systems are always integral and there are no holes in the corresponding polyhedron. The stride informations are explicitly contained in the constraint systems in the form of equations.

The cost of this method is to add new dimensions to the constraint systems. This can be a relevant issue since first, it increases the complexity of the scanning step and second, it increases the constraint system size while high-level code generation typically requires a lot of memory. In practice processing of additional dimensions is often trivial with the method presented in part III. In fact, our prototype is more efficient and needs less memory than those based on other methods (see section 7.5).

### 3.2.2 Rational Transformations

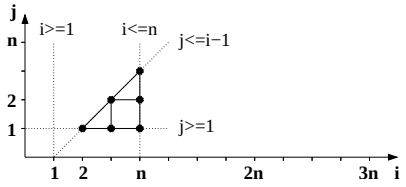
Some automatic allocators or schedulers ask for rational transformations [45]. Thus scattering functions can have a more general shape:

$$\theta(\vec{x}) = (T\vec{x} + \vec{t})/\vec{d},$$

where / means integer division and $\vec{d}$ is a constant vector such that each element divides the corresponding dimension of $\theta(\vec{x})$. In practice, divisors often correspond to resource constraints (e.g. the number of processors, of functional units etc.). Wetzel proposed the first solution to solve this problem, but only for one divisor value for the whole scattering function, and leading to a complex control [104].

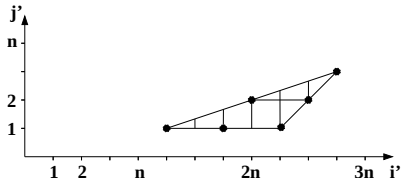
Again, we propose to add dimensions to solve the problem. For each rational element in

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



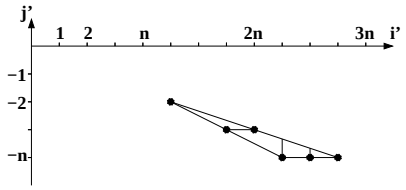

(a) original polyhedron $A\vec{x} + \vec{a} \geq \vec{0}$

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



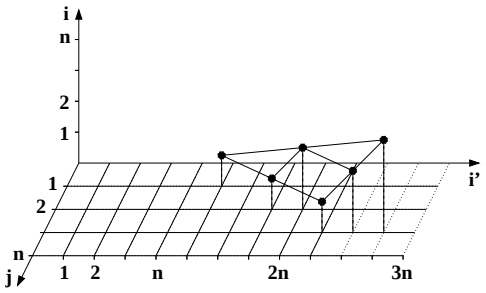

(b) usual transformation $(AT^{-1})\vec{y} + \vec{a} \geq \vec{0}$

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




(c) intermediate transformation using the Hermite Normal Form $(AU^{-1})\vec{y} + \vec{a} \geq \vec{0}$

$$\left[\begin{array}{c|cc} 1 & -2 & -1 \\ \hline 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -1 \end{array}\right] \left(\begin{array}{c} i' \\ \hline \\ i \\ j \\ \\ \end{array}\right) + \left(\begin{array}{c} 0 \\ -1 \\ n \\ -1 \\ -1 \end{array}\right) \begin{array}{c} = \\ \hline \\ \geq \\ \\ \\ \end{array} \vec{0}$$



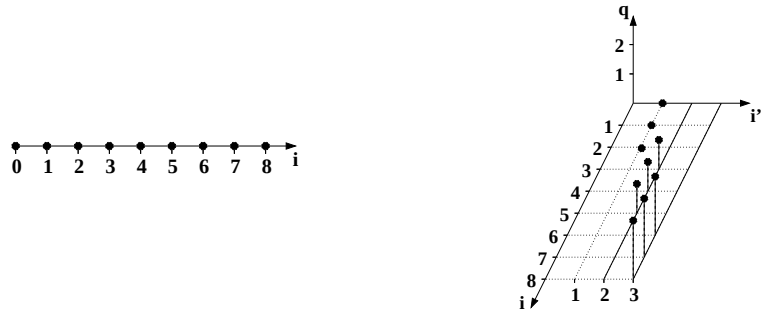

(d) our transformation $\mathcal{T}$

Figure 3.5: Transformation policies for $\mathcal{D}_{S2}$ in Figure 3.2 with $\theta_{S2}(i,j) = 2i + j$

$(T\vec{x})/\vec{d}$, we introduce an auxiliary variable standing for the quotient of the division. For instance let us consider the original polyhedron in Figure 3.6(a) and the scheduling function $\theta(i) = i/3+1$. We introduce $q$ and $r$ such as $i = 3q + r$, with by definition $0 \leq r = i - 3q \leq 2$. Then we can deal with an equivalent integral transformation $\theta'(q) = q + 1$ with $0 \leq i - 3q \leq 2$. This amounts to *strip-mine* the dimension $i$, as shown in Figure 3.6(b). With several non-integer coefficients, we just need more auxiliary variables standing for the result of the divisions.

### 3.2.3 Non-Uniform Transformations

As the power of program analysis increased with time, program transformations became more and more complex in order to deal with new optimization opportunities. Starting from simple transformation for a single loop nest, they evolved to statement-wise functions and more recently to several transformations per statement, each of them applying to a subset of the iteration




$$\begin{bmatrix} 1 \\ -1 \end{bmatrix} ( \ i \ ) + \begin{pmatrix} 0 \\ 8 \end{pmatrix} \geq \vec{0}$$

$$\left[\begin{array}{c|cc} 1 & -1 & 0 \\ \hline 0 & -3 & 1 \\ 0 & 3 & -1 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{array}\right] \left(\begin{array}{c} i' \\ \hline \\ q \\ i \\ \\ \end{array}\right) + \left(\begin{array}{c} -1 \\ \hline 0 \\ 2 \\ 0 \\ 8 \end{array}\right) \begin{array}{c} = \\ \hline \\ \geq \\ \\ \end{array} \vec{0}$$

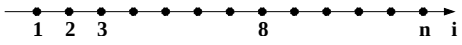
(a) original polyhedron $A\vec{x} \geq \vec{c}$ (b) our transformation $\mathcal{T}$

Figure 3.6: Rational transformation with $\theta(i) = i/3 + 1$

domain. Thus a scattering function for a statement with the iteration domain $\mathcal{D}$ may be of the following form:

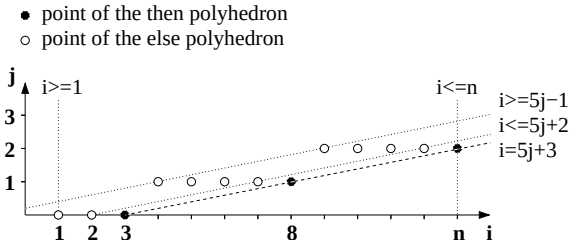
$$\theta(\vec{x}) = \begin{cases} \text{if } \vec{x} \in \mathcal{D}_1 \text{ then } T_1\vec{x} + \vec{t_1} \\ \text{if } \vec{x} \in \mathcal{D}_2 \text{ then } T_2\vec{x} + \vec{t_2} \\ \ldots \\ \text{if } \vec{x} \in \mathcal{D}_n \text{ then } T_n\vec{x} + \vec{t_n} \end{cases}$$

where the $\mathcal{D}_i$, $1 \leq i \leq n$ are a partition of $\mathcal{D}$. It is quite simple to handle such transformations, at least when the code generator deals efficiently with more than one polyhedron, by explicitly splitting the considered polyhedra into partitions. When the iteration domain is split using affine conditions, as in *index set splitting* [54], building the partition is trivial, but more general partitions with non-affine criteria are possible as long as we can express each subset as a polyhedron. For instance, Slama et al. found programs where the best parallelization requires non-uniform transformations, e.g. $\theta(i) =$ if $(mod(i, d) = n)$ then ... else ... where $d$ is a scalar value and $n$ a constant possibly parametric. They propose a code generation scheme dedicated to this problem [99]. It is possible to handle this in our framework by adding new dimensions. For instance the iteration domain corresponding to the *then* part of $\theta(i)$ would be the original one with the additional constraint $i = jd + n$, while the additional constraints for the *else* part could be $i \leq jd + n - 1$ and $i \geq jd + n + 1 - d$. Then we can apply the transformations to the resulting polyhedra as shown in section 3.2.1. Let us illustrate the separation into disjoint polyhedra with the example from the Slama et al. paper [99] shown in Figure 3.7. They exhibit a problem where the best placement (of statement instances on processors) is a non-uniform function $\theta(i) =$ if $(mod(i, 5) = 3)$ then 1 else 0. We propose to consider the two polyhedra in Figure 3.7(b), then to apply the corresponding scheduling to these polyhedra as shown in previous sections.




$$\begin{bmatrix} 1 \\ -1 \end{bmatrix} ( \ i \ ) + \begin{pmatrix} -1 \\ n \end{pmatrix} \geq \vec{0}$$

(a) Original polyhedron




$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 1 & -5 \\ -1 & 5 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -3 \\ -3 \end{pmatrix} \geq \vec{0} \qquad \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 5 \\ 1 & -5 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ 2 \\ 1 \end{pmatrix} \geq \vec{0}$$

Then polyhedron Else polyhedron

(b) Separated polyhedra for the non-uniform scheduling $\theta(i) =$ if $(mod(i,5) = 3)$ then 1 else 0

Figure 3.7: Example of index set splitting for non-uniform transformation

## 3.3 Legality

It is not possible to apply any transformation to a program without changing its semantics. Two statement instances $a$ and $b$ accessing a given memory location should be executed in any order if no access is a writing:

$$\begin{cases} \mathcal{W}_a \cap \mathcal{W}_b = \emptyset \\ \mathcal{W}_a \cap \mathcal{R}_b = \emptyset \\ \mathcal{R}_a \cap \mathcal{W}_b = \emptyset \end{cases}$$

Bernstein conditions

where $\mathcal{R}_x$ and $\mathcal{W}_x$ are respectively the sets of memory cells read and written by the operation $x$. Two operations are said to be *dependent* on each other if they do not respect these conditions. This definition suggested by Bernstein [21] is the most widely used while it is not always true since two operations writing a memory location may commute as the well known case of *reduction* shows. In this thesis we will consider this definition satisfactory. If the operation $a$ is executed before $b$ in the original program, we can distinguish between three dependence types:

1. *output-dependence* when the two operations writes one after the other onto the same memory cell $\mathcal{W}_a \cap \mathcal{W}_b \neq \emptyset$,

2. *flow-dependence* when $b$ reads the memory cell after $a$ wrote it $\mathcal{W}_a \cap \mathcal{R}_b \neq \emptyset$,

3. *anti-dependence* when $a$ reads the memory cell before $b$ writes it $\mathcal{R}_a \cap \mathcal{W}_b \neq \emptyset$.

A program transformation is said to be *legal* if it do not modify the order of dependent pair of operations. The dependence relations inside a program must direct the transformation construction. It is possible to avoid some dependences called *false-dependences*: output-dependences, anti-dependences and some flow-dependences known as *flow-indirect*. The only real dependences are known as *flow-direct* dependences: two operations $a$ and $b$ being in flow-direct dependence relation if they are in flow-dependence and there doesn't exist an operation $c$ such that $c$ writes on the same memory cell than $a$ between the executions of $a$ and $b$. To delete false dependences, it is possible to apply a *total memory expansion* or a single assignment form to the program [72]. Then there will be only one write for each memory cell during the program execution. Unfortunately such methods are not welcome when the question is memory traffic.

Many tests have been designed for dependence checking between operations. Most of them try to find efficiently a quite reliable, approximative but conservative (they overestimate data dependences) solutions, e.g. the GCD-test [8], Banerjee test [9] and its multi-dimensional version $\lambda$-test [67]. Other solutions are a combination of GCD and Banerjee test called I-test [67] or $\Delta$-test [51] that gives an exact solution when there is at most one variable in the subscript functions. On the opposite, a few methods allow to find an exact solution (assuming the Bernstein definition) like the Omega-test [89] and the Simplex-Gomory test [43]. In the same way many dependence representations are possible, from the simplest ones as *dependence levels* [3] to the most precise as *dependence polyhedra* [59]. We chose for our purpose to use the most precise extraction technique and representation of dependences as possible: the dependence polyhedra. However, many authors have noticed that approximate dependences are special cases of dependence polyhedra. Hence, our method applies whatever representation is chosen, provided the approximation is conservative.

In section 3.3.1, we recall how dependences in a SCoP can be expressed exactly using linear (in)equalities. Then we show in section 3.3.2 how to build the legal transformation space where each program transformation has to be found. We explain in section 3.3.3 how to use this information to check whether a transformation is legal and how to correct it if possible in section 3.3.4.

### 3.3.1 Dependence Graph

A convenient way to represent the scheduling constraints between the program operations is the *dependence graph*. In this directed graph, each program statement is represented using a unique vertex, and the existing dependence relations between statement instances are represented using edges. Each vertex is labelled with the iteration domain of the corresponding statement and the edges are labelled with the dependence polyhedra describing the dependence relation between the source and destination statements. The dependence relation can be defined in the following way:

**Definition 3.1** *A statement $R$ **depends** on a statement $S$ (written $S\delta R$) if there exit an operation $S(\vec{x}_1)$, an operation $R(\vec{x}_2)$ and a memory location $m$ such that:*

1. *$S(\vec{x}_1)$ and $R(\vec{x}_2)$ refer the same memory location $m$, and at least one of them writes to that location;*
2. *$\vec{x}_1$ and $\vec{x}_2$ respectively belong to the iteration domain of $S$ and $R$;*

3. *in the original sequential order,* $S(\vec{x}_1)$ *is executed before* $R(\vec{x}_2)$.

From this definition, we can easily describe the *dependence polyhedra* of each dependence relation between two statements with affine (in)equalities. A dependence polyhedron is a subset of the cartesian product of the iteration spaces of $S$ and $T$. In these polyhedra, every integral point represents a dependence between two instances of the corresponding statements. The systems have the following components:

1. *Same memory location*: assuming that $m$ is an array location, this constraint is the equality of the subscript functions of a pair of references to the same array: $F_S\vec{x}_S+\vec{f}_S = F_R\vec{x}_R+\vec{f}_R$.

2. *Iteration domains*: both $S$ and $R$ iteration domains can be described using affine inequalities, respectively $A_S\vec{x}_S + \vec{a}_S \geq \vec{0}$ and $A_R\vec{x}_R + \vec{a}_R \geq \vec{0}$.

3. *Precedence order*: this constraint can be separated into a disjunction of as many parts as there are common loops to both $S$ and $R$. Each case corresponds to a common loop depth and is called a *dependence level*. For each dependence level $l$, the precedence constraints are the equality of the loop index variables at depth lesser to $l$: $x_{R,i} = x_{S,i}$ for $i < l$ and $x_{R,l} > x_{S,l}$ if $l$ is less than the common nesting level. Otherwise, there are no additional constraints and the dependence only exists if $S$ is textually before $R$. Such constraints can be written using linear inequalities: $P_S\vec{x}_S - P_R\vec{x}_R + \vec{p} \geq \vec{0}$.

Thus, the dependence polyhedron for $S\delta R$ at a given level $l$ and for a given pair of references $p$ can be described using the following system of (in)equalities:

$$\mathcal{D}_{S\delta R,l,p} : D\begin{pmatrix}\vec{x}_S\\ \vec{x}_R\end{pmatrix} + \vec{d} = \left[\begin{array}{cc} F_S & -F_R \\ \hline A_S & 0 \\ 0 & A_R \\ P_S & -P_R \end{array}\right]\begin{pmatrix}\vec{x}_S\\ \vec{x}_R\end{pmatrix} + \left(\begin{array}{c} \vec{f}_S - \vec{f}_R \\ \hline \vec{a}_S \\ \vec{a}_R \\ \vec{p} \end{array}\right) \begin{matrix} = \\ \hline \geq \end{matrix} \vec{0} \tag{4}$$

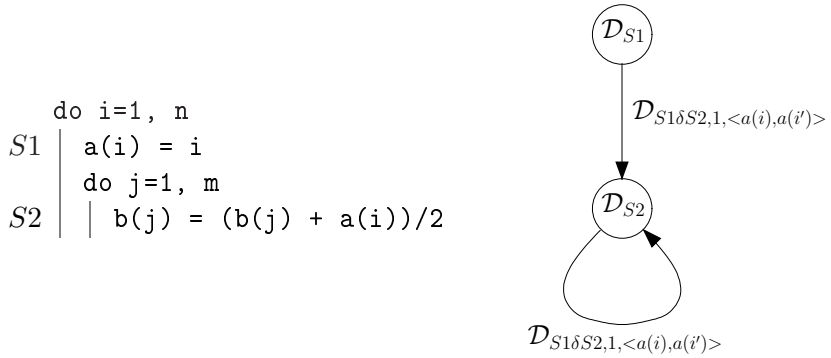
There is a dependence $S\delta R$ if there exists an integral point inside $\mathcal{D}_{S\delta R,l,p}$. This can be easily checked with linear integer programming tools like PipLib[1] [42]. If this polyhedron is not empty, there is an edge in the dependence graph from the vertex corresponding to $S$ up to the one corresponding to $R$ and labelled with $\mathcal{D}_{S\delta R,l,p}$. Figure 3.8 shows an example of dependence graph on a small program where the empty dependence polyhedra have been deleted. Two dependences remains, one from $S1$ to $S2$ and another from $S2$ to itself, the dependence polyhedra in Figure 3.8(b) have been calculated with (4). We will use this example for illustrating further concepts. For the sake of simplicity in the following we will ignore subscripts $l$ and $p$ and refer in the following to $\mathcal{D}_{S\delta R}$ as the only dependence polyhedron describing $S\delta R$.

### 3.3.2 Legal Transformation Space

Considering the transformations as scheduling functions, the time interval in the target program between the executions of two operations $R(\vec{x}_R)$ and $S(\vec{x}_S)$ is

$$\Delta_{R,S}\begin{pmatrix}\vec{x}_S\\ \vec{x}_R\end{pmatrix} = \theta_R(\vec{x}_R) - \theta_S(\vec{x}_S). \tag{5}$$

[1] PipLib is freely available at `http://www.prism.uvsq.fr/~cedb`




(a) Original program and corresponding dependence graph

$$\mathcal{D}_{S1} : \begin{bmatrix} 1 \\ -1 \end{bmatrix} \begin{pmatrix} i \end{pmatrix} + \begin{pmatrix} -1 \\ n \end{pmatrix} \geq \vec{0}$$

$$\mathcal{D}_{S2} : \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ m \end{pmatrix} \geq \vec{0}$$

$$\mathcal{D}_{S1\delta S2,1,<a(i'),a(i)>} : \left[\begin{array}{ccc} 1 & -1 & 0 \\ \hline 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{array}\right] \begin{pmatrix} i' \\ i \\ j \end{pmatrix} + \left(\begin{array}{c} 0 \\ \hline -1 \\ n \\ -1 \\ n \\ -1 \\ m \end{array}\right) \frac{=}{\geq} \vec{0}$$

$$\mathcal{D}_{S2\delta S2,1,<b(j'),b(j)>} : \left[\begin{array}{cccc} 0 & 1 & 0 & -1 \\ \hline 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 \end{array}\right] \begin{pmatrix} i' \\ j' \\ i \\ j \end{pmatrix} + \left(\begin{array}{c} 0 \\ \hline -1 \\ n \\ -1 \\ m \\ -1 \\ n \\ -1 \\ m \\ -1 \end{array}\right) \frac{=}{\geq} \vec{0}$$

(b) Iteration domains and dependence polyhedra

Figure 3.8: Dependence graph example

If there exists a dependence $S\delta R$, i.e. if $\mathcal{D}_{S\delta R}$ is not empty, then $\Delta_{R,S} \begin{pmatrix} \vec{x}_S \\ \vec{x}_R \end{pmatrix}$ must be lexicopositive in $\mathcal{D}_{S\delta R}$ (intuitively, the time interval between two operations $R(\vec{x}_R)$ and $S(\vec{x}_S)$ such that $R(\vec{x}_R)$ depends on $S(\vec{x}_S)$ must be at least $(0, \ldots, 0, 1)^T$, the smallest time interval: this guarantees that the operation $R(\vec{x}_R)$ is executed after $S(\vec{x}_S)$ in the target program). This condition represents as many constraints as there are points in $\Delta_{R,S}$. Fortunately, all these constraints can be compacted in a small set of affine constraints with the help of Farkas Lemma (see [44] and Appendix B).

**Lemma 3.1** *(Affine form of Farkas Lemma [97]) Let $\mathcal{D}$ be a nonempty polyhedron defined by the inequalities $A\vec{x} + \vec{a} \geq \vec{0}$. Then any affine function $f(\vec{x})$ is nonnegative everywhere in $\mathcal{D}$ iff it is a positive affine combination:*

$$f(\vec{x}) = \lambda_0 + \vec{\lambda}^T(A\vec{x} + \vec{a}), \text{ with } \lambda_0 \geq 0 \text{ and } \vec{\lambda}^T \geq \vec{0}.$$

*$\lambda_0$ and $\vec{\lambda}^T$ are called Farkas multipliers.*

$\Delta_{R,S}$ is a vector. For it to be lexicopositive, some of its components must be constrained to be either non negative or strictly positive. Let us apply Farkas Lemma to one of the constrained components. We can find a non-negative scalar $\lambda_0$ and a non-negative vector $\vec{\lambda}^T$ such that:

$$T_{R,\bullet}\vec{x}_R + t_R - (T_{S,\bullet}\vec{x}_S + t_S) - \delta = \lambda_0 + \vec{\lambda}^T \left( D \begin{pmatrix} \vec{x}_S \\ \vec{x}_R \end{pmatrix} + \vec{d} \right) \tag{6}$$

In this formula, $T_{R,\bullet}$ and $T_{S,\bullet}$ are corresponding rows in the $T_R$ and $T_S$ matrices, and $\delta$ is zero or one according to the position of the rows.

This formula can be split in as many equalities as there are independent variables ($\vec{x}_S$ and $\vec{x}_R$ components and parameters) by equating their coefficients in both sides of the formula. The Farkas multipliers can be eliminated by using the Fourier-Motzkin projection algorithm (see [97] and Appendix A). The result is a system of affine constraints on the coefficients of the transformation (the elements of $T_{R,\bullet}$ and $T_{S,\bullet}$). The important point is that this system is the same for all rows of the scheduling matrices and depends only on the dependence to be satisfied. Furthermore, it depends linearly on the value of $\delta$. These systems completely characterize the legal transformations of a program, and can be computed once and for all as soon as the dependences are known. For instance let us continue the example began in Section 3.3.1. We show in Figure 3.10 the expressions of both transformation functions and dependence constraints using Farkas Lemma. In these expressions, the coefficients of each independent variable has been defined, then there remains to achieve the equalities as in equation (6) for the two dependences to find the set of constraints describing the legal transformation space shown in Figure 3.9. The

$$\left\{ \begin{array}{rcrcl}
S1\delta S2 & & & & \\
i & : & \lambda_{S2,1} - \lambda_{S2,2} - \lambda_{S1,1} + \lambda_{S1,2} & = & \lambda_{S1\delta S2,1} - \lambda_{S1\delta S2,2} \\
j & : & \lambda_{S2,3} - \lambda_{S2,4} & = & \lambda_{S1\delta S2,3} - \lambda_{S1\delta S2,4} \\
m & : & \lambda_{S2,4} & = & \lambda_{S1\delta S2,4} \\
n & : & \lambda_{S2,2} - \lambda_{S1,2} & = & \lambda_{S1\delta S2,2} \\
1 & : & \lambda_{S2,0} - \lambda_{S2,1} - \lambda_{S2,3} - \lambda_{S1,0} + \lambda_{S1,1} - 1 & = & \lambda_{S1\delta S2,0} - \lambda_{S1\delta S2,1} - \lambda_{S1\delta S2,3} \\
S2\delta S2 & & & & \\
i' & : & -\lambda_{S2,1} + \lambda_{S2,2} & = & \lambda_{S2\delta S2,1} - \lambda_{S2\delta S2,2} - \lambda_{S2\delta S2,7} \\
i & : & \lambda_{S2,1} - \lambda_{S2,2} & = & \lambda_{S2\delta S2,3} - \lambda_{S2\delta S2,4} + \lambda_{S2\delta S2,7} \\
j & : & 0 & = & \lambda_{S2\delta S2,5} - \lambda_{S2\delta S2,6} \\
m & : & 0 & = & \lambda_{S1\delta S2,6} \\
n & : & 0 & = & \lambda_{S1\delta S2,2} + \lambda_{S1\delta S2,4} \\
1 & : & -1 & = & \lambda_{S2\delta S2,0} - \lambda_{S2\delta S2,1} - \lambda_{S2\delta S2,3} - \lambda_{S2\delta S2,5} - \lambda_{S2\delta S2,7}
\end{array} \right.$$

Figure 3.9: Legal transformation space constraints

Farkas multipliers can be eliminated by using the Fourier-Motzkin projection algorithm [97] in order to find the constraints on the transformation functions. The constraint system describe the legal transformation space, where each integral point corresponds to a legal solution.

### 3.3.3 Checking for Legality

Once we know the legal transformation space, checking whether a transformation is legal or not is a simple operation. The only information we need is to know if the suggested transformation belongs to the legal space or not. Then we have to equate the transformation functions with their expressions as given by Farkas Lemma. The transformation is legal, if and only if the linear system builded with the transformation expressions and the legal transformation space

$$\begin{aligned}
\theta_{S1}(i) = T_{S1}(i) + \vec{t}_{S1} &= \lambda_{S1,0} + \begin{pmatrix}\lambda_{S1,1}\\ \lambda_{S1,2}\end{pmatrix}^T \left(\begin{bmatrix}1\\ -1\end{bmatrix}(i) + \begin{pmatrix}-1\\ n\end{pmatrix}\right) \\
&= \begin{pmatrix}\lambda_{S1,1} - \lambda_{S1,2}\\ \lambda_{S1,2}\\ \lambda_{S1,0} - \lambda_{S1,1}\end{pmatrix}^T \begin{pmatrix}i\\ n\\ 1\end{pmatrix} \\
\theta_{S2}\begin{pmatrix}i\\ j\end{pmatrix} = T_{S2}\begin{pmatrix}i\\ j\end{pmatrix} + \vec{t}_{S2} &= \lambda_{S2,0} + \begin{pmatrix}\lambda_{S2,1}\\ \lambda_{S2,2}\\ \lambda_{S2,3}\\ \lambda_{S2,4}\end{pmatrix}^T \left(\begin{bmatrix}1 & 0\\ -1 & 0\\ 0 & 1\\ 0 & -1\end{bmatrix}\begin{pmatrix}i\\ j\end{pmatrix} + \begin{pmatrix}-1\\ n\\ -1\\ m\end{pmatrix}\right) \\
&= \begin{pmatrix}\lambda_{S2,1} - \lambda_{S2,2}\\ \lambda_{S2,3} - \lambda_{S2,4}\\ \lambda_{S2,4}\\ \lambda_{S2,2}\\ \lambda_{S2,0} - \lambda_{S2,1} - \lambda_{S2,3}\end{pmatrix}^T \begin{pmatrix}i\\ j\\ m\\ n\\ 1\end{pmatrix}
\end{aligned}$$

(a) Transformation function expressions

$$\begin{aligned}
\theta_{S2}\begin{pmatrix}i\\ j\end{pmatrix} - \theta_{S1}(i) - 1 &= \lambda_{S1\delta S2,0} + \begin{pmatrix}\lambda_{S1\delta S2,1}\\ \lambda_{S1\delta S2,2}\\ \lambda_{S1\delta S2,3}\\ \lambda_{S1\delta S2,4}\end{pmatrix}^T \left(\begin{bmatrix}1 & 0\\ -1 & 0\\ 0 & 1\\ 0 & -1\end{bmatrix}\begin{pmatrix}i\\ j\end{pmatrix} + \begin{pmatrix}-1\\ n\\ -1\\ m\end{pmatrix}\right) \\
&= \begin{pmatrix}\lambda_{S1\delta S2,1} - \lambda_{S1\delta S2,2}\\ \lambda_{S1\delta S2,3} - \lambda_{S1\delta S2,4}\\ \lambda_{S1\delta S2,4}\\ \lambda_{S1\delta S2,2}\\ \lambda_{S1\delta S2,0} - \lambda_{S1\delta S2,1} - \lambda_{S1\delta S2,3}\end{pmatrix}^T \begin{pmatrix}i\\ j\\ m\\ n\\ 1\end{pmatrix} \\
\theta_{S2}\begin{pmatrix}i\\ j\end{pmatrix} - \theta_{S2}\begin{pmatrix}i'\\ j\end{pmatrix} - 1 &= \lambda_{S2\delta S2,0} + \begin{pmatrix}\lambda_{S2\delta S2,1}\\ \lambda_{S2\delta S2,2}\\ \lambda_{S2\delta S2,3}\\ \lambda_{S2\delta S2,4}\\ \lambda_{S2\delta S2,5}\\ \lambda_{S2\delta S2,6}\\ \lambda_{S2\delta S2,7}\end{pmatrix}^T \left(\begin{bmatrix}1 & 0 & 0\\ -1 & 0 & 0\\ 0 & 1 & 0\\ 0 & -1 & 0\\ 0 & 0 & 1\\ 0 & 0 & -1\\ -1 & 1 & 0\end{bmatrix}\begin{pmatrix}i'\\ i\\ j\end{pmatrix} + \begin{pmatrix}-1\\ n\\ -1\\ n\\ -1\\ m\\ -1\end{pmatrix}\right) \\
&= \begin{pmatrix}\lambda_{S2\delta S2,1} - \lambda_{S2\delta S2,2} - \lambda_{S2\delta S2,7}\\ \lambda_{S2\delta S2,3} - \lambda_{S2\delta S2,4} + \lambda_{S2\delta S2,7}\\ \lambda_{S2\delta S2,5} - \lambda_{S2\delta S2,6}\\ \lambda_{S2\delta S2,6}\\ \lambda_{S2\delta S2,2} + \lambda_{S2\delta S2,4}\\ \lambda_{S2\delta S2,0} - \lambda_{S2\delta S2,1} - \lambda_{S2\delta S2,3} - \lambda_{S2\delta S2,5} - \lambda_{S2\delta S2,7}\end{pmatrix}^T \begin{pmatrix}i'\\ i\\ j\\ m\\ n\\ 1\end{pmatrix}
\end{aligned}$$

(b) Legality condition expressions

Figure 3.10: Transformation expressions using Farkas Lemma

constraints has a solution. Then we can use any linear algebra tool (like PipLib) to check whether the system has a solution and then whether the transformation is legal.

Let us suppose we want to use the scheduling functions $\theta_{S1}(i) = i$ and $\theta_{S2}\begin{pmatrix} i \\ j \end{pmatrix} = j$ to transform the code in Figure 3.8(a), then we have to write the equalities between each independent variable coefficients and their expressions in Figure 3.10(a). The new constraints are shown in Figure 3.11. There remains to check if the system built from both constraints of Figure 3.11 and of Figure 3.9 has a solution. We proposed this problem to PipLib and the answer was there is no solution, thus the transformation is not legal.

$$\left\{\begin{array}{rcrcl} \theta_{S1}(i) & & & & \\ i & : & \lambda_{S1,1} - \lambda_{S1,2} & = & 1 \\ n & : & \lambda_{S1,2} & = & 0 \\ 1 & : & \lambda_{S1,0} - \lambda_{S1,1} & = & 0 \\ \theta_{S2}(i,j) & & & & \\ i & : & \lambda_{S2,1} - \lambda_{S2,2} & = & 0 \\ j & : & \lambda_{S2,3} - \lambda_{S2,4} & = & 1 \\ m & : & \lambda_{S2,4} & = & 0 \\ n & : & \lambda_{S2,2} & = & 0 \\ 1 & : & \lambda_{S2,0} - \lambda_{S2,1} - \lambda_{S2,3} & = & 0 \end{array}\right.$$

Figure 3.11: Constraints for checking $\theta_{S1}(i) = i$ and $\theta_{S2}(i,j) = j$

### 3.3.4 Correcting for Legality

Experts or optimizing compilers have a wide choice of optimizing transformations for a given program. Each transformation has a more or less precise cost model which helps in deciding whether to apply the transformation or not. In the polyhedral framework, many transformations are related to well chosen scheduling functions [10, 44, 35]. For instance, generalized loop interchange is associated to schedules whose matrix is a permutation matrix [10]. Trying to use these transformations directly may result in a negative dependence test. Let us consider the code in Figure 3.12. An expert or an optimizing compiler may decide that moving the $i$-loop innermost would result in better locality. But because of complex dependences, using directly the loop interchange transformation $\theta_S\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix}$ is not legal as it can be shown by the method presented in section 3.3.3. This will lead usually to rejecting the transformation.

The polyhedral model allow more flexibility when defining such transformations. Moreover, we can work with an incompletely specified transformation and use the legality constraints as a way to find the missing coefficients. The method consists in stating the constraints the transformation has to satisfy, then solving these constraints and the legality constraints (6), using a linear algebra tool such as PipLib. If the system does not have a solution, we conclude that there is no legal instance of the proposed transformation. For example, "innermosting" the $i$-loop in the code in Figure 3.12 means that we are looking for a transformation function of the following form: $\theta_S\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} T_{1,1} & T_{1,2} \\ T_{2,1} & T_{2,2} \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}$ with as only constraints $T_{2,1} = 1$ and $T_{2,2} = 0$ to ensure that the second dimension of the target code will correspond to the $i$ loop. By solving the system we find the solution $\theta_S\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix}$. Expressed using classical transformation techniques, it is a combination of loop skewing and loop interchange. It leads to the target program in Figure 3.13 and as expected to a better cache behavior (on a i386 1GHz system with 128KB L1 cache memory and n=33000 the number of cache misses of the original program is

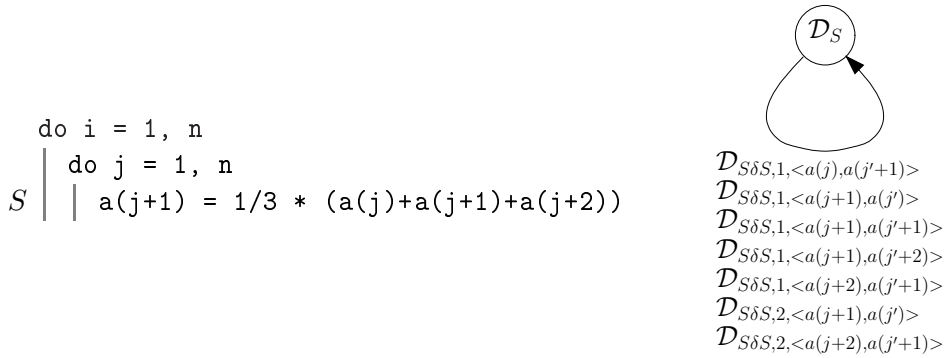



Figure 3.12: Original Hyperbolic-PDE program and dependence graph

68M but 43M for the target one). Correcting a transformation for legality is also possible when we have to consider non-linear constraints as in the data locality transformation case [19], this will be addressed in chapter 5.

```
do  i' = 2, 2*n
  do j' = max(i'-n,1), min(i'-1,n)
    j = i'-j' ;
    i = i' ;
S   a(j+1) = 1/3 * (a(j)+a(j+1)+a(j+2))
```

Figure 3.13: Final Hyperbolic-PDE program

## 3.4 Conclusion

Applying transformations is the heart of any program restructuring scheme. Affine schedules are known to be a convenient way to express most of the usual transformations. However current polyhedral frameworks handle only a restrictive set of possible transformation functions (unimodular or at least invertible functions). The main reason is that they are not able to modify the input polyhedra in a convenient way for the code generation process. In this chapter we discussed a general transformation framework able to deal with non-unimodular, non-invertible, non-integral or even non-uniform functions. Our only limit is the legality of the transformation. We showed for that purpose how to build the legal transformation space in the same way as in [44, 45]. Then we showed how it is possible to use this space to check whether a transformation is legal or not, or to correct a transformation for legality when it can be described using explicit constraints. To be able to apply the most general transformation as long as it is legal is a direct consequence of the results of this chapter.

# Part II

# Improving Data Locality

# Chapter 4

# A Model For Evaluating The Memory Traffic

Technological advances in the realization of integrated chips result in faster clocks for processors, and in larger capacity for memories. As a consequence, if nothing is done, processors will starve because their memory systems cannot supply data at the required speed. Memory hierarchies are a good solution to this problem: they are cheap and efficient, at least for ordinary programs and situations. Nevertheless, their efficiency decreases dramatically for scientific computing and signal processing codes, where large data sets are accessed according to highly regular patterns. Next, their temporal behavior is difficult to predict; this forbids their use in systems with hard real time constraints. Lastly, moving data from level to level uses a lot of power, which renders them unsuitable for embedded systems. A lot of work has been devoted to improving the behavior of memory hierarchies. There are two kinds of approaches for this problem. The first approach consists in designing highly optimized libraries (LAPACK is a good example [6]) for the most common linear algebra and signal processing algorithms. This method often gives the best results, provided the source problem and the target architecture are within the scope of the available library. The second approach tries to optimize the source program at compile time. This method is not restricted to a given set of algorithms and can be adapted, with minor modifications, to any memory hierarchy architecture. The present work belongs to the later approach.

Most optimizing compilers try to transform the source program in order to improve the behavior of the memory hierarchy. The basic principle is to regroup all accesses to a given memory cell, in order to take a maximum advantage of possible reuses. This is obtained first by applying loop transformations [106, 77] according to some cost model [85], then by tiling the resulting loop nest [107] with tiles having a carefully chosen size [37]. All methods mentioned earlier are based on a heuristic cost model. Let us consider for instance two accesses to the same memory cell. It seems probable that the longer the time interval between these accesses is, the higher the probability of the first reference to be evicted from the cache is. Hence, loop transformations aim at moving these references to neighboring iterations of some innermost loop. On the opposite, we propose a technique based on an estimate of the memory traffic, and trying to find the loop transformation that minimizes this estimate, under the constraint that all dependences are satisfied. This program transformation scheme, which we call *chunking* is based on the duality of the target architecture properties and a singular execution model.

This chapter is organized as follows. Section 4.1 details the hypothesis on the target architecture and explains the chunking mechanism. In section 4.2 we show how it is possible to take advantage of this framework to provide an asymptotic evaluation of the traffic. Section 4.3 shows how this evaluation may be affected by each kind of data locality improvement.

## 4.1 Chunking

There are many reasons for the limitation of previous techniques to heuristics. The main one is that cache management is left to hardware. Thus it is difficult to find an accurate solution to the traffic evaluation problem for a particular cache type. But there exists already some ways, although often limited, for controlling cache content (such as the primitive *Write Back Invalidate* WBINVD of the old i386 instruction set that flushes the cache memory and write the modified data back to memory). More cache control features appear in new architectures (as IA64) and still better, most embedded architectures offer a software-managed cache in order to allow real-time guarantees. This section presents *chunking*, a program transformation policy for locality optimization based on the availability of these tools. The basic idea is to combine a target architecture as defined in section 4.1.1 to avoid conflict misses and a particular execution model described in section 4.1.2 to avoid capacity misses and to bypass the replacement mechanism. Then a traffic evaluation becomes possible.

### 4.1.1 Target Architecture

To allow the functioning of the chunking or to simplify the analysis of the problem, some assumptions were made about the target architecture. It should have a main memory and one cache level between it and the processor, the main memory being big enough to contain the program data set. The goal is to know at any moment which data are stored in the cache, i.e. to be able to manage it precisely and to avoid conflict misses. For this purpose we will consider two possibilities : the ideal one corresponding to *scratch pad memory* systems, and its approximation achieved by conventional systems.

#### Scratch Pad Memories

A scratch pad memory (SPM) is a software-managed fast SRAM, i.e. driven by the application or the compiler [20, 103]. Unlike conventional cache memories, SPMs reduce hardware management overhead (complex tag checking and comparison logic) by relying on the software to move data (or instructions) into the cache. Eventually, SPMs are more efficient in area, power and performance prediction. Thus they are widely used in modern embedded systems. Typically the SPM occupies a range of the physical address space as shown in Figure 4.1: the address space is divided between off-chip memory and the on-chip SPM. Thus, copying data to the SPM can be done by a standard Memory to Memory Move (MMM) instruction. More elaborate devices, like Direct Memory Access (DMA) channels can be provided in order to improve performance.

Using such memory device allows to know precisely which data are stored in the cache at a given execution date. Moreover, the SPM design obviously avoid the problem of conflict misses of conventional cache memories.

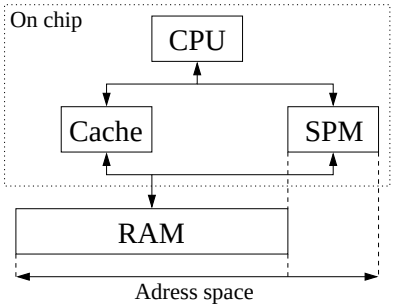



Figure 4.1: Scratch pad memories basic principle

### Classical Cache Memories

Conventional cache memories can be a good approximation of SPMs if we assume that some instructions to manage the cache are provided and that conflict misses are negligible. A simple directive like *Write Back Invalidate* WBINVD offered by the i386 instruction set that flushes the cache and updates the modified data is sufficient to have the knowledge of which data is in the cache at a given date. The two conditions are, first, to use it before the cache is full (this will be guaranteed by the execution model presented in section 4.1.2). And second, there are no conflict misses. This last condition is satisfied by *fully associative* cache memories, in which a line can occupy any place [58], and is approximated by modern caches with high associativity. It is possible to compensate any discrepancy by assuming a cache size smaller than the real one. Moreover several techniques allow to reduce the number of conflict misses using e.g. well known *array padding* [108, 94], array copying, or spacing out the variables as far as possible in the cache [84]. Eventually classical cache memories are a good approximation of SPMs once combined with the execution model presented in the next section. For opportunity reasons, we transposed our work on these architectures and the provided experimental results throughout this thesis have been measured on conventional cache memory systems.

With SPMs, there is no need for special instructions to manage the cache. All data movements can be done with standard instructions. To be able to manage precisely a traditional cache memory and to emulate a SPM, a few instructions dedicated to cache control would be necessary:

| Instruction | Description |
|---|---|
| LOAD CACHED | a memory line is fetched to the cache and to the processor |
| LOAD NOT CACHED | a memory line is fetched to the processor but not to the cache |
| STORE CACHED | a cache line is modified from the processor |
| STORE NOT CACHED | a memory line is modified directly from the processor |
| INVALIDATE | a line is evicted from the cache and, if modified, written back to memory |
| FLUSH | INVALIDATE all the contents of the cache |

In the following we will refer to these instructions to exhibit cache management, even in the SPM case.

### 4.1.2 Execution Model

The principle of our method is to partition the set of operations of a program in subsets small enough that their accessed data fit together in the cache: the *chunks*. The target program is thus executed chunk by chunk, with a cache flush between each of them. These subsets must be such that their sequential execution is equivalent to the execution of the original program, i.e. the chunking transformation must respect dependences. In practice, chunks will be numbered then executed in order of increasing numbers. A chunk number will be assigned to each operation, i.e. to each instance of each statement. In other words, for each statement $S$ we seek a *chunking function* $\theta_S$ associating a chunk number $\theta_S(\vec{x}_S)$ to each iteration vector $\vec{x}_S$. The original operations will be rescheduled according to these chunking functions. Operations with the same chunk number will be executed in the original sequential order. At the end of each chunk, we have to execute the cache management code. This consists in writing INVALIDATE instructions for all memory cells accessed by the chunk. If this solution is found to be too complicated, a brute force solution is to FLUSH all the cache. The set of memory cells accessed by the operations of a chunk $\vec{c}$ is called the *footprint* of the chunk, $\mathcal{F}(\vec{c})$:

$$\mathcal{F}(\vec{c}) = \cup_{S \in \mathcal{S}_P} \cup_{<A,f> \in S} \left\{ A[f(\vec{x}_S)] \mid \vec{x}_S \in \mathcal{D}_S, \theta_S(\vec{x}_S) = \vec{c} \right\}. \quad (1)$$

Thus, the progress of a program submitted to chunking can be summarized by the following pseudo-code:

```
FLUSH
do c=1, N
| execution of chunk c according to <_P
| INVALIDATE F(c)
```

where N is the total number of chunks and $<_P$ is the original sequential order of the program $P$. Each chunk must satisfy the *capacity condition*: each chunk footprint fits in the cache,

$$\forall \vec{c},\ \text{Card}\ \mathcal{F}(\vec{c}) \leq C \quad (2)$$

where $C$ is the cache size. A set of chunks answering the capacity condition and dependences requirements is called a *chunk system* for $P$.

The quality of a chunking can be assessed by using two valuations. First, the *footprint size* which is the number of memory cells accessed by the operations of a chunk. Next, the *traffic* which is the number of data movements between main and cache memory. We want to build an optimal chunk system i.e. where each chunk footprint fits in the cache and the traffic is minimal. Then during the execution of the target program, the only misses are compulsory misses at the beginning of each chunk. To be able to achieve a precise dependence analysis (see chapter 3) and to generate the target code (see part III), we are looking for affine chunking functions. Hence, for an operation $S(\vec{x}_S)$, instance of the statement $S$ with the iteration vector $\vec{x}_S$ in the iteration domain $D_S$, the chunk number can be written:

$$\theta_S(\vec{x}_S) = T_S \vec{x}_S + \vec{t}_S.$$

$T_S$ is a matrix called the *chunking matrix*; its dimensions are $g \times \rho(S)$ with $\rho(S)$ the number of loops surrounding $S$. The choice of the value of $g$ is postponed till chapter 5. $\vec{t}_S$ is a constant

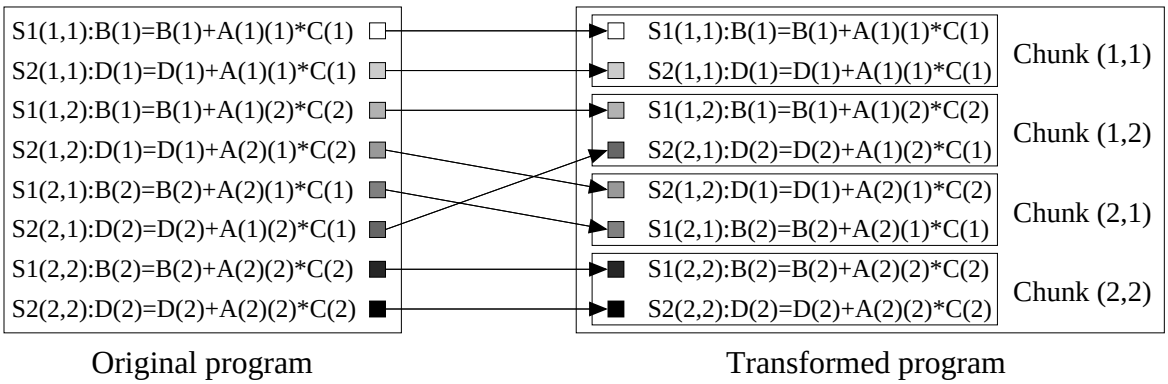
vector. We present in figure 4.2 an example of chunking on a simple program: two matrix-vector multiplies where the first statement uses a matrix $A$ and the second one uses its transpose $A^T$. Thus there is inter-statement group-reuse for each memory cell of $A$ and only the diagonal elements benefit from this reuse in the same iteration. In Figure 4.2(b) are shown possible chunking functions to solve this problem (how to find these function is the subject of the next chapter). Figure 4.2(c) shows how a new execution order is assigned to each operation thanks to

```
   do i=1, n
     do j=1, n
S1     B(i) = B(i) + A(i)(j) * C(j)
S2     D(i) = D(i) + A(j)(i) * E(j)
```

(a) source program

$$\theta_{S1}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} \; ; \; \theta_{S2}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix}$$

(b) chunking functions




(c) transformation at the operation level (with n=3)

```
   do i'=1, n
     do j'=1, n
S1     B(i') = B(i') + A(i')(j') * C(j')
S2     D(j') = D(j') + A(i')(j') * E(i')
       FLUSH
```

(d) target program

Figure 4.2: Chunking example

the chunking functions: the statement instances are divided into chunks, and inside a chunk, the original execution order is enforced. We can observe that in the target execution order, a given memory cell of $A$ is accessed only in a given chunk. The resulting target code implementing the new order is shown in Figure 4.2(d) (how to build the target code from the original one and the chunking functions is the subject of part III of this thesis). Reasoning with directive-based program transformation schemes, we can see that building this target code is not a trivial problem. It could be found by (1) splitting the original loop nest into two loop nests, (2) interchanging the loops of the resulting loop nest that surrounds $S2$ and (3) merging the two loop nests. But most existing techniques are not able to find either the useful transformations or the

order in which they have to be applied. By the way, the chosen example exhibits non-uniformly generated references [49] (the subscript functions differ by more than a constant term) where group-locality was supposed to be non-exploitable [49, 106]. Using transformation functions shows here its power and compositional properties.

Eventually, chunking bypasses the replacement mechanism of hardware-managed caches since the basic principle is never to use it. Coupling this property with a hardware model as described in section 4.1.1 makes it possible to achieve a traffic evaluation as demonstrated in the next section. Moreover, the choice of a technique using cache controls is going to allow guarantees of real-time type. Indeed, the prediction of the contents of the cache and its stability at the same point in various executions become possible in particular when using scratch pad memories where conflict misses are non-existent. When using this method on conventional cache systems for improving performance or/and reducing power dissipation and if real-time is not a matter, there is no need to insert a FLUSH instruction between chunks provided that the replacement mechanism always selects data from previous chunks for eviction. This is true for LRU, likely for FIFO, false for RANDOM replacement policy. Then we avoid the FLUSH instruction overhead and we can benefit from inter-chunk locality even if it is not considered by our model.

## 4.2 Computing the Memory Traffic

Memory traffic $\mathcal{T}$ is the number of comings and goings between main memory and cache memory. Evaluating it precisely is a challenge even with our proposed scheme. Modeling the replacement mechanism is quite difficult, but it is bypassed by chunking. Our hardware model assumes that conflict misses are non-existent or at least they do not change the order of magnitude of the traffic. Hence, we will be satisfied with asymptotic evaluation of the traffic. Since in many cases, program transformations can change the order of magnitude of the traffic, it would be useless to fiddle with constant factors or worse, units in the last decimal place. In some cases, *i.e.* when self-reuse has already been exploited, only the constant factors can be improved. The question of deciding if a more precise evaluation can influence the target code is left for future work. In the chunking case, we can estimate the traffic to the double of the total size of chunk footprints. Indeed, every data will be at first read from the main memory then rewritten there after use. This value is an overestimate because only data which were modified are being updated in memory:

$$\mathcal{T} = 2 \sum_{\vec{c}} \text{Card } \mathcal{F}(\vec{c}).$$

Since we are only considering the order of magnitude of $\mathcal{T}$, we can ignore the factor of 2, and we can then define the traffic as:

$$\mathcal{T} = \sum_{a \in \mathcal{M}_P} \text{Card } \left\{ \vec{c} \mid a \in \mathcal{F}(\vec{c}) \right\}. \tag{3}$$

*Proof.* Let $i_S(a)$ be the indicator function of a set $S$, i.e. the function which is equal to 1 if $a \in S$ and zero otherwise. We have:

$$\mathcal{T} = \sum_{\vec{c}} \text{Card } \mathcal{F}(\vec{c}) = \sum_{\vec{c}} \sum_{a \in \mathcal{M}_P} i_{\mathcal{F}(\vec{c})}(a).$$

Inverting the order of summation gives:

$$\mathcal{T} = \sum_{a \in \mathcal{M}_P} \sum_{\vec{c}} i_{\mathcal{F}(\vec{c})}(a)$$

which is clearly equivalent to (3). ∎

We deduce directly that the minimum value of $\mathcal{T}$ is obtained when Card $\{\vec{c} \mid a \in \mathcal{F}(\vec{c})\} = 1$, or, equivalently, when footprints are disjoint. The value of this minimal traffic, that we will call the *compulsory traffic* is:

$$\mathcal{T} = \text{Card } \mathcal{M}_P,$$

where $\mathcal{M}_P$ is the data set of the program $P$. A chunk system with this property is said to be perfect. All programs have at least one perfect chunk system which corresponds to $\forall u \in \mathcal{O}_P : \theta(u) = 0$. This is useful only if the chunk system satisfies the capacity condition Card $\mathcal{M}_P \leq C$, which rarely happens in practice. Most programs do not have any other perfect chunk system. The reason is that a program which has a non trivial perfect chunk system corresponds to several independent programs. Such programs, of the *embarrassingly parallel* variety, are neither frequent nor interesting. In the general case, we should form imperfect chunk systems, where a given memory cell may be accessed by operations belonging to different chunks. The objective in such a case is to minimize the traffic, that is to minimize the number of accesses to identical memory cells in different chunks.

### 4.2.1 Decompositions

Let us rewrite the formula of the traffic (3) by introducing the *usage relation*: the set of all pairs ⟨cell,chunk⟩ where *cell* is accessed during the execution of *chunk*:

$$U = \{\langle a, \vec{c} \rangle \mid \exists u \in \mathcal{O}_P : \theta(u) = \vec{c}, a \in \mathcal{D}_u\}. \tag{4}$$

Given the equality

$$\text{Card } \{\vec{c} \mid a \in \mathcal{F}(\vec{c})\} = \text{Card } \{\langle a, \vec{c} \rangle \mid \langle a, \vec{c} \rangle \in U\}$$

and because the sets $\{\langle a, \vec{c} \rangle \mid \langle a, \vec{c} \rangle \in U\}$ are disjoint for different values of $a$, we have:

$$\begin{aligned} \mathcal{T} &= \sum_{a \in \mathcal{M}_P} \text{Card } \{\langle a, \vec{c} \rangle \mid \langle a, \vec{c} \rangle \in U\} \\ &= \text{Card } \cup_{a \in \mathcal{M}_P} \{\langle a, \vec{c} \rangle \mid \langle a, \vec{c} \rangle \in U\} \\ &= \text{Card } U. \end{aligned}$$

Hence to find a correct and optimal chunk system, our job is to minimize Card $U$ under the constraints $\forall \vec{c} : \text{Card } \mathcal{F}(\vec{c}) \leq C$.

We can give a more detailed definition of $U$, knowing that:

- a memory cell $a \in \mathcal{M}_P$ is in fact an array cell $A[\vec{i}],\ A \in \mathcal{A}_P,\ \vec{i} \in \mathcal{D}_A$;
- an operation $u \in \mathcal{O}_P$ is an instance of a statement $S(\vec{x}_S),\ S \in \mathcal{S}_P,\ \vec{x}_S \in \mathcal{D}_S$;

- $a \in \mathcal{D}_u$ iff $\exists \langle A, f \rangle \in S : \vec{\imath} = f(\vec{x}_S)$;

this gives finally:

$$U = \cup_{A \in \mathcal{A}_P} \cup_{S \in \mathcal{S}_P} \cup_{<A,f> \in S} \{ \langle A[\vec{\imath}], \vec{c} \rangle \mid \exists \vec{x}_S \in \mathcal{D}_S : f(\vec{x}_S) = \vec{\imath}, \theta_S(\vec{x}_S) = \vec{c} \}.$$

The first observation is that the sets:

$$U_A = \cup_{S \in \mathcal{A}_P} \cup_{<A,f> \in S} \{ \langle A[\vec{\imath}], \vec{c} \rangle \mid \exists \vec{x}_S \in \mathcal{D}_S : f(\vec{x}_S) = \vec{\imath}, \theta_S(\vec{x}_S) = \vec{c} \}$$

are disjoint. Hence, when computing the traffic, we can compute their size independently and sum the results. Next, we observe that we can now dispense with $A$ and introduce sets:

$$u_A = \cup_{S \in \mathcal{A}_P} \cup_{<A,f> \in S} \{ \langle \vec{\imath}, \vec{c} \rangle \mid \exists \vec{x}_S \in \mathcal{D}_S : f(\vec{x}_S) = \vec{\imath}, \theta_S(\vec{x}_S) = \vec{c} \}.$$

We finally arrive at the traffic decomposition:

$$\begin{aligned} u_{S,A,f} &= \{ \langle \vec{\imath}, \vec{c} \rangle \mid \exists \vec{x}_S \in \mathcal{D}_S : f(\vec{x}_S) = \vec{\imath}, \theta_S(\vec{x}_S) = \vec{c} \} \\ &= \{ \langle f(\vec{x}_S), \theta_S(\vec{x}_S) \rangle \mid \vec{x}_S \in \mathcal{D}_S \}. \end{aligned} \tag{5}$$

The sets $u_{S,A,f}$ are not disjoint. Indeed, an array $A$ can be accessed in various instructions, which means that we should have the union $\cup_{S \in \mathcal{A}_P}$ so that these sets are separate. Furthermore, the same array can be accessed with several subscript functions in the same instruction, we should also keep the union $\cup_{<A,f> \in S}$ so that they are disjoint. However, for heuristics reasons, we will suppose that they are actually disjoint.

The same decomposition applies also to footprints. We are lead to the study of the following sets:

$$\begin{aligned} v_{S,A,f}(\vec{c}) &= \{ \vec{\imath} \mid \exists \vec{x} \in \mathcal{D}_S : f(\vec{x}) = \vec{\imath}, \theta_S(\vec{x}) = \vec{c} \} \\ &= \{ f(\vec{x}) \mid \vec{x} \in \mathcal{D}_S, \theta_S(\vec{x}) = \vec{c} \}. \end{aligned} \tag{6}$$

### 4.2.2 Asymptotic Evaluation

Let us first study a set:

$$H = \{ U\vec{x} \mid V\vec{x} = \vec{0}, \vec{x} \in \mathcal{D} \}$$

where $U$ and $V$ are arbitrary integral matrices of the right dimension, and where $D$ is a bounded full dimensional domain. Let us first study the dimension of the supporting subspace:

$$K = \{ U\vec{x} \mid V\vec{x} = \vec{0} \}.$$

Let $\{\vec{v}_1, \ldots \vec{v}_k\}$ be a basis for $\ker U \cap \ker V$. This basis can clearly be extended to a basis $\{\vec{v}_1, \ldots \vec{v}_q\}$ for $\ker V$.

**Lemma 4.1** *The vectors $\{U\vec{v}_{k+1}, \ldots U\vec{v}_q\}$ are a basis for $K$.*

*Proof.* Vectors such that $V\vec{x} = \vec{0}$ are linear combinations of $\vec{v}_1, \ldots \vec{v}_q$, and, when multiplied by $U$, vectors $\vec{v}_1, \ldots \vec{v}_k$ disappear. Hence, $U\vec{v}_{k+1}, \ldots U\vec{v}_q$ generate $H$. Suppose they are not linearly independent. There exists $\lambda_{k+1}, \ldots \lambda_q$ not all zero such that:

$$\sum_{i=k+1}^{q} \lambda_i U\vec{v}_i = \vec{0}.$$

This implies that the vector $\vec{x} = \sum_{i=k+1}^{q} \lambda_i \vec{v}_i$ is in $\ker U$. Since it already is in $\ker V$, it belongs to $\ker U \cap \ker V$, hence is a linear combination of $\vec{v}_1, \ldots \vec{v}_k$. From this we deduce that the $\vec{v}_1, \ldots, \vec{v}_q$ are linearly dependent, a contradiction. ∎

The dimension of $K$ is thus $l = \dim\ \ker V - \dim\ (\ker U \cap \ker V)$. This means in fact that, given $l$ of the components of $\vec{y} = F\vec{x}$, we can compute the remaining ones by a linear transformation:

$$(y_{l+1}, \ldots, y_d)^T = L(y_1, \ldots y_l)^T.$$

From this follows:

$$\text{Card } \left\{U\vec{x} \mid V\vec{x} = \vec{0}, \vec{x} \in \mathcal{D}\right\} = \text{Card } \left\{(U\vec{x})[1..q] \mid V\vec{x} = \vec{0}, \vec{x} \in \mathcal{D}\right\}.$$

Suppose now that $\mathcal{D}$ is such that the value of each component of $\vec{x}$ is an integer in a segment of length $m$. It follows that each component of $U\vec{x}$ also is integral and belongs to a segment of length proportional to $m$. Hence, the size of $H$ if of the order of $m^l$.

Since $\dim\ \ker V + \text{rank } V = \text{number of column } V$, we have finally

$$\text{Card } H \text{ is of the order of } m^l, \text{ with } l = \text{rank } \begin{bmatrix} V \\ U \end{bmatrix} - \text{rank } V. \tag{7}$$

### 4.2.3 Application to the Estimation of the Traffic and Footprint Sizes

Let us consider first formula (6); because we suppose that the program has static control, the index function is affine:

$$f(\vec{x}_S) = F\vec{x}_S + \vec{f}$$

where $F$ is a matrix of dimension $\rho(A) \times \rho(S)$ and $\vec{f}$ is a constant vector which can be ignored for size computation. Similarly, let us recall that the chunking function is supposed to be of the form:

$$\theta_S(\vec{x}_S) = T_S\vec{x}_S + \vec{t}_S$$

where $T_S$ is a matrix of dimension $g \times \rho(S)$ and $\vec{t}_S$ is a constant vector. We choose as first dimension of $T_S$: $g = \max \rho(S)$ in order to have chunking functions of the same dimension, given that to add null lines to a matrix does not modify its rank. $F$ can be extracted by analysis of the source code, while $T_S$ is the unknown of the problem.

In the case of footprints, there are two situations, depending on the value of $\vec{c}$:

1. The system of constraints $\vec{x}_S \in \mathcal{D}_S, \theta_S(\vec{x}_S) = \vec{c}$ is infeasible, and $u_{S,A,f}(\vec{c})$ is empty. This means that $S$ does not belong to chunk $\vec{c}$, and the size of the corresponding footprint is 0. Deciding which statement belongs to which chunk is a problem in program partitioning, and is left for future research.

2. The above system of constraints is feasible. The asymptotic size of the corresponding footprint is directly given by the result in formula (7), with $T_S$ as $V$ and $F$ as $U$. The size is of the order of $O(m^l)$, with $l = \text{rank} \begin{bmatrix} T_S \\ F \end{bmatrix} - \text{rank } T_S$.

$$\text{Card } \mathcal{F}_{S,A,f}(\vec{c}) = O(m^l), \text{ with } l = \text{rank} \begin{bmatrix} T_S \\ F \end{bmatrix} - \text{rank } T_S. \tag{8}$$

Consider now the estimation of the traffic as given by (5). In this case there is no constraint on $\vec{x}_S$, which can be represented by saying that $V$ is the null matrix 0. The rank of the null matrix is 0. $U$ is clearly the block matrix $\begin{bmatrix} T_S \\ F \end{bmatrix}$. Hence, we can finally estimate traffic:

$$\mathcal{T}_{S,A,f} = O(m^k), \text{ with } k = \text{rank} \begin{bmatrix} T_S \\ F \end{bmatrix}. \tag{9}$$

### 4.2.4 Evaluation of the Segment Length

One of the main components of the evaluation formula of the traffic and the footprint sizes generated by a reference to an array $A$ with the subscript function $f(\vec{x}) = F\vec{x} + \vec{f}$ is the segment length $m$. Although we are looking for asymptotic evaluations, the more precise is our knowledge about the segment value, the better our comparisons between evaluations are. The segment length is the number of values that each component of $F\vec{x}$ can take. Let $\vec{F}_{i,\bullet}$ be the row vector corresponding to the $i^{th}$ row of $F$, and $m_i$ the number of values that $\vec{F}_{i,\bullet}\vec{x}$ can take. The problem of finding how many values an affine expression $\vec{F}_{i,\bullet}\vec{x}$ can take knowing $\vec{x} \in \mathcal{D}$ is a well known problem. The solution has relevant applications such as array size reduction. The most simple way to find an estimate is to use the Fourier-Motzkin elimination algorithm (see appendix A). Using this method we will find an overestimate of the range of the possible values. Then if there are holes in the range or a big overestimate, this method may not be sufficiently precise. In practice this method gives most of the time very good results. Another way is to compute the exact number of possible values thank to Ehrhart polynomials [32] at the price of a heavier processing than by using the Fourier-Motzkin method.

Thus we can precise the evaluation formula:

$$\text{Card } \mathcal{F}_{S,A,f}(\vec{c}) = O\left(\prod_{i=0}^{\rho(A)} m_i^{l_i}\right) \text{ with } l_i = \text{rank} \begin{bmatrix} T \\ F_{i,\bullet} \end{bmatrix} - \text{rank } T$$

$$\mathcal{T}_{S,A,f} = O\left(\prod_{i=0}^{\rho(A)} m_i^{k_i}\right) \text{ with } k_i = \text{rank} \begin{bmatrix} T \\ F_{i,\bullet} \end{bmatrix}.$$

For convenience reasons in the following we will use formula 8 and 9 instead of the precise ones by using $m = \text{MAX}_{i=1}^{\rho(A)} m_i$, except if stated otherwise.

## 4.3 Correspondence Between Locality and Generated Traffic

We may distinguish several types of reuse as defined by Wolf and Lam [106]. When a given reference in a statement generates several accesses to the same memory cell over different instances of

this statement, the reference is said to provide *self-temporal* reuse. In the same way, if a reference accesses several memory cells from the same cache line over different instances of its enclosing statement, the reference is said to provide *self-spatial* reuse. When considering several references that generate accesses onto the same memory cell, we say that they exhibit *group-temporal* reuse. Lastly if several references refer to a given cache line we call this *group-spatial* reuse. If the reuse actually results in accesses to the considered memory cells or blocks directly from the cache, we say that there is *locality*.

Achieving locality from reuse has direct consequences on the memory traffic. We may see easily how the different types of locality may interact with our traffic model. According to our evaluation scheme, the overall traffic generated by a program is $\mathcal{T} = \sum_i n_i O(m_i^{k_i})$. Notice that we do not strictly respect the asymptotic calculation rules since we neither eliminate the factors $n_i$ nor consider only the greatest $m_i^{k_i}$. The reasons are, first some interesting optimizations may only improve the constant factor and second, even the optimal transformation may not be able to improve the reference that generates the heaviest traffic. Our estimation is sensitive to every kind of data locality improvement. By order of interest, improving self-temporal locality may reduce the power $k_i$. Intuitively when the number of dimensions of an array is less than the depth of its enclosing loop we may at best access the same memory cell during each iteration of the innermost loops while at worst we may have to scan the whole array before reusing a given datum. For each *local* array dimension, we reduce the memory traffic by a factor $m_i$, thus we reduce the power $k_i$. Improving self-spatial locality may reduce the value $m_i$ at best by a factor equal to the cache line length since accessing data of a given cache line may result in a single transfer from main memory to the cache. Lastly, group-locality may change the value of $n_i$ since only one transfer would be actually necessary. Hence if group-locality is achieved for $p$ references, $n_i$ is reduced by a factor of $p$.

## 4.4 Conclusion

Evaluating the memory traffic is a tough challenge because of the difficulty to anticipate the cache behavior. We presented in this chapter a transformation model called *chunking* that allows an evaluation of the traffic, at least in an asymptotic sense. The originality of this model is that it is inspired by the availability of tools to manage the cache from the software, even if they are quite basic (as the i386 instruction WBINVD to flush the cache). We showed that our asymptotic model is precise enough to be sensitive to every kind of data locality improvement. Our evaluations depend on transformation functions to restructure the input program. Hence they may drive the transformation construction in order to achieve the best possible locality.

# Chapter 5

# Chunking, Algorithms and Experiments

A wide range of solutions has been proposed to find suitable program transformations for improving data locality. The standard framework that emerged is to use some unimodular transformations [106, 77, 10, 85, 96, 108], then to apply tiling [107, 112, 1]. Another data-centric [66] approach starts from a memory cell and tries to build the slice that accesses this cell. Dependences greatly complicate the transformation process. As a result, most of these methods apply only to loop nests in which dependences are non-existent or uniform (dependences are uniform if they can be expressed using constant distance vectors) or have a special form, i.e. *fully permutable* loop nests (a loop nest being fully permutable when every possible loop interchange transformations are legal). Moreover, previous methods require most of the time severe structural limitations on the input program, i.e. to be a perfect loop nest. To improve this situation, Sass and Mutka proposed a method to transform a program into a perfect loop nest [96], well known *loop skewing* or *code sinking* are widely used with the same intention [108, 1, 53]. But this transformation is not always possible: complex data dependences, multiple loops may forbid it. Lim and Lam showed how to extract the largest fully permutable innermost loop nest in a arbitrary nested program [80]. Another limitation of previously mentioned techniques comes from the set of considered program transformations since it is most of the time restricted to unimodular transformations.

In this chapter we present a data locality improvement method that can be applied to a wide domain since it can be used in arbitrary static control programs. This program class includes a large range of problems which are discussed in depth in chapter 2. Next, we do not lay down any requirement on dependences. Lastly, we will be able to find arbitrary affine transformation functions. Our method is not guaranteed to always transform the source program. In fact, if the source program is already optimized, our method should leave it as is. We will show that this is indeed the case, with the exception of a few purely syntactical modifications.

In the previous chapter we defined *chunking*, a program transformation scheme that makes possible an asymptotic evaluation of the traffic. Chunking aims at generating operation sets from a static control program. These sets should be built in such a way that they benefit from a lot of data locality and all their accessed data fit in the cache. Each set, called a *chunk*, has a unique number corresponding to its execution order. The data set accessed by a given chunk is called the chunk *footprint*. The problem is addressed in this way: for each statement $S$ we have to find a chunking function $\theta_S$ associating to each instance of a statement a chunk number. This

function is supposed to be affine, hence, a chunking function can be written:

$$\theta_S(\vec{x}_S) = T_S\vec{x}_S + \vec{t}_S$$

where $\vec{x}_S$ is the iteration vector, $T_S$ is the chunking matrix and $\vec{t}_S$ a constant vector. All the $T_S$ and $\vec{t}_S$ are the unknown of the problem. In this model, it is possible to make estimates of footprint sizes and traffic with respect to the chunking functions. These evaluations depend on the subscript matrices which can be extracted by analysis of the source code and $T_S$ which is the unknown of the problem. Thus we can find the constraints that $T_S$ has to satisfy in order that the footprints fit in the cache and the traffic is minimal.

Our present purpose is to know how to build, in a totally automatic way, a chunk system for every static control program. The construction should supply an optimal chunk system. It will be the case when each memory cell accessed during the program execution appears in as few footprints as possible. This condition is enough to guarantee that we have found the best possible chunk system, and to limit the number of chunks. Chunking functions are calculated thanks to several sets of constraints which are discussed in next sections. In section 5.1 we exhibit the constraints that the chunking functions have to satisfy in order to minimize the traffic by improving self-temporal locality. Section 5.2 shows how to choose the functions in such a way that the transformation is legal for dependences. Sections 5.3 and 5.4 gives respectively the constraints which have to be satisfied by the chunking functions in order to achieve group-locality and spatial-locality. Section 5.5 shows some directions to extend our framework. Lastly, section 5.6 presents experimental results, then section 5.7 concludes.

## 5.1 Self-Temporal Locality Constraints

Thanks to the evaluations, we know which rank constraints must be satisfied by the chunking matrices to minimize the traffic and to respect the capacity condition (see chapter 4). In this section, we show how to build such matrices, in particular, we will show that there always exists a solution such that each associated footprint fits in the cache. In section 5.1.1 we study the simplified case of statements with only one reference, in section 5.1.2 we generalize for arbitrary number of references.

### 5.1.1 Unique References

In this section, we address the problem of finding chunking matrices for programs where each statement has a unique reference. A statement has a unique reference if it accesses only one array. This implies there is only one subscript function $F_S$ per statement $S$. The goal is to find for each statement the function $T_S$ generating least traffic while satisfying the capacity condition. $T_S$ and $F_S$ are the only parameters for traffic and footprint size evaluations. Hence, we will search the best possible rank constraint pair $\left\langle \text{rank } T_S, \text{rank } \begin{bmatrix} T_S \\ F_S \end{bmatrix} \right\rangle$ for each statement, and build the matrix $T_S$ according to these constraints. Thus, the basic method is for each statement :

1. find the best rank constraint pair $\left\langle \text{rank } T_S, \text{rank } \begin{bmatrix} T_S \\ F_S \end{bmatrix} \right\rangle$,
2. build a $T_S$ matrix with respect to these constraints.

In the following we will describe both steps. We will show that we can always find a constraint pair satisfying the capacity condition and that the second step always succeeds, provided that rank $T_S$ and rank $\begin{bmatrix} T_S \\ F_S \end{bmatrix}$ respect obvious inequalities.

To find the best rank constraint pair for a given statement $S$, we have to evaluate both traffic and footprint size of each chunk that would be generated by a transformation fulfilling these constraints. We have to achieve the evaluations for every possible pair. The evaluation coefficients $l$ and $k$ of expressions (8) and (9) from chapter 4 are not arbitrary; it is easy to check that:

$$\begin{cases} \max\left(\text{rank } F_S - \text{rank } T_S, 0\right) & \leq & l & \leq & \min\left(\rho(S) - \text{rank } T_S, \text{rank } F_S\right) \\ \max\left(\text{rank } F_S, \text{rank } T_S\right) & \leq & k & \leq & \min\left(\rho(S), \text{rank } T_S + \text{rank } F_S\right) \end{cases}$$

Thus, possibilities are very limited since both the loop depth and the number of array subscripts are usually small. Hence the algorithm to find the best constraints is very simple: the principle is to scan the possible values of rank $T_S$ and rank $\begin{bmatrix} T_S \\ F_S \end{bmatrix}$ then to compute the generated traffic when the capacity condition is satisfied then lastly to return the best solution. The algorithm is shown in Figure 5.1. It needs the $F_S$ matrix in order to calculate the rank and use the number of columns $\rho(S)$. It is also necessary to know the order of magnitude of the cache size $C = O(m^\alpha)$ in order to check whether the capacity condition is respected. For instance, if the reference accesses a vector of length $m$ which does not fit in the cache, we would look for a solution such that $C = O(m^0)$. Theorem 5.1 proves this algorithm always find a solution, it corresponds to the trivial chunking where each chunk has only one operation.

**Theorem 5.1** *Algorithm 5.1 always find a solution that fulfills the capacity condition.*

*Proof.* The hardest constraint for the capacity condition is $\alpha = 0$, and the last choice tried by the algorithm will always be the constraint rank $T_S = \text{rank } \begin{bmatrix} T_S \\ F_S \end{bmatrix} = \rho(S)$ that gives $l = 0$, and where the traffic is maximum: $\mathcal{T} = O(m^{\rho(S)})$. ∎

Let us begin with an example to illustrate the construction process. We will try to improve the locality of the program presented in chapter 1 to illustrate the limits of the hierarchical memory systems. This program has only one statement $S$ with one array reference, $a(j)$:

```
      do i=1, times
   |  do j=1, m
S  |  | a(j) = a(j) + scalar
```

The subscript matrix of the reference $a(i)$ is $F = \begin{bmatrix} 0 & 1 \end{bmatrix}$, we can compute rank $F = 1$ to find the optimal constraint pair $\left\langle \text{rank } T_S, \text{rank } \begin{bmatrix} T_S \\ F_S \end{bmatrix} \right\rangle = \langle 1, 1 \rangle$ among all possible solutions checked by the algorithm 5.1 (there is a cross in the traffic evaluation when the footprint size is not acceptable):

| rank $T$ | rank $\begin{bmatrix} T_S \\ F_S \end{bmatrix}$ | Footprint | Traffic |
|---|---|---|---|
| 0 | 1 | $O(m^1)$ | × |
| 1 | 1 | $O(m^0)$ | $O(m^1)$ |
| | 2 | $O(m^1)$ | × |
| 2 | 2 | $O(m^0)$ | $O(m^2)$ |

**Best constraint algorithm**: find the rank constraints with the best properties

**Input:** the subscript matrix $F_S$, the loop depth $\rho(S)$, and the cache size $C = O(m^\alpha)$

**Output:** the best constraint pair $\left\langle \text{rank } T_S, \text{rank } \begin{bmatrix} T_S \\ F_S \end{bmatrix} \right\rangle$

1. traffic_power $= \infty$
2. for rank $T_S$ from 0 to $\rho(S)$
   (a) for rank $\begin{bmatrix} T_S \\ F_S \end{bmatrix}$ from $\max(\text{rank } F_S, \text{rank } T_S)$ to $\min(\rho(S), \text{rank } T_S + \text{rank } F_S)$
      i. if rank $\begin{bmatrix} F_S \\ T_S \end{bmatrix} - \text{rank } T_S < \alpha$ and rank $\begin{bmatrix} T_S \\ F_S \end{bmatrix} <$ traffic_power
         A. traffic_power $=$ rank $\begin{bmatrix} T_S \\ F_S \end{bmatrix}$
         B. constraints $= \left\langle \text{rank } T_S, \text{rank } \begin{bmatrix} T_S \\ F_S \end{bmatrix} \right\rangle$
3. return constraints

Figure 5.1: Algorithm to find the best constraints

Once the optimal constraint pair is found, it remains to build the matrix $T$ accordingly. For a statement $S$ with one reference, it is always possible to find a matrix $T_S$ such that rank $T_S = v$ and rank $\begin{bmatrix} T_S \\ F_S \end{bmatrix} = w$. The basic mechanism is to compose a generating matrix $G$ having the basis vectors of $\ker F$ as column vectors, which we extend to a non singular matrix. Then we compute the inverse of the generating matrix. $T$ is made of $v$ rows of the inverse and completed with null rows if necessary. The process is formally described in the algorithm in figure 5.2, the correctness is assessed by theorem 5.2.

**Theorem 5.2** *The algorithm 5.2 builds a matrix $T_S$ according to the rank constraints.*

*Proof.* Since the matrix $T_S$ is composed of $v$ linearly independent rows, the first constraint rank $T_S = v$ is satisfied. These rows are those of $G^{-1}$ from $\rho(S) - w + 1$ to $\rho(S) - w + v$. Hence, the kernel of $T_S$ is generated by the column vectors of $G$ from 1 to $\rho(S) - w$ and from $\rho(S) - w + v + 1$ to $\rho(S)$. The kernel of $\begin{bmatrix} T_S \\ F_S \end{bmatrix}$ is the intersection of the kernel of $T_S$ with the kernel of $F_S$, hence it is generated by the $\rho(S) - w$ first column vectors of $G$ and the constraint rank $\begin{bmatrix} T_S \\ F_S \end{bmatrix} = w$ is satisfied. As for the choice of $g$ (the number of rows of $T_S$), it is clear that bordering a matrix by null rows does not change its rank. Since when reordering the program it is useful to have all chunking function of the same dimension, we may take $g = \max \rho(S)$ ∎

This algorithm can always provide a solution if $v$ and $w$ have compatible values, i.e. when $\rho(S) \geq w \geq v$. The algorithm to find the best constraint pairs guarantees that it is always the case, because it looks for the solutions only in the correct domains.

Let us resume the example above. The best constraint pair is $\langle v, w \rangle = \langle 1, 1 \rangle$ The various steps of the chunking matrix construction are:

- we compute a basis of $\ker F$: $\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}$;

**Construction algorithm**: Build a matrix under rank constraints

**Input:** the subscript matrix $F_S$ and the rank constraints rank $T_S = v$, rank $\begin{bmatrix} T_S \\ F_S \end{bmatrix} = w$

**Output:** a matrix $T_S$ respecting the rank constraints

1. compute a basis $B$ of $\ker F_S$ and complete it to a basis of $\mathbb{N}^{\rho(S)}$
2. let $G$ be the matrix of these vectors (vectors added at step 1 to complete to a basis of $\mathbb{N}^{\rho(S)}$ are the last columns)
3. compute $G^{-1}$, inverse of $G$ possibly multiplying each line by the respective common denominator to stay with integral coefficients
4. build matrix $T_S$:
   (a) for $i$ from 1 to $v$:
   $i^{th}$ row of $T_S = (\rho(S) - w + i)^{th}$ row of $G^{-1}$
   (b) Complete $T_S$ with null rows

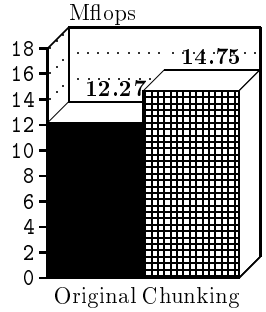
Figure 5.2: Construction algorithm

- we complete the basis of $\ker F$ to provide a basis of $\mathbb{N}^2$: $\left\{\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\}$;
- $G = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$;
- we compute $G'$, inverse of $G$; since $G$ is the identity matrix, $G = G'$;
- we build $T_S$ from $\rho(S) - n = 1$ lines of $G'$ from $w + 1 = 2$ to $\rho(S) + w - v = 2$ then until the last line ($g = 2$) by null lines: $T_S = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$.

The matrix $T_S$ corresponds to the following chunking function $\theta_S \begin{pmatrix} i \\ j \end{pmatrix} = j$. There is only one statement and the only operation is commutative, thus every operation reordering is legal. Then we can apply the transformation i.e. a loop interchange to build the target program. Assuming that the overhead of both FLUSH and INVALIDATE instructions is null, we can simulate the transformation and see the performance evolution, a 20% improvement on an i386 machine with one cache level of 32KB with $n = 50000$:

```
FLUSH
do c=1, n
S | do i=1, times
  |   a(c) = a(c) + scalar
  | INVALIDATE &a(c)
```

(a) target program




(b) performance evolution

### 5.1.2 Multiple References

In this section we study the general case of statements with multiple references. A statement $S$ has multiple references if it accesses various arrays, or several times the same array with different subscript functions. Thus the chunking matrix has to be computed with regard to several constraints from several references. Let $r_S$ be the number of references in statement $S$. Our purpose is to find for each statement $S$ the best constraints $\left\langle \text{rank } T_S, \left\{ \text{rank } \begin{bmatrix} T_S \\ F_{S,i} \end{bmatrix} \text{ for } 1 \leq i \leq r_S \right\} \right\rangle$, i.e. one rank constraint for the transformation matrix, and one rank constraint for each reference, such that each $\left\langle \text{rank } T_S, \text{rank } \begin{bmatrix} T_S \\ F_{S,i} \end{bmatrix} \right\rangle$ satisfies the capacity condition associated to the array indexed by $F_{S_i}$, and such that the global traffic generated for $S$: $\mathcal{T}_S = \sum_{i=1}^{r_S} \mathcal{T}_{S,i}$ is minimal. Unlike the unique reference case, we will see that it is not always possible to build a chunking matrix corresponding to a given set of rank constraints, even if we will see one more time that at least one solution is always possible. Thus we have to build a list of possible constraints ordered by decreasing quality. The algorithm to build such a list needs the subscript matrices and the expressions of the cache size relatively to the segment length for each reference. As in the unique reference case, the basic principle is to scan all the possible rank values (since the value space is very small) and to check if they respect the capacity condition. Then the possible constraint sets are sorted by order of increasing traffic. Figure 5.3 detailed this algorithm. Theorem 5.3 shows that this algorithm always find at least one solution. In the same way as for unique references, it corresponds to the trivial chunking where each chunk has only one operation.

**Theorem 5.3** *The constraint list produced by algorithm 5.3 has at least one element.*

*Proof.* This is the multiple reference version of theorem 5.1: the hardest capacity condition constraint is $\forall i, \alpha_i = 0$. For each $F_{S,i}$, we have to consider the constraint $\text{rank } \begin{bmatrix} T_S \\ F_{S,i} \end{bmatrix} = \rho(S)$ that always respects the hardest condition: $l_i = 0$. Then the element including these constraints $\left\langle \rho(S), \left\{ \text{rank } \begin{bmatrix} T_S \\ F_{S_i} \end{bmatrix} = \rho(S) \text{ for } 1 \leq i \leq r_S \right\} \right\rangle$ is always a solution where the traffic is maximum: $\mathcal{T} = O(\sum_{i=1}^{r_S} m_i^{\rho(S)})$. ∎

Let us illustrate the chunking matrix construction for multiple references with the program used as example for the legal transformation space computation in Figure 3.8:

```
   do i=1, n
S1 |  a(i) = i
   |  do j=1, m
S2 |  |  b(j) = (b(j) + a(i))/2
```

We assume as input hypothesis that $n$ array elements can fit in the cache, but $m$ cannot. Then, the acceptable orders of magnitude for the footprint size are $O\left(n^1\right)$ and $O\left(m^0\right)$. Such a simple code yet exhibits several difficulties: non-perfect loop nest, dependences between different statements, parameters and multiple references. The first statement has a unique reference and the best chunking function can be found by using algorithms described in the previous section (the best properties corresponds to the constraint pair $\langle 1, 1 \rangle$ and the chunking matrix is $T_{S1} = \begin{bmatrix} 1 \end{bmatrix}$). The second statement has multiple references, the first to the array $b$ with the subscript matrix $F_{S2,1} = \begin{bmatrix} 0 & 1 \end{bmatrix}$, and the second one to the array $a$ with subscript matrix $F_{S2,2} = \begin{bmatrix} 1 & 0 \end{bmatrix}$.

**Best constraint list algorithm**: Build a sorted list of possible constraints

**Input:** the subscript matrices $F_{S,i}$, the loop depth $\rho(S)$, and the cache size expressions with respect to each reference $C = O(m_i^{\alpha_i})$ for $1 \leq i \leq r_S$

**Output:** the list of possible constraint pairs

1. for each $S$ statement
   (a) list of constraints = $\emptyset$
   (b) for rank $T_S$ from 0 to $\rho(S)$
      i. Find all sets $\left\{\text{rank} \begin{bmatrix} T_S \\ F_{S,i} \end{bmatrix} \text{for } 1 \leq i \leq r_S\right\}$ such that for the array indexed by $F_{S,i}$, rank $\begin{bmatrix} T_S \\ F_{S,i} \end{bmatrix}$ − rank $T_S < \alpha_i$ then add the new constraint pair $\left\langle \text{rank } T_S, \left\{\text{rank} \begin{bmatrix} T_S \\ F_{S,i} \end{bmatrix} \text{for } 1 \leq i \leq r_S\right\}\right\rangle$ in the list
   (c) Sort the constraint pairs in order of increasing $\mathcal{T}_S$

Figure 5.3: Best constraint list algorithm

Then we have to sort the possible constraint pairs (respecting the capacity condition) by increasing traffic order. The following table shows all the tested solutions, the possible sorted constraints pairs $\left\langle \text{rank } T_S, \left\{\text{rank} \begin{bmatrix} T_S \\ F_{S,1} \end{bmatrix}, \text{rank} \begin{bmatrix} T_S \\ F_{S,2} \end{bmatrix}\right\}\right\rangle$ are $\langle 1\{1,1\}\rangle$, $\langle 1\{1,2\}\rangle$, $\langle 2\{2,2\}\rangle$:

| rank $T_S$ | rank $\begin{bmatrix} T_S \\ F_{S,1} \end{bmatrix}$ | rank $\begin{bmatrix} T_S \\ F_{S,2} \end{bmatrix}$ | Footprint | Traffic |
|---|---|---|---|---|
| 0 | 1 | 1 | $O(m^1 + n^1)$ | × |
| 1 | 1 | 1 | $O(m^0 + n^0)$ | $O(m^1 + n^1)$ |
| | | 2 | $O(m^0 + n^1)$ | $O(m^1 + n^2)$ |
| | 2 | 1 | $O(m^1 + n^0)$ | × |
| | | 2 | $O(m^1 + n^1)$ | × |
| 2 | 2 | 2 | $O(m^0 + n^0)$ | $O(m^2 + n^2)$ |

Next, we have to build a matrix $T_S$ that fulfills the best possible constraints. It is not possible to state the existence of a solution directly from the constraints. We showed in the unique reference case that the matrix $T_S$ is closely linked to the properties of matrices $F_{S,i}$ i.e. to the basis vectors of their kernels. Hence the construction of a matrix $T_S$ will be tried considering each possible constraint pair.

The major difference between the generalized construction algorithm and the unique reference version is the construction of the set $B$. Considering rank $T_S = v$ and rank $\begin{bmatrix} T_S \\ F_{S,i} \end{bmatrix} = w_i$, to respect a given $w_i$, $B$ should include a limited number of linearly independent vectors of the basis of $\ker F_{S,i}$. Indeed, to achieve a solution $T_S$ such that rank $T_S = v$ , $T_S$ has to be composed of $v$ linearly independent vectors. Hence the kernel of $T_S$ is generated by $\rho(S) - v$ linearly independent vectors, then $B$ has to encompass at most $\rho(S) - v$ vectors. The principle of the construction of $B$ is to begin with $B = \{\emptyset\}$ and to successively include for each $F_{S,i}$, $\rho(S) - w_i$ vectors of a basis of $\ker F_i$. The choice of vectors to be included in $B$ is essential. The order in which

we add vectors of $\ker F_{S,i}$ is not relevant. But we may have for the same $F_{S,i}$ several choices of vectors. Some choices will lead to exceed the $\rho(S) - v$ vectors limit in $B$ or to make impossible the compliance with a $w_j$, $j \neq i$. Thus we should benefit from those in span $B \cap \ker F_{S,i}$ that have been already included in $B$ because this limits the number of vectors in $B$, without interference on a constraint $w_j$, $\forall j \neq i$. Then we have to choose the remaining ones. Since we can not foresee the best choice of $(\rho(S) - \text{rank } F_{S,i}) - \dim(\text{span } B \cap \ker F_{S,i})$ vectors of $\ker F_i$ that remains to be included before having studied every $F_{S,i}$, the proposed algorithm uses a tree to remember every possible set $B$. The $T_S$ construction algorithm is formally described in Figure 5.4 and its correctness is addressed in theorem 5.4.

**Theorem 5.4** *Algorithm 5.4 builds matrices $T_S$ according to the rank constraints.*

*Proof.* This algorithm guarantees that the set $B$ is composed of exactly $\rho(S) - w_i$ vectors of a basis of $\ker F_{S,i}$ for $1 \leq i \leq r_S$. The vectors of $B$ are linearly independent since for each $F_{S,i}$ we add vectors from span $B \cap \ker F_i$. Finally, $B$ has at most $\rho(S) - v$ vectors. In these conditions, we can refer to the demonstration of the theorem 5.2 for the construction of $T_S$ in the unique reference case to prove that when such a set $B$ is found, a correct matrix $T_S$ is built. ∎

The generalized construction algorithm can not always find a solution since it may not exist. However, it will find the solution if it exists because the algorithm study all the possibilities for $B$ (up to a multiplication by a constant for each row vector). Eventually there is always a solution that respects the capacity condition since the pair $\langle 0, \{m_i = 0 \text{ for } 1 \leq i \leq r_S\}\rangle$ is inevitably in the list, and the solution: $T_S = I$ always exists.

Let us resume our example. The first constraint pair to study is $\langle 1, \{1, 1\}\rangle$. We have:

- for the matrix $F_{S2,1} = \begin{bmatrix} 0 & 1 \end{bmatrix}$, span $\ker F_{S2,1} = \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}$;
- for the matrix $F_{S2,2} = \begin{bmatrix} 1 & 0 \end{bmatrix}$, span $\ker F_{S2,2} = \left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$.

Thus $\dim$ span $\ker F_{S2,1} \cap$ span $\ker F_{S2,2} = 2$ and it is not possible to achieve both constraints $w_1 = 1$ and $w_2 = 1$ since we cannot add a second vector in $B$ without violating a constraint. Hence we have to try with the next pair $\langle 1, \{1, 2\}\rangle$, here the complete tree is easily calculated:

$$\emptyset \quad \rightarrow \quad \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\} \quad \rightarrow \quad \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}.$$

Then we can refer to the example in section 5.1.1 to build the $T_{S2}$ matrix. Eventually, the chunking matrices are:

$$T_{S1} = \begin{bmatrix} 1 \end{bmatrix}, \; T_{S2} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

In this example the dependences are complex, as shown in section 3.3. Thus we have to check the legality of the chunking:

$$\theta_{S1} \begin{pmatrix} i \end{pmatrix} = \begin{bmatrix} 1 \end{bmatrix} \begin{pmatrix} i \end{pmatrix}, \; \theta_{S2} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

We can use the method shown in section 3.3.3 to test the transformation and see that the chunking is not legal. How to solve this problem is the subject of the next section.

**Generalized construction algorithm**: Build matrices under rank constraints

**Input:** the subscript matrices $F_{S,i}$, the loop depth $\rho(S)$, and the rank constraints rank $T_S = v$, rank $\begin{bmatrix} T_S \\ F_{S,i} \end{bmatrix} = w_i$ for $1 \leq i \leq r_S$

**Output:** a set of matrices $T_S$ respecting the rank constraints

1. tree initialization: root at level 0 and a leaf at level 1 carrying $B = \{\emptyset\}$
2. for i from 1 to $r_S$
   for all the leaves of the tree at level $i$
   if dim $B + (\rho(S) - w_i) -$ dim (span $B \cap \ker F_i) \leq (\rho(S) - v)$
   (a) find the $q$ combinations of $(\rho(S) - w_i) -$ dim (span $B \cap \ker F_i$) vectors of a basis of $\ker F_i$ such that for every combinations $C$ we have:
   - dim (span $B \cap$ span $C) = 0$
   - $\forall j, j \neq i,$ dim $B +$ dim $C +$ dim $\ker F_j -$ dim $((\text{span } B \cup \text{span } C) \cap \ker F_j) \leq (\rho(S) - w_j)$

   (b) transform the current leaf in a node with $q$ leaves at level $i+1$, each leaf being labelled with a new set $B$ builded with one combination:
   $B = \{B,$ combination$\}$
3. set of matrices $= \emptyset$
4. for each leaves at level $r_S$
   (a) complete $B$ to a basis of $\mathbb{N}^{\rho(S)}$
   (b) let $G$ be the matrix of these vectors (vectors added at step 1 to complete to a basis of $\mathbb{N}^{\rho(S)}$ are the last columns)
   (c) compute $G^{-1}$, inverse of $G$ possibly multiplying each line by the respective common denominator to stay with integral coefficients
   (d) build matrix $T_S$:
      i. for $i$ from 1 to $v$:
      $i^{th}$ row of $T_S = (\dim B + i)^{th}$ row of $G^{-1}$
      ii. Complete $T_S$ with null rows
   (e) add $T_S$ to the set of matrices

Figure 5.4: Generalized construction algorithm

## 5.2 Building Legal Transformations

As a program restructuring scheme, chunking transformation modifies the operation execution order. Thus the existence of a good solution highly depends on data dependences. To bypass the dependence problem, most of the existing methods apply only to perfect loop nests in which dependences are non-existent or have a special form (fully permutable loop nests) [108]. To enlarge their application domain some preprocessing, e.g. *loop skewing* or *code sinking*, may enable them [108, 1, 53]. More ambitious techniques do not lay down any requirement on dependences, but are limited to propose *solution candidates* having some locality properties then to *check* them for legality [66, 17]. If the candidate is proven to violate dependences, then another candidate having less interesting properties is studied. In this section, we present a method that goes beyond checking by adjusting an optimizing transformation for dependence satisfaction, without modifying its locality properties. This technique can be used to correct a transformation candidate as well as to replace preprocessing. While designed at first for chunking purpose, this method can be adapted to most of the existing program transformation frameworks for data locality optimization.

This section is organized as follows. In section 5.2.1 we recall the properties of transformations for locality. Section 5.2.2 shows how it is possible to correct a transformation for legality.

### 5.2.1 Properties of Data Locality Transformations

Program transformations for locality aim at bringing the processing of some memory cells closer. The general framework using affine schedules is to find partial transformation functions (only the first few dimensions of the functions are defined) such that the partial execution dates of the operations referring to a given datum are the same. In this way, the operations have neighboring schedules and the datum may stay in the cache during the time intervals between the accesses. The framework ends by applying a completion procedure to achieve an invertible transformation function [108, 78, 55].

Let us consider self-temporal locality and a transformation candidate before completion $\theta_{Sc}(\vec{x}_S) = T_{Sc}\vec{x}_S + \vec{t}_{Sc}$. This function has the property that, modified in the following way:

$$\theta_S(\vec{x}_S) = C_S T_{Sc}\vec{x}_S + \vec{t}_S, \tag{1}$$

where $C_S$ is an invertible matrix and $\vec{t}_S$ is a constant vector, the locality properties are left unmodified for each time step. Intuitively, if $\theta_{Sc}$ gives the same execution date for $\vec{x}_1$ and $\vec{x}_2$, then the transformed function $\theta_S$ does it as well. In the same way if the dates are different with $\theta_{Sc}$, then the transformed function $\theta_S$ returns different dates. But while the values of $C_S$ and $\vec{t}_S$ do not change the self-temporal locality properties (basically, this transformation do not change the rank properties of $T_{Sc}$), they can change the legality of the transformation.

Transformation expressions similar to (1) and having the same type of degrees of freedom can be used to achieve every type of locality (*self* or *group* - *temporal* or *spatial*) [106, 17]. The challenge is, considering the candidate transformation matrices $T_{Sc}$, to find the *corrected matrices* $C_S T_{Sc}$ and the constant vectors $\vec{t_S}$ in order for the transformation system to be legal for dependences.

### 5.2.2 Finding Legal Transformations

Optimizing compilers typically decouple the properties that the transformation functions have to satisfy to achieve optimization and legality. The basic framework is first to find the best transformations (e.g. for data locality improvement, which references carry the most reuse and necessitate new access patterns, which rank constraints should be respected by the corresponding transformation functions, etc.), then to *check* whether a transformation candidate is legal or not. If not, build and try another candidate, and so on. The major advantage of such a framework is to focus firstly on the most interesting properties, and the main drawback is to forsake these properties if a legal transformation is not directly found after a simple check of a candidate solution. We saw in section 5.2.1 that there exists an infinity of transformation functions having the same properties as a candidate transformation (see formula 1). Thus, it is not possible to check all these transformations to find a legal one. In this section we study another way: we show how to find, when possible, the unknown components $C_S T_{Sc}$ and $\vec{t}_S$ of formula 1 in order to correct the transformations for legality.

This problem can be solved in an iterative way, each dimension being considered as a stand-alone transformation. Each row of $C_S T_{Sc}$ is a linear combination of the rows of $T_{Sc}$. Thus, the unknown in the $i^{th}$ algorithm iteration are, for each statement, the linear combination coefficients building the $i^{th}$ row of $C_S T_{Sc}$ from $T_{Sc}$ and the constant factor of the corresponding $\vec{t}_S$ entry. The constraints on both constant factors and linear combination coefficient are affine. Hence we can find a legal solution by solving these constraints and the legal transformation space constraints (see section 3.3.2) using any linear algebra tool. After each iteration, we have to update the dependence graph and to compute the new legal transformation space because there is no need to consider the dependences already satisfied. Thus, to find a solution is easier as the algorithm iterates. The algorithm is shown in figure 5.5. The correctness of the algorithm comes from two properties: (1) the target transformations are legal, (2) the $C_S$ matrices are invertible. The legality is achieved because each transformation part is chosen in the legal transformation space (see section 3.3.2) (step 1a). The second property follows from the updating policy (step 1(c)iv): at start the $C_S$ matrices are identities. During each iteration, we exchange their rows, multiply some rows by non null constants (as guaranteed by step 1b) and add to these rows a linear combination of the other rows. Each of these transformations does not modify the invertibility property.

Let us illustrate how the algorithm works using the example in figure 5.6. Suppose that an optimizing compiler would like to exploit the data reuse generated by the references to the array $A$ of the program in figure 5.6(a) and that it suggests the transformation candidates in figure 5.6(b). As shown by the graph describing the resulting operation execution order, where each arrow represents a dependence relation and each backward arrow is a dependence violation, the transformation system is not legal. The correction algorithm modifies successively each transformation dimension. Each stand-alone transformation splits up the operations into sets such that there are no backward arrows between sets. The algorithm stops when there are no more backward arrows or when every dimension has been corrected. Then any polyhedral code generator (see part III), can generate the target code. Choosing transformation coefficients as small as possible (step 1(c)i) is a heuristic helping code generators to avoid control overhead.

Let us resume the example began in section 5.1.2. We checked chunking function candidate and stated that they were not legal. However, applying our correction algorithm leads to the

**Correction Algorithm**: Adjust a transformation system to respect dependences

**Input:** a dependence graph DG, the transformation candidates $\theta_{Sc}(\vec{x}_S) = T_{Sc}\vec{x}_S$

**Output:** the legal transformations $\theta_S(\vec{x}_S) = C_S T_{Sc}\vec{x}_S + \vec{t}_S$

1. for dimension i = 1 to maximum dimension of $T_{Sc}$
   (a) build the legal transformation space with:
      - for each edge in DG, the constraints of formula (6) in chapter 3 for the $i^{th}$ row of $T_{Rc}$ and $T_{Sc}$
      - the constraints equating the $i^{th}$ row entries of each $C_S T_{Sc}$ with a linear combination of $T_{Sc}$ entries whose coefficients are unknown
   (b) for each statement, remove from the solution space the trivial solution where $\forall j \geq i$ the linear combination coefficient of the $j^{th}$ row of $T_{Sc}$ is null
   (c) if the solution space is empty, return $\emptyset$, else
      i. pick the solution giving for each statement the minimum values for the entries of the $i^{th}$ row of $C_S T_{Sc}$ and the $i^{th}$ element of $\vec{t}_S$
      ii. update DG: for each edge in DG, add to the dependence polyhedron the constraint equating the $i^{th}$ dimension of $C_S T_{Sc}\vec{x}_S + \vec{t}_S$ of the statements labelling the source and destination vertices (this may empty the polyhedron for integral solutions)
      iii. if every dependence polyhedra in DG are empty, goto 2
      iv. for each statement, update the candidate transformation $T_{Sc}$:
         - replace a row such that the corresponding linear combination coefficient is not null with the $i^{th}$ row
         - replace the $i^{th}$ row with the $i^{th}$ row of $C_S T_{Sc}$
2. return the transformation functions $\theta_S(\vec{x}_S) = C_S T_{Sc}\vec{x}_S + \vec{t}_S$

Figure 5.5: Algorithm to correct the transformation functions

following corrected transformation functions:

$$\theta_{S1}\begin{pmatrix} i \end{pmatrix} = \begin{bmatrix} 1 \end{bmatrix}\begin{pmatrix} i \end{pmatrix} + \begin{pmatrix} 0 \end{pmatrix} = \begin{pmatrix} i \end{pmatrix};\ \theta_{S2}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} n \\ 0 \end{pmatrix} = \begin{pmatrix} j+n \\ 0 \end{pmatrix}$$

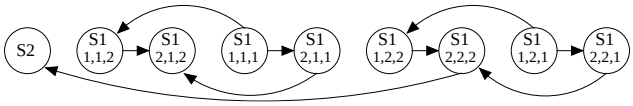
These functions lead to the following target code (the method for generation the target code will be described in part III):

```
   do c=1, n
S1 | a(c) = c
   do c=n+1, n+m
   | do i=1, n
S2 | | b(c-n) = (b(c-n) + a(i))/2
```

```
      do i=1, n
      |  do j = 1, n
      |  |  do k = 1, n
S1    |  |  |    A(j,k) = A(j,k) + B(i,j,k) / A(j,k-1)
S2    c = A(n,n) + 1
```

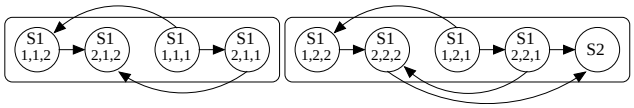
(a) Original program

$$\theta_{S1c}\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix};\ \theta_{S2c} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$



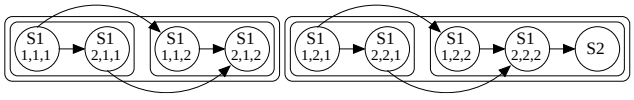

(b) Transformation function candidates

$$\theta_{S1c}\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ \hline 0 \\ 0 \end{pmatrix};\ \theta_{S2c} = \begin{pmatrix} n \\ \hline 0 \\ 0 \end{pmatrix}$$




(c) First correction iteration

$$\theta_{S1c}\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \hline 0 & -1 & 0 \\ \hline 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ \hline 0 \\ \hline 0 \end{pmatrix};\ \theta_{S2c} = \begin{pmatrix} n \\ \hline n \\ \hline 0 \end{pmatrix}$$




(d) Second and last correction iteration

```
      do j = 1, n
      |  do k = 1, n
      |  |  do i = 1, n
S1    |  |  |    A(j,k) = A(j,k) + B(i,j,k) / A(j,k-1)
S2    c = A(n,n) + 1
```

(e) Target program

Figure 5.6: Iterative transformation correction principle ($n = 2$ for graphs)

## 5.3 Group-Reuse Constraints

There is group-reuse when two statements, $S1$ and $S2$, access the same array `A` through subscript matrices $F_1$ and $F_2$ (for the sake of readability, we will use homogeneous coordinates[1] in this section). There is reuse if there exist two iteration vectors $\vec{x}_1$ and $\vec{x}_2$ such that $F_2\vec{x}_2 = F_1\vec{x}_1$, and this reuse is exploited if these two operations are in the same chunk:

$$\forall\vec{x}_1\forall\vec{x}_2, F_2\vec{x}_2 - F_1\vec{x}_1 = \vec{0} \Rightarrow T_2\vec{x}_2 - T_1\vec{x}_1 = \vec{0}. \quad (2)$$

Observe that this constraint has the same shape as a dependence constraint, and that we do not require that $S1$ and $S2$ belong to the same loop nest. If $F_2\vec{x}_2 = F_1\vec{x}_1$, then $S1(\vec{x}_1)$ and $S_2(\vec{x}_2)$ are in dependence. This dependence may be a read-read dependence, which may not be taken into account in other circumstances, but which exists nevertheless. As to the right-hand side of (2), it is similar but more restrictive than the right-hand side of the dependence relation:

$$\forall S1, S2 \in \mathcal{O}_P, S1(\vec{x}_1)\delta_P S2(\vec{x}_2) \Rightarrow \theta_{S1}(\vec{x}) \leq \theta_{S2}(\vec{x}_2).$$

As a consequence, we can give a more precise result:

**Theorem 5.5** *(2) is true iff* $\left[T_2\ ,\ -T_1\right] = N\left[F_2\ ,\ -F_1\right]$ *where* $N$ *is a matrix of full row rank.*

*Proof.* Let $\vec{x}$ be the concatenation of vectors $\vec{x}_1$ and $\vec{x}_2$. Formula (2) can be written

$$\forall\vec{x}, \left[F_2\ ,\ -F_1\right]\vec{x} = \vec{0} \Rightarrow \left[T_2\ ,\ -T_1\right]\vec{x} = \vec{0}.$$

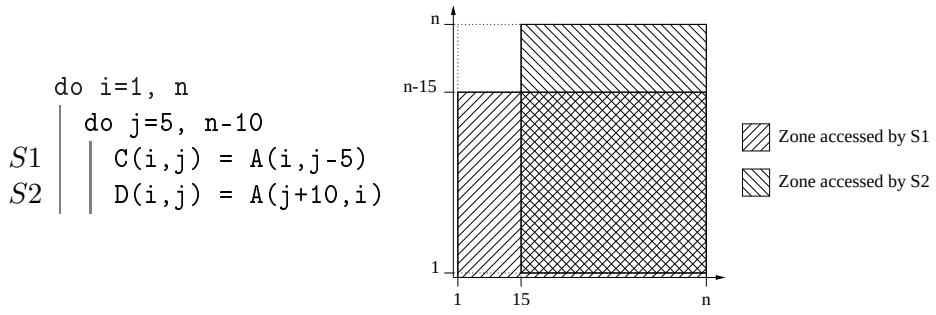
$\left[F_2\ ,\ -F_1\right]\vec{x} = \vec{0}$ and $\left[T_2\ ,\ -T_1\right]\vec{x} = \vec{0}$ describe two sets where one point belonging to the first one necessarily belongs to the second one too. Therefore the first one is a subset of the second one. So it can be written as the second one with $b$ additional constraints:

$$\left[F_2\ ,\ -F_1\right]\vec{x} = \vec{0} \Leftrightarrow \left\{ \begin{array}{c} \left[T_2\ ,\ -T_1\right]\vec{x} = \vec{0} \\ Q\vec{x} = \vec{0} \end{array} \right.$$

then $\left[ \begin{array}{c} T_2\ ,\ -T_1 \\ Q \end{array} \right] = M\left[F_2\ ,\ -F_1\right]$ with $M$ a matrix such that $\det M \neq 0$ (the system is not modified by linear transformations). Let us write $M$ as $\left[ \begin{array}{c} N \\ N' \end{array} \right]$ where $N'$ is the matrix made with the $b$ last lines of $M$. Now we have $\left[ \begin{array}{c} T_2\ ,\ -T_1 \\ Q \end{array} \right] = \left[ \begin{array}{c} N \\ N' \end{array} \right]\left[F_2\ ,\ -F_1\right]$ and finally $\left[T_2\ ,\ -T_1\right] = N\left[F_2\ ,\ -F_1\right]$. ∎

The unknowns are the entries of $N$, which define the linear transformations to apply to $\left[F_2\ ,\ -F_1\right]$ in such a way that the chunking functions respect the dependences. This is clearly the same problem as the correction for dependences in section 5.2. We solve them at the same time, by adding the necessary constraints (a set of constraints by pairs of references in which group-reuse is detected) to the initial problem. This theory, which does not assume that group-reuse is associated to constant dependences, can even be used for "self-group-reuse", when the

[1] This means that the expression constant parts are included in subscript and chunking matrices thanks to a new row per coefficient (one row for each global parameter and another one for the scalar), and that the iteration vectors are extended accordingly (an entry for each global parameter and a 1 for the scalar).




(a) sample code (b) Accessed zones of A

Figure 5.7: Example of group reuse

two accesses to A are in the same statement. Here, we deduce from (2) that the linear subspace $G = \{\vec{x}_2 - \vec{x}_1 | F_1\vec{x}_1 - F_2\vec{x}_2 = \vec{0}\}$ is included in the kernel of $T = T_1 = T_2$. It is easy to find a basis for $G$ by gaussian elimination techniques. The resulting vectors can be taken into account when building the chunking matrices. Improving group-locality do not change the order of magnitude of the traffic. It can divide the traffic generated by $n$ references by a factor of $n$.

For instance, let us consider the source code in figure 5.7(a). All control centric methods will estimate that there is no self reuse and no exploitable group-reuse. The reason is that they fail to consider non uniformly generated references (uniformly generated references are such as their subscript functions differ in at most the constant term [49]). In fact there is good reuse between the two statements for a part of the array A as shown by the figure 5.7(b). In this example, there is no dependence, then we can use the trivial solution of $\left[T_2\ ,\ -T_1\right] = N\left[F_2\ ,\ -F_1\right]$, that is $T_1 = F_1$ and $T_2 = F_2$. The computed chunking functions are:

$$\theta_{S1}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ j-5 \end{pmatrix}; \theta_{S2}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j+10 \\ i \end{pmatrix}.$$

This transformation leads to the target code below. The group-locality is now maximal: in the shared zone of A, the two statements access the same memory cell during the same iteration.

```
   do c1=1, 14
   |  do c2=0, n-15
S1 |  |  C(c1,c2+5) = A(c1,c2)
   do c1=15, n
S1 |  C(c1,5) = A(c1,0)
   |  do c2=1, n-15
S1 |  |  C(c1,c2+5)  = A(c1,c2)
S2 |  |  D(c2,c1-10) = A(c1,c2)
   |  do c2=n-14, n
S2 |  |  D(c2,c1-10) = A(c1,c2)
```

## 5.4 Spatial-Reuse Constraints

There is spatial reuse for a reference if it accesses data in the same cache line during different iterations. As for group locality, improving spatial locality does not change the order of magni-

tude of the traffic. It can divide the traffic generated by a reference by a factor of $d$, where $d$ is the cache line length in words. Spatial locality is achieved if the operations accessing the same cache line are in the same chunk. Let us consider a reference to an array $A$ with the subscript function $F$. Let $i$ be the number of the major dimension of $A$, i.e. the dimension with data lines ordered successively in memory. $i$ is programming language dependent. For instance, $i$ is 1 for C and $\rho(A)$ for FORTRAN. Then spatial locality is achieved for $A$ if the operations accessing the memory cells of the major dimension are in the same chunk. In other words, spatial locality is achieved if $F_{i,\bullet} \in \ker T$.

This constraint is added in the $T$ construction algorithm seen in section 5.1 by asking for a more accurate choice of vectors to be included in the matrix $G$. If the new constraint prevents the construction of $T$, we can try with another line of the subscript function and suggest the corresponding data layout transformation. This result can be compared with the Kandemir et al. method [61], where both loop and data transformations are used to improve spatial locality. Chunking does not require a non-singular transformation matrix, but it can achieve spatial locality only for a given loop level. However, in practice results are often alike.

## 5.5 Dealing With Large Arrays

In previous sections, the number of dimensions for each chunking function was bounded by $\rho(S)$ for each statement $S$. This constraint is the same for every statement-wise scheduling-based transformation frameworks that do not modify the original iteration domains except by using scheduling functions. In practice this may lead to the trivial result where each chunk has only one instance of a given statement, and where the traffic is maximum. To avoid this situation, we will propose some extensions to our framework that result in adding new dimensions to both transformation functions and input iteration domains.

A trivial solution to the chunking problem may be obtained even if data dependences allow every kind of transformations, in particular when some capacity conditions are contradictory. To remove some capacity constraints is possible. For instance we may not consider those from references that generate no self-temporal data reuse (this may be checked easily by verifying that the kernel of the corresponding subscript function is the null vector $\vec{0}$). This makes sense in the scratch pad memory case since no memory cells from these arrays should be considered to be placed in the SPM. In the classical cache memory case this property is not that clear since in any situation these references will pollute the data cache. However if the replacement policy is LRU then pollution effects are limited since in the chunking strategy there should not be capacity misses when considering a given cache size. Thus because of the data reuse of some references, the cache blocks that will be chosen for eviction will be most of the time those from references without data reuse. Hence we should consider them neither as a part of the chunk footprints nor to build the constraint system.

For instance let us consider the code in Figure 5.8 that computes traces of a matrix $A$ according to a given weight for each column. If $n$ is greater than the cache size then the reference to the array $A$ will lead to the trivial solution according to the method presented in previous sections. Nevertheless we may remove it from the footprint calculation since it does not generate reuse. Even with such simplification, the capacity conditions of the two remaining references are *not compatible* i.e. they lead together to the trivial solution but not separately. The according

```
      do i=1, n
      |  do j=1, n
S     |  |  trace(i+j) = trace(i+j) + w(j)*A(i,j)
```

Figure 5.8: Trace calculation kernel, an example of non-compatible constraints

footprint sizes and generated traffic are:

$$\text{Card } \mathcal{F} = O\left(2n^{\left(\text{rank}\begin{bmatrix} T \\ 1\ 1 \end{bmatrix} - \text{rank } T_S\right)} + n^{\left(\text{rank}\begin{bmatrix} T \\ 0\ 1 \end{bmatrix} - \text{rank } T_S\right)}\right)$$

$$\mathcal{T} = O\left(2n^{\left(\text{rank}\begin{bmatrix} T \\ 1\ 1 \end{bmatrix}\right)} + n^{\left(\text{rank}\begin{bmatrix} T \\ 0\ 1 \end{bmatrix}\right)}\right).$$

If $n$ is too big, we should have Card $\mathcal{F} = O(3n^0)$ and the only solution is $T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, i.e. the trivial one that generates the worst traffic $\mathcal{T} = O(3n^2)$.

We propose in this section two solutions to improve the traffic in the challenging case of non-compatible capacity constraints. Section 5.5.1 presents the first one that suggests to eliminate some embarrassing capacity conditions by decreasing the segment length. Section 5.5.2 shows how it is possible, starting from a trivial or a bad solution for a particular statement, to improve the traffic afterward.

### 5.5.1 Capacity Constraint Elimination

In spite of the expressivity power of scheduling functions, they do not offer sufficient flexibility in many cases. Our transformation framework asks for constructing chunks according to the size of the accessed data set. It may not be possible to provide solutions generating satisfying footprint sizes by using the transformation method presented in section 5.2. Let us illustrate this need for flexibility by considering the following code[2] and iteration domain:

```
      do i=0, n
S     |  a(i) = 0
```

$$\mathcal{D}_S : \begin{bmatrix} 1 \\ -1 \end{bmatrix}(i) + \begin{pmatrix} 0 \\ n \end{pmatrix} \geq \vec{0}$$

If the chunking function was $\theta(i) = T(i) + \vec{t}$, there would be only two possibilities:

- $T$ is the null matrix: there is only one operation set containing all the operations of the original program. This is not a solution when all data cannot fit in the cache together.
- $T$ is not the null matrix: there are $n$ operation sets containing only one operation from the original program. This is always a possible solution.

[2]Obviously such program should not be considered for chunking since it exhibits no self-temporal reuse and spatial locality is yet exploited, it is considered here for example reasons.

This choice can be too limited to find an interesting solution. Even in this simple case, if $n$ is too large, applying strictly the chunking transformation including the cache flush between each chunk would generate a high flushing overhead and would not allow to benefit for spatial locality. Hence, a basic idea is to split the data space accessed by a given loop. For the preceding example this boils down to strip-mine the loop and to add the resulting new dimension to the iteration vector. The corresponding code and iteration domain would be:

$$S \left| \begin{array}{l} \texttt{do ii=0, floor(n/b)} \\ \quad \texttt{do i=b*ii+1, min(b*ii+b,n)} \\ \quad\quad \texttt{a(i) = 0} \end{array} \right. \qquad \mathcal{D}'_S : \begin{bmatrix} b & -1 \\ -b & 1 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} ii \\ i \end{pmatrix} + \begin{pmatrix} b-1 \\ 0 \\ 0 \\ n \end{pmatrix} \geq \vec{0}$$

Where $b$ is any strictly positive scalar[3]. Using the new dimension, the chunking function should be $\theta \begin{pmatrix} ii \\ i \end{pmatrix} = T \begin{pmatrix} ii \\ i \end{pmatrix} + \vec{t}$, then there is some new intermediate possibilities:

- The column of $T$ corresponding to the new dimension $ii$ is not null while the other one is null: there are $n/b$ operation sets containing $b$ operations. This solution is always possible if $b$ is carefully chosen.
- The column of $T$ corresponding to the new dimension $ii$ is null while the other one is not null: there are $b$ operation sets containing $n/b$ operations. This solution depends on the value of the parameter $n$.
- Both columns of $T$ are non null, and rank $T = 1$: the number of chunks and operations inside each chunks depends on the entries of $T$. We can trivially estimate the upper bound of operations inside each chunk to $\min(n/b,b)$ (i.e. the maximum number of pairs $< ii, i >$ that give the same result for the affine expression $ii + i$). This solution is always possible if $b$ is carefully chosen.
- Both columns of $T$ are non null, and rank $T = 2$: this corresponds to the trivial solution, with $n$ chunks and each of them includes one operation.

Both the first and the third cases provide solutions whose capacity constraints only depend on the value $b$. By choosing it smaller than the cache size, we can ensure that these solutions are possible.

We can derive from this example a general method to achieve flexibility in three steps. The first one is to choose the capacity constraints to remove. Using a precise evaluation of the segment length (see section 4.2.4) may give useful informations about which references provide the most reuse. This allows to sort them according to their reuse volume in order to consider only the main opportunities to improve data locality and to remove the constraints from other references leading to a trivial or a bad solution. Another criterion is to regroup as many equivalent constraints as possible to consider for improving or to consider for removing. Hence the processing is easier since to treat one constraint may affect many references at a same time. For instance let us consider the input code in Figure 5.8, we will not take into account the reference to the array $A$. The two others references generate non-compatible constraints. They provide appreciably the

[3]There are at present no solutions to handle the fully-parametric case (where $b$ is any integral parameter) in the whole framework and in particular in the code generation step. Ongoing works are addressing this issue [56].

same amount of reuse, we choose to remove the constraints generated by $w$ since the processing for the second step is easier.

The second step is to split the challenging data spaces in order to be able to remove their respective capacity constraints. The basic idea for each case is to build a loop that scans the whole data space of the considered reference dimension and to strip-mine it. When the array dimension has not a coupled subscript (the subscript function for this dimension depends at most on one outer loop counter e.g. the reference to $b$ in Figure 5.8), we only have to strip-mine the loop corresponding to the iterator in the subscript function. This transformation is always legal. When the subscript is coupled (as the reference to *trace* in Figure 5.8) we have to build the loop that scans each values of the subscript function (basically by applying a *skewing* for the coupled case that may be easily computed: the chunking function is equal to the subscript function up to a constant factor, and checked for legality by using our methods presented in chapter 5) and strip-mine this loop. Then we can add to the chunking matrix a column and a row corresponding to the new dimension. For instance let us consider the input code in Figure 5.8 where we want to remove the capacity constraint generated by $w$. We need to strip-mine the loop $j$ according to the subscript function of $w$. In our model this only imply modifying its iteration domain by adding a new column corresponding to the new loop and the associated constraints:

```
   do i=1, n
   |  do jj=1, floor(n/b)
S  |  |  do j=max(b*jj,1), min(b*jj+b-1,n)
   |  |  |  trace(i+j) = trace(i+j) + w(j)
```

$$\mathcal{D}'_S : \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & b & -1 \\ 0 & -b & 1 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{pmatrix} i \\ jj \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ b-1 \\ 0 \\ -1 \\ n \end{pmatrix} \geq \vec{0}$$

To ensure that each chunk will include only one part of the split data space, we add the constraint to the transformation function construction process that the new column of the chunking function can not be null (this is included in the correction algorithm 5.5 by adding the constraint that the coefficient of the new dimension iterator is not null at each step of the algorithm: if there is no solution we try without the constraint and we delay it to the next dimension until it has been satisfied, if there is no solution we return $\emptyset$). Because of this constraint on the transformation function we know that the number of accessed data for the considered array dimension is at most the size of each resulting block, $b$. Thus we can write the new evaluation of the footprint sizes for our example:

$$\text{Card } \mathcal{F} = O\left( (n+b)^{\left(\text{rank } \begin{bmatrix} T \\ 1\ 1 \end{bmatrix} - \text{rank } T_S\right)} + b^{\left(\text{rank } \begin{bmatrix} T \\ 0\ 1 \end{bmatrix} - \text{rank } T_S\right)} \right)$$

If $b$ is carefully chosen, the second part of the evaluation does not matter. Then we can optimize using the classical chunking mechanism with the additional constraint on $T$.

The third part is then to apply the chunking transformation method as presented in previous sections. In our example only a reference is still challenging and ask for rank constraints on $T$. The computed solution is $\theta \begin{pmatrix} i \\ jj \\ j \end{pmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ jj \\ j \end{pmatrix}$, and leads to the following target code produced by the method described in part III:

```
do c1=0, floor(n/d)
  do c2=max(b*c1+1,2), min(b*c1+n+b-1,2*n)
    do i=max(-b*c1+c2-b+1,c2-n,1), min(-b*c1+c2,n,c2-1)
S     j = c2 - i
      trace(i+j) = trace(i+j) + w(j)
```

To conclude the example, we measured the cache misses generated by the two code versions and we shows the results in Figure 5.9. The target computer had a 64KB L1 data cache and a 256KB L2 unified cache, the chosen value for $b$ was 100. We can see that once the two arrays no more fit in the cache together (at $n = 8K$ for L1 and $n = 32K$ for L2), the chunked code achieves better results. Unfortunately because of the high control overhead, we achieved a limited speedup of 9% when $n = 16K$ but of 75% when $n = 64K$ since L2 cache misses are more costly.

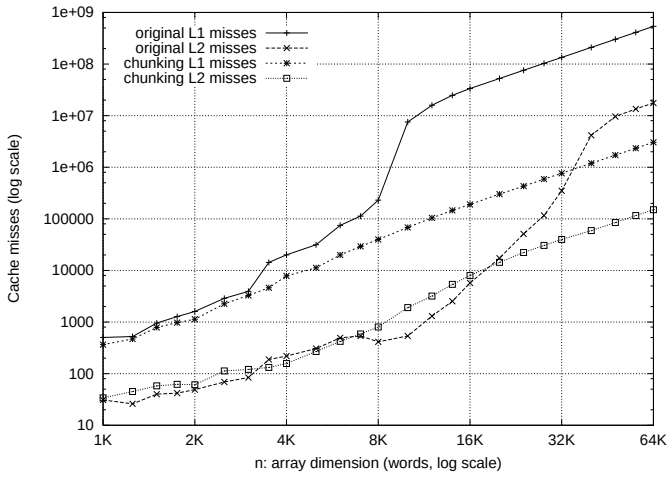



Figure 5.9: Cache misses of original and chunked codes of Figure 5.8 example (without $A$)

### 5.5.2 Chunk Fusion

Once the chunking functions have been constructed, it is still possible to post-process the transformation in order to achieve more flexibility and to reduce the traffic. If too many constraints have to be taken into account, the chunking process may produce extremely small operation sets with very small footprints. In this case it may be interesting to take advantage of *inter-chunk locality* i.e. when different chunks access similar memory locations by merging some of them. Merging chunks is possible as long as both dependences and capacity constraints are satisfied. A trivial particular case is that it is always legal to merge consecutive chunks with a direct benefit on the traffic if the accessed data overlap (but the capacity condition has still to be satisfied). Let us consider a reference that generates some reuse. Merging $b$ chunks that access the same memory cells of the corresponding array will divide the traffic generated by the reference by a factor of $b$. The footprint size generated by this reference is not modified since no additional memory cells are accessed inside each chunk. On the opposite, the footprint size of the references that do not benefit from the same inter-chunk reuse are multiplied by a factor of $b$.

The basic principle of chunk fusion is to reorder the chunks in such a way that consecutive

chunks access the same data for a given reference. Then we may merge them by applying a rational transformation as described in section 3.2.2 where the division coefficient is $b$. Finding the transformation that reorders the chunks according to a reference may be achieved easily by using our methods presented in the previous sections. For instance let us consider the code in Figure 5.8 without the reference to $A$. If $n$ is too large and we do not use the method presented in section 5.5.1, the chunking process will achieve the trivial result:

```
   do c1=1, n
   |  do c2=1, n
S  |  |  trace(c1+c2) = trace(c1+c2) + w(c2)
```

The corresponding generated footprint size is Card $\mathcal{F} = O(3n^0)$ and the traffic evaluation is $\mathcal{T} = O(3n^2)$. We may reduce the generated traffic by merging $b$ chunks that access the same memory cells of $w$, i.e. merging chunks perpendicularly to a reuse vector of the considered reference $w$. In this example, the dimension $c2$ is perpendicular to the reuse vector of $w$, then we merge several loops $c2$ in the same chunk by applying the transformation $c1/b$. The result is:

```
   do c1=1, floor(n/b)
   |  do c2=1, n
   |  |  do i=max(b*c1,1), min(b*c1+b-1,n)
S  |  |  trace(i+c2) = trace(i+c2) + w(c2)
```

The generated footprint size is now Card $\mathcal{F} = O(2n^0 * b + n^0)$ and the generated traffic has been decreased to $\mathcal{T} = O(2n^2 + (n^2)/b)$.

Both methods presented in this section add dimensions to the input problem and allow to achieve *tiling*-like transformations [107], with the strength that we benefit from our correction algorithm in Figure 5.5 to find automatically some useful preprocessing to make the tiling possible. The weakness is that we do not consider the order between the tiles for locality while it is usually taken into account for tiling. Nevertheless this is a consideration for classical cache memories, since in the scratch pad case, taking advantage of data previously stored then evicted has no sense. These methods allow to handle strong capacity conditions in our framework. To ensure to propose the best transformation, we may compute the chunking transformation and generate the target code for every possible conditions and switch on the best solution at execution time when the parameters are known.

## 5.6 Experimental Results

We are implementing the chunking approach in *WRAP-IT*[4] [16], our polyhedral framework in Open64/ORC. This implementation is still not complete and we used our separate source-to-source optimizing tool *chunky*[5] [13, 18, 17]. This prototype implements at present the chunking function calculation to the code generation in C; the dependence calculation and application of Farkas lemma still use a Maple implementation. The dependence correction and code generation

[4]WRAP-IT is freely available under GNU public license at `http://www.lri.fr/~girbal/site_wrapit`
[5]Parts of Chunky are freely available under GNU public license at `http://www.prism.uvsq.fr/~cedb`

make an intensive use of polyhedral operations implemented by PolyLib[6] [105] and PipLib[7] [42, 47]. This prototype already allows us to test various non-trivial problems.

Experiments were conducted on a PC workstation with a Pentium III processor running at 1GHz. This processor comes with two cache levels: a split first level (L1) for instructions and data of 16KB each and an unified second level (L2) of 256KB. L1 is a 4-way set associative cache with a miss penalty of 3 cycles. L2 is an 8-way set associative cache with a miss penalty of 44 cycles. Both cache levels are non-blocking and have a line size of 32 bytes. To make the best evaluations, we choose to use the hardware counters of the Pentium III to compare the number of cache misses and the overall number of CPU cycles for each calculation. The compiler option was O3 for the original programs, but O1 for the transformed programs in order to prevent any compiler optimization that can disturb chunking. Furthermore we have selected by hand the data layout that gives the best results for the input problems.

Figure 5.10 summarizes some results on several computational kernels from various sources. The first column of the array gives the problem name, the second one shows which type of constraints have been useful for improving locality in the corresponding program (ST: self-temporal locality constraints, see section 5.1, SS: self-spatial, see section 5.4, G: group, see section 5.3). The three remaining columns give the problem size description and the observed results on the target architecture. Each problem have been tested for two array sizes, one where the accessed data set of the problem cannot fit in the first cache level but fits in the second cache level and another where the data set cannot fit either in the L1 cache or in the L2 cache. For each case we measured both the cache misse reduction and the overall performance improvement of the transformed program with respect to the original one. The capacity constraints were set in such a way that for each array accessed in the input program at least one dimension can fit in the cache, in order to avoid to achieve systematically a trivial transformation. Methods discussed in section 5.5 can then be used to handle stronger constraints.

The first four rows of Figure 5.10 present the results achieved with our running example and several well known kernels. For each of these problems, the correction algorithm had to modify the suggested transformations for dependences in order to propose the best possible data locality properties. Input codes, chunking functions, target code and cache miss measurements are detailed in Figures 5.11 and 5.12. For the running example, the ratio $m/n$ was set to 64 in order to better show the impact of our method. For this example, Figure 5.11(a)(4) shows the evolutions of the number of cache misses observed with hardware counters for the original and target versions, according to the value of the parameter $m$. The number of cache misses sharply grows when the array $b$ becomes larger than a cache level in the original program. The chunked program has a better behavior: the miss growth comes later, when the input hypothesis are no longer satisfied, *i.e.* when the array $a$ cannot fit in the cache. We have observed the same phenomenon on most of the programs with good data reuse we have tested. This is shown in Figures 5.11(b), 5.12(c) and 5.12(d) for the three others kernels, LU decomposition, Cholesky factorization and Gauss-Jordan elimination. When capacity conditions used to build the chunking functions are no longer respected, we have to build new functions accordingly.

As for the running example, chunking can reduce the number of cache misses of other kernels by more than one order of magnitude. This cache miss reduction can imply a significant performance improvement. The speedup is better with big problems. Since the miss penalty for

[6] PolyLib is freely available under GNU public license at `http://icps.u-strasbg.fr/PolyLib`

[7] PIP/PipLib is freely available under GNU public license at `http://www.prism.uvsq.fr/~cedb`

| problem | optimization | array size (words) | missdown (%) | speedup (%) |
|---|---|---|---|---|
| running example | ST | 16K | 99.1 (L1) | 7 |
| | | 1M | 99.9 (L2) | 427 |
| LU decomposition | ST+SS | $80 * 80$ | 79.3 (L1) | 2 |
| | | $256 * 256$ | 84.1 (L2) | 43 |
| Cholesky factorization | ST+SS | $80 * 80$ | 70.3 (L1) | 2 |
| | | $256 * 256$ | 85.5 (L2) | 46 |
| Gauss-Jordan | ST | $80 * 80$ | 70.2 (L1) | -13 |
| | | $256 * 256$ | 93.1 (L2) | 26 |
| Matvect transpose | G | $80 * 80$ | 59.6 (L1) | 6 |
| | | $256 * 256$ | 96.3 (L2) | 28 |
| Matrix multiply | SS | $80 * 80$ | 62.4 (L1) | 5 |
| | | $256 * 256$ | 76.0 (L2) | 9 |
| Reduction factorization | ST+G | 8K | 50.1 (L1) | 21 |
| | | 512K | 51.0 (L2) | 31 |
| Swim | SS | $80 * 80$ | 0.4 (L1) | 24 |
| | | $256 * 256$ | 49.8 (L2) | 44 |
| Ocean | ST+G | $80 * 80$ | 52.0 (L1) | 33 |
| | | $256 * 256$ | 93.2 (L2) | 42 |
| Hydro fragment | ST | 6K | 99.9 (L1) | 73 |
| | | 300K | 99.9 (L2) | 99 |
| Equation of state fragment | ST | 4K | 92.6 (L1) | 86 |
| | | 256K | 99.0 (L2) | 94 |
| First difference | ST | 8K | 99.9 (L1) | 14 |
| | | 512K | 98.9 (L2) | 88 |
| Planckian distribution | ST | 4K | 99.9 (L1) | 1 |
| | | 64K | 99.9 (L2) | 3 |

Figure 5.10: Experimental Results

an L2 miss is of the order of 10 times an L1 miss, these results are not surprising. The situation of Gauss-Jordan for $80 * 80$ arrays shows how it is necessary to avoid control overhead. In this (rare) case, despite the attention given to code generation (see part III) and a significant cache miss reduction, our method fails to improve performance on small problems (because of a the need to compute a minimum deeply inside a loop: see Figure 5.12(d)(3)). The point of view is quite different when the critical resource is energy, like in embedded systems. Cathoor et al. [29] show that data movements in the hierarchy is one of the main cause of energy consumption. In this case, a cache miss reduction is always a benefit.

The next four rows depict the results on others kernels, *Matvect transpose* is a double matrix-vector multiply where the matrix in the second calculation is the transposition of the matrix in the first one. It shows how our method can improve group locality even in the case of non-uniformly generated references. *Matrix multiply* shows that our evaluation method it able to find the best loop order and to improve this computational kernel only with loop transformation (i.e. without *tiling*), as also demonstrated by Mc Kinley et al. [85]. *Reduction factorization* is a kernel from [83], *Swim* and *Ocean* have been extracted from the SPEC2000fp and PerfectClub benchmark suites respectively. Our *Swim* kernel has been extracted after converting the original FORTRAN program to C, our transformation solved the spatial locality loss due to the data layout modification. The remaining four kernels are from Livermore Loops and are very simple

codes we used to check our tool. Overall, the prototype showed how it is possible to improve locality and to achieve better performance in many different situations thanks to chunking coupled with a code generation scheme that avoids control overhead as far as possible.

While some parts of our method have high theoretical complexity in the worst case (code generator and parametrized linear programming solver have exponential worst case complexities), the prototype offers good performance for our kernels. The reason is that the main parameters are loop nest depths and array dimension which are usually small. To give an idea, the chunking of a Cholesky factorization with 7 statements, a maximal loop nest depth of 3 and a maximal array dimension number of 2 requires about 20 seconds on the test machine. Most of the time is spent in Maple code and we have many reasons to think that a better implementation will significantly improve the prototype performance. Nevertheless, the question of scalability remains, and will be tested on a larger benchmark suite.

## 5.7 Conclusion

We showed in this chapter how, starting from a method based on memory traffic evaluations, it is possible to build transformation functions to decrease this traffic. We described each property of the resulting code as constraints on the chunking functions. These properties include achieving self-temporal locality, self-spatial locality, group locality as well as legality. We defined the algorithms that choose and build the transformation functions according to these constraints. The method requires nothing besides the original code but the relative sizes of the cache and data. It exhibits many interesting properties. First of all, the computed solution always fulfills the imposed memory requirements. Next, it can be applied to any static control part of a program; in particular there is no requirement on dependences and we compute the space of all legal transformations directly. Lastly, we deal with general programs which have static control regions but do not have static control *in toto*. Locality optimization has the nice property that there is no need of applying it to far away statements, since the hope of having reuse in this situation is very small. Hence chunking can be applied locally, i.e. to loop nests or small subroutines, and there is only a limited danger of an excessive compilation time. Experimental evidence shows the ability of our framework to improve both locality and performance in various situations, even when data dependences are complex thanks to our transformation correction algorithm. Overall our framework allows to handle more general programs (e.g. not only perfectly nested loops, or programs with simple data dependences), to apply more general transformations (e.g. not only non-unimodular or non-invertible transformations) and to find them in a wider space (we may correct them for legality automatically) than many other methods.

Once the transformation functions have been constructed and applied in the polyhedral domain, we have to generate the target code in such a way that control overhead do not offset our optimizations. This problem is addressed in the next part of this thesis.

```
   do i=1, n
S1 | a(i) = i
   | do j=1, m
S2 | | b(j) = (b(j)+a(i))/2
```

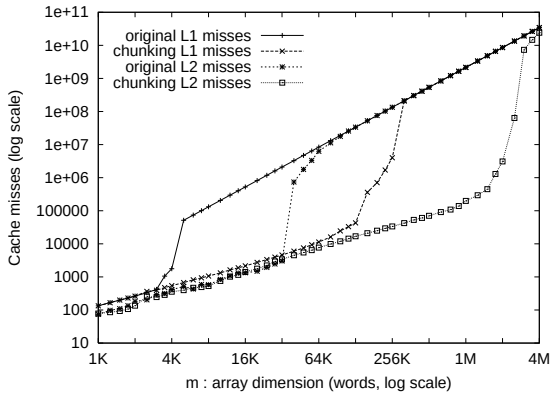
(1) Input code

$$\theta_{S1}(i) = (i),\ \theta_{S2}(i,j) = (j+n)$$

(2) Chunking functions

```
   do c1=1, n
S1 | a(c1) = c1
   do c1=n+1, n+m
   | do i=1, n
S2 | | b(c1-n) = (b(c1-n)+a(i))/2
```

(3) Target code




(4) Cache misses of original and chunked codes

(a) Chunking of the running example

```
   do i=1, n
   | do j=i+1, n
S1 | a(j,i) = a(j,i)/a(i,i)
   | | do k=i+1, n
S2 | | | a(j,k) = a(j,k)-a(j,i)*a(i,k)
```

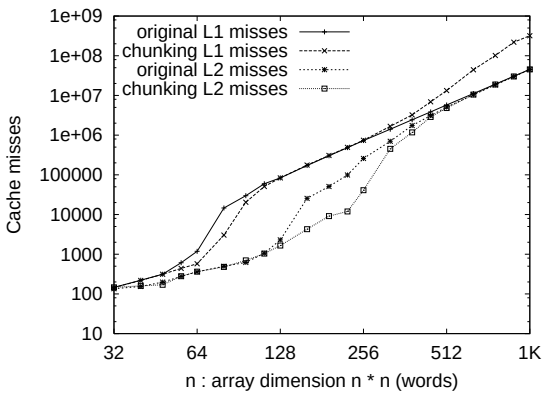
(1) Input code ($a$ is row major)

$$\theta_{S1}(i,j) = (i,n),\ \theta_{S2}(i,j,k) = (k,j)$$

(2) Chunking functions

```
   do j=2, n
S1 | a(j,1) = a(j,1)/a(1,1)
   do c1=2, n-1
   | do c2=2, n-1
   | | do i=1, min(c2-1,c1-1)
S2 | | | a(c2,c1) = a(c2,c1)-
   | | |            a(c2,i)*a(i,c1)
   | do i=1, c1-1
S2 | | a(n,c1) = a(n,c1)-a(n,i)*a(i,c1)
   | do j=c1+1, n
S1 | | a(j,c1) = a(j,c1)/a(c1,c1)
   do c2=2, n
   | do i=1, c2-1
S2 | | | a(c2,n) = a(c2,n)-
   | | |           a(c2,i)*a(i,n)
```

(3) Target code ($a$ is column major)




(4) Cache misses of original and chunked codes

(b) Chunking of a LU decomposition

Figure 5.11: Chunking of the running example and LU decomposition

```
   do i=1, n
S1 | a(i,i) = sqrt(a(i,i))
   | do j=i+1, n
S2 | | a(j,i) = a(j,i)/a(i,i)
   | | do k=i+1, j
S3 | | | a(j,k) = a(j,k)-a(j,i)*a(k,i)
```

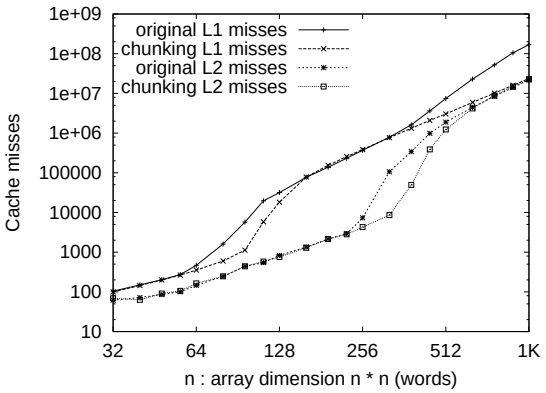
(1) Input code ($a$ is column major)

$$\theta_{S1}(i) = (i, 0),\ \theta_{S2}(i, j) = (i, n),\ \theta_{S3}(i, j, k) = (k, j)$$

(2) Chunking functions

```
S1 a(1,1) = sqrt(a(1,1))
   do j=2, n
S2 | a(j,1) = a(j,1)/a(1,1)
   do c1=2, n-1
S1 | a(c1,c1) = sqrt(a(c1,c1))
   | do c2=c1, n-1
   | | do i=1, c1-1
S3 | | | a(c2,c1) = a(c2,c1)-
   | | |            a(c2,i)*a(c1,i)
   | do i=1, c1-1
S3 | | a(n,c1) = a(n,c1)-a(n,i)*a(c1,i)
   | do j=i+1, n
S2 | | a(j,c1) = a(j,c1)/a(c1,c1)
S1 a(n,n) = sqrt(a(n,n))
   do i=1, n-1
S3 a(n,n) = a(n,n)-a(n,i)*a(n,i)
```

(3) Target code ($a$ is row major)




(4) Cache misses of original and chunked codes

(c) Chunking of a Cholesky factorization

```
   do i=1, n
   | do j=1, i-1
   | | do k=i+1, n
S1 | | | a(j,k) = a(j,k)-
   | | |          a(i,k)*a(j,i)/a(i,i)
   | do j=i+1, n
   | | do k=i+1, n
S2 | | | a(j,k) = a(j,k)-
   | | |          a(i,k)*a(j,i)/a(i,i)
```

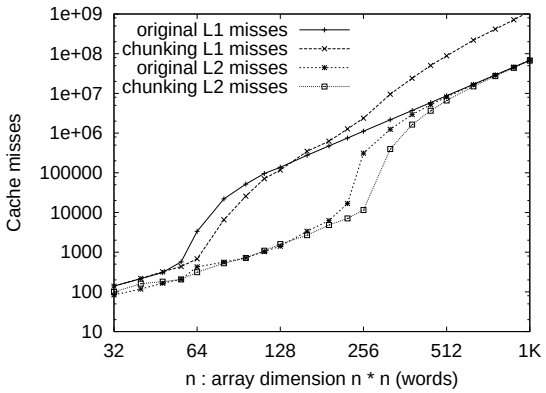
(1) Input code ($a$ is row major)

$$\theta_{S1}(i, j, k) = (j, k, 0),\ \theta_{S2}(i, j, k) = (j, 0, k)$$

(2) Chunking functions

```
   do c1=2, n
   | do c3=2, n
   | | do i=1, min(c3-1,c1-1)
S1 | | | a(c1,c3) = a(c1,c3)-
   | | |            a(i,c3)*a(c1,i)/a(i,i)
   do c1=n+1, 2*n-2
   | do c2=c1-n+2, n
   | | do i=c1-n+1, c2-1
S2 | | | a(c1-n,c2) = a(c1-n,c2)-
   | | |              a(i,c2)*a(c1-n,i)/a(i,i)
```

(3) Target code ($a$ is row major)




(4) Cache misses of original and chunked codes

(d) Chunking of a Gauss-Jordan elimination

Figure 5.12: Chunking of a Cholesky factorization and a Gauss-Jordan elimination

# Part III

# Code Generation

# Chapter 6

# Scanning Polyhedra

Let us continue the analogy of the polyhedral framework for compiling with the Fourier Transform for signal processing as begun in chapter 3. Once the useful transformations are done in the intermediate space, we have to return back to the original space by using the Inverse Fourier Transform. In the polyhedral model, this inverse transformation starting from the polyhedral representation of a SCoP to return back to an abstract syntax tree (or directly to a target source code) is called *code generation*. The focus of recent compiler research is optimized code generation. The reason is that, with a higher level of integration, modern processors have acquired high-level features (vector processing, hidden and apparent parallelism, memory hierarchies), which are not available in so-called high-level languages, which were patterned after the much simpler processors of the 1960's. Hence, there is a semantic gap which grows larger and larger with time. A striking example is the Multimedia Instruction Set of many popular processors, which so far cannot be used directly in any high-level language.

Usual compiler intermediate representations like abstract syntax trees are not appropriate for complex program restructuring. Simple optimizations e.g. constant folding or scalar replacement may be achieved without hard modifications of such stiff data structures. But more complex transformations such as loop inversion, skewing, tiling etc. modify the execution order and this is far away from the syntax. In many cases, optimization is done in three steps: (1) select a new execution order, most of the time the result is not a program, but a reordering function (a schedule, or a placement, or a chunking function); (2) build a loop nest (or a set of loop nests) which implement the execution order implied by the reordering function; (3) apply the local optimizations which have been enabled by the new execution order: for instance, mark some loops for parallel execution.

The main theme of this part is the second step of the above scheme. Finding suitable execution orders has been the subject of the previous part and of most of the research on the *polyhedral model* [17, 24, 25, 40, 45, 54, 78, 92, 99, 110] and the post-processing is easy in general. On the other hand, simple-minded schemes for loop building have a tendency to generate inefficient code, which may offset the optimization they are enabling.

In this chapter we define the polyhedron scanning problem. We present related work and strategies in section 6.1. We show in section 6.2 that a good code generation highly depend on the chosen transformations. We specify some constraints to achieve an efficient code and we apply them to chunking. We show also that an intermediate step between transformation

function construction and code generation may improve the effectiveness of the generated code.

## 6.1 Polyhedron Scanning Problem

We showed in section 2 that any static control program can be specified using a set of iteration domains and scheduling functions that can be merged to create new polyhedra with the appropriate lexicographic order. Generating code in the polyhedral model amounts to finding a set of nested loops visiting each integral point of each polyhedra, once and only once, following this order. This is a critical step in the framework since the final program effectiveness highly depends on the target code quality. In particular, we must ensure that a bad control management does not spoil performance, for instance by producing redundant conditions, complex loop bounds or under-used iterations. On the other hand, we have to avoid code explosion for instance because a large code may pollute the instruction cache.

Ancourt and Irigoin [5] proposed the first solution to the polyhedron scanning problem. Their seminal work was based on the Fourier-Motzkin pair-wise elimination [97] (see also Appendix A). The scope of their method was very restrictive, since it could be applied to only one polyhedron, with unimodular transformation matrices. The basic idea was to apply the transformation function as a change of basis of the loop index (see chapter 3), then for each new dimension, to project the polyhedron onto the axis and deduce the corresponding loop bounds. For a given axis $i_k$, the Fourier-Motzkin algorithm can establish that $L(i_1, ..., i_{k-1}) + \vec{l} \leq c_k i_k$ and $c_k i_k \leq U(i_1, ..., i_{k-1}) + \vec{u}$, where $\vec{l}$ and $\vec{u}$ are constant vectors of size $m_l$ and $m_u$ respectively. Thus we can derive the corresponding scanning code for the subscript $i_k$:

```
do i_1 = ...
|  ...
|  do i_k = MAX_{j=1}^{m_l} ⌈(L_j(i_1,...,i_{k-1}) + l_j)/c_k⌉, MIN_{j=1}^{m_u} ⌊(U_j(i_1,...,i_{k-1}) + u_j)/c_k⌋
|  |  ...
|  |  |  Body
```

The main drawback of this method is the large amount of redundant control since eliminating a variable with the Fourier-Motzkin algorithm may generate up to $n^2/4$ constraints for the loop bounds where $n$ is the initial constraint number. Many of these constraints are obviously redundant. For instance let us consider the polyhedron in Figure 6.1. We can observe that the Fourier-Motzkin projection overestimates the real projection and that this will generate a control overhead. Ancourt and Irigoin presented a redundant constraint elimination method but not adapted to high-dimensionality, geometrically-complex polyhedra [4, 69].

Most further works tried to extend this first technique in order to reduce the redundant control and to deal with more general transformations. Le Fur presented a new redundant constraint elimination policy by using the simplex method [69]. Li and Pingali [78], Xue [110], Darte [40] and Ramanujam [92] relaxed the unimodularity constraint to an invertibility constraint and then proposed to deal with non-unit strides (loop increments can be something different than one). They all use the Hermite Normal Form [97] to find the strides, and the classical Fourier-Motzkin elimination to compute the loop bounds. We saw in chapter 3 that these methods are obsolete thanks to the use of our transformation policy. In addition, Li and Pingali proposed a completion algorithm to build a non-unimodular transformation function from a partial matrix, such as the

Context:

$$n \leq m \leq 2n$$

Polyhedron:

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \\ m \end{pmatrix} \geq \vec{0}$$

Projections:

$$\begin{array}{rcccl} 1 & \leq & i & \leq & min(n, m-1) \\ 1 & \leq & j & \leq & min(n, m-i) \end{array}$$

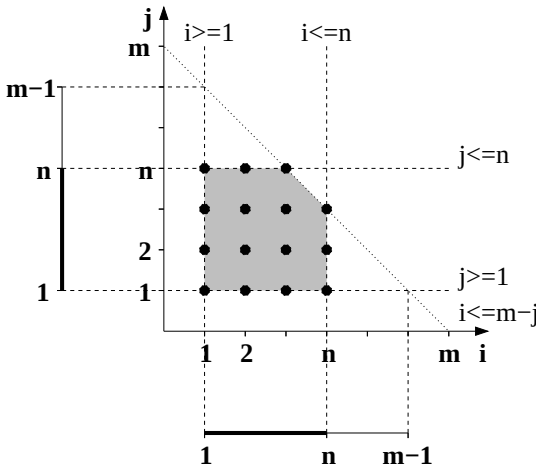


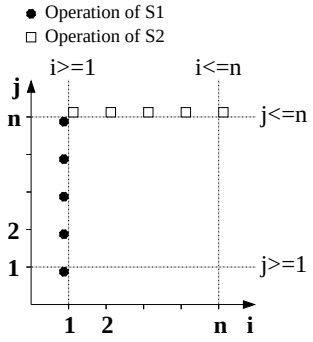

Figure 6.1: Projection onto axis using Fourier-Motzkin elimination method

transformation stay legal for dependences [78]. In the same spirit, Griebl et al. relaxed the invertibility constraint and proposed to deal with arbitrary matrices by using a square invertible extension [55]. We shown here how to deal with general affine transformation functions without constraints on unimodularity, invertibility or even regularity (see chapter 3).

Alternatively to the Fourier-Motzkin elimination method, Collard et al. [37] presented a loop bound calculation technique based on a parameterized version of the dual simplex algorithm [42]. Another method makes successive projections of the polyhedron on the axis as in [5] but use the Chernikova algorithm [70] to work with a polyhedron represented as a set of rays and vertices [71]. These two techniques have the good property of producing a code without any redundant control (for only one polyhedron), but while the second one can generate a very compact code, the first one can quickly explode in length.

The problem of scanning more than one polyhedron in the same code was firstly solved by generating a naive perfectly nested code and then (partially) eliminating redundant guards [63]. Another way was to generate the code for each polyhedron separately, and then to merge them [55, 25]. This solution generates a lot of redundant control, even if there were no redundancies in the separated code. Figure 6.2 shows three well knows implemented code generation policies and their consequences on control overhead for a simple problem of scanning two one-dimensional polyhedra in a 2-dimensional space. The simplest policy is to calculate the bounding box of the polyhedra then to scan each point in the bounding box and verify if a statement has to be executed with guards. The result in Figure 6.2(b) shows both the large amount of control overhead and the problem of eliminating redundant constraints in the code generated by LooPo[1] [55]. Here, for $O(n)$ operations, we have to check $O(n^2)$ points. If the transformation allows to avoid few cache misses, it is clearly not a benefit because of code generation. An interesting improvement is to use the convex hull of the polyhedra instead of the bounding box. We can see in Figure 6.2(c) that it can reduce significantly the generated control overhead, but in our example this is still not satisfactory. The code has been generated by the Omega Code Generator, CodeGen[2] [63]. Quilleré et al. proposed to recursively generate a set of loop nests scanning several unions of polyhedra by separating them into subsets of disjoint polyhedra and generating the

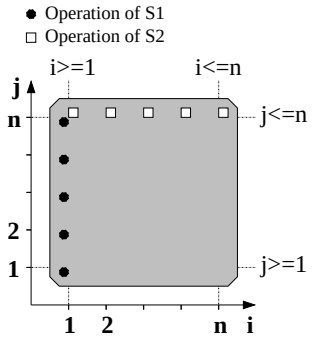
[1] LooPo is available at http://www.infosun.fmi.uni-passau.de/cl/loopo/
[2] CodeGen is available at http://www.cs.umd.edu/projects/omega/




$$\mathcal{D}_{S1}: \left[\begin{array}{cc} 1 & 0 \\ \hline 0 & 1 \\ 0 & -1 \end{array}\right] \begin{pmatrix} i \\ j \end{pmatrix} + \left(\begin{array}{c} -1 \\ \hline -1 \\ n \end{array}\right) \begin{array}{c} = \\ \hline \geq \end{array} \vec{0}$$

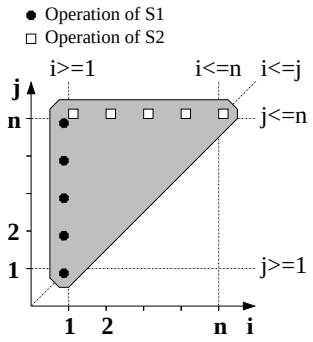
$$\mathcal{D}_{S2}: \left[\begin{array}{cc} 0 & 1 \\ \hline 1 & 0 \\ -1 & 0 \end{array}\right] \begin{pmatrix} i \\ j \end{pmatrix} + \left(\begin{array}{c} -n \\ \hline -1 \\ n \end{array}\right) \begin{array}{c} = \\ \hline \geq \end{array} \vec{0}$$

(a) Original polyhedra to scan



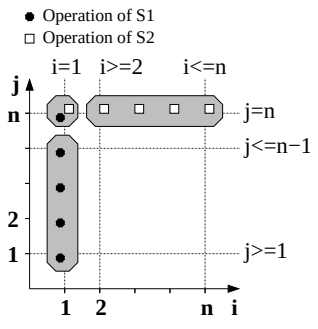

```
do i=1, max(1,n)
| do j=min(1,n), n
| | if (1 <= i .and. i <= n .and. j == n)
| | | S1
| | if (i == 1 .and. 1 <= j .and. j <= n)
| | | S2
```

(b) Bounding box method, code generated by LooPo (without CLooG option)




```
do i=1, n
| do j=i, n
| | if (i <= 1)
| | | S1
| | if (n <= j)
| | | S2
```

(c) Convex hull method, code generated by CodeGen (option effort -2)




```
if (n == 1)
| S1(i=1,j=1)
| S2(i=1,j=n)
if (n >= 2)
| do j=1, n-1
| | S1(i=1)
| S1(i=1,j=n)
| S2(i=1,j=n)
| do i=2, n
| | S2(j=n)
```

(d) Separation method, code generated by CLooG (default behavior)

Figure 6.2: Code generation policies

corresponding loop nests from the outermost to the innermost levels [91]. This later approach provides at present the best solutions since it guarantees that there is no redundant control. This separation strategy is illustrated through Figure 6.2(d), the code generated by CLooG[3] [14, 15], an implementation of this algorithm. However, it suffers from some limitations, e.g. high complexity or needless code explosion. The next chapter presents some solutions to these drawbacks.

## 6.2 Code Generation Aware Transformations

Another way of avoiding control overhead consists in taking care of it when selecting the transformation function. We can consider two kinds of control overhead. First, complex loop bounds with maxima or minima calculation may impose function calls and comparison instructions at each iteration. Second, strides on a given dimension that cannot be translated into loop steps result in costly modulo calculation at each iteration of the corresponding loop. In this section we propose methods to prevent such problems if possible. In section 6.2.1 we show how it is possible to avoid some complex loop bounds at the transformation construction step. Next we show in section 6.2.2 methods to avoid modulo calculations.

### 6.2.1 Avoiding Complex Loop Bounds

A loop bound is said to be complex if it needs the calculation of minima or maxima of affine expressions. We have to distinguish between two cases being more or less worrying. On one hand, min-max calculation between constant values (parameters and scalars) that should be evaluated outside the loops (but some compilers may not succeed in achieving this optimization by themselves...). On the other hand evaluations with expressions including loop iterators that may imply control overhead since they have to be computed as many times as the involved loop counters iterate. This may happen because of the properties of a polyhedron before transformation, i.e. if a dimension is bounded by several inequalities including at least an outer loop counter (e.g. $j \leq i$ and $j \leq m$). Moreover this can be the consequence of a coupled transformation, i.e. if a component of a transformation function is an affine expression including more than one loop counter (e.g. $\theta_S \begin{pmatrix} i \\ j \end{pmatrix} = i+j$), since to compensate the amplitude of the corresponding scheduling dimension, the scanning process on other dimensions have to adapt the number of points to scan with respect to this scheduling dimension. While the first situation is a consequence of the input problem, the second one can be avoided during the transformation construction. In our case, this means that we have to adapt two steps. (1) We should accept after the generalized construction algorithm in Figure 5.4 only the chunking matrices without coupled components. (2) We have to adapt the step 1b of the algorithm of Figure 5.5 in such a way that we remove from the solution space the linear combinations with more than one non-null element. With such a modification, the correction algorithm can only exchange some rows, multiply rows by non-null scalars and set the constant part of the chunking functions. Then to add to a row a linear combination of the other rows is no longer possible and a non-coupled transformation will stay non-coupled after correction. The direct consequence of such a policy is to forbid improving self-temporal locality of references with coupled subscripts. For instance let us consider the program in Figure 6.3(a), there are two references: $a(i+j)$ and $b(j)$. The segment length is $2n$

[3] CLooG is available at http://www.prism.uvsq.fr/~cedb

for the first reference but $n$ for the second one, thus it should be more interesting to improve locality for the first reference. Chunking transformation has been applied for both cases shown in Figure 6.3(b) and Figure 6.3(c), we can observe that on the testing system (an i386 1GHz system with 128KB L1 cache memory) the results for the first cache level correspond to chunking theory, but because of the control overhead the solution with non-coupled transformation is the more interesting for performance (the conclusion may be different when the critical resource is power).

```
do i=1, n
  do j=1, n
S   a(i+j) = a(i+j) + b(j)
```

| L1 Miss | Mflops |
|---|---|
| 113M | 36.7 |

(a) Original program and performance

```
do c=1, n
  do i=1, n
    j = c
S   a(i+j) = a(i+j) + b(j)
```

| L1 Miss | Mflops |
|---|---|
| 56M | 67.1 |

(b) Program optimized for reference $b(j)$: $\theta_S \begin{pmatrix} i \\ j \end{pmatrix} = j$

```
do c=2, 2*n
  do i=max(c-n,1), min(c-1,n)
    j = c - i
S   a(i+j) = a(i+j) + b(j)
```

| L1 Miss | Mflops |
|---|---|
| 31M | 49.7 |

(c) Program optimized for reference $a(i+j)$: $\theta_S \begin{pmatrix} i \\ j \end{pmatrix} = i+j$

Figure 6.3: Influence of complex loop bounds ($n = 30000$ for experiments)

Going further it may be interesting to prevent any minima or maxima calculation. For instance we may want to avoid them when the SCoP we are trying to optimize is inside a deeply nested loop and when at least one parameter vary outside the SCoP (e.g. a parameter is a loop counter of a surrounding loop). In addition to the constraint of non-coupled transformation, we need to avoid to scan polyhedra with incompatible dimension together. Two polyhedra have incompatible dimensions if it is not possible to compare precisely the limit of their bounding box. For instance $\mathcal{D}_1 : \{i \mid i \in \mathbb{Z}, 1 \leq i \leq n\}$ is not compatible with $\mathcal{D}_2 : \{i \mid i \in \mathbb{Z}, 1 \leq i \leq m\}$ except if the context gives a constraint between the bounds (e.g. $m \geq n$). This information can easily be calculated using e.g. Fourier-Motzkin [69], PIP [37] or PolyLib [71]. Using the separation strategy, the bound of the intersection of the two polyhedra will need the computation of a min-max when the dimensions are incompatible. These informations can be summarized in an incompatibility graph where each vertex is labelled with a statement and where there is an edge between two vertices if the iteration domains of the corresponding statements are incompatible.

To avoid min-max calculations, the transformation construction has to take into account the incompatibility graph informations. When two statements are incompatible, the intersection of these polyhedra in the transformation space should be empty. Let us consider two incompatible

statements $S1$ and $S2$, then the condition $\forall \vec{x}_{S1} \in \mathcal{D}_{S1}, \forall \vec{x}_{S2} \in \mathcal{D}_{S2}, \theta_{S2}(\vec{x}_{S2}) - \theta_{S1}(\vec{x}_{S1}) \neq \vec{0}$ ensure the polyhedra intersection is empty. This condition may be split in two inequalities:

$$\forall \vec{x}_{S1} \in \mathcal{D}_{S1}, \forall \vec{x}_{S2} \in \mathcal{D}_{S2},\ \theta_{S2}(\vec{x}_{S2}) - \theta_{S1}(\vec{x}_{S1}) \geq \vec{0}$$
$$\text{or}$$
$$\forall \vec{x}_{S1} \in \mathcal{D}_{S1}, \forall \vec{x}_{S2} \in \mathcal{D}_{S2},\ \theta_{S1}(\vec{x}_{S1}) - \theta_{S2}(\vec{x}_{S2}) \geq \vec{0}$$

Thus for each edge in the graph we have to consider 2 possibilities and at worst we have to solve two times more constraint systems for each incompatibility we want to avoid. Fortunately we can often reduce this number. When there is an edge in the dependence graph corresponding to $S1\delta S2$ (see section 3.3.1) this condition can be simplified in $\theta_{S2}(\vec{x}_{S2}) - \theta_{S1}(\vec{x}_{S1}) \geq \vec{0}$, if $S2\delta S1$ the condition become $\theta_{S1}(\vec{x}_{S1}) - \theta_{S2}(\vec{x}_{S2}) \geq \vec{0}$. Moreover any connected component of the dependence graph may be scanned individually, hence the transformation construction should study them individually if there are incompatibilities between them. On the opposite some min-max calculations may not be avoided; it is useless to consider the condition to bypass them. When there is a cycle in the dependence graph, the condition cannot be satisfied for every statement couples of the cycle and some min-max calculations have to be generated. In the same way the constraint for improving inter-statement group-locality seen in section 5.3 cannot be used with this condition.

Each part of the condition is not a set of affine constraints. Both the transformation matrices and the iteration vectors are unknown, thus components like $\theta_{S1}(\vec{x}_{S1}) = T_{S1}\vec{x}_{S1} + \vec{t}_{S1}$ are non-linear. However the expressions have to be non-negative everywhere in the iteration domains. Thus we can use the affine form of Farkas Lemma in the same way as for building the legal transformation space in section 3.3.2 to state that:

$$T_{S2}\vec{x}_{S2} + \vec{t}_{S2} - (T_{S1}\vec{x}_{S1} + \vec{t}_{S1}) = \lambda_0 + \vec{\lambda}^T \left( \begin{bmatrix} A_{S1} & 0 \\ 0 & A_{S2} \end{bmatrix} \begin{pmatrix} \vec{x}_{S1} \\ \vec{x}_{S2} \end{pmatrix} + \begin{pmatrix} \vec{a}_{S1} \\ \vec{a}_{S2} \end{pmatrix} \right), \lambda_0 \geq 0, \vec{\lambda}^T \geq \vec{0}.$$

Thus as in legal transformation space construction, we can split this equality in as many equalities as there are independent variables and find the new constraints the transformation function has to satisfy in order to avoid min-max calculations.

### 6.2.2 Avoiding Complex Loop Strides

While our transformation framework guarantees that there are no additional *holes* in the target polyhedra because of the transformation functions (see section 3.2), it may happen that some values of some dimensions must not be scanned. This information is explicitly included in the constraint systems defining the polyhedra as equalities of the form $x_1 = sx_2 + n$, where $x_1$ and $x_2$ are entries of the iteration vector, $s$ is a scalar and $n$ a constant (equalities may be more complex after the transformation application but they will always be reduced to such a form after the projection step of every code generation method, see chapter 7). Such equality imposes to the values of $x_2$ to be a coefficient of the $x_1$ values, for instance $j = 2i$ means that $i$ must be even. To avoid scanning integral points that should not be considered, guards including modulo calculation may be inserted, for instance the constraint $j = 2i$ should impose the insertion of a guard `if (mod(i,2)==0)` after the loop $i$. The complexity of modulo calculations are such that they may be the main source of control overhead. Fortunately in simple cases they can be avoided by adjusting the loop step, this solution is widely explored in section 7.2.

A simple brute-force solution to avoid modulo calculation overhead in any case is to restrict iterator coefficients of transformation functions to be one of the three values $-1$, 0 or 1. This condition may be easily inserted when constructing the legal transformation space construction (see section 3.3.2) by adding two constraints per coefficient of the transformation matrix: considering the coefficient $a$, we have to add the constraints $a \geq -1$ and $a \leq 1$. Then the resulting search space will be the legal transformation space without production of guards with modulo calculations during the code generation step.

The previous condition is sufficient but not necessary. Modulo calculations may be avoided thanks to loop steps as explained in section 7.2. Another solution is to post-process the transformation functions to generate a semantically equivalent scheduling that corresponds to a less complex code. A useful method is to add dimensions to remove the modulo calculation. When $s$ iteration domains have to be scanned and for $1 \leq i \leq s$ the $i^{th}$ constraint system has an equality of the form $x_1 = sx_3 + n_i$ such that the $n_i$ are a succession of $s$ integers, then there are eventually no *hole* in the $x_1$ dimension since for each value (in the domain) we have to scan one and only one integral point of a given polyhedron. In this situation all the $n_i$ give the ordering between the corresponding statements onto $x_1$ then we may add a new dimension $x_2$ after $x_1$ to represent this ordering and replace the original equality by two equalities: $x_1 = x_3$ and $x_2 = n_i$. For instance let us consider two equal iteration domains $1 \leq i \leq n$ for two statements $S1$, $S2$ and the scheduling functions $\theta_{S1}(i) = (2i)$ and $\theta_{S2}(i) = (2i + 1)$. The resulting code is shown in Figure 6.4(a), we can observe that each iteration will only execute one statement instance. We can translate these scheduling functions to $\theta_{S1}(i) = (i, 0)$ and $\theta_{S2}(i) = (i, 1)$ to achieve a semantically equivalent code shown in Figure 6.4(b) where the statements are well ordered in the loop and the control overhead is removed.

```
do t=2, 2*n+1
| if(mod(t-1,2)==0)
|   | S2(i=(c1-1)/2)
| if(mod(t,2)==0)
|   | S1(i=c1/2)
```

(a) $\theta_{S1}(i) = (2i)$ and $\theta_{S2}(i) = (2i + 1)$

```
do t1=1, n
| t2 = 0
| S1(i=t1)
| t2 = 1
| S2(i=t1)
```

(b) $\theta_{S1}(i) = \begin{pmatrix} i \\ 0 \end{pmatrix}$ and $\theta_{S2}(i) = \begin{pmatrix} i \\ 1 \end{pmatrix}$

Figure 6.4: Equivalent codes with different scheduling functions

## 6.3 Exploiting Degrees of Freedom

The polyhedron scanning orders specified by the scheduling functions may leave some dimensions unspecified. This means that the code generator is free to choose their scanning order. Basically, this can happen when the operations are parallel onto these dimensions, and when there is no dependences between the considered statements. Then it is the code generator responsibility to provide the best target code i.e. with the minimum control overhead. This work is essential since it concerns the innermost loops of the generated code, where the consequences of a bad control management are the most disturbing. For instance let us consider the matrix multiply codes in figure 6.5. These target codes can result from a generation where the scheduling functions are $\theta_{S_1}(\vec{x}_{S_1}) = \vec{c}$ and $\theta_{S_2}(\vec{x}_{S_2}) = \vec{c}$ (this is possible since the computations on remaining dimensions

are fully parallel[4]).

```
do c=1, n
| do i=1, n
| | do j=1, n
| | | c1(i,j) = c1(i,j) + a1(i,c)*b1(c,j)
| do i=1, n
| | do j=1, n
| | | c2(i,j) = c2(i,j) + a2(i,c)*b2(c,j)
```

iterations: $2n^3 + 2n^2 + n$

(a) split version

```
do c=1, n
| do i=1, n
| | do j=1, n
| | | c1(i,j) = c1(i,j) + a1(i,c)*b1(c,j)
| | do j=1, n
| | | c2(i,j) = c2(i,j) + a2(i,c)*b2(c,j)
```

iterations: $2n^3 + n^2 + n$

(b) semi-split version

```
do c=1, n
| do i=1, n
| | do j=1, n
| | | c1(i,j) = c1(i,j) + a1(i,c)*b1(c,j)
| | | c2(i,j) = c2(i,j) + a2(i,c)*b2(c,j)
```

iterations: $n^3 + n^2 + n$

(c) merged version

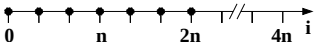
Figure 6.5: Equivalent target codes for matrix multiplies

We have tested these simple codes on a x86 architecture at 1 GHz, compiled with the GCC 3.2.2 compiler and the option -O3. For $n = 500$ the code in figure 6.5(a) takes $5.31s$ while the code in figure 6.5(c) takes $4.75s$, a 15% improvement. The code in figure 6.5(c) can be generated by performing the separation strategy for every free dimensions of the polyhedra. After each separation, if some polyhedra are fully scanned, the corresponding statement bodies have to be printed out. A new list with the remaining polyhedra is created to continue the separation.

Unfortunately this solution is only partial, since it allows us to reduce the control overhead

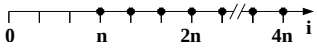
[4]Notice that this is a consideration for the automatic parallelizer or optimizer, not for the code generator: it only knows that it has to do its best to avoid control overhead when the scheduling offers some opportunities.

only according to the original lexicographic order. For instance, let us consider the two polyhedra in figures 6.6(a) and 6.6(b) and the fact that there is no scheduling function (this can be a loop body generation with no more scheduling constraints to respect). The generated code for scanning the two polyhedra in figure 6.6(c) is not optimal since there are iterations used only for $S_1$ and only for $S_2$.




$$\begin{bmatrix} 1 \\ -1 \end{bmatrix} ( \ i \ ) \geq \begin{pmatrix} 0 \\ -2n \end{pmatrix}$$

(a) first polyhedron




$$\begin{bmatrix} 1 \\ -1 \end{bmatrix} ( \ i \ ) \geq \begin{pmatrix} n \\ -4n \end{pmatrix}$$

(b) second polyhedron

```
do i=1, n-1
| S1
do i=n, 2*n
| S1
| S2
do i=2*n+1, 4*n
| S2
```

(c) scanning code

Figure 6.6: Suboptimal code generation for a free dimension

The compulsory control for a polyhedron scanning problem without scheduling constraints is the number of iterations necessary to scan the biggest polyhedron in term of integral points. The control is minimum when it is limited to this compulsory control. We are free to modify the polyhedra scanning orders to achieve this goal. Formally, the general problem is to find for each polyhedron defined by $A_S\vec{x}_S + \vec{a}_S \geq \vec{0}$ an invertible transformation matrix $Z_S$ and a translation vector $\vec{z}_S$ such that the number of integral points in the intersection of all the polyhedra $\cap_S(A_S Z_S \vec{x}_S + \vec{a}_S + \vec{z}_S \geq \vec{0})$ is maximum. Because of the $Z^S$ matrices, the general problem is not affine. Then we simplify it by considering only the translation vector $\vec{z}_S$.

Counting integer points inside a parameterized polyhedron is possible using Ehrhart polynomials [32]. These polynomials can have periodic coefficients when a vertex of the polyhedron is not integral. In a code generation framework this case is rare, but when this happens we are not able to use this method for our purpose. Let us continue the example in figure 6.6, we can consider that the first polyhedron is fixed in space and the other one can move. The intersection of the two polyhedra is defined by $\mathcal{D} = \{i \in \mathbb{Z} | 1 \leq i \leq 2n \wedge n + z \leq i \leq 4n + z\}$. The result of the computation of the Ehrhart polynomial $EP(i)$ given by PolyLib is three adjacent domains of the parameter values, each being associated to an Ehrhart polynomial $EP_1(i)$, $EP_2(i)$ and $EP_3(i)$:

- $EP_1(i) = n - z + 1$, when $-n \leq z \leq n$.
- $EP_2(i) = 2n + 1$, when $-2n \leq z \leq -n$.
- $EP_3(i) = 4n + z + 1$, when $-4n \leq z \leq -2n$.

It is not hard to find that the maximum number of integral points is $2n+1$, given by every $z$ such as $-2n \leq z \leq -n$. Let us choose $z = -n$, apply this translation onto the second polyhedron, and set the corresponding offset $i = i - z$ in the statement body. Eventually, we can generate one of the optimal scanning code in control as shown in figure 6.7.

```
do i=1, 2*n
| S1
| S2(i->(i+n))
do i=2*n+1, 3*n
| S2(i->(i+n))
```

Figure 6.7: An optimal version of the scanning code in figure 6.6(c)

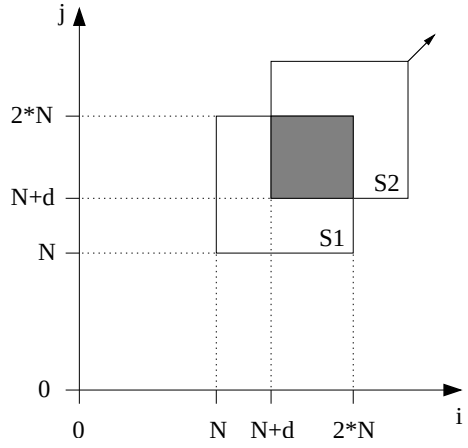
This technique is guaranteed to reduce the control overhead. At worst, it will leave the original polyhedra intact. Using $Z_S$ to find new solutions to the polyhedron overlapping problem is left for future work. Nevertheless a simple transformation that may help is to interchange dimensions. This transformation do not generate overhead when updating the iterators in the statement bodies since the only modification is the interchange between the considered dimension iterators (for instance, moving a polyhedra by $n$ on the dimension $i$ results in updating $i$ with $i - n$ while interchanging $i$ with e.g. $j$ do not generate new calculation). Then we can try the previous transformation for maximizing the intersection for every possible interchange and find the solution with the maximum overlapping. On a small number of polyhedra this method can speedily achieve impressive results, for instance the two polyhedra in Figure 6.2 can be obviously scanned using a single loop if their scheduling are free.

**Discussion on predicated instruction architectures** Improving control overhead may seem not to be useful for architectures with predicated instruction like Itanium. In these architectures, every operation is executed while the control statements are calculated in parallel. Then the result of the control calculation will validate or not the instruction thanks to its predicate. Nevertheless, maximizing the intersection of polyhedra on architecture with explicitly parallel instruction sets (EPIC) as Itanium has the side-effect to maximize the operation number at the innermost loop level. Hence, the compiler work for maximizing the use of the bundles is highly simplified. This improvement is *not* equivalent to loop unrolling since first, loop unrolling is typically avoided because it heavily increases the complexity of further processing and second, loop unrolling adds only the same kind of instructions while maximizing polyhedra overlapping may concern very different instructions and thus a better use of the functional units.

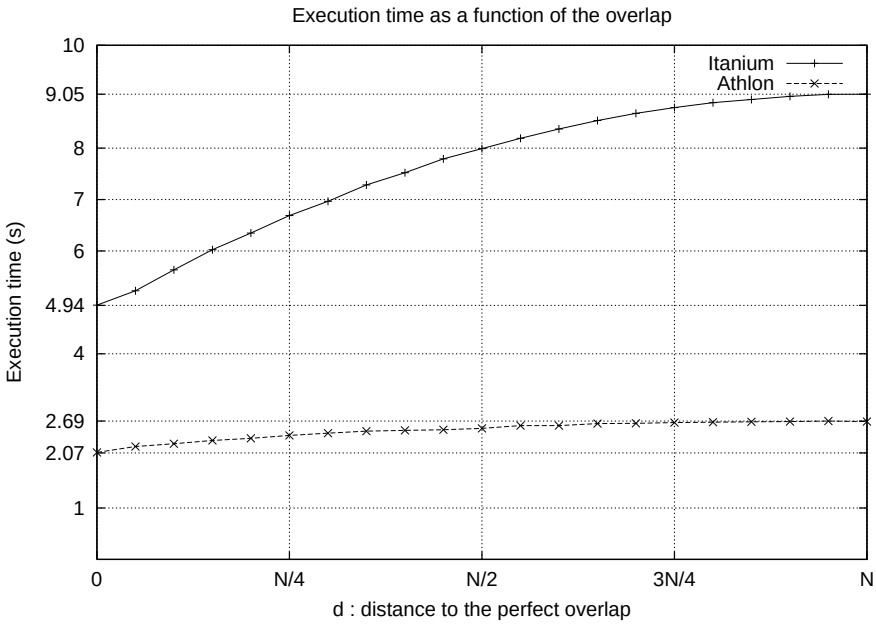
As an illustration let us consider the problem in Figure 6.8(a); we decreased the overlap of two iteration domains of two statements achieving independent computations[5] and we measured the execution time as shown in Figure 6.8(b). Our target machines, compilers and compiler options were an Itanium I 1GHz with Intel compiler ECC 5.0.1 called with -O3 and -restrict options for the EPIC target, and an Athlon 2Ghz with GCC 3.3.2 called with -O3 option for the non-EPIC target. Our results show that, not surprisingly, improving control overhead may achieve better performance on classical architectures: we measured a 30% improvement between null and total overlapping. Results are even more impressive on Itanium since we achieve a 83% improvement with total overlapping. We checked the assembly code to conclude that this improvement is the direct consequence of a better use of the Itanium bundles, i.e. instruction-level parallelism.

[5]for our experiments, the parameter $N$ was set to 500, we repeated each computation 400 times to achieve significative execution times and the exact statements were:
$S1$: `A(i-N,j-N) = A(i-N,j-N) * (B(i-N,j-N) + cst)`
$S2$: `C(i-(N+d),j-(N+d)) = C(i-(N+d),j-(N+d)) * (D(i-(N+d),j-(N+d)) + cst)`

```
do i=N, N+d-1
|  do j=N, 2*N-1
|  |  S1
do i=N+d, 2*N-1
|  do j=N, N+d-1
|  |  S1
|  do j=N+d, 2*N-1
|  |  S1
|  |  S2
|  do j=2*N, 2*N+d-1
|  |  S2
do i=2*N, 2*N+d-1
|  do j=N+d, 2*N+d-1
|  |  S2
```




(a) Scanning code for the two iteration domains




(b) Execution time with regard to the overlapping area

Figure 6.8: Checking overlapping impact on EPIC and non-EPIC architectures

## 6.4 Conclusion

The strength of our polyhedral framework is to provide a convenient model to apply easily any kind of statement-wise affine transformations as described in chapter 3. Once these transformations have been applied in the polyhedral domain, generating the target code that actually implements the transformation boils down to a polyhedron scanning problem. This chapter presented this problem and the strategies that have been proposed to solve it. We showed how the final code efficiency depends on the properties of the transformation for code generation purpose. We proposed ways to ensure that the target code will not suffer from a high control overhead, from removing transformations in the search space when they are proven to lead to an inefficient code, to post-processing the scheduling functions. We proposed to modify them for improving the generated code in two directions: (1) to remove modulo calculations and (2) to optimize control sharing by maximizing polyhedra overlapping. Once the transformations have been carefully constructed or improved for code generation purpose, we have to build the target code. The most powerful state-of-the-art technique to achieve this task is known as the Quilleré et al. algorithm [91], this method will be presented and extended in depth in the next chapter.

# Chapter 7

# Extended Quilleré et al. Algorithm

Code generation is the last step to the final program. It is often ignored in spite of its impact on the target code quality. In particular, we must ensure that a bad control management does not spoil performance, for instance by producing redundant guards or complex loop bounds. At present, the Quilleré et al. method give the best results when we have to generate a scanning code for several polyhedra [91]. This technique is guaranteed to avoid redundant control while scanning the scheduling dimensions. However, it suffers from several limitations, e.g. code generation with unit strides only, needless code explosion and high complexity. In this chapter, we will show that, starting from one of the best algorithms known so far [91], we can generalize and improve it in two directions:

- The quality of the target code is such that many loop nests in familiar benchmarks can be regenerated with the least possible control overhead.
- The implementation is efficient enough to handle problems with thousands of statements and tens of free parameters.

The chapter is organized as follows. We present the general algorithm with many adaptations to our purpose in section 7.1. We show how it is possible to deal efficiently with non unit strides in section 7.2. We address the problem of reducing the code size without consequence on code efficiency in section 7.3, then section 7.4 discusses complexity issues and solutions. In section 7.5, experimental results obtained through the algorithm implementation are shown. Lastly section 7.6 summarizes our work on code generation.

## 7.1 Extended Quilleré et al. Algorithm

Quilleré et al. proposed recently the first code generation algorithm building the target code without redundant control directly instead of starting from a naive code and trying to improve it [91]. As a consequence, this method never fails to remove a guard and the processing is easier. Eventually it generates a better code more efficiently. The algorithm rely on polyhedral operations that can be implemented by e.g. PolyLib[1] [105]. The basic mechanism is, starting from the list of polyhedra to scan, to recursively generate each level of the abstract syntax

[1]PolyLib is available at http://icps.u-strasbg.fr/PolyLib

tree of the scanning code (AST). The nodes of the AST are labelled with a polyhedron $\mathcal{T}$ and have a list of children (notation $\mathcal{T} \rightarrow (...)$). The leaves are labelled with a polyhedron and a statement (notation $\mathcal{T}_S$). Each recursion builds an AST node list as described by the algorithm in Figure 7.1. It starts with the following input:

1. the list of transformed polyhedra to be scanned $(\mathcal{T}_{S_1}, ..., \mathcal{T}_{S_n})$;
2. the context, i.e. the set of constraints on the global parameters;
3. the first dimension $d = 1$.

Generating the code from the AST is a trivial step: the constraint system labelling each node can be directly translated as loop bounds for the constraints concerning the dimension corresponding to the node level, and as surrounding conditional for the other constraints.

This algorithm is somewhat different from the one presented by Quilleré et al. in [91] and its improved version in [16]; our main contributions are the following:

- To take advantage of free scheduling (see section 6.3). The code in Figure 6.5(c) can be generated by performing the Quilleré et al. recursion for every free dimensions of the polyhedra. After each recursion, if some polyhedra are fully scanned, the corresponding statement bodies have to be printed out. A new list with the remaining polyhedra is created to continue the recursion. This is the aim of step 5b of the algorithm.
- To handle non-unit strides (step 5a and section 7.2).
- To reduce the code size without degrading code performance (step 7 and section 7.3).
- To reduce the code generation processing time by using pattern matching (step 3 and section 7.4).

Let us describe this algorithm with a non-trivial example. We propose to scan the two polyhedral domains presented in Figure 7.2(a). The iteration vector is $(i, j, k)$ and the parameter vector is $(n)$. We first compute intersections with the context (i.e. at this point, the constraints on the parameters, supposed to be $n \geq 6$). We project the polyhedra onto the first dimension, $i$, then we separate them into disjoint polyhedra. This means that we compute the domains where there are points to scan for $\mathcal{T}_{S_1}$ alone, both $\mathcal{T}_{S_1}$ and $\mathcal{T}_{S_2}$, and $\mathcal{T}_{S_2}$ alone (as shown in Figure 7.2(b), this last domain is empty). Here, we notice there is a constraint on an inner dimension implying a non-unit stride; we can determine this stride and update the lower bound. We finally generate the scanning code for this first dimension. We now recurse on the next dimension, repeating the process for each polyhedron list (in this example, there are now two lists: one inside each generated outer loop). We intersect each polyhedron with the new context, now the outer loop iteration domains; then we project the resulting polyhedra onto the outer dimensions, and finally we separate these projections into disjoint polyhedra. This last process is trivial for the second list but yields several domains for the first list, as shown in Figure 7.2(c). Eventually, we generate the code associated with the new dimension, and since this is the last one, the scanning code is fully generated.

**CODEGENERATION**: Build a polyhedra scanning code without redundant control AST.

**Input:** a polyhedron list $(\mathcal{T}_{S_1}, ..., \mathcal{T}_{S_n})$, a context $C$, the current dimension $d$.
**Output:** the abstract syntax tree of the code scanning the polyhedra in the input list.

1. Intersect each polyhedron $\mathcal{T}_{S_i}$ in the list with the context $C$ in order to restrict the domain (and subsequently the code that will be generated) under the context of the surrounding loop nest.

2. Compute for each resulting polyhedron $\mathcal{T}_{S_i}$ its projection $\mathcal{P}_i$ onto the outermost `d` dimensions and consider the new list of $\mathcal{P}_i \rightarrow \mathcal{T}_{S_i}$.

3. Separate the projections into a new list of disjoint polyhedra: given a list of $m$ polyhedra, start with the first two polyhedra $\mathcal{P}_1 \rightarrow \mathcal{T}_{S_1}$ and $\mathcal{P}_2 \rightarrow \mathcal{T}_{S_2}$ by computing $(\mathcal{P}_1 - \mathcal{P}_2) \rightarrow \mathcal{T}_{S_1}$ (i.e. $S_1$ alone), $(\mathcal{P}_1 \cap \mathcal{P}_2) \rightarrow (\mathcal{T}_{S_1}, \mathcal{T}_{S_2})$ (i.e. $S_1$ and $S_2$) and $(\mathcal{P}_2 - \mathcal{P}_1) \rightarrow \mathcal{T}_{S_2}$ (i.e. $S_2$ alone), then for the three resulting polyhedra, make the same separation with $\mathcal{P}_3 \rightarrow \mathcal{T}_{S_3}$ and so on.

4. Build the lexicographic ordering graph where there is an edge from a polyhedron $\mathcal{P}_1 \rightarrow (\mathcal{T}_{S_p}, ..., \mathcal{T}_{S_q})$ to another polyhedron $\mathcal{P}_2 \rightarrow (\mathcal{T}_{S_v}, ..., \mathcal{T}_{S_w})$ if its scanning code has to precede the other to respect the lexicographic order, then sort the list according to a valid order.

5. For each polyhedron $\mathcal{P} \rightarrow (\mathcal{T}_{S_p}, ..., \mathcal{T}_{S_q})$ in the list:

   (a) Compute the stride that the inner dimensions impose to the current one, and find the lower bound by looking for stride constraints in the $(\mathcal{T}_{S_p}, ..., \mathcal{T}_{S_q})$ list.

   (b) While there is a polyhedron in $(\mathcal{T}_{S_p}, ..., \mathcal{T}_{S_q})$:

      i. Merge successive polyhedra with another dimension to scan in a new list.

      ii. Recurse for the new list with the new loop context $C \cap \mathcal{P}$ and the next dimension $d + 1$.

6. For each polyhedron $\mathcal{P} \rightarrow$ (inside) in the list, apply steps 2 to 4 of the algorithm to the *inside* list in order to remove dead code. Then consider the concatenation of the resulting lists as the new list.

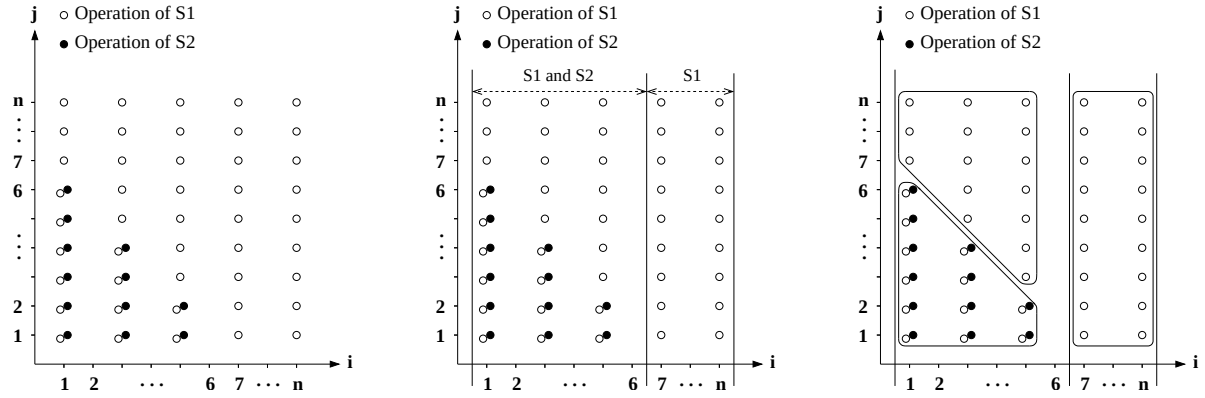
7. Make all the possible unions of *host* polyhedra with *point* polyhedra to reduce code size.

8. Return the polyhedron list.

Figure 7.1: Extended Quilleré et al. Algorithm

## 7.2 Non-Unit Strides

To scan a transformed polyhedron it may be necessary to avoid some values in some dimensions. This happens when there exists a set of dimensions such that the transformed polyhedron projection onto these dimensions has integer points without a corresponding preimage in the original space.




$$\mathcal{T}_{S_1} : \begin{cases} i = 2k+1 \\ 1 \le i \le n \\ 1 \le j \le n \end{cases}$$

$$\mathcal{T}_{S_2} : \begin{cases} i = 2k+1 \\ 1 \le i \le 6 \\ 1 \le j \le 7-i \end{cases}$$

(a) Initial domains to scan

```
do i=1, 6, 2
| T_S1 : {1 ≤ j ≤ n}
| T_S2 : {1 ≤ j ≤ 7 − i}
do i=7, n, 2
| T_S1 : {1 ≤ j ≤ n}
```

(b) Projection and separation on the first dimension

```
do i=1, 6, 2
| do j=1, 7-i
| | S1
| | S2
| do j=8-i, n
| | S1
do i=7, n, 2
| do j=1, n
| | S1
```

(c) Recursion on next dimension

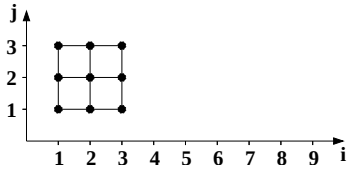
Figure 7.2: Step by step code generation example

Previous works were challenged by this problem, which occurs when the transformation function is non-unimodular (i.e. the transformation matrix $T$ has non unit determinant). We can observe the phenomenon with the transformation of the polyhedron in Figure 7.3(a) by the function $\theta \begin{pmatrix} i \\ j \end{pmatrix} = i+2j$ (the corresponding transformation matrix $T = \begin{bmatrix} 1 & 2 \end{bmatrix}$ is not invertible, but it can be extended[2] to $T = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}$). The target polyhedron is shown in Figure 7.3(b). The integer points without dots have no images in the original polyhedron. The original coordinates can be determined from the target ones by $\vec{original} = T^{-1}\vec{target}$. Because $T$ is non-unimodular, $T^{-1}$ has rational elements. Thus some integer target points have a rational image in the original space; they are called *holes*. We show in section 3.2.1 that in order to avoid scanning the holes, the loop strides (the steps the iterators make at the end of the loop body) and the loop lower bounds had to be found. Previous works used the Hermite Normal Form [97] in different ways to solve the problem (see section 3.2.1).

We do not change the basis of the original polyhedra, but we only change their scanning order as discussed in section 3.2.1. As a consequence, our target systems are always integral and there are no holes in the corresponding polyhedra. As an illustration, the target polyhedron given

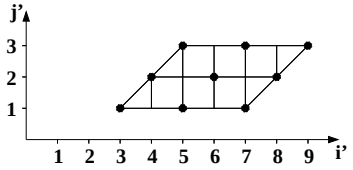
[2]Obviously we could also use the extension $\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ which is unimodular. The example is chosen for educational purpose. We may ask directly for the 2-dimensional non-unimodular scheduling but it would lead to a 4-dimensional polyhedron with our transformation scheme.

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



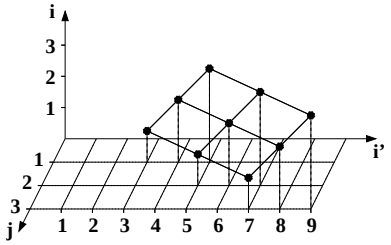

(a) original polyhedron $A\vec{x} + \vec{a} \geq \vec{0}$

$$\begin{bmatrix} -1/2 & 1/2 \\ 1/2 & -1/2 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} -1 \\ 3 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0}$$




(b) usual transformation $(AT^{-1})\vec{y} + \vec{a} \geq \vec{0}$

$$\left[\begin{array}{c|cc} 1 & -1 & -2 \\ \hline 0 & 1 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{array}\right] \left(\begin{array}{c} i' \\ \hline i \\ j \end{array}\right) + \left(\begin{array}{c} 0 \\ \hline -1 \\ 3 \\ -1 \\ 3 \end{array}\right) \begin{array}{c} = \\ \hline \geq \end{array} \vec{0}$$




(c) our transformation $\mathcal{T}$

Figure 7.3: Non-unimodular transformation with $\theta(i,j) = i + 2j$

by our transformation policy is shown in Figure 7.3(c). The stride informations are explicitly contained in the constraint systems thanks to the equalities. Without the step 5a in the code generation algorithm, the successive projections lead to the equality constraints, as shown in Figure 7.4(a) for our example. This leads to an inefficient scanning code since heavy guards (with modulo operations) are inside the loops.

We solve this problem by finding the stride for the current dimension $x$ and the new lower bound, i.e. the first value in the dimension with an integral point to scan (for instance if we want to scan only odd or even points onto a dimension, the stride is the same but we need to start the loop from a convenient odd or even point). The stride can be directly read from the equality constraints $y_S = s_S x_S + n_S$ of the transformed polyhedra, where for the polyhedron $S$, $y_S$ is the striding dimension (the dimension whose meaning is to state that the values of the dimension $x_S$ that have to be considered for scanning are integral coefficients of the values of $y_S$), $x_S$ is the strided dimension (the dimension where some integral values have not to be considered for scanning), $s_S$ is the stride (the size of the step between two integral points that have to be scanned) and $n_S$ is a merging of the parameters (outer dimensions, structure parameters and constant). The lower bound has to be found under the context of the outermost loop counters and the structure parameters. This is a problem in parametric integer programming, that can be solved by PipLib [42]. For our example, the stride of the $i$ loop is 2, directly given by the equality constraint, and the lower bound is the minimum value of $i$ in the polyhedron defined by

$\begin{cases} 1 \leq i \\ i' - 6 \leq i \\ i = i' - 2j \end{cases}$ and under the context $3 \leq i' \leq 9$. The solution given by PipLib as a quasi affine selection tree is shown in the improved code in Figure 7.4(b). The final optimized code can be generated after backtracking the guard in the algorithm and is shown in Figure 7.4(c).

```
do i'=3, 9
| do i=max(1,i'-6), min(i'-2,3)
| | if (mod(i'-i,2) == 0)
| | | j=(i'-i)/2
| | | S1
```

(a) guarded version

```
do i'=3, 9
| if (i'<=7)
| | lower = i' - 2*((i'+1)/2) + 2
| else
| | lower = i' - 6
|
| do i=lower, min(i'-2,3), 2
| | j=(i'-i)/2
| | S1
```

(b) strided version

```
do i'=3, 7
| do i=i'-2*((i'+1)/2)+2, min(i'-2,3), 2
| | j=(i'-i)/2
| | S1
do i'=8, 9
| do i=i'-6, min(i'-2,3), 2
| | j=(i'-i)/2
| | S1
```

(c) backtracked version

Figure 7.4: Scanning codes for the polyhedron in Figure 7.3(c)

When there is a set of polyhedra $K$ to scan, for each dimension $x_S$ we have to consider a set of striding constraints $y_S = s_S x_S + n_S$. Like in [63], the greatest common step $gcs$ is given by:

$$gcs = gcd(\{s_{S_i} | S_i \in K\}, \{n_{S_i} - n_{S_j} | S_i, S_j \in K \wedge i \neq j\}).$$

The meaning of the GCD of $s_{S_i}$ is obvious: in a given loop we can only consider a common step between the different statements, and the greatest one will give the best performance by avoiding the greatest number of guards. The GCD of the $n_{S_i} - n_{S_j}$ is connected to the first integral point to consider for each statement: since the step will occur after the first iteration and will bypass some integral points we must ensure that the first valid point of every statement will be scanned.

Then we have to find the lower bound, i.e. the first value of $x_S$ where an integer point has to be scanned. This can be achieved in the same way as for one polyhedron by merging all the constraints of the different polyhedra for the dimension $x_S$. We have successfully implemented this method with the restriction that the $n_{S_i} - n_{S_j}$ have to be constants. The problem to find the *gcs* when the $n_{S_i} - n_{S_j}$ are parameterized is under investigation.

## 7.3 Reducing Code Size

The power of optimizing methods in the polyhedral model are of a particular interest for embedded system compiling. One of the main constraint for such applications is the object code size because of inherent hardware limitations. Generated code size may be under control for this purpose or simply to avoid instruction cache pollution. It is possible to manage it easily with iterative code generation methods [63]: they start from a naive (inefficient) and short code and eliminate the control overhead by selecting conditions to remove and performing code hoisting (splitting the code on the chosen condition and copying the original guarded code in the two branches). Thus, to stop code hoisting stops code growing. With recursive code generation methods as discussed in this paper, it is always possible to choose not to separate the polyhedra and to generate a smaller code with conditions [91]. These techniques always operate at the price of a less efficient generated code. This section presents another way, with a quite small impact on control overhead and a possibly significant code size improvement. It is based on a simple observation: separating polyhedra often results in isolating some points, while this is not always necessary. Figure 7.2 shows a dramatic example of this phenomenon (hoisting-based code generators as the Omega CodeGen have to meet the same issue). Integrating these points inside *host* loops when possible will reduce the code size by adding new iterations. The problem was first pointed out by Bouchebaba in the particular case of 2-dimensional loop nest fusion [24]. He extracted the 14 situations where a vertex should not be fused with a loop for his purpose and apply the fusion in the other cases. In the following is presented a solution for general code generation based on the properties of the code construction algorithm in Figure 7.1.

To ensure that the separation step will not result in needless polyhedron peeling, it is necessary to compute this separation. In addition we have to achieve the recursion on every dimensions since the projection hide some of them during the separation process. Thus, we can remove isolated points at the end of each recursion (step 7). At a given depth of the recursion, the removing process is applied for each *loop* node in the list (i.e. such that the dimension corresponding to the current depth is not constant):

1. Define the point candidate to merge with the node: scan the node branch in depth first order and build the list of statements in the leaves. The statement candidate has to fit this statement list since it is guaranteed after dead code elimination that each statement in the leaves is executed at least once. Thus only a point with this structure may be merged with the node.

2. Check if such a point directly precedes or follows the node in the lexicographic ordering graph built in step 4 and 6 (details on this graph construction can be found in [91]). This graph is only based on the projecting dimensions, however if a point candidate directly follows the node in the ordering graph and cannot be merged, this means that an input polyhedron is not convex, a contradiction.

3. Merge the point candidates with the node if the previous test was a success by using a polyhedral *union*, and remove the points from the list of polyhedra.

We propose to illustrate this algorithm through the example in Figure 7.2. We have to generate the scanning code for the three polyhedra in Figure 7.5(a). For the sake of simplicity, we will show directly the translations of the node constraint systems into source code. We first compute the intersections with the context (i.e., at this point, the constraints on the parameters, supposed to be $n \geq 2$ and $m \geq n$). We project the polyhedra onto the first dimension, $i$, then we separate them into disjoint polyhedra. As shown in Figure 7.5(b) this results in two disjoint polyhedra. Thus we generate the scanning code for this first dimension. Then we recurse on the next dimension, repeating the process for each polyhedron list (in this example, there are now two lists: one inside each generated outer loop). We intersect each polyhedron with the new context, now the outer loop iteration domains; then we project the resulting polyhedra on the outer dimensions, and finally we separate these projections into disjoint polyhedra. This last process is trivial for the second list but yields several domains for the first list, as shown in Figure 7.5(c). Then we generate the code associated with the new dimension, and since this is the last one, a scanning code is fully generated. Lastly, we remove dead code (for instance in the first loop nest in Figure 7.5(c), the iteration $i = n$ is useful only for a small part of the loop body) by applying a new projection step during the recursion backtrack. The final code is shown in Figure 7.5(d).

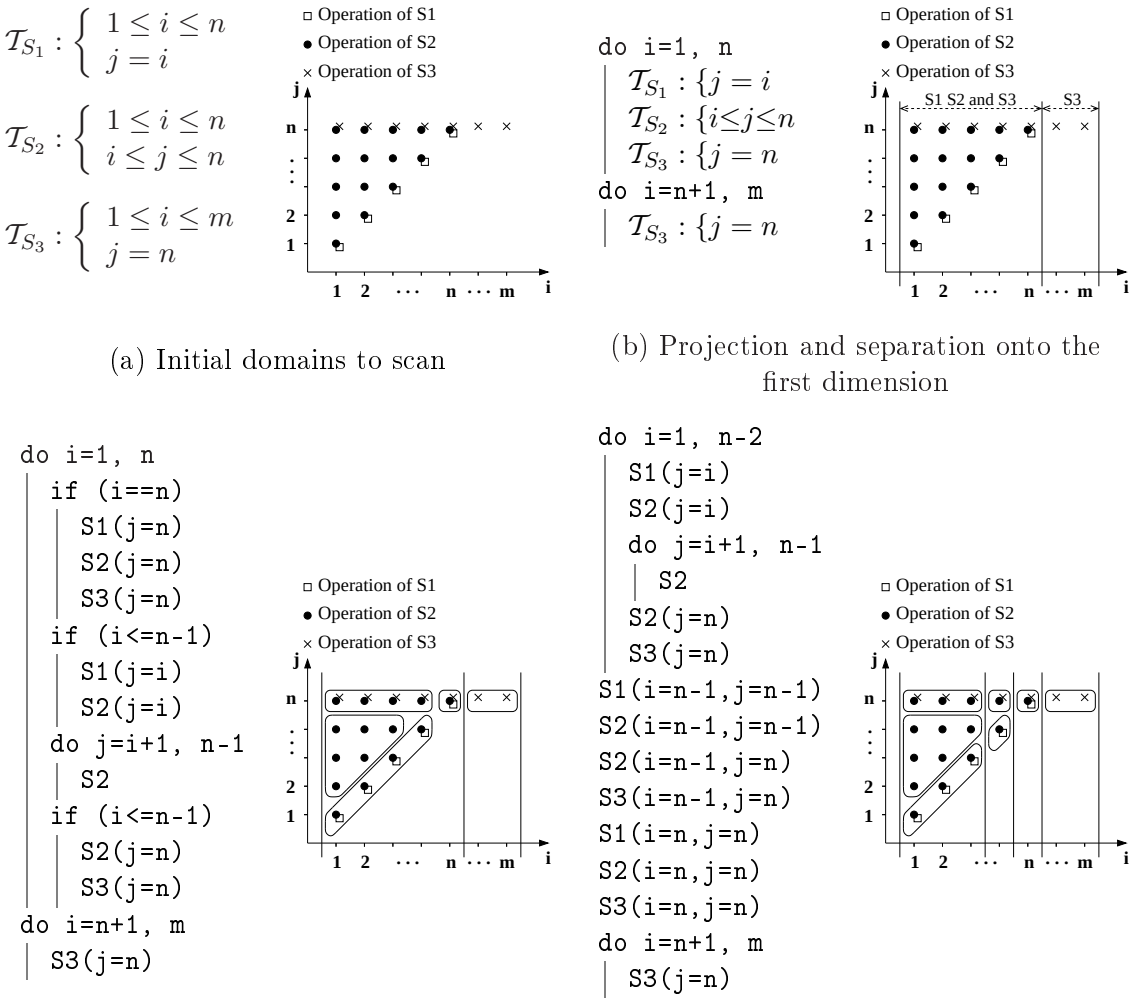
We can apply the code size reduction process to this example. The translation of the AST after dead code removing for the dimension $j$ is equivalent to the code in Figure 7.5(c). The statement candidate for the $j$ loop is $S_2$. We can merge both $S_2$ points before and after this loop. Then the dead code removing for dimension $i$ would only isolate the point corresponding to $i = n$, the new candidate would be $S_1S_2S_3$. It can be merged and the final code is shown in Figure 7.6 with an object code size of 176B while the previous one in Figure 7.5(d) is 464B (each statement is a 2-dimensional array entry increment).

## 7.4 Complexity Issues

The main computing kernel in the code generation process is the separation into disjoint polyhedra (step 3). Given a list of $n$ polyhedra, the worst-case complexity is $\mathcal{O}(3^n)$ polyhedral operations (exponential themselves). In addition, the memory usage is very high since we have to allocate memory for each separated domain. For both issues, we propose a partial solution.

We use pattern matching to reduce the number of polyhedral computations: at a given depth, the domains are often the same (this is a property of the input codes, this happens for 17% of the operations in the benchmark set presented in section 7.5), or disjoint (this is a property of the scheduling matrices, this happens for 36% of the operations in the benchmark set of section 7.5). Thus we check quickly for these properties before any polyhedral operation by comparing directly the elements of the constraint systems (this allows to find 75% of the equalities), and by comparing the unknowns having fixed values (this allows to find 94% of the disjunctions). When one of these properties is proved, we can directly give the trivial solution to the operation. This method improves performance by a factor near to 2.

To avoid a memory allocation explosion, when we detect a high memory consumption, we continue the code generation process for the remaining recursions with a more naive algorithm,




(a) Initial domains to scan

(b) Projection and separation onto the first dimension

(c) Recursion on next dimension

(d) Backtrack with dead code removing

Figure 7.5: Code size explosion example

leading to a less efficient code but using far less memory. Instead of separating the projections into disjoint polyhedra (step 3 of the algorithm), we merge them when their intersections are not empty. Then we work with a set of unions, significantly smaller than a set of disjoint polyhedra. Other parts of the algorithm are left unmodified. The drawback of this method is the generation of costly conditionals ruling whether an integral point has to be scanned or not. This method can be compared to using the convex hull of the polyhedra [63, 104, 55, 25], but is more general since it can deal with complex parameterized bounds (typically maximum or minimum of parameterized affine constraints e.g. $max(m, n)$) that do not describe a convex polyhedron.

## 7.5 Experimental Results

We implemented this algorithm and integrated it into a complete polyhedral transformation infrastructure inside Open64/ORC. Such a modern compiler provides many steps enabling the extraction of large static control parts (e.g. function inlining, loop normalization, `goto` elimina-

```
do i=1, n
| S1(j=i)
| do j=i, n
| | S2
| S3(j=n)
do i=n+1, m
| S3(j=n)
```

Figure 7.6: Compacted version of code in Figure 7.5(d)

tion, induction variable substitution etc.). This implementation of this algorithm is called CLooG (the Chunky Loop Generator) and was originally designed for a locality-improvement algorithm and software, Chunky (see part II). In this section is presented a study on the applicability of the presented framework to large, program representative SCoPs that have been extracted from SPECfp2000 and PerfectClub benchmarks. The chosen methodology was to perform the code regeneration of all these static control parts. Thus the transformation functions were set to the original scheduling functions (then the generated codes are semantically equivalent to the original ones).

Figure 7.7 gives a coverage of static control parts for a set of SPECfp 2000 and PerfectClub benchmarks, it gives some precisions with regard to Figure 7.7 relatively to iteration domains form and code generation. The first two columns recall some general informations about the SCoPs: the first one gives the total number of SCoPs in the corresponding benchmark, the next one count the number of SCoPs with at least one global parameter. Not surprisingly, the set of problems appears to be heavily parametric, supporting the works on fully parametric methods, but challenging the code generators since free parameters are one of the main source of memory explosions. The *Iteration Domains* section describes the shape of the polyhedra: a *point* means that the corresponding statement is executed only once, a *hyper-rectangle* is an iteration domain bounded by constants (possibly parameters), a *prism* is bounded by constants except for one bound, and an *other* has at least one varying bound. Excluding the points, the large majority of rectangular iteration domains is a sign that considering pattern matching is a good way to reduce the time spent in polyhedral operations. The *Code Generation* section describes our code generator, behavior on a Intel Pentium III 1 GHz architecture with 512 MB RAM. The first column shows how many SCoPs have to be partially regenerated in a suboptimal way because of a memory explosion on the testing system. The three challenging problems have the common property to be heavily parametric (13 or 14 free parameters). The *Time* column shows the time spent during the code generation processing. These results are very encouraging since the code generator proved its ability to regenerate real-life problems with hundreds of statements and a lot of free parameters. Both code generation time and memory requirement are acceptable in spite of a worst-case exponential algorithm complexity. Moreover, the performance between the original and the regenerated codes are quite the same, demonstrating the presented method to efficiently avoid control overhead. Previously related experiences with Omega [63] or LooPo [55] showed how it was challenging to producing efficient code just for ten or so polyhedra without time or memory explosion.

We compared the results achieved by our code generator, CLooG, with a previous implementation of the Quilleré et al. algorithm, LoopGen 0.4 [91] (the differences between CLooG and

LoopGen are a direct consequence of the improvements discussed in this thesis), and the most widely used code generator in the polyhedral model, i.e. Omega's CodeGen 1.2 [63]. Because

| | SCoPs | | Iteration Domains | | | | | Code Generation | | Robustness | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | All | Param. | All | Point | HRect. | Prism | Other | Sub. | Time (s) | CodeGen | LoopGen |
| applu | 25 | 15 | 757 | 245 | 506 | 4 | 2 | 0 | 28.16 | 39% | 53% |
| apsi | 109 | 80 | 2192 | 1156 | 1036 | 0 | 0 | 1 | 42.13 | 98% | 98% |
| art | 62 | 27 | 499 | 357 | 142 | 0 | 0 | 0 | 1.50 | 99% | 100% |
| equake | 40 | 14 | 639 | 414 | 216 | 9 | 0 | 0 | 6.80 | 73% | 73% |
| lucas | 11 | 4 | 2070 | 317 | 1753 | 0 | 0 | 1 | 47.58 | 1% | 1% |
| mgrid | 12 | 12 | 369 | 314 | 55 | 0 | 0 | 0 | 4.53 | 54% | 54% |
| swim | 6 | 6 | 123 | 63 | 60 | 0 | 0 | 0 | 0.58 | 100% | 100% |
| adm | 109 | 80 | 2260 | 1224 | 1036 | 0 | 0 | 1 | 43.94 | 92% | 92% |
| dyfesm | 112 | 70 | 1497 | 923 | 540 | 33 | 1 | 0 | 14.81 | 84% | 86% |
| mdg | 33 | 17 | 530 | 358 | 167 | 5 | 0 | 0 | 4.52 | 82% | 100% |
| mg3d | 63 | 39 | 1442 | 586 | 856 | 0 | 0 | 0 | 18.56 | 85% | 85% |
| qcd | 74 | 23 | 819 | 458 | 361 | 0 | 0 | 0 | 28.23 | 79% | 86% |

Figure 7.7: Coverage of static control parts in high-performance applications

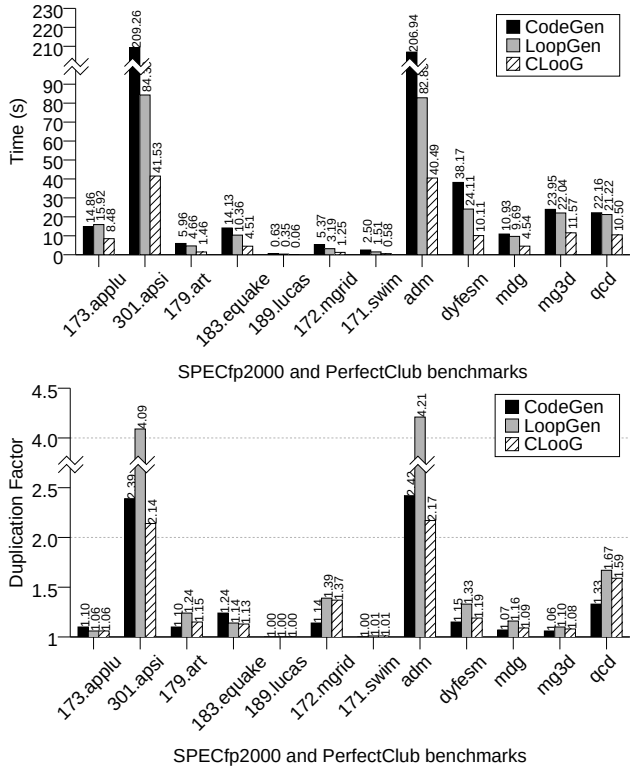



Figure 7.8: Code generation times and sizes

of inherent limitations (mainly memory explosion), these generators are not able to deal with all the real-life code generation problems in the benchmark set, the section *robustness* in Figure 7.7 gives the percentages of the input problems they are able to deal with. These results illustrate the existing need for scalability of code generation schemes. Hence, comparisons are done with the only common subset. The two valuations are the code generation time and the generated code size with respect to the original code size. The results are given in Figure 7.8. It shows

that generating directly a code without redundant control is far more efficient than trying to improve a naive one. Our pattern matching strategy demonstrates its effectiveness, since we observe a significant speedup of 4.05 between CLooG and CodeGen and of 2.57 between CLooG and LoopGen. Generated code sizes by LoopGen are typically greater than CodeGen results by 38% on average because it removes more control overhead at the price of code size. The code size improvement methodology presented in this paper significantly reduces this increase to 6% on average while keeping up the generated code effectiveness.

In conclusion, our algorithm is much faster than CodeGen and noticeably faster than LoopGen. LoopGen generates larger code, while our code and the CodeGen code are of about the same size. It remains to compare the run time overheads: our code has the same performance as the original code (it may be even more efficient because of versioning, guard or dead iteration elimination), and we believe this should be true also for LoopGen. For technical reasons, assessing the performances of CodeGen is difficult, and is left for future work.

## 7.6 Conclusion

The complexity of code generation has long been a deterrent for using polyhedral representations in optimizing or parallelizing compilers. Moreover, most existing solutions only address a subset of the possible polyhedral transformations. Thus the contribution of this thesis on code generation is twofold. First, it presents a general transformation framework in section 3.2. This results in opening new opportunities to optimize the target program, e.g. to benefit from more freedom while generating the code. Efficient solutions have been proposed to use them. Second, it demonstrates the ability of a code generator to produce an optimal control on real-life problems, with a possibly very high statement number, in spite of a worst-case exponential complexity. Many directions to improve the target code quality have been investigated, as non unit stride support or avoiding complex loop bounds. Lastly, we addressed the problem of the generated code size differently from the usual tradeoff between size and control overhead.

# Chapter 8

# Conclusion

Exploiting data locality is one of the keys to achieve high performance level in most computer systems and hence one of the main challenges for optimizing compilers. Program transformations are one of the most valuable technique for this purpose. Classical approaches only cover a limited set of programs. For instance, perfect loop nests or fully permutable loop nests are preferably considered. Similarly, the set of acceptable transformations is limited, for instance to unimodular transformations. In this thesis we presented a transformation framework to improve both data locality and performance that may automatically achieve complex transformations on a large range of programs. All aspects of high level data locality improvement framework have been explored, from extracting useful informations of an input code to the target code generation.

## 8.1 Contributions

**Improved transformation scheme** In chapter 2 we defined our program model, a slight extension of the well known static control loop nest. This model aims at including most of the intensive compute kernels that fit the polyhedral model in a general program by separating static control and static references. We showed experimentally that this model is relevant to real-life applications and that there exist many ways to increase automatically the ratio of program parts with such properties.

Static control programs can be manipulated by using the polyhedral representation of each statement. Previous transformation schemes were challenged by the problem of using general instance-wise affine scheduling functions to restructure the code because they considered each transformation as a change of basis of the polyhedral representation. As a result they need tedious processing at many steps of the transformation framework such as extending transformation functions when they are not invertible, or computing their inverses and the Hermite Normal Form of these inverses if they are not unimodular. Instead, we suggested in chapter 3 a straightforward transformation scheme able to deal with non-unimodular, non-invertible, non-integral or even non-uniform transformations by modifying the lexicographic order of the polyhedra thanks to additional dimensions. Hence we are able to deal with complex transformations.

**Improving data locality by chunking** We proposed in chapter 4 a new way to improve data locality inspired by the availability of tools to manage the cache from software. We suggested to

restructure the program in operation sets called *chunks*. At the beginning of each set the cache memory is empty. The chunk size is chosen in such a way that all the accessed data can fit in the cache. Thus we bypass the replacement mechanism which is very difficult to model. At the end of each chunk, the cache is flushed then we start the execution of the next chunk. We even showed that flushing can be omitted when the replacement mechanism is LRU or FIFO.

In this model, it is possible to provide an asymptotic evaluation of the memory traffic and to use this information to find the best program restructuring. We showed in chapter 5 that the properties of the target program, including legality and every type of locality, may be expressed as constraints on the transformation functions. We defined the algorithms that build the transformation functions according to these constraints. Overall our method exhibits several advantages since its application domain is any static control program, without any structural or dependence requirement. General affine transformation functions are automatically constructed by our algorithm and prototype. Experimental evidence shows that our framework can improve both data locality and performance in traditionally challenging programs with e.g. non-perfectly nested loops, complex dependences or non-uniformly generated references. As the method needs nothing beside the original code but the relative sizes of the cache and data, it may provide adaptable solutions for various sizes of the accessed data set.

**A method for correcting transformation for dependences** We presented in chapter 5 a general method correcting a program transformation for legality with no consequence on the data locality properties. It has been implemented in our prototype, advantageously replacing usual *enabling* preprocessing techniques and saving a significant amount of interesting transformations from being ignored. It could be used in combination with a wide range of existing data locality improvement methods, for the single processor case as well as for parallel systems using space-time mappings [73]. We believe that this method can be extended to correct transformations dedicated to other problems e.g. automatic parallelization, but the demonstration is left for future work.

**Code generation for fragile optimizations** The current trend in program optimization is to separate the selection of an optimizing transformation and its application to the source code. Most transformations are reorderings, followed optionally by modifications to the statements themselves. The program transformer must be informed of the selected reordering, and this is usually done by way of directives, like *tile* or *fuse* or *skew*. It is difficult to decide the completeness of a set of directives, or to understand their interactions. We claim that giving a scheduling function is another way of specifying a reordering, and that it has several advantages over the directive method. It is more precise, it has better compositionality properties, and there are many cases in which automatic selection of scattering functions is possible. The only drawback was that deducing a program from a scheduling function took time, and was likely to introduce much runtime overhead. For instance it may not be acceptable to save a cache miss that costs 10 CPU cycles by generating a control that costs 11 CPU cycles.

We discuss in both chapter 6 and 7 several methods to avoid control overhead from the selection of program transformations to the code generation step itself. We deeply improve a state-of-the-art algorithm in order to generate efficient codes. Our contributions include improving control sharing between statements, avoiding complex control and code size explosion. We also propose solutions to reduce the high complexity of the method. We believe that tools like

CLooG have removed the difficulty of generating efficient code in a reasonable amount of time. The whole source-to-polyhedra-to-source transformation was successfully applied to 12 benchmarks from SPEC2000fp and PerfectClub suites with a significant speedup of 4.05 with respect to the most widely used code generator, for the benchmark parts it is able to deal with.

## 8.2 Future work

Our work leaves many open problems, exciting questions and future application opportunities. The transformation framework described in this thesis is not well adapted to every kind of programs. For instance, chunking a static control program without any data reuse may only increase the control overhead. In the same way, when dealing with large data sets, we saw that chunking may only achieve a trivial solution or a *tiling*-like transformation with simple tile shapes. In other cases it may be very powerful in many situations and in particular it may optimize locality where existing methods have no effects. Hence the need of a criterion to decide whether using chunking is a benefit or not. Moreover, further implementation work is necessary to handle real-life benchmarks in our prototype and to provide full statistics on, for instance, corrected transformations. Furthermore, the question of scalability is left open since, for several tenth of deeply nested statements, the number of unknown in the constraint systems can become embarrassingly large. Splitting up the problem according to the dependence graph is a solution under investigation.

Ongoing work on code generation aims at finding new improvements on the target code quality. Two major challenges are to solve the general greatest common step problem for parameterized non-unit stride, and to find new answers to the polyhedra overlapping question. There are still problems leading to time or memory explosion. Pattern matching, i.e. avoiding the general polyhedral calculations for simple cases (e.g. rectangular domains), seems to be a promising way to reduce the time spent in code generation. It has been shown in this thesis that the main explosion cause is the number of free parameters, since the variability of parameter interactions leads to an exponential growth of the generated code. Upstream from code generation, it is possible for compilers to reduce both complexity and code versioning by finding and using the affine constraints on and between every static control part parameters [38]. Another way is to point out the most compute intensive parts in the source programs and to drive the code generator to avoid meaningless control overhead elimination which is both time and code size consuming.

One of our goals when we designed chunking was to compile for systems including scratch pad memory. This memory device often replace the classical cache in embedded systems. It has neither replacement policy nor associative addressing systems and they have to be managed by the compiler. Since there are typicaly no parameters on embedded applications (for instance the image size for video processing is always fixed), we believe that our framework may be extended and conveniently adapted for such purpose. In the same way, it may be reused for *out-of-core* processing, i.e. for improving virtual memory management for programs manipulating very large data sets. Because these memory systems are fully associative, they are an interesting target for our transformation policy.

# Appendix A

# Fourier-Motzkin Elimination Method

The Fourier-Motzkin pair-wise elimination method and its integer extension, Omega [90], is one of the most widely used algorithm in the automatic parallelization and optimization field. It may be used e.g. to check for the existence of a real (an by extension integral) solution in a linear constraint system (application to e.g. data dependence graph calculation) or to compute the range of possible values of a given variable of a linear constraint system (application to compute the projection of a polyhedron onto a given axis for e.g. code generation). This method has been designed by the French mathematician Joseph Fourier in 1826 to solve a system of affine inequalities using variable elimination.

The key idea is that a system of affine inequalities with $m$ variables can be projected onto a space with $m-1$ dimensions without altering the solution space. Let us consider a linear inequality system $A\vec{x} \geq \vec{0}$, where $A$ is a matrix of $m$ lines (constraints) and $n$ columns (component number of $\vec{x}$) of rationals. The steps of the algorithm are described below:

1. choose a variable for elimination, for instance the first component of $\vec{x}$, $x_1$
2. rearrange the inequalities by separating the $m_l$ ones with positive coefficients of $x_1$, the $m_u$ one with negative coefficients and the $m_0$ ones with null coefficients, then the inequalities can be rewritten in the following form:

$$\begin{array}{rcl} L(x_2,...,x_n) & \leq & l\,x_1 \\ & & u\,x_1 \leq U(x_2,...,x_n) \\ & & \vec{0} \leq A'(x_2,...,x_n) \end{array}$$

3. rearrange the first $m_l + m_u$ inequalities to

$$u\,L(x_2,...,x_n) \leq l\,u\,x_1 \leq l\,U(x_2,...,x_n)$$

4. eliminate $x_1$ by setting each lower bound less or equal to each upper bound and creating $m_l * m_u$ inequalities. Add to this list the $m_0$ inequalities where the coefficient of $x_1$ was 0. The new system is:

$$\begin{array}{rcl} u\,L(x_2,...,x_n) & \leq & l\,U(x_2,...,x_n) \\ \vec{0} & \leq & A'(x_2,...,x_n) \end{array}$$

5. repeat the processing until there remains a single variable

If the last range is non-empty, then the original system has a real solution. Each solution may be built by choosing a value for the last variable in the last range, then by performing a back substitution, i.e. choosing a value for the previous eliminated variable that satisfy the previous range built at step 3 for that value of the last variable and so on. If we are looking for an integral solution, we may check first that there is a integral solution in the last range. Then for each integral solution, we have to look for integral solutions in the previous range and so on. In the parametric case, $u\,L(x_2,...,x_n) \leq l\,U(x_2,...,x_n)$ is not a sufficient condition for the existence of an integer in the interval, except if $u$ or $l$ is $\pm 1$, hence we must use the Omega extension [90].

As an illustration, let us check whether the following system of affine inequalities has a solution or not:

$$\left\{\begin{array}{rcrcrcl} i & + & 2j & - & 3 & \geq & 0 \\ i & - & 2j & + & 4 & \geq & 0 \\ -i & & & + & 3 & \geq & 0 \end{array}\right.$$

we start the elimination of the variable $i$ by rearranging the inequalities according to the coefficients of $i$. Thus, inequalities are rewritten in this way:

$$\left\{\begin{array}{rcrcl} -2j & + & 3 & \leq & i \\ 2j & - & 4 & \leq & i \\ & & & i & \leq 3 \end{array}\right.$$

we eliminate the variable $i$ and we consider the following new inequalities:

$$\left\{\begin{array}{rcrcl} -2j & + & 3 & \leq & 3 \\ 2j & - & 4 & \leq & 3 \end{array}\right.$$

next, we rearrange the inequalities to exhibit the range of the variable $j$:

$$\left\{\begin{array}{rcrcl} 0 & \leq & 2j & & \\ & & 2j & \leq & 7 \end{array}\right.$$

since the range

$$0 \leq j \leq 7/2$$

is non-empty, the original system of inequalities has a real solution. Moreover, the range includes integral points, hence the system may have integral solutions for $0 \leq j \leq 7/2$. Choosing e.g. $j = 0$ we can check that the previous range on $i$ has the integral solution 3. Thus the original system of affine inequalities has also integral solutions.

# Appendix B

# Affine Form of Farkas Lemma

Farkas Lemma is fundamental to the theory of polyhedra; it was proved by the Hungarian physicist and mathematician Gyula Farkas in 1901. There exists many forms of this Lemma. The affine form is of a particular interest for affine schedule purpose since it allows to solve the dependence constraints (see section 3.3.2 or [44]). More generally it may be used to *linearize* non-affine constraints (e.g. to find the constraints to avoid min-max calculations in section 6.2.1) when we can show that they are nonnegative in a given polyhedral domain.

**Lemma B.1** *(Affine form of Farkas Lemma [97]) Let $\mathcal{D}$ be a nonempty polyhedron defined by the inequalities $A\vec{x}+\vec{b} \geq \vec{0}$. Then any affine function $f(\vec{x})$ is nonnegative everywhere in $\mathcal{D}$ iff it is a positive affine combination:*

$$f(\vec{x}) = \lambda_0 + \vec{\lambda}^T(A\vec{x}+\vec{b}), \text{ with } \lambda_0 \geq 0 \text{ and } \vec{\lambda}^T \geq \vec{0},$$

*where $\lambda_0$ and $\vec{\lambda}^T$ are called Farkas multipliers.*

As an illustration, let us consider an affine function $f(x) = ax + b$, supposed to be non-negative everywhere in the domain $[1, 3]$. To find the constraints that $a$ and $b$ have to satisfy is geometrically simple: if $a > 0$ then every function $f(x) = ax + b$ with $b \geq -a$ is non-negative in $[1, 3]$, and if $a < 0$ then we must have $b \geq -3a$ (see Figure B.1). We can use Farkas Lemma to find these constraints algebraically. The domain $\mathcal{D}$ is defined by the following inequalities

$$\mathcal{D} : \left\{ \begin{array}{rcrcl} x & - & 1 & \geq & 0 \\ -x & + & 3 & \geq & 0 \end{array} \right.$$

Thus according to the Lemma, $f(x)$ is nonnegative everywhere in this domain if and only if $f(x) = \lambda_0 + \lambda_1(x-1) + \lambda_2(3-x)$ where $\lambda_0 \geq 0$, $\lambda_1 \geq 0$ and $\lambda_2 \geq 0$. The we can equate the coefficients of the components of $f(x)$ and build the constraint system:

$$\left\{ \begin{array}{rcrcrcl} & & \lambda_1 & - & \lambda_2 & = & a \\ \lambda_0 & - & \lambda_1 & + & 3\lambda_2 & = & b \\ \lambda_0 & & & & & \geq & 0 \\ & & \lambda_1 & & & \geq & 0 \\ & & & & \lambda_2 & \geq & 0 \end{array} \right.$$

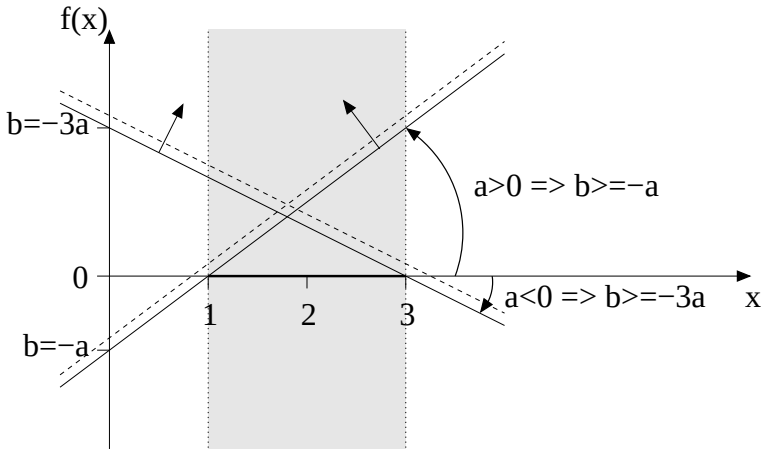



Figure B.1: Geometric example of Farkas Lemma

Then we can use the Fourier-Motzkin elimination method to remove the $\lambda_i$ from the constraint system as described in Appendix A. First, we eliminate $\lambda_0$:

$$\begin{cases} \lambda_1 - \lambda_2 = a \\ \lambda_1 - 3\lambda_2 \geq -b \\ \lambda_1 \geq 0 \\ \lambda_2 \geq 0 \end{cases}$$

then we eliminate $\lambda_1$:

$$\begin{cases} \lambda_2 \geq -a \\ -2\lambda_2 \geq -a-b \\ \lambda_2 \geq 0 \end{cases}$$

hence the range of the last variable is

$$a+b \geq 2\lambda_2 \geq max(0,-2a)$$

we deduce that the constraint on $a$ and $b$ is $a+b \geq max(0,-2a)$ which is clearly equivalent to the result of our geometric reasoning.

# Appendix C

# Chunky Loop Generator

Since this thesis is not just a theoretical exercise, our code transformation and generation scheme has been implemented and provided to the community. The aim of this appendix is to briefly introduce the Chunky Loop Generator (aka CLooG) to demonstrate how we can easily apply any affine transformations to an input code then generate an efficient target code. While these software and library have been developed during this thesis, many people from various areas are already using them for different purposes [74, 28, 23, 22, 52, 102, 101, 26, 30, 46, 35, 88]. Moreover it has been included in well known projects like LooPo, the automatic source-to-source parallelizer of the Passau university.

A large range of problems are solved by reasoning in the polyhedral model. When these problems deal with program transformations for performance, as automatic parallelization or effective cache management, code generation is the very end of the processing and it can be essential to have a fine control on this step. On one hand, generated code size has to be under control for sake of readability or instruction cache use. On the other hand, we must ensure that a bad control management does not spoil performance, for instance by producing redundant guards or complex loop bounds. CLooG implements the Quilleré et al. algorithm [91] and/or our own improvements (see [14, 15] and Part III) to provide such a control. It is specially designed to avoid control overhead and to produce a very effective code. In this appendix, we present the basic informations about the organization and writing of the CLooG input, then we show an example of using the software for parallelizing an original code.

## C.1 Organization of the CLooG Input

Basically, an input problem for CLooG has three main parts: (1) the context that represents global constraints, (2) the original statements represented by their iteration domains, and (3) the scattering functions that describes the execution order of each statement instance. Scattering is a shortcut for scheduling, allocation, chunking functions and the like. Because CLooG is oriented toward the generation of compilable code, we may associate some additional data to each iteration domain. We call this association a *statement* and the whole data about all statements and the context a *program*. Thus the input file follows the grammar in Figure C.1, where:

**Grammar 1** :

```
File             ::= Program
Program          ::= Context Statements Scattering
Context          ::= Language      Domain         Naming
Statements       ::= Nb_statements Statement_list Naming
Scattering       ::= Nb_functions  Domain_list    Naming
Naming           ::= Option Name_list
Name_list        ::= String ...
Statement_list   ::= Statement ...
Statement        ::= Iteration_domain 0 0 0
Iteration_domain ::= Domain_union
Domain_union     ::= Nb_domains Domain_list
Domain_list      ::= Domain ...
Language         ::= c | f
Option           ::= 0 | 1
Nb_statements    ::= Integer
Nb_domains       ::= Integer
Nb_functions     ::= Integer
```

Figure C.1: CLooG input file grammar

- `Context` is the properties shared by all statements. It consists of the language used (which can be `c` for C or `f` for FORTRAN 90) and the constraints on size parameters. These constraints are essential since they give to CLooG the number of parameters. If there is no parameter or no constraints on parameters, we may give a constraint always satisfied like $0 \geq 0$. `Naming` sets the parameter names. If the naming option `Option` is 1, parameter names will be read on the next line. There must be exactly as many names as parameters. If the naming option `Option` is 0, parameter names are automatically generated. The name of the first parameter will be `M`, and the name of the $(n+1)^{th}$ parameter directly follows the name of the $n^{th}$ parameter in ASCII code.

- `Statements` is the data about all statements. `Nb_statements` is the number of statements in the program, and thus the number of `Statement` items in the `Statement_list`. `Statement` is the data defining a statement. To each statement is associated a polyhedron (the statement iteration domain: `Iteration_domain`). `Naming` sets the iterator names. If the naming option `Option` is 1, iterator names will be read on the next line. There must be exactly as many names as nesting level in the deepest iteration domain. If the naming option `Option` is 0, parameter names are automatically generated. The iterator name of the outermost loop will be `i`, and the iterator name of the loop at level $n+1$ directly follows the iterator name of the loop at level $n$ in ASCII code.

- `Scattering` is the definition of scattering functions. `Nb_functions` is the number of functions `Nb_functions` (it should be equal to the number of statements or 0 if there is no scattering function) and the function themselves as a `Domain_list`. `Naming` sets the scattering dimension names. If the naming option `Option` is 1, dimension names will be read on the next line. There must be exactly as many names as scattering dimensions. If the naming option `Option` is 0, parameter names are automatically generated. The name of the $n^{th}$ scattering dimension will be `cn`.

This grammar describes an input program and scattering functions. Eventually, the input uses only characters, integers and domains. Each domain is defined by a set of constraints in the PolyLib format [105] (they can be written thanks to the PolyLib function `Matrix_Print`). They have the following syntax:

- some optional comment lines beginning with `#`,
- the row and column numbers, possibly followed by comments,
- the constraint rows, each row corresponds to a constraint which the domain must satisfy. Each row must be on a single line and is possibly followed by comments. The constraint is an equality $p(x) = 0$ if the first element is 0, an inequality $p(x) \geq 0$ if the first element is 1. The next elements are the unknown coefficients, followed by the parameter coefficients. The last element is the constant factor.

For instance, assuming that `i`, `j` and `k` are iterators and `m` and `n` are parameters, the domain defined by the following constraints :

$$\begin{cases} -i+m & \geq 0 \\ -j+n & \geq 0 \\ i+j-k & \geq 0 \end{cases}$$

may be written as follows

```
# This is the domain
3 7                 # 3 lines and 7 columns
#  i  j  k  m  n  1
1 -1  0  0  1  0  0 #    -i + m >= 0
1  0 -1  0  0  1  0 #    -j + n >= 0
1  1  1 -1  0  0  0 # i + j - k >= 0
```

Each statement iteration domain (`Iteration_domain`) is considered as a union of domains (`Domain_union`). A union is defined by its number of elements (`Nb_domains`) and the elements themselves (`Domain_list`). For instance, let us consider the following pseudo-code:

```
do i=1, n
| if (i >= m) .or. (i <= 2*m)
| | S1
| do j=i+1, m
| | S2
```

The iteration domain of `S1` can be divided into two domains and written as follows:

```
2 # Number of domains in the union
# First domain
3 5           # 3 lines and 5 columns
#  i  m  n  1
1  1  0  0 -1 #  i >= 1
```

```
1 -1  0  1  0 #  i <= n
1  1 -1  0  0 #  i >= m
# Second domain
3 5           # 3 lines and 5 columns
#  i  m  n  1
1  1  0  0 -1 #  i >= 1
1 -1  0  1  0 #  i <= n
1 -1  2  0  0 #  i <= 2*m
```

We use a similar representation for scattering functions (`Scattering`): an integer gives the number of functions (`Nb_functions`) and each function is represented by a domain. Each line of the domain corresponds to an equality defining a dimension of the function. Note that all functions must have the same scattering dimension number. For instance, let us consider the above code and the following scheduling functions:

$$\theta_{S1}\left( \begin{array}{c} i \end{array} \right) = \left( \begin{array}{c} i \\ 0 \end{array} \right), \theta_{S2}\left( \begin{array}{c} i \\ j \end{array} \right) = \left( \begin{array}{c} n \\ i+j \end{array} \right).$$

This scheduling can be written as follows:

```
2 # Number of scattering functions
# First function
2 7                   # 2 lines and 7 columns
# c1 c2  i  m  n  1
0  1  0 -1  0  0  0    #  c1 = i
0  0  1  0  0  0  0    #  c2 = 0
# Second function
2 8                   # 2 lines and 8 columns
# c1 c2  i  j  m  n  1
0  1  0  0  0  0 -1  0 #  c1 = n
0  0  1 -1 -1  0  0  0 #  c2 = i+j
```

## C.2 Example

Let us consider the allocation problem (the statement instances have to be distributed on several processors) of a Gaussian elimination:

```
do i=1, n-1
| do j=i+1, n
| | c(i,j) = a(j,i)/a(i,i)             ! S1
| | do k=i+1, n
| | | a(j,k) = a(j,k) - c(i,j)*a(i,k) ! S2
| | end do
| end do
end do
```

Allocation functions that will associate a processor number to each statement instance can be found by any good automatic parallelizer like PAF or LooPo:

$$\theta_{S1}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \end{pmatrix}, \theta_{S2}\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} k \end{pmatrix}.$$

The computations onto a given processor may respect the original order. A trivial code analysis can give us the original scheduling (see section 3.1) of the statement instances:

$$\theta_{S1}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 0 \\ i \\ 0 \\ j \\ 0 \end{pmatrix}, \theta_{S2}\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 0 \\ i \\ 0 \\ j \\ 1 \\ k \\ 0 \end{pmatrix}.$$

Thus, the fully-defined scattering functions (they may be simplified by the automatic parallelizer) with additional zeros to bring them to the same dimensionality are:

$$\theta_{S1}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ 0 \\ i \\ 0 \\ j \\ 0 \\ 0 \\ 0 \end{pmatrix}, \theta_{S2}\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} k \\ 0 \\ i \\ 0 \\ j \\ 1 \\ k \\ 0 \end{pmatrix}.$$

Then, the input file could be:

```
# -------------------------- CONTEXT ----------------------------
f # language is FORTRAN 90

1 # Context (no constraints on parameters)
1 3                       # 1 line and 3 columns
#  n  1
1  0  0                   # 0 >= 0, always true

1 # We want to set manually the parameter name
n                         # parameter name

# ------------------------ STATEMENTS --------------------------
2 # Number of statements

1 # First statement: one domain
4 5                       # 4 lines and 3 columns
#  i  j  n  1
```

```
1  1  0  0 -1             # i >= 1
1 -1  0  1 -1             # i <= n-1
1 -1  1  0 -1             # j >= i+1
1  0 -1  1  0             # j <= n
0  0  0                   # for future options

1 # Second statement: one domain
6 6                       # 6 lines and 3 columns
#  i  j  k  n  1
1  1  0  0  0 -1          # i >= 1
1 -1  0  0  1 -1          # i <= n-1
1 -1  1  0  0 -1          # j >= i+1
1  0 -1  0  1  0          # j <= n
1 -1  0  1  0 -1          # k >= i+1
1  0  0 -1  1  0          # k <= n
0  0  0                   # for future options

0 # We let CLooG set the iterator names

# ------------------------ SCATTERING -------------------------
2 # Scattering functions
# First function
8 13                                    # 8 lines and 13 columns
# p1 p2 p3 p4 p5 p6 p7 p8  i  j  n  1
0  1  0  0  0  0  0  0  0 -1  0  0  0    # p1 = i
0  0  1  0  0  0  0  0  0  0  0  0  0    # p2 = 0
0  0  0  1  0  0  0  0  0 -1  0  0  0    # p3 = i
0  0  0  0  1  0  0  0  0  0  0  0  0    # p4 = 0
0  0  0  0  0  1  0  0  0  0 -1  0  0    # p5 = j
0  0  0  0  0  0  1  0  0  0  0  0  0    # p6 = 0
0  0  0  0  0  0  0  1  0  0  0  0  0    # p7 = 0
0  0  0  0  0  0  0  0  1  0  0  0  0    # p8 = 0
# Second function
8 14                                    # 8 lines and 14 columns
# p1 p2 p3 p4 p5 p6 p7 p8  i  j  k  n  1
0  1  0  0  0  0  0  0  0  0  0 -1  0  0 # p1 = k
0  0  1  0  0  0  0  0  0  0  0  0  0  0 # p2 = 0
0  0  0  1  0  0  0  0  0 -1  0  0  0  0 # p3 = i
0  0  0  0  1  0  0  0  0  0  0  0  0  0 # p4 = 0
0  0  0  0  0  1  0  0  0  0 -1  0  0  0 # p5 = j
0  0  0  0  0  0  1  0  0  0  0  0  0 -1 # p6 = 1
0  0  0  0  0  0  0  1  0  0  0 -1  0  0 # p7 = k
0  0  0  0  0  0  0  0  1  0  0  0  0  0 # p8 = 0

1 # We want to set manually the scattering dimension names
p1 p2 p3 p4 p5 p6 p7 p8       # scattering dimension names
```

A call to the CLooG software will generate the pseudo-code in Figure C.2, where according to the allocation the value of $p1$ gives the processor number. Thus each processing with different values of $p1$ may be executed in parallel. CLooG has no information on the meaning of a scattering function, thus generating the parallel code is under the responsibility of the user, but many facilities are provided for that purpose and the post-processing of the CLooG software output or library representation is basically easy. Further technical informations on CLooG may be found in [12].

```
if (n >= 2) then
| p1 = 1
| do p5=2, n
|   | S1(i = 1,j = p5)
| end do
end if
do p1=2, n-1
| do p3=1, p1-1
|   | do p5=p3+1, n
|   |   | S2(i = p3,j = p5,k = p1)
|   | end do
| end do
| do p5=p1+1, n
|   | S1(i = p1,j = p5)
| end do
end do
if (n >= 2) then
| p1 = n
| do p3=1, n-1
|   | do p5=p3+1, n
|   |   | S2(i = p3,j = p5,k = n)
|   | end do
| end do
end if
```

Figure C.2: Target pseudo-code of the Gaussian elimination

# Personnal Bibliography

**International Journals**

- C. Bastoul, P. Feautrier. Adjusting a program transformation for legality. *Parallel Processing Letters*, accepted, planned to appear in the March 2005 issue.

**International Conferences**

- C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 7-16, Juan-les-Pins, September 2004. (Student Award)
- C. Bastoul, P. Feautrier. More legal transformations for locality. In *Euro-Par'10 International Euro-Par Conference, LNCS 3149*, pages 272-283, Pisa, August 2004. (Distinguished Paper Award)
- C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'03 IEEE International Symposium on Parallel and Distributed Computing*, pages 23-30, Ljubljana, October 2003.
- C. Bastoul, A. Cohen, A. Girbal, S. Sharma, O. Temam. Putting polyhedral loop transformations to work. In *LCPC'16 International Workshop on Languages and Compilers for Parallel Computing*, LNCS 2958, pages 209-225, College Station, September 2003.
- C. Bastoul, P. Feautrier. Improving data locality by chunking. In *CC'12 International Conference on Compiler Construction*, LNCS 2622, pages 320-335, Warsaw, April 2003.
- C. Bastoul, P. Feautrier. Reordering methods for data locality improvement. In *CPC'10 Tenth International Workshop on Compilers for Parallel Computers*, pages 187-196, Amsterdam, January 2003.

**National Conferences**

- C. Bastoul. Une méthode d'amélioration de la localité basée sur des estimations asymptotiques du trafic. In *Renpar'14*, pages 127-134, Hammamet, April 2002.

**Technical Reports Linked to Tools** :

- **WRAP-IT**: C. Bastoul, A. Cohen, S. Girbal, S. Sharma and O. Temam. Putting polyhedral transformations to work. Technical Report 4902, INRIA Rocquencourt, 2003.

- **CLooG**: C. Bastoul. Generating loops for scanning polyhedra. Technical Report 2002/23, PRiSM, Versailles University, 2002.
- **PipLib**: P. Feautrier, J.F. Collard, C. Bastoul. Solving systems of affine (in)equalities. Technical Report, PRiSM, Versailles University, 2002.
- **MHAOTEU**: J. Abella, C. Bastoul, J.-L. Bechennec, N. Drach, C. Eisenbeis, P. Feautrier, B. Franke, G. Fursin, A. Gonzalez, T. Kisku, P. Knijnenburg, J. Llosa, M. O'Boyle, J. Sebot, X. Vera. Guided Transformations. In *Report M3.D2 for the MHAOTEU ESPRIT project No 24942*, February, 2001.

# Bibliography


[1] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *SC'2000 High Performance Networking and Computing*, Dallas, november 2000.

[2] F. Allen and J. Cocke. Design and optimization of compilers. In *A catalogue of optimizing transformations*, pages 1–30. Prentice-Hall, Englewood Cliffs, 1978.

[3] J. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, october 1987.

[4] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed-memory machines. In *SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 126–138, Albuquerque, june 1993.

[5] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, june 1991.

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide, Third Edition*. SIAM, 1999.

[7] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 166–178, Santa Barbara, California, 1995.

[8] U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1976.

[9] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.

[10] U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*, pages 192–219, Irvine, august 1990.

[11] U. Banerjee. *Loop Transformations for Restructuring Compilers*. Kluwer Academic, 1993.

[12] C. Bastoul. Generating loops for scanning polyhedra. Technical Report 2002/23, PRiSM, Versailles University, 2002.

[13] C. Bastoul. Une méthode d'amélioration de la localité basée sur des estimations asymptotiques du trafic. In *RENPAR'14 Rencontres Francophones du Parallélisme*, pages 127–134, Hammamet, avril 2002.

[14] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'03 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, Ljubljana, october 2003.

[15] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, Juan-les-Pins, september 2004.

[16] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral transformations to work. In *LCPC'16 International Workshop on Languages and Compilers for Parallel Computing, LNCS 2958*, pages 209–225, College Station, october 2003.

[17] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *CC'12 International Conference on Compiler Construction, LNCS 2622*, pages 320–335, Warsaw, april 2003.

[18] C. Bastoul and P. Feautrier. Reordering methods for data locality improvement. In Peter Knijnenburg and Paul van der Mark, editors, *CPC'10 Tenth International Workshop on Compilers for Parallel Computers*, pages 187–196, Amsterdam, january 2003. Leiden Institute of Advanced Computer Science Publisher.

[19] C. Bastoul and P. Feautrier. More legal transformations for locality. In *Euro-Par'10 International Euro-Par conference, LNCS 3149*, pages 272–283, Pisa, august 2004. (Distinguished paper).

[20] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation. *IEEE Design & Test*, 17(2):74–85, April 2000.

[21] A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, october 1966.

[22] K. Beyls. *Software methods to improve data locality and cache behavior*. PhD thesis, Universiteit Gent, 2004.

[23] K. Beyls and E. D'Hollander. Compiler-generated dynamic scratch pad memory management. In *WASP'03 Proceedings of the Second International Workshop on Application Specific Processors*, San Diego, december 2003.

[24] Y. Bouchebaba. *Optimisation des transferts de données pour le traitement du signal: pavage, fusion et réallocation des tableaux*. PhD thesis, École des mines de Paris, 2002.

[25] P. Boulet, A. Darte, G-A. Silber, and F. Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3):421–444, 1998.

[26] N. Brisebarre and J.-M. Muller. Finding the truncated polynomial that is closest to a function. Technical Report 4787, INRIA-Rhone-Alpes, april 2003.

[27] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6), november 1994.

[28] P. Carribault and A. Cohen. Applications of storage mapping optimization to register promotion. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 247–256, Saint Malo, 2004.

[29] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom memory managament methodology*. Kluwer Academic, 1998.

[30] Z. Chamski, A. Cohen, M. Duranton, C. Eisenbeis, P. Feautrier, and D. Genius. *Application Domain-Driven System Design for Pervasive Video Processing*, pages 251–270. Kluwer Academic Publishers, 2003.

[31] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 205–217, 1995.

[32] Ph. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs. In *International Conference on Supercomputing*, pages 278–285, Philadelphia, may 1996.

[33] Ph. Clauss and B. Meister. Automatic memory layout transformation to optimize spatial locality in parameterized loop nests. *ACM SIGARCH, Computer Architecture News*, 28(1), march 2000.

[34] Ph. Clauss and I. Tchoupaeva. A symbolic approach to bernstein expansion for program analysis and optimization. In *CC'13 International Conference on Compiler Construction, LNCS 2985*, pages 120–133, Barcelona, april 2004.

[35] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In *Euro-Par'10 International Euro-Par Conference*, Pisa, august 2004.

[36] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *ACM Symp. on Principles and Practice of Parallel Programming*, pages 92–102, Santa Barbara, California, July 1995.

[37] J-F. Collard, T. Risset, and P. Feautrier. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3):421–436, 1995.

[38] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Fifth ACM Symposium on Principles of Programming Languages*, pages 84–97, Tucson, January 1978.

[39] K. Kennedy D. Callahan, S. Carr. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 53–65, New York, 1990.

[40] A. Darte and Y. Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20(5):679–710, 1994.

[41] R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. on Parallel and Distributed Systems*, 9(1):5–23, January 1998.

[42] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[43] P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, february 1991.

[44] P. Feautrier. Some efficient solutions to the affine scheduling problem: one dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, october 1992.

[45] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, december 1992.

[46] P. Feautrier. Scalable and modular scheduling. Technical Report 19, École Normale Supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme, april 2004.

[47] P. Feautrier, J. Collard, and C. Bastoul. Solving systems of affine (in)equalities. Technical report, PRiSM, Versailles University, 2002.

[48] B. Franke and M. O'Boyle. Combining array recovery and high level transformation: an empirical evaluation for embedded processors. In *CPC'10 International Workshop on Compilers for Parallel Computers*, pages 29–38, Amsterdam, January 1995.

[49] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memories management by global program transformation. *Journal of Parallel and Distributed Computing*, (5):587–616, 1988.

[50] G. Gao, R., V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *LCPC'5 Fifth Workshop on Languages and Compilers for Parallel Computing, LNCS 757*, pages 281–295, New Haven, august 1992.

[51] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29, New York, june 1991.

[52] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. Facultät für mathematik und informatik, universität Passau, 2004.

[53] M. Griebl, P. Faber, and C. Lengauer. Space-time mapping and tiling – a helpful combination. *Concurrency and Computation: Practice and Experience*, 16(3):221–246, march 2004.

[54] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *Int. Journal of Parallel Programming*, 28(6):607–631, 2000.

[55] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *PACT'98 International Conference on Parallel Architectures and Compilation Techniques*, pages 106–111, 1998.

[56] A. Größlinger, M. Griebl, and C. Lengauer. Introducing non-linear parameters to the polyhedron model. In Michael Gerndt and Edmond Kereku, editors, *Proc. 11th Workshop on Compilers for Parallel Computers (CPC 2004)*, Research Report Series, pages 1–12, Seeon, Germany, july 2004. LRR-TUM, Technische Universität München.

[57] M. Hall, B. Murphy, S. Amarasinghe, S. Liao, and M. Lam. Interprocedural analysis for parallelization. In *LCPC'8 International Workshop on Languages and Compilers for Parallel Computing*, pages 61–80, Columbus, August 1995.

[58] J.L. Hennessy and D.A. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[59] F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report ENSMP-CAI-87-E94, Ecole des Mines de Paris, Fontainebleau (France), 1987.

[60] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *ICS'11 Proceedings of the 11th ACM International Conference on Supercomputing*, pages 269–276, Vienna, july 1997.

[61] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving cache locality by a combination of loop and data transformations. *IEEE Transactions on Computers*, 48(2):159–167, february 1999.

[62] R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, july 1967.

[63] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symposium on the frontiers of massively parallel computation*, McLean, 1995.

[64] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, pages 301–320, Portland, 1993.

[65] I. Kodokula. *Data-centric compilation*. PhD thesis, Cornell University, 1993.

[66] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 346–357, Las Vegas, june 1997.

[67] X. Kong, D. Klappholz, and K. Psarris. The i test: A new test for subscript data dependence. In *ICPP'90 International Conference on Parallel Processing*, pages 204–211, St. Charles, august 1990.

[68] D. Kuck. *The Structure of Computers and Computations*. John Wiley & Sons, Inc., 1978.

[69] M. Le Fur. Parcours de polyèdres paramétrés avec l'élimination de Fourier-Motzkin. Technical Report 2358, INRIA, 1994.

[70] H. Le Verge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.

[71] H. Le Verge, V. Van Dongen, and D. Wilde. Loop nest synthesis using the polyhedral library. Technical Report 830, IRISA, 1994.

[72] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3–4):649–671, 1998.

[73] C. Lengauer. Loop parallelization in the polytope model. In *International Conference on Concurrency Theory, LNCS 715*, pages 398–416, Hildesheim, August 1993.

[74] C. Lengauer. *Program Optimization in the Domain of High-Performance Parallelism*, pages 73–91. Number 3016 in Lecture Notes in Computer Science. Springer-Verlag, 2004.

[75] S. Leung and J. Zahorjan. Optimizing data locality by array restructuring. Technical Report TR-95-09-01, CSE Dept. University of Washinton, 1995.

[76] E. Lewis, C. Lin, and L. Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 50–59, Montreal, June 1998.

[77] W. Li. *Compiling for NUMA parallel machines.* PhD thesis, Cornell University, 1993.

[78] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, April 1994.

[79] A. Lim. *Improving Parallelism and Data Locality with Affine Partitioning.* PhD thesis, Stanford University, 2001.

[80] A. Lim and M. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL'24 ACM Symposium on the Principles of Programming Languages*, pages 201–214, Paris, January 1997.

[81] Y. Lin. *Compiler analysis of sparse and irregular computations.* PhD thesis, University of Illinois Urbana-Champaign, 2000.

[82] V. Loechner, B. Meister, and Ph. Clauss. Precise data locality optimization of nested loops. *Journal of Supercomputing*, 21(1):37–76, january 2002.

[83] B. Lu and J. Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the 12th IEEE International Parallel Processing Symposium*, pages 42–51, Orlando, april 1998.

[84] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, february 1997.

[85] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, july 1996.

[86] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., 1997.

[87] M. O'Boyle and P. Knijnenburg. Non-singular data transformations : definition, validity and applications. In *Sixth Workshop on Compilers for Parallel Computers*, pages 287–298, 1996.

[88] D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Toward a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *SC'2004 Proceedings of the 16th ACM conference on Supercomputing*, Pittsburgh, november 2004.

[89] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the third ACM/IEEE conference on Supercomputing*, pages 4–13, Albuquerque, august 1991.

[90] W. Pugh. Uniform techniques for loop optimization. In *ICS'5 ACM International Conference on Supercomputing*, pages 341–352, Cologne, june 1991.

[91] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, october 2000.

[92] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing*, 9(4):365–389, 1995.

[93] L. Rauchwerger and D. Padua. The LRPD test: Speculative run–time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Compilers and Languages for Parallel and Distributed Computers*, 10(2):160–180, 1999.

[94] G. Rivera and C-W. Tseng. Eliminating conflict misses for high performance architectures. In *ICS'98 International Conference on Supercomputing*, pages 353–360, Melbourne, july 1998.

[95] V. Sarkar and G. Gao. Optimization of array accesses by collective loop transformations. In *ICS'91 Proceedings of the 5th ACM International Conference on Supercomputing*, pages 194–205, Cologne, june 1991.

[96] R. Sass and M. Mutka. Enabling unimodular transformations. In *In Proc. of Supercomputing '94*, pages 753–762, Washington, november 1994.

[97] A. Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, 1986.

[98] Z. Shen, Z. Li, and P. Yew. An empirical study on array subscripts and data dependencies. In *ICPP'6 International Conference on Parallel Processing Vol 2*, pages 145–152, The Pennsylvania State University, August 1989.

[99] Y. Slama and M. Jemni. Vers l'extension du modèle polyédrique aux transformations irrégulières. In *CARI'5 International Conference on African Research in Computer Science*, Antananarivo, october 2000.

[100] A. Tanenbaum. *Structured Computer Organization, third edition.* Prentice-Hall, Englewood Cliffs, 1990.

[101] S. Verdoolaege, F. Catthoor, M Bruynooghe, and G. Janssens. Feasibility of incremental translation. Technical Report CW 348, Katholieke Universiteit Leuven Department of Computer Science, october 2002.

[102] S. Verdoolaege, K. Danckaert, F. Catthoor, M Bruynooghe, and G. Janssens. An access regularity criterion and regularity improvement heuristics for data transfer optimization by global loop transformations. In *ODES'03 International Workshop on Optimizations for DSP and Embedded Systems*, pages 49–59, San Francisco, March 2003.

[103] L. Wang, W. Tembe, and S. Pande. Optimizing on-chip memory usage through loop restructuring for embedded processors. In *CC'9 International Conference on Compiler Construction, LNCS 1781*, pages 141–156, Berlin, March 2000.

[104] S. Wetzel. Automatic code generation in the polytope model. Master's thesis, Facultät für Mathematik und Informatik, Universität Passau, 1995.

[105] D. Wilde. A library for doing polyhedral operations. Technical report, IRISA, 1993.

[106] M. Wolf and M. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, New York, june 1991.

[107] M. Wolfe. Iteration space tiling for memory hierarchies. In *3rd SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, december 1987.

[108] M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley Publishing Company, 1995.

[109] D. Wonnacott and W. Pugh. Nonlinear array dependence analysis. In *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995. Troy, New York.

[110] J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, 1994.

[111] J. Xue. Transformations of nested loops with non-convex iteration spaces. *Parallel Computing*, 22(3):339–368, 1996.

[112] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

# Index

**Résumé**

Les mémoires cache ont été introduites pour réduire l'influence de la lenteur des mémoires principales sur les performances des processeurs. Elles n'apportent cependant qu'une solution partielle, et de nombreuses recherches ont été menées dans le but de transformer les programmes automatiquement afin d'optimiser leur utilisation. Les enjeux sont considérables puisque les performances ainsi que la consommation d'énergie dépendent du trafic entre les niveaux de mémoire.

Nous revisitons dans cette thèse le processus de transformation de code basé sur le modèle polyédrique afin d'en améliorer le spectre d'application comme la qualité des résultats. Pour cela nous avons redéfini le concept de parties à contrôle statique (SCoP) dont nous montrons expérimentalement la grande présence dans les applications réelles. Nous présentons une politique de transformation libérée des limitations classiques aux fonctions unimodulaires ou inversibles. Nous étendons enfin l'algorithme de Quilleré et al. pour être capables de générer en des temps raisonnables des codes particulièrement efficaces, levant ainsi une des principales idées reçues sur le modèle polyédrique. Nous avons proposé au dessus de ce schéma de transformation une méthode d'amélioration de la localité s'appuyant sur un modèle d'exécution singulier qui permet une évaluation du trafic en mémoire. Cette méthode considère chaque type de localité de même que la légalité comme autant de contraintes composant un système dont la transformation recherchée est une solution. Nous montrons expérimentalement que cette technique permet l'amélioration à la fois de la localité et des performances dans des cas traditionnellement embarrassants comme les structures de programmes complexes, les dépendances de données complexes ou les références non uniformément générées.

**Mots clés :** Localité des données, mémoire cache, contrôle statique, modèle polyédrique, transformation de programmes, compilation.

**Abstract**

Cache memories were invented to decouple fast processors from slow memories. However, this decoupling is only partial, and many researchers have attempted to improve cache use by program optimization. Potential benefits are significant since both energy dissipation and performance highly depend on the traffic between memory levels.

This thesis will visit most of the steps of high level transformation frameworks in the polyhedral model in order to improve both applicability and target code quality. To achieve this goal, we refine the concept of static control parts (SCoP) and we show experimentally that this program model is relevant to real-life applications. We present a transformation policy freed of classical limitations like considering only unimodular or invertible functions. Lastly, we extend the Quilleré et al. algorithm to be able to generate very efficient codes in a reasonable amount of time. To exploit this transformation framework, we propose a data locality improvement method based on a singular execution scheme where an asymptotic evaluation of the memory traffic is possible. This information is used in our optimization algorithm to find the best reordering of the program operations, at least in an asymptotic sense. This method considers legality and each type of data locality as constraints whose solution is an appropriate transformation. The optimizer has been prototyped and tested with non-trivial programs. Experimental evidence shows that our framework can improve both data locality and performance in traditionally challenging programs with e.g. non-perfectly nested loops, complex dependences or non-uniformly generated references.

**Keywords:** Data locality, cache memory, static control, polyhedral model, program transformations, compilation.