XFOR (Multifor): A New Programming Structure to Ease the Formulation of Efficient Loop Optimizations

Imen Fassi

Université de Strasbourg

École Doctorale Mathématiques, Sciences de l'Information et de l'Ingénieur

Laboratoire des sciences de l'ingénieur, de l'informatique et de l'imagerie (ICube)

THÈSE présentée par :

Imen FASSI

Soutenue le : **27 Novembre 2015**

pour obtenir le grade de : **Docteur de l'Université de Strasbourg**
discipline/spécialité : **Informatique**

XFOR (Multifor): A New Programming Structure to Ease the Formulation of Efficient Loop Optimizations

Thèse dirigée par :	
M. Philippe CLAUSS	Professeur, Université de Strasbourg.
Rapporteurs :	
M. François IRIGOIN	Directeur de Recherches, MINES ParisTech, Paris.
M. Denis BARTHOU	Professeur, Université de Bordeaux.
Examinateur :	
M. Steven DERRIEN	Professeur, Université de Rennes 1.

À ma famille, à mes ami(e)s. . .

" إِذَا مَا طَمَحْتُ إِلَى غَايَةٍ رَكِبْتُ ٱلْمُنَى وَ نَسِيتُ ٱلْحَذَرْ ... "

إِرَادَةُ ٱلْحَيَاةْ ـ أَبُو ٱلْقَاسِمْ ٱلشَّابِّي

"When i aspire to a goal, i ride ambition and i forget caution. . . "

Abu-AlQasim AlShabi, Will to Live

Contents

List of Figures

List of Tables

Résumé en Français

0

0.1 Introduction

Le travail réalisé dans le cadre de cette thèse de doctorat s'inscrit dans les travaux d'optimisation de programme et de nids de boucles en particulier. En effet, nous proposons une nouvelle structure itérative permettant de définir plusieurs boucles simultanément. La nouvelle structure, nommée "XFOR" ou "multifor", permet de définir pour chaque indice un ordonnancement particulier pour une exécution offrant les meilleures performances. Dans l'entête d'une boucle XFOR, on retrouve la définition de la borne inférieure, la borne supérieure et l'incrément que l'on connaît dans une boucle for classique. Mais également, nous introduisons de nouveaux paramètres, à savoir l'$offset$ et le $grain$, afin de synchroniser les indices figurant dans la boucle XFOR les uns par rapport aux autres, mais aussi, pour appliquer, de façon intuitive, des transformations polyédriques (ou composition de transformations) plus ou moins complexes sur ces boucles. Pour chaque indice, nous définissons un $offset$ (une expression affine qui est fonction des indices des XFOR englobants, et de l'indice courant). Cet $offset$ définit le décalage entre la première itération du référentiel et la première itération de l'indice correspondant. Le deuxième paramètre, le $grain$ ($grain \geq 1$), définit la fréquence d'exécution de chaque indice par rapport au référentiel.

Ce chapitre est organisé comme suit; dans la Section 0.2 nous définissons le modèle polyédrique sur lequel repose notre structure XFOR. Ensuite, dans la Section 0.3, nous énumérons les travaux marquants qui ont été réalisés afin d'améliorer la performance de programmes. La Section 0.4, représente la syntaxe d'une boucle XFOR, et décrit sa sémantique. Notre compilateur source-à-source IBB est représenté dans la Section 0.5. La Section 0.6 est dédiée aux stratégies de programmation XFOR. Par la suite, dans la Section 0.7, nous introduisons notre environnement de développement XFOR-WIZARD. Puis, dans la Section

Ce chapitre est un résumé en français du manuscrit. Chaque chapitre est représenté par une section en respectant l'organisation originale du document. Les résultats expérimentaux sont représentés d'une manière concise. Si le lecteur est intéressé par un sujet particulier, il est invité à consulter le chapitre correspondant.

0.8, nous montrons comment la structure XFOR permet de combler les lacunes des optimiseurs automatiques de programmes comme Pluto [6]. La Section 0.9 montre comment appliquer des transformations polyédriques sur les boucles d'un XFOR. Elle présente également notre outil logiciel XFORGEN permettant de générer automatiquement des boucles XFOR à partir d'une description au format OpenScop [7] générée par Pluto, dans le but d'améliorer ce programme transformé. Et finalement, dans la section 0.10, nous présentons la conclusion et les perspectives.

0.2 Modèle Polyédrique

Le modèle polyédrique [8] est une abstraction mathématique permettant de modéliser les boucles affines. Une boucle est dite affine lorsque sa borne inférieure, sa borne supérieure et les références aux tableaux qui figurent dans son corps sont des expressions affines fonction des indices de boucles englobantes et de paramètres invariants. Les paramètres qui apparaissent dans les bornes de boucles définissent la taille du problème.

Dans le modèle polyédrique, les exécutions d'une instruction sont représentées par un ensemble de points, contenu dans un $\mathbb{Z}$-polyèdre défini par une conjonction d'inégalités linéaires. Ce système d'inégalités est équivalent à un problème d'optimisation et peut être résolu grâce aux techniques de la programmation linéaire. Ainsi, le modèle polyédrique fournit une abstraction mathématique de l'instruction en représentant chaque instance dynamique d'une instruction par un point dans un espace bien défini. Étant donné que le modèle considère chaque instance d'instruction, il représente plus concrètement l'exécution d'un programme, lorsque comparé à d'autres représentations syntaxiques. En revanche, les représentations habituelles de programmes, tels que les graphes de flot de contrôle (CFG), les arbres de syntaxiques abstraits (AST), ne sont pas suffisantes pour modéliser les dépendances. Les analyses d'alias et le graphe d'espace d'itérations ne décrivent pas de façon précise les dépendances ou reposent sur des descriptions exhaustives qui ne sont pas pratiques.

Plusieurs outils logiciels ont été développés autour du modèle polyédrique. Comme exemple, nous citons; OpenScop [7], Clan [9], Clay [10], CLooG [11, 12, 13], Candl [14], Clint [15], Pluto [16, 17, 18] et Pluto+ [19].

0.3 Travaux Connexes

Dans les dernières décennies, de nombreux efforts ont été fournis pour pourvoir de nouveaux langages de programmation, ou des extensions à des langages existants, afin de permettre aux utilisateurs d'étendre l'expressivité de leurs

codes sources dans le but de mieux profiter des ressources de calcul offerts par les nouvelles architectures de processeurs. Ces propositions se divisent en deux catégories; la première est la fourniture de nouvelles fonctionnalités à des langages de programmation existants ou nouveaux – comme Titanium qui étend le langage Java [20], ou bien de nouveaux langages comme Chapel [21] ou X10 [22] –. La deuxième catégorie comprend les extensions permettant de transmettre au compilateur des informations pertinentes au moment de l'exécution – comme par exemple le *non-aliasing* de pointeurs en utilisant le mot-clé `restrict` en C, ou en utilisant des *pragmas* informatifs. Outre leur pertinence fonctionnelle, il est généralement difficile de proposer une stratégie comme norme pour tous les utilisateurs qui sont, la plupart du temps, habitués à une pratique et un langage de programmation donné. Les expériences antérieures ont montré que les extensions de langages ont eu, généralement, plus de succès car ils peuvent être adoptées progressivement sans gêner les pratiques de programmation habituelles. Un autre avantage est que ces extensions aident les programmeurs à étendre leur façon de raisonner.

De nombreux efforts ont été faits dans le domaine de l'optimisation et de la parallélisation automatique de programmes, ces travaux fournissent de nombreux résultats intéressants. Ces études ont particulièrement considéré les nids de boucles car ils représentent les parties les plus consommatrices de temps dans un programme. Ainsi, plusieurs outils de compilation et des langages spécifiques à un domaine ont été proposés, certains d'entre eux ciblant les codes *stencil* comme le compilateur Pochoir [4], d'autres ciblant les codes de traitement d'image comme Halide [3] ou Polymage [2], ou bien traitant les nids de boucles linéaires en général comme Pluto [16]. Ces outils logiciels appliquent automatiquement des transformations de boucles (comme la fusion, le tiling, l'échange de boucle, le skewing, etc..) soit pour améliorer la localité de données ou pour exposer des boucles parallèles. Cependant, du point de vue de l'utilisateur, ces compilateurs sont des boîtes noires qui peuvent échouer dans certaines circonstances ou faire quelques mauvais choix en ce qui concerne l'optimisation ou la transformation appliquée. En effet, les heuristiques implémentées ne peuvent pas être toujours efficaces et ne peuvent pas gérer simultanément touts les aspects qui peuvent affecter la performance. D'autre part, l'utilisateur a très peu de moyens pour influencer la forme des solutions générées et ainsi, il est obligé d'utiliser le code généré tel qu'il est. Ou, comme alternative unique, il peut écrire une autre version du code entièrement à la main, ce qui le limite fortement aux transformations de code simples et abordables. Ainsi, un vaste champ de stratégies de programmation est définitivement inaccessible pour l'utilisateur.

0.4 Syntaxe et Sémantique du XFOR

0.4.1 Boucle XFOR Simple

La syntaxe générale d'une boucle XFOR est la suivante :

```
xfor ( index = expr, [index = expr, ...] ;
       index relop expr, [index relop expr, ...] ;
       index += incr, [index += incr, ...] ;
       grain, [grain, ...] ; offset, [offset, ...] ) {
       label: {statements}
       [label: {statements} ...]
}
```

Listing 1 – *Syntaxe d'une Boucle XFOR Simple.*

Où [...] dénote des arguments optionnels, *index* dénote les indices des boucles qui composent le XFOR, *expr* dénote une expression arithmétique affine paramétrée fonction des indices de boucles englobantes ou une constante, *incr* dénote une constante entière, *relop* est un opérateur relationnel dans $\{ <, \leq, >, \geq, == \}$. Les valeurs des indices *index* ne sont affectées que dans l'entête de la boucle XFOR à travers l'incrémentation à chaque itération par une constante entière *incr* (on ne modifie pas les indices dans le corps de la boucle). Le *label* est un entier positif associant un groupe d'instructions à un indice de la boucle XFOR, selon l'ordre dans lequel les indices sont définis (de gauche à droite: 0 pour le premier indice, 1 pour le second indice, etc...). Les corps des boucles for composant la boucle XFOR peuvent être n'importe quel code basé sur le langage *C*. Cependant, leurs instructions ne peuvent accéder qu'à leurs indices de boucle respectifs, et non à un autre indice de boucle dont la valeur peut être incohérente dans leur domaine. Les trois premiers éléments de l'entête de la boucle XFOR sont similaires à l'initialisation, le test et l'incrémentation d'une boucle for traditionnelle, sauf que ces éléments décrivent deux ou plusieurs indices de boucle. Les deux derniers éléments définissent l'$offset$ et le *grain* pour chaque indice; l'$offset$ est une expression affine des indices englobants et courant, ou une constante. Le *grain* est une constante positive ($grain \geq 1$). Tous les indices doivent être présents dans toutes les composantes de l'entête du XFOR [23, 24].

La liste des indices définit plusieurs boucles for dont les domaines d'itérations respectifs sont tous projetés dans un même domaine "virtuel" de référence global. La façon avec laquelle ces domaines se chevauchent est définie uniquement par leurs $offsets$ et *grains* correspondants, et non par les valeurs de leurs indices respectifs, qui ont leurs propres intervalles de valeurs. Le *grain* définit la fréquence de la boucle associée par rapport au domaine de référence. Par exemple, si nous considérons une boucle XFOR à deux indices; si le premier indice a un *grain* égal à 1 et le second a un *grain* égal à 2, alors,

le premier indice sera déroulé deux fois, par conséquent, à chaque itération de la boucle de référence, nous avons une seule exécution du deuxième indice et deux exécutions du premier. L'$offset$ définit l'écart entre la première itération du référentiel et la première itération de la boucle associée. Par exemple, si l'$offset$ est égal à 3, alors la première itération de la boucle associée sera exécutée à la quatrième itération de la boucle de référence [24].

0.4.2 Nid de Boucles XFOR

```
1  xfor ( index1 = expr, index2 = expr ;
2         index1 < expr, index2 < expr ;
3         index1 += cst, index2 += cst ;
4         grain1 , grain2 ;
5          offset1 , offset2 ) {
6      label: {statements}
7      xfor ( index3 = expr, index4 = expr ;
8             index3 < expr, index4 < expr ;
9             index3 += cst, index4 += cst ;
10            grain3 , grain4 ;
11             offset3 , offset4 ) {
12         label: {statements}
13     }
14     label: {statements}
15 }
```

Listing 2 – *Syntaxe d'un Nid de Boucles XFOR de Profondeur 2 Ayant 2 Indices par Niveau.*

Les boucles XFOR imbriquées présentent quelques particularités et une sémantique spécifique doit être décrite. Sans perte de généralité, soit le nid de boucle XFOR représenté dans le Listing 2. Un tel nid se comporte comme deux nids de boucle for ; $(index_1, index_3)$ et $(index_2, index_4)$, respectivement, qui s'exécutent simultanément de la même manière que pour une boucle XFOR unique. Le $grain$ et l'$offset$ sont appliqués, à chaque profondeur, avec le même raisonnement que dans le cas d'une boucle XFOR simple. Les bornes inférieures et supérieures sont des fonctions affines des indices de boucles englobantes.

0.5 Traduction des Boucles XFOR : le Compilateur IBB

IBB ("*Iterate But Better !*") [25] est compilateur source-à-source permettant de traduire un code source contenant des boucles XFOR en un code C sémantiquement équivalent composé de boucles for classiques. Cela se fait en deux étapes ; Tout d'abord, les domaines d'indices sont transformés en polyèdres (représentés au format OpenScop [7]) sur un domaine de référence commun, et ensuite, le code de balayage est généré pour leur union. La deuxième étape est réalisée en utilisant la librairie ClooG [11, 12, 13] consacrée à générer un code pour le parcours d'union de polyèdres à partir d'une représentation OpenScop.

0.5.1 Utilisation d'IBB

Le compilateur IBB prend en entrée un fichier source *C* utilisant les boucles XFOR. Il peut identifier automatiquement ces nids de boucles et les traduire en des nids de boucles for sémantiquement équivalents. En outre, Le compilateur IBB offre à l'utilisateur un certain nombre d'options (arguments en ligne de commande) permettant de personnaliser le code cible comme décrit dans la sous-section 5.2.4 ([*IBB Options*], page 74). IBB supporte également les pragmas OpenMP [26] pour la parallélisation de boucles. Cela veut dire que l'utilisateur peut appliquer les pragmas "`#pragma omp [parallel] for`" aux boucles XFOR. De plus, IBB tolère l'utilisation des éléments de tableaux ou bien des fonctions (y compris les fonctions *min* et *max*) dans les bornes des indices. Le compilateur IBB peut être invoqué en utilisant la commande suivante :

```
IBB [ options ] input_file
```

Le comportement par défaut d'IBB est de lire le fichier en entrée et écrire le code généré dans un fichier de sortie (si l'utilisateur ne spécifie pas un nom particulier, le fichier de sortie sera nommé '`a.out.c`').

0.5.2 Domaine de Référence

Les indices d'une boucle XFOR sont exprimés relativement à un domaine de référence global. Par conséquent, les boucles for "de référence" qui sont générées dans le code final, itèrent sur ce domaine. Dans cette sous-section, nous décrivons comment IBB calcule les domaines de ces boucles for.

Soit une boucle XFOR de profondeur 1. Le boucles for de référence équivalentes à cet XFOR doivent balayer un certain nombre d'itérations. Soit f le nombre d'indices dans l'entête du XFOR. En calculant l'union disjointe de tout les domaines d'itérations des boucles for (*i.e.* les indices de la boucle XFOR), nous obtenons un ensemble de domaines adjacents D_i dans lesquels certaines boucles des f indices se chevauchent. Soient lb_i, ub_i, $grain_i$ et $offset_i$, $i = 1..f$, les paramètres caractérisant chaque boucle for de l'entête du XFOR. La borne inférieure nlb_i et la borne supérieure nub_i de chaque boucle de référence sont définies ainsi :

- $nlb_i = offset_i$
- $nub_i = (ub_i - lb_i + 1)/(lcm(grain_k)/grain_i) + offset_i, k = 1..f$

nlb_i consiste à décaler le domaine et nub_i consiste à le contracter par un facteur égal au *plus petit commun multiple* (PPCM) de tout les *grains* divisé par le *grain* courant. En effet, le *grain* est traduit par un déroulement de boucles dans le but d'avoir un code équivalent qui est plus efficace. Chaque indice j est déroulé d'un facteur égal à : $ppcm(grain_i)/grain_j, i = 1..f$.

L'union disjointe des domaines D_i est calculée en utilisant les bornes nlb_i et nub_i, résultant en un ensemble de domaines disjoints R_j. La valeur initiale de l'indice de la boucle de référence est égale à : $MIN_{i=1..f}\,(nlb_i)$. Par conséquent, le nombre total d'itérations dans le domaine de référence est égal à :

$$MAX_{i=1..f}\,(nub_i) - MIN_{i=1..f}\,(nlb_i) + 1 \tag{1}$$

De façon plus générale, étant donné un nid de boucles XFOR, le calcul du domaine de référence se fait en quatre étapes. Premièrement, chaque domaine d'itérations associé à une boucle for du nid de boucles XFOR est normalisé (*i.e.* Les bornes inférieures sont décalées à 0). Deuxièmement, chaque domaine est translaté depuis l'origine suivant la valeur de son $offset$[1]. Troisièmement, chaque domaine D_i est déroulé par un facteur égal à $ppcm(grain_k)/grain_i, k = 1..f$. Enfin, l'union disjointe est calculée et résulte en une union de domaines convexes R_j.

0.5.3 Génération de Code

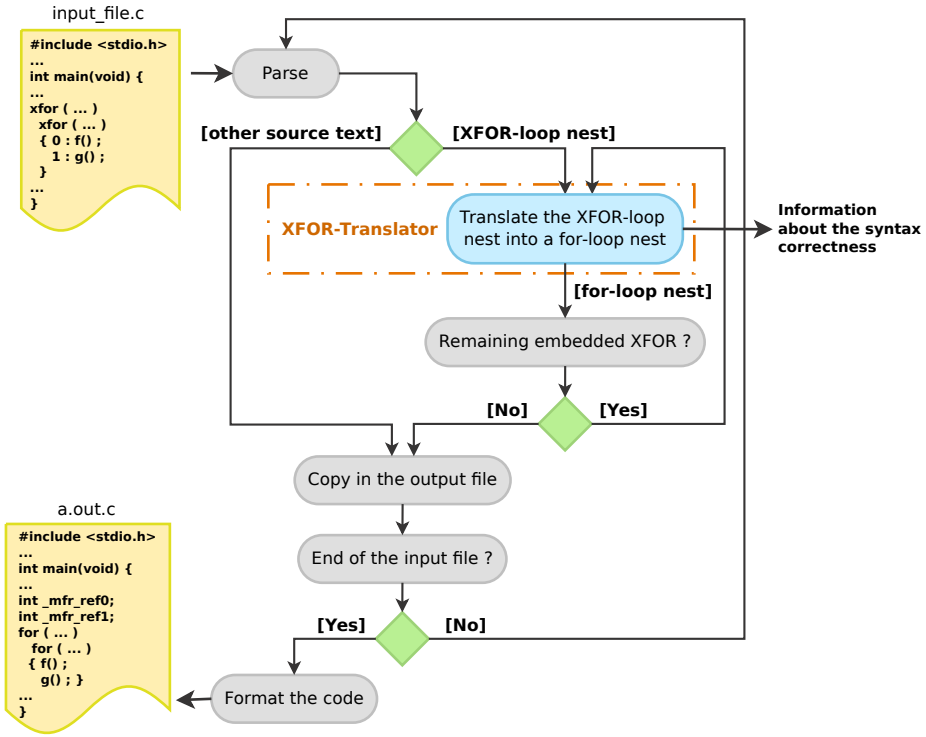


Figure 1 – *IBB : le Compilateur XFOR.*

La Figure 1 illustre le fonctionnement du compilateur du XFOR. Pendant la l'analyse du fichier d'entrée, IBB copie le code dans le fichier de sortie jusqu'à l'identification d'un nid de boucles XFOR. Le nid identifié est renvoyé au module

[1]Notez que les valeur des indices ne définissent pas leurs positions dans le domaine de référence.

de traduction d'un XFOR (*XFOR-Translator*). Ce dernier le traduit en des nids de boucles *for* sémantiquement équivalents. Ensuite, IBB copie les boucles générées dans le fichier de sortie. Cette procédure est répétée jusqu'à la fin du fichier d'entrée. Une fois l'analyse terminée, le code généré est formaté.

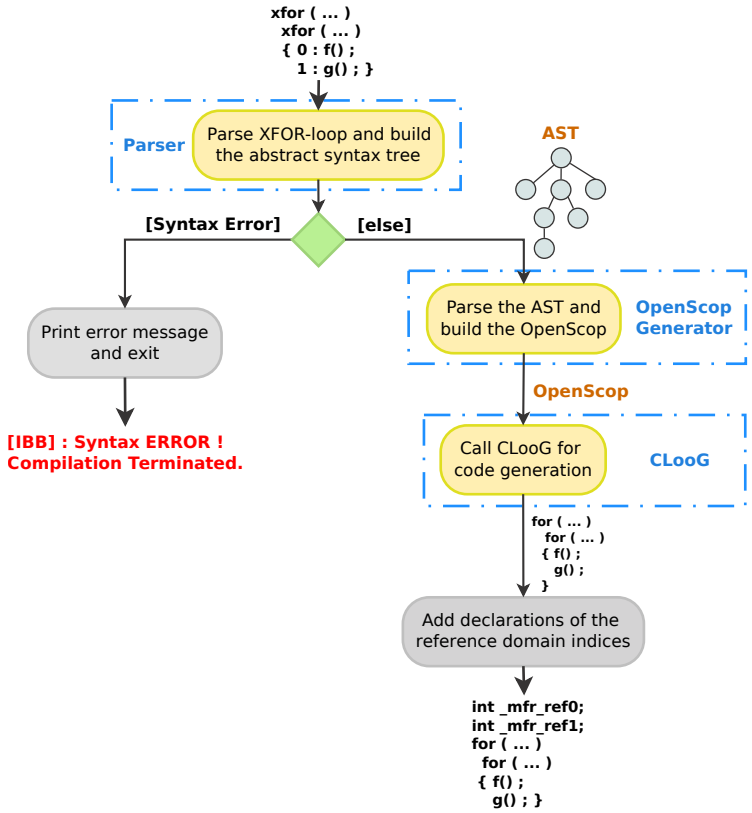


Figure 2 – *Traducteur du XFOR (XFOR-Translator).*

Le traducteur de XFOR (*XFOR-Translator*) est responsable de la génération d'un code *C* composé de boucles *for* classiques sémantiquement équivalentes à partir d'un nid de boucles XFOR donné. Cette tâche est accomplie en trois étapes (voir Figure 2) ; Tout d'abord, l'analyseur syntaxique analyse le nid de boucles XFOR et génère un arbre syntaxique abstrait (AST) qui comprend toutes les informations relatives aux indices du XFOR. Ensuite, le générateur de description OpenScop (*OpenScop generator*) exprime les domaines d'indices par rapport à un domaine de référence commun. Finalement, le générateur de code CLooG [11] génère le code de balayage pour l'union de ces domaines.

0.6 Stratégies de Programmation XFOR

XFOR est une structure dédiée à la programmation orientée reutilisation de données. Elle permet de définir explicitement les distances de reutilisation entre les différentes instructions figurant dans le corps d'une boucle XFOR. Les programmes à calculs stencils, et de façon générale, les programmes dans lesquels il y a un accès fréquent aux mêmes données, sont de bons

candidats pour une reécriture en XFOR. Dans cette Section, nous présentons des techniques permettant d'écrire des codes XFOR efficaces et d'améliorer les temps d'exécution. Le programmeur peut choisir entre ; minimiser les distances de reutilisation inter-instructions ou intra-instruction en même temps que la parallélisation. Nous évaluons les performances des codes XFOR vis-à-vis les codes originaux dans les exécutions séquentielles-vectorisées et parallèles.

0.6.1 Minimisation des Distances de Reutilisation Inter-Instructions

Lors de la manipulation de plusieurs domaines d'itérations balayés par des nids de boucles *for* successifs, ces domaines peuvent être réordonnancés en les chevauchant à travers le décalage ($offset$) et le déroulement (*grain*). Ceci peut réduire les distances de réutilisation de données tout en respectant les dépendances. L'ordonnancement final peut être décrit par un nid de boucles XFOR [24].

0.6.2 Minimisation des Distances de Reutilisation Intra-Instruction

La deuxième stratégie de programmation consiste à minimiser les distances de reutilisation entre les instructions qui appartiennent au même corps de boucle. Ici, un domaine d'itérations est divisé en plusieurs domaines, chacun étant associé à un sous-ensemble des instructions du corps de la boucle d'origine ou bien à un calcul partiel d'une expression arithmétique d'origine (si une telle décomposition est autorisée). Cette stratégie peut être également utile pour l'optimisation des codes qui contiennent des test de *modulo* sur les indices de boucles. Ainsi, les instructions peuvent être réordonnancées en superposant ces domaines comme décrit dans la stratégie précédente.

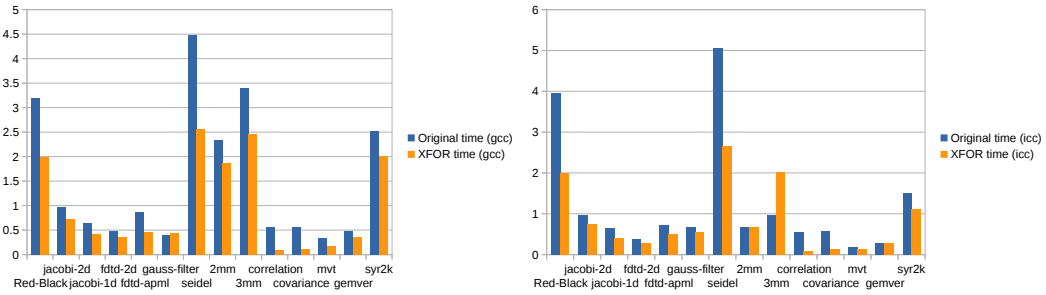
0.6.3 XFOR Parallèle

La structure XFOR permet également au programmeur d'écrire des codes parallèle ; soit avec l'utilisant les pragmas "`#pragma omp [parallel] for`" ou bien à travers la vectorisation. La vectorisation est activée en choisissant une valeur adéquate de l'$offset$.

0.6.4 Résultats Expérimentaux

Les expérimentations on été faites sur un processeur *Intel Xeon X5650 6-core, 2.67 GHz*. Les codes testés font partie du benchmark polyédrique *Polybench* [27]. Chaque code a été reécrit en utilisant la structure XFOR. Les codes originaux ainsi que les codes XFOR ont été compilés en utilisant *GCC*4.8.2 et *ICC*14.0.3 avec les options `O3` et `march=native`. La vectorisation automatique a été activée afin d'en profiter lorsque c'est possible. Les temps d'exécution sont

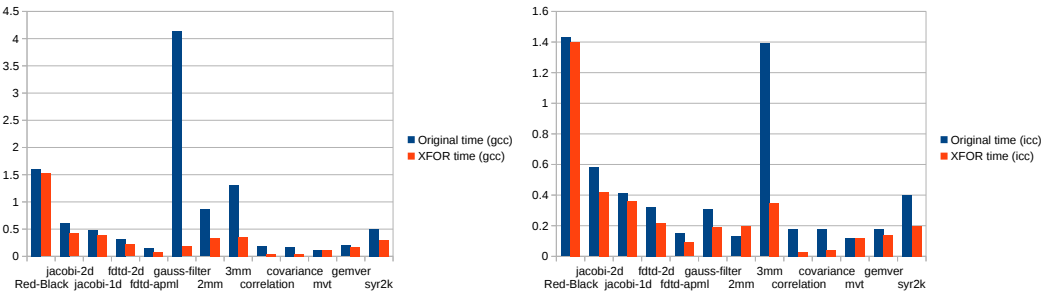
donnés en secondes. La Figure 3 donnent les temps d'exécution pour une exécution séquentielle-vectorisée. Les codes XFOR montrent des accélérations impressionnantes.



(a) *gcc* (b) *icc*

Figure 3 – *Les Temps d'Exécution pour les Codes XFOR et Originaux Séquentiels-Vectorisés.*

La parallélisation OpenMP a été activée en *GCC* avec l'option `-fopenmp`, et en *ICC* avec l'option `-openmp`. La vectorisation automatique a été activée afin d'en profiter lorsque c'est possible. Suivant les dépendances, la boucle la plus externe dans les codes XFOR et les codes originaux à été parallélisée. Les codes ont été exécutés en utilisant 6 *threads* parallèles. Les Figures 4 donnent une comparaison entre les codes originaux et le code XFOR. Nous remarquons que les codes XFOR vont jusqu'à 6 fois plus vite que les codes originaux (cas du benchmark *correlation* compilé avec *ICC*).



(a) *gcc* (b) *icc*

Figure 4 – *Les Temps d'Exécution pour les Codes XFOR et Originaux OpenMP Parallèles.*

0.7 XFOR-WIZARD : L'Environnement de Programmation XFOR

Pour une utilisation efficace de la structure XFOR, nous avons développé un outil d'aide à la programmation appelé «XFOR-WIZARD». Il s'agit d'un assistant qui guide le programmeur dans le processus de reécriture d'un programme donné en un programme équivalent utilisant des boucles XFOR. XFOR-WIZARD comporte

un analyseur de dépendances permettant de vérifier si le programme XFOR respecte bien les dépendances exprimées dans un programme de référence.

La mise en œuvre d'XFOR-WIZARD a nécessité l'extension de l'outil polyédrique Clan afin d'analyser les boucles XFOR. Clan est un analyseur qui génère une description de nids de boucles *for* au format OpenSCop, cette description peut être soumise à plusieurs outils polyédriques pour faire des traitements divers.

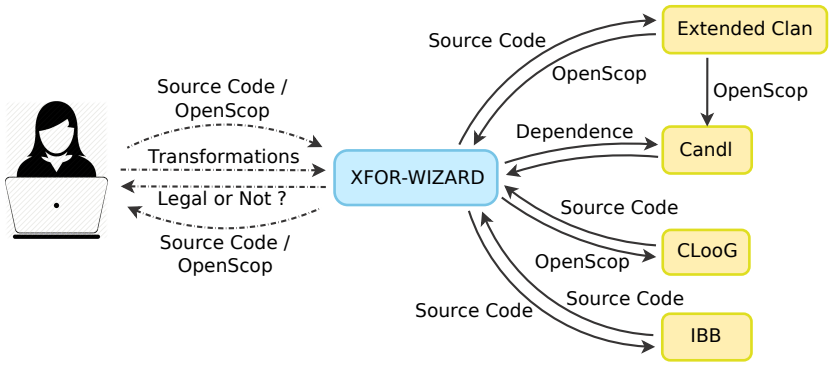


Figure 5 – *XFOR-WIZARD : Architecture et Interactions.*

Comme la Figure 5 le montre, l'utilisateur interagit seulement avec l'assistant. Il applique les manipulations et les transformations de boucles. Puis, XFOR-WIZARD lui informe sur la légalité de la transformation. Ceci est assuré en invoquant les outils polyédriques Clan (la version étendue XFOR) et Candl.

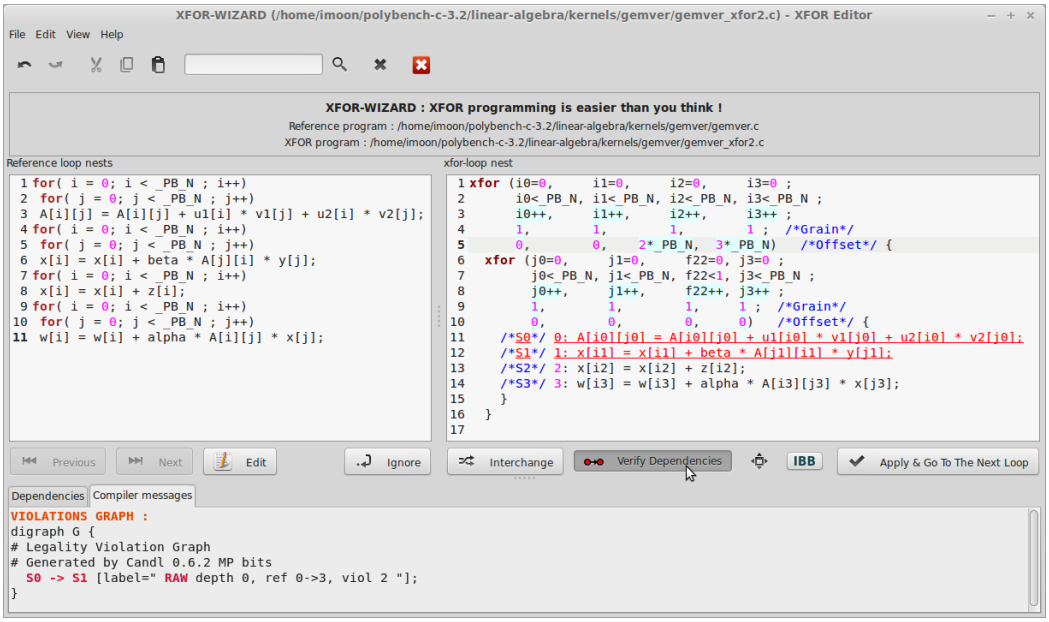


Figure 6 – *XFOR-WIZARD : L'Éditeur XFOR.*

Nous avons implémenté une interface graphique (voir Figure 6) permettant une utilisation facile et efficace de XFOR-WIZARD. Avec cette interface,

l'utilisateur peut éditer un nid de boucles XFOR et vérifier étape par étape la validité de la modification.

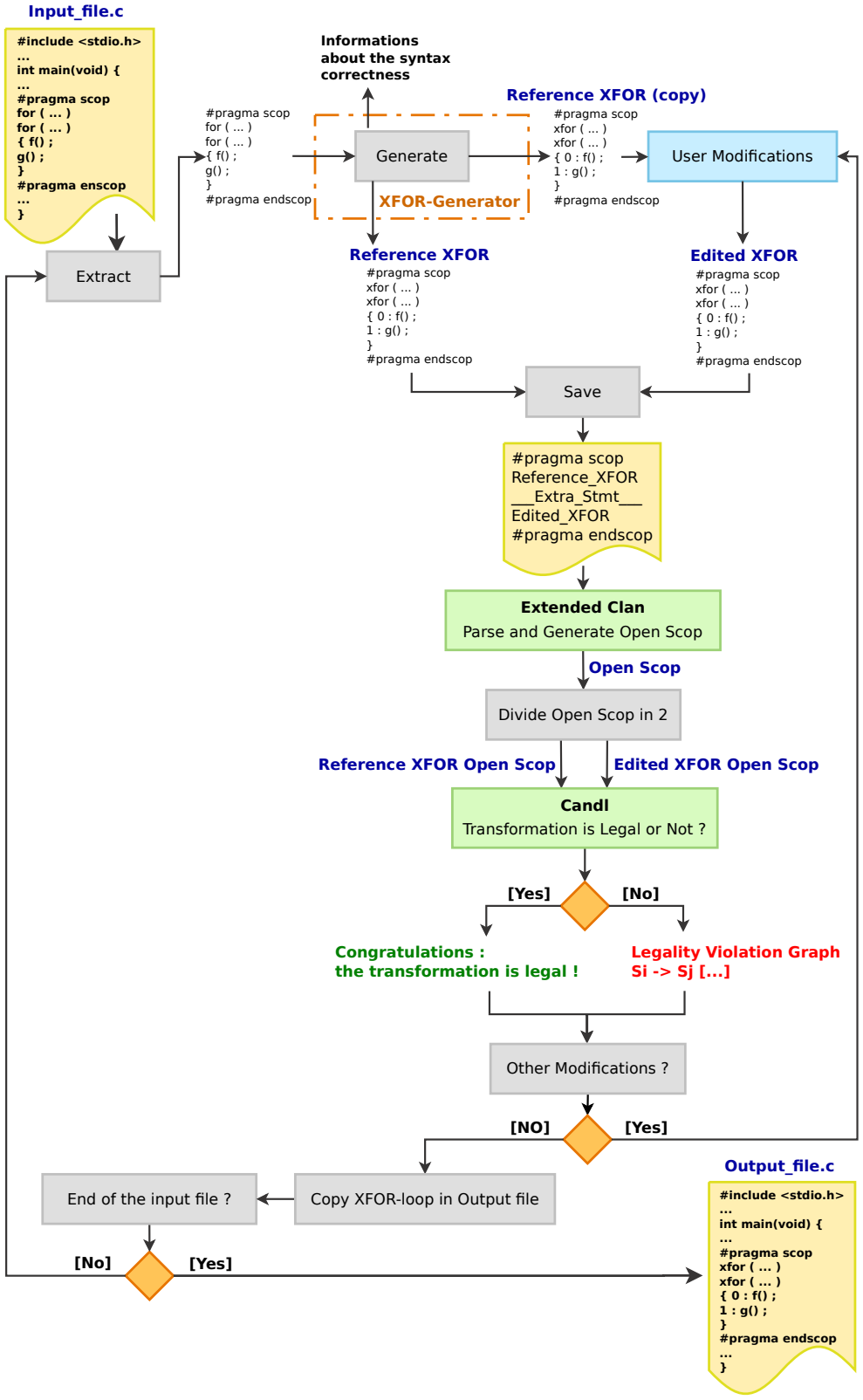


Figure 7 – *Génération de Code XFOR et Vérification des Transformations de Boucles.*

La Figure 7 illustre le fonctionnement et l'architecture générale de XFOR-WIZARD. Essentiellement, l'assistant prend comme entrée un programme utilisant des boucles *for*, mais il est également capable d'analyser les boucles XFOR, ce qui est particulièrement utile lorsque le programmeur veut reprendre un projet qu'il a commencé antérieurement. Ainsi, il peut continuer à transformer les boucles XFOR et vérifier étape par étape la validité des transformations. Afin d'aider le programmeur dans la génération d'un programme XFOR équivalent mais plus efficace, XFOR-WIZARD commence par l'analyse du programme de référence et identifie les parties de code placées entre `#pragma scop` et `#pragma endscop`. Ensuite, pour chaque scop, il génère automatiquement un nid de boucles XFOR parfaitement imbriqué qui est sémantiquement équivalent à la séquence de nids de boucles *for* initiale, où toutes les instructions sont ordonnancées de façon identique au code de référence. Ceci est réalisé en fusionnant tous les nids de boucles *for* dans un nid XFOR unique, où la boucle XFOR a autant d'indices que le nombre d'instructions dans les nids de boucles de référence, et les *offsets* de la boucle XFOR la plus externe sont fixés au valeurs maximales (typiquement le nombre d'itérations de la boucle initiale) qui garantissent une exécution séquentielle similaire à la séquence des nids d'origine. Une fois que le nid de boucles XFOR a été généré, il est considéré comme boucle de référence. XFOR-WIZARD fournit à l'utilisateur une copie du XFOR de référence sur laquelle il peut appliquer des transformations et vérifier, étape par étape, si elles sont légales ou non.

Afin d'utiliser l'analyseur de dépendance Candl pour vérifier les dépendances, nous procédons comme suit; nous sauvegardons la boucle de référence et la boucle éditée dans le même fichier[2]. Cela garantit que lorsque la version étendue du Clan sera invoquée, les deux représentations OpenScop auront le même ordre de paramètres et les mêmes identifiants pour les tableaux. Après cela, Clan est invoqué. Ensuite, l'OpenScop résultant est divisé en deux parties ; chaque nid de boucle a sa propre représentation OpenScop. Enfin, Candl est invoqué. Ce dernier commence par le calcul des dépendances de la boucle de référence, puis les compare à l'ensemble des dépendances du XFOR transformé, et informe ensuite l'utilisateur sur la légalité de ses modifications.

0.8 XFOR et Optimiseurs Automatiques

La structure XFOR permet de combler les lacunes en performances non traitées par les optimiseurs automatiques de boucles. Nous considérons l'optimiseur polyédrique bien connu *Pluto* [6, 16] qui implémente des stratégies et des transformations avancées pour l'optimisation de boucles. Les expérimentations

[2]Nous insérons une instruction supplémentaire entre les deux boucles pour pouvoir les séparer facilement après la génération de l'OpenScop.

montrent que la structure XFOR aide à découvrir des aspects importants de la performance qui n'étaient pas, pour certains d'entre eux, identifiables avant. Ceci a été fait en comparant les codes XFOR aux codes Pluto, mais aussi, en comparant les codes XFOR entre eux. Nous avons identifié cinq lacunes en performance dans les stratégies d'optimisation automatiques de boucles :

1. Insuffisance de l'optimisation de la localité de données.
2. Excès de branches conditionnelles dans le code généré.
3. Code trop long en instructions machine.
4. Optimisation excessive de la localité de données causant des cycles d'inactivité du processeur.
5. Non exploitation de la vectorisation.

0.8.1 Cycles de Processeur Gaspillés

Le temps d'exécution d'un programme est directement lié au nombre total de cycles dépensés par le processeur pour exécuter ses instructions. Parmi ces cycles, certains peuvent être bloqués (dans l'attente de la terminaison de certains évènements duquel la continuation de la séquence d'instructions en cours d'exécution dépend) et d'autres peuvent être consommés inutilement à exécuter une séquence d'instructions trop longue qui aurait pu être accomplie en utilisant un nombre réduit d'instructions ou bien en profitant de la vectorisation. Ces cycles perdus consomment inutilement du temps et de l'énergie. Bien que le blocage du processeur ne puisse pas être évité complètement, ou bien masqué partiellement par l'exécution simultanée d'instructions, le nombre de cycles bloqués doit être minimal. Pour cela, les causes de blocage doivent être traitées convenablement durant la phase d'optimisation de programme. Ces causes peuvent être classées en quatre catégories :

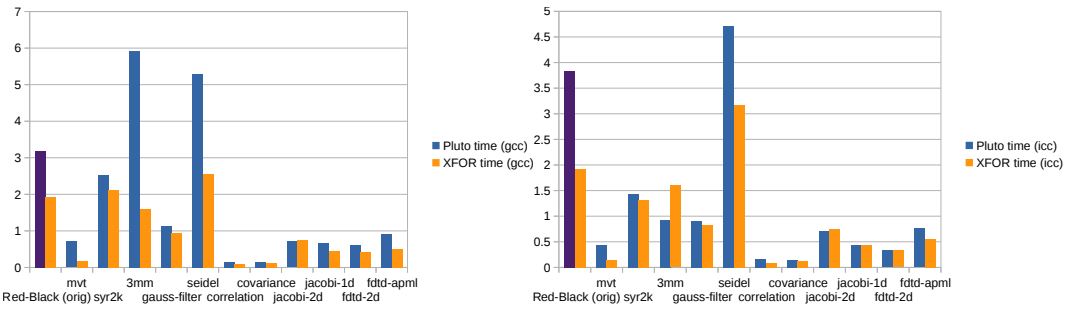
1. Blocage du à l'insuffisance de ressources de calcul.
2. Blocage du à la latence mémoire.
3. Blocage du aux dépendances entre les instructions.
4. Blocage du aux mauvaises prédictions des branches conditionnelles.

Le premier point peut être résolu en utilisant plus de matériel. Le point 2 est traité par la majorité des compilateurs qui implémentent les techniques d'optimisation de la localité de données qui sont plus ou moins efficaces. Cependant les heuristiques utilisées vont nécessairement rater certaines possibilités d'optimisation qui peuvent être appliquées par un

programmeur expert, particulièrement en utilisant la structure XFOR. Les stratégies implémentées ne sont pas conscientes des autres aspects de la performance, et cela peut avoir un mauvais impact qui peut annihiler le gain offert par l'amélioration de la localité de données. En ce qui concerne le point 3, ce problème n'a jamais été explicitement traité par les optimiseurs automatiques comme la localité de données a toujours été la seule finalité. De plus, la minimisation de la distance de reutilisation de données entre les instructions peut empêcher la vectorisation de ces instructions. Concernant le point 4, comme les prédicteurs de branches conditionnelles ne peuvent pas être contrôlé par logiciel, le risque de mauvaises prédictions est proportionnel au nombre de branches dans le code généré. Ce dernier dépend de la transformation appliquée. Par exemple la transformation classique "*tiling*" génère des boucles avec un contrôle compliqué particulièrement quand elle implique des formes non rectangulaires.

0.8.2 Résultats Expérimentaux

Les expérimentations ont été faites sur un processeur *Intel Xeon X5650 6-core, 2.67GHz (Westmere)* qui tourne sous *Linux 3.2.0*. Les codes testés font partie du benchmark polyédrique *Polybench* [27]. Le code XFOR de *Red-Black Gauss-Seidel* est comparé à sa version originale comme ce code n'est pas supporté par Pluto. Cependant, chaque code XFOR est comparé à la meilleur version Pluto. Les codes XFOR et Pluto ont été compilés avec *GCC*4.8.1 et *ICC*14.0.3 en utilisant les options `O3` et `march=native`. Les temps d'exécution des boucles principales sont donnés en secondes. Le *grain* des boucles XFOR est toujours égal à 1, sauf pour le code de *Red-Black Gauss-Seidel* où il est égal à 2.



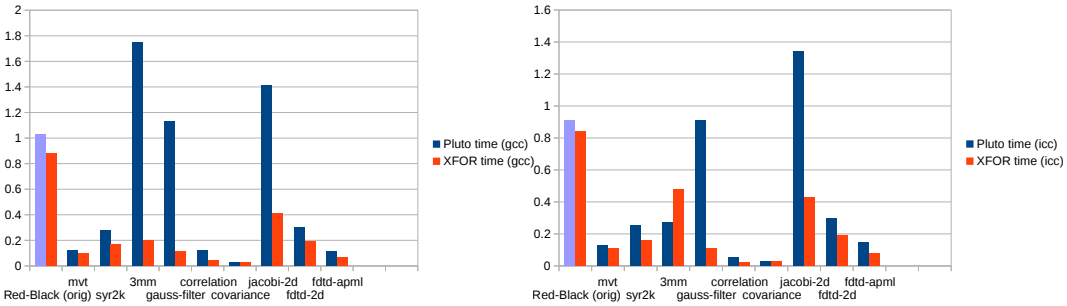
(a) *gcc* (b) *icc*

Figure 8 – *Les Temps d'Exécution pour les Codes XFOR et Pluto Séquentiels-Vectorisés.*

Codes Séquentiels et Vectorisés. Nous avons remarqué qu'*ICC* arrive à vectoriser certains codes, alors que *GCC* échoue. Des exemples typiques sont les codes : `jacobi-1d`, `fdtd-2d` et `fdtd-apml`. *ICC* est capable de vectoriser les codes Pluto, tandis que *GCC* vectorise seulement les codes XFOR. Les mesures

sont représentés dans les Figures 8. Ces mesures montrent que la structure XFOR offre des accélérations intéressantes.

Codes Parallèles OpenMP. Nous avons également comparé les codes XFOR et Pluto parallèles. Pluto génère un code parallèle grâce à l'option `-parallel`. La parallélisation a été activée dans *GCC* en utilisant l'option `-fopenmp`, et en *ICC* en utilisant l'option `-openmp`. Chaque code a été exécuté en utilisant 12 *threads* parallèles placés sur les 6 coeurs *hyperthreadés* du processeur *Xeon* X5650. Les mesures obtenues sont représentées à la Figure 9. Les mesures montrent que les codes XFOR sont généralement plus rapides.



(a) *gcc* (b) *icc*

Figure 9 – *Les Temps d'Exécution pour les Codes XFOR et Pluto Parallèles.*

0.9 XFOR : une Représentation Intermédiaire Polyédrique et une Approche Collaborative Manuelle-Automatique pour l'Optimisation de Boucles

0.9.1 Transformations de Boucles

XFOR permet d'exprimer de façon simple diverses transformations Polyédriques comme : le décalage de boucles, la déroulement de boucles, la fission de boucles, la fusion de boucles, la torsion de boucles, l'épluchage de boucles, la permutation de boucles, l'inversion de boucles, etc

Composition Flexible de Transformations. En plus de la facilité de l'application des transformations de boucles, les boucles XFOR permettent également de composer les transformations de façon simple et intuitive. Une dilatation de boucles est implémentée tout simplement en fixant le *grain* correspondant au coefficient de la dilatation. Cette transformation sera traduite par un déroulement des autres indices i_k par un coefficient égal à $ppcm(grain_j)/grain_k$. Où $ppcm(grain_j)$ est le *plus petit commun multiple* de tout les *grains* qui apparaissent dans l'entête du XFOR et $grain_k$ est le *grain*

correspondant à l'indice i_k. Le reste des transformations peut être appliqué en fixant l'$offset$ comme décrit dessous :

Généralisation. Sans aucune perte de généralité, considérons une boucle XFOR de profondeur 2. La transformation affine : $((i, j) \rightarrow (ai + bj + c, a'i + b'j + c'))$ peut être exprimée en utilisant les $offsets$ suivant :

```
xfor (i=... ; ... ; (a-1)*i+b*j+c, ...)
xfor (j=... ; ... ; a'*i+(b'-1)*j+c', ...)
```

Plus généralement, soit I un vecteur d'itérations de dimension n. Une transformation t d'un indice $i_j, 0 \leq j < n$:

$$t(i_j) = \sum_{k=0}^{j-1} \alpha_k i_k + \alpha_j i_j + \sum_{k=j+1}^{n-1} \alpha_k i_k$$

est appliquée en fixant l'$offset$ de l'indice i_j à l'expression suivante :

$$\sum_{k=0}^{j-1} \alpha_k i_k + (\alpha_j - 1) i_j + \sum_{k=j+1}^{n-1} \alpha_k i_k$$

0.9.2 XFORGEN : Générateur Automatique de Boucles XFOR à Partir d'une Description OpenScop

Afin de généraliser l'utilisation de boucles XFOR comme représentation intermédiaire polyédrique pour les transformations de boucles, nous avons développé un outil logiciel qui génère des boucles XFOR à partir de l'OpenScop généré par les optimiseurs automatiques de boucles. Ce logiciel est appelé XFORGEN. Comme optimiseur automatique de la localité de données, nous considérons Pluto [4]. C'est l'outil logiciel le plus connu qui a émergé dans les compilateurs polyédriques source-à-source. Pluto met en œuvre les stratégies d'optimisation de boucles et les transformations les plus avancées.

Le fonctionnement de XFORGEN est illustré par la Figure 10. XFORGEN prend en entrée un fichier; le fichier peut être une description OpenScop, ou un programme contenant des boucles $for/xfor$. Tout d'abord le fichier est analysé et les boucles à optimiser sont représentées en utilisant la structure de données OpenScop. Ensuite, si l'utilisateur a choisi d'appeler l'optimiseur automatique Pluto, la première étape consiste à configurer ses options, puis, Pluto est invoqué pour optimiser le Scop initial. Ensuite, la structure OpenScop est scannée pour générer un arbre de syntaxe abstrait (AST) qui représente le nid de boucle XFOR équivalent. Enfin, le code XFOR est généré à partir de cette structure intermédiaire.

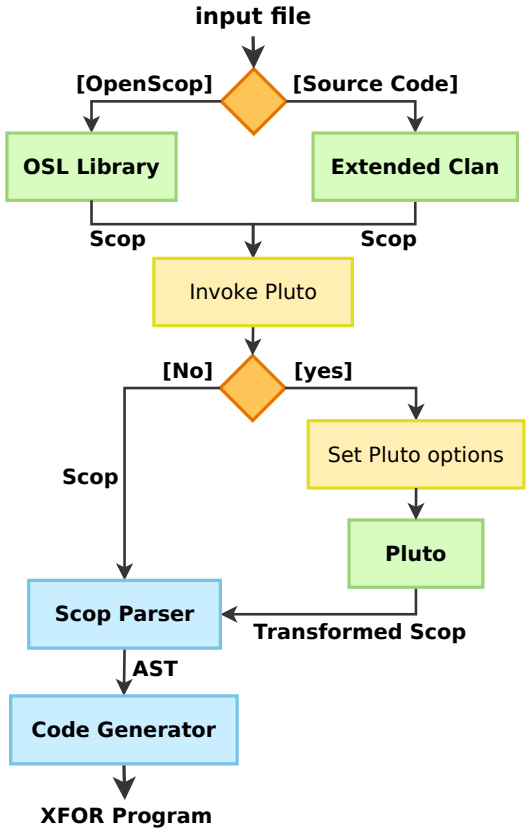


Figure 10 – *Fonctionnement de XFORGEN.*

La Figure 11 détaille la génération de la structure XFOR. La première étape consiste à éliminer les unions de domaines de la structure OpenScop. Deuxièmement, la nouvelle structure OpenScop (sans union de domaines) est analysée. Pour chaque *statement*, les opérations suivantes sont appliquées :

1. Lire la relation `Domain` et calculer les bornes de nids de boucles. Ici, non seulement un indice est calculé, mais tout un ensemble d'indices imbriqués.

2. Lire la relation `Scattering` et calculer l'$offset$ de chaque indice dans le nid.

3. Ajouter le nid à l'AST qui représente le nid de boucles XFOR d'une manière à avoir à la fin un nid parfait.

4. Insérer les instructions correspondantes dans la liste des instructions. La position d'une instruction dans la liste est déterminée en utilisant les composantes constantes de la matrice `Scattering`.

5. Une fois que tous les *statements* ont été traités, les indices du nid de boucles XFOR sont renommés.

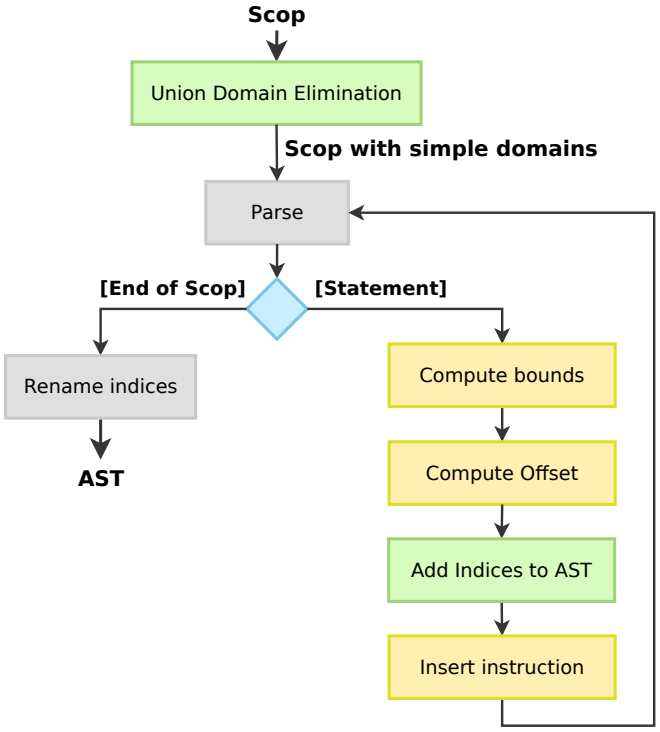


Figure 11 – *Analyseur de Scop et Génération de Boucles XFOR.*

0.10 Conclusion

Dans cette thèse, nous présentons une nouvelle structure de contrôle en programmation nommée "XFOR" permettant de définir plusieurs boucles de type *for* simultanément ainsi que l'application de transformations (simples ou composées) de boucles d'une façon aisée et intuitive. Les expérimentations ont montré des accélérations significatives des codes XFOR par rapport aux codes originaux, mais aussi par rapport au codes générés automatiquement par l'optimiseur polyédrique de boucles Pluto [6, 16].

Nous avons mis en œuvre la structure XFOR par le développement de trois outils logiciels: (1) un compilateur source-à-source nommé IBB (*Iterate-But-Better!*), qui traduit automatiquement tout code basé sur le langage *C* contenant des boucles XFOR en un code équivalent où les boucles XFOR ont été remplacées par des boucles *for* sémantiquement équivalentes. L'outil IBB bénéficie également des optimisations implémentées dans le générateur de code polyédrique CLooG [11] qui est invoqué par IBB pour générer des boucles *for* à partir d'une description OpenScop; (2) un environnement de programmation XFOR nommé XFOR-WIZARD qui aide le programmeur dans la ré-écriture d'un programme utilisant des boucles *for* classiques en un programme équivalent, mais plus efficace, utilisant des boucles XFOR; (3) un outil appelé XFORGEN, qui génère automatiquement des boucles XFOR à partir de toute représentation

OpenScop de nids de boucles transformée générée automatiquement par un optimiseur automatique (Pluto par exemple).

Comme perspectives, nous proposons de traiter le cas de boucles XFOR non linéaires et de faire des expérimentations sur des benchmarks non polyédriques comme le benchmark SPARKoo [28] par exemple. De plus, une exécution parallèle des différents indices d'un XFOR semble être intéressante. Finalement, rendre le XFOR disponible pour la programmation GPU est une thématique prometteuse, le XFOR pouvant aider considérablement le programmeur à exprimer de façon simple et naturelle des accès complexes aux données.

1 Introduction

1.1 Context

In [29], E. W. Dijkstra wrote: *the programmer finds his task in the field of tension between the available machines and the computations we want to have performed by them. As available machines become more and more powerful, mankind will become more and more ambitious in their applications and programs will grow in size and complexity. It takes no greek prophet to forecast that in the years to come the mechanical execution of the program once it is there will be the minor problem, whereas the major problem will be the process of program composition itself.*

In the last decade, many efforts have been made in providing new programming languages or extensions enabling users to enlarge expressiveness of their source code, in order to take better advantage of the computing resources of new challenging processor architectures. These proposals either focus on providing new features to new or existing programming languages – as Titanium extending the Java language [20], or new languages as Chapel [21] or X10 [22] – or in proposing extensions for transmitting relevant information to the compiler or the runtime system – as for instance non-aliasing of pointers using the `restrict` keyword in C, or by using informative pragmas. Beside their functional relevancy, it is generally difficult to make any proposal a standard for all users who are mostly used to to a given programming language and practice. Past experiences have shown that extensions made on forefront languages are usually more successful, since they can be progressively adopted without abruptly disturbing the usual programming practices. Another advantage is that such extensions push programmers to enlarge their way of reasoning.

On the other hand, recent processor architectures are providing more and more optimization opportunities that the main programming languages are not able to exploit as efficiently as they should in their current releases. A main reason is that new hardware features are provided at the price of less transparency given to the software. A recent major example is obviously the multiplication of cores on a chip, requiring programmers to develop parallel code. While compilers and runtime systems are playing an important role

in improving code efficiency regarding current hardware, it is also crucial to help programmers in writing efficient code thanks to adequate control and data programming structures, that a compiler will then translate into efficient executable code.

Efficient implementations for challenging programs require a combination of high-level algorithmic insights and low-level implementation details. Deriving the low-level details is a natural job for the compiler-computer couple (CCC), but it cannot replace the human insight. Therefore, one of the central challenges for programming languages is to establish a synergy between the programmer and the CCC, exploiting the programmer's expertise to reduce the burden on the CCC. However, programmers must be able to provide their insight effortlessly, using programming structures they can use efficiently. In order to reach both low-level efficiency and programmability simultaneously, a solution is to assist programmers with automatic code transformations that translate into an efficient program what they express at their level of understanding. In the last decades, many efforts have been made in providing automatic program optimization and parallelization software tools that analyze and transform source codes. However, such approaches face three major challenges: fully automatic analysis and transformation is highly complex and can never fully replace the human insight, programming control structures of current mainstream languages are almost never directly addressing the main performance issues of current computers, while super effective codes cannot reasonably be written by programmers due to very convoluted shapes.

Many efforts have been made on automatic optimization and parallelization of programs, providing many interesting progresses. Loops have been addressed extensively since they often represent the most time-consuming parts of a software. A mathematical framework called the polytope model [8] is specifically dedicated to loops characterized by linear loop bounds and memory accesses. Thus, several compiling tools and domain-specific languages for loop nests have been proposed, some of them targeting stencil codes like the Pochoir compiler [4], some others targeting image processing codes like Halide [3] or Polymage [2], or targeting linear loop nests in general like Pluto [16]. These software tools are automatically applying loop transformations like fusion, tiling, loop interchange or skewing, either to improve data locality or to expose parallel loops. However, from a user point of view, such compilers are black boxes that may fail in some circumstances or make some unfortunate choices regarding the selected applied optimization. Indeed, the implemented heuristics cannot always be successful and cannot simultaneously handle all issues that may affect performance.

On the other hand, the user has very few means for influencing the shape of the generated solutions and so, he is obliged to use the generated code as it is. Or, as unique alternative, he may write another code version entirely by hand,

which limits him strongly to affordable and simple code transformations. Thus, a large field of programming strategies is definitely unavailable to the user.

In addition to that, work on iterative and machine learning compilation frameworks [30, 31, 32, 33] are proof of the high complexity of code optimization, even when handling loops exhibiting seemingly simple shapes, as loops with linear bounds and linear memory references that are all in the scope of Pluto [30, 31]. Finally, the ever evolving hardware complexity and the nature of the codes generated by back-end compilers are also important issues preventing automatic optimizers of being wholly foolproof, since they can never address all the possible and forthcoming performance issues simultaneously.

Thus there will always be a significant gap between the runtime performance that may be reached thanks to the best automatic optimizers, and the peak performance that could be expected from an optimized code that is run on a given hardware, whose resources are still underused.

To fill this gap, we propose to make available programming structure to users, enabling them to apply, with relative ease, advanced and efficient optimizing transformations to their codes, while alleviating the related burden thanks to the assistance of automatic code generators.

Following this idea, we propose a computer-assisted control structure called XFOR, helping programmers in addressing directly and accurately three main issues regarding performance: well balanced data locality improvement through generalized loop fusion and loop fission, vectorization and loop parallelization. Two parameters in this structure, the *offset* and the *grain*, afford to adjust precisely the schedule of statements and their interactions regarding data reuse. In addition of being directly usable as a programming structure, XFOR's expressiveness makes it a language for intermediate representation of polyhedral loop transformations.

We implemented the XFOR structure through the development of three software tools: (1) a source-to-source compiler named IBB for Iterate-But-Better!, which automatically translates any C/C++ code containing XFOR-loops into an equivalent code where XFOR-loops have been translated into for-loops. The IBB XFOR support tool takes also benefit of optimizations implemented in the polyhedral code generator CLooG [11] which is invoked by IBB to generate for-loops from an OpenScop specification; (2) an XFOR programming environment named XFOR-WIZARD that assists the programmer in re-writing a program with classical for-loops into an equivalent but more efficient program using XFOR-loops; (3) a tool named XFORGEN, which automatically generates XFOR-loops from any OpenScop representation of transformed loop nests automatically generated by an automatic optimizer.

1.2 Outline

The manuscript is organized as follows ; In Chapter 2, we remind the polyhedral model. In Chapter 3, we enumerate state-of-art languages that are targeting program optimization and performance. After, in Chapter 4, we present the syntax and semantics of XFOR loops by giving illustrative examples. The XFOR source-to-source compiler "IBB" is presented in Chapter 5. Chapter 6 is a key tutorial for an efficient and aware use of XFOR loops. It describes in details different programming strategies that yield for improving program performance. Next, we present the XFOR programming envirement "XFOR-WIZARD" in Chapter 7. Then, in Chapter 8, we show why the fully-automatic code optimizers may suffer from a lack of flexibility in some circumstances and how the XFOR construct helps in filling this gap. After that, in Chapter 9, we describe how to apply and compose in a clear and concise way advanced loop transformations using XFOR structure together with our software tool XFORGEN. And finally, conclusion and perspectives are addressed.

The Polyhedral Model 2

2.1 Introduction

In this Chapter, we first present the theoretical notions of the polyhedral model in Section 2.2. After that, in Section 2.3, we enumerate and describe the most well known polyhedral software tools and libraries for loop optimizations and transformations; such as OpenScop, Clan, Clay, CLooG, Candl, Clint, Pluto and Pluto+. Finally, we exhibit some limitations of the fully automatic polyhedral tools.

2.2 The Polyhedral Model

The polyhedral model (or polytope model) is a mathematical abstraction to analyze programs. It was born from the seminal work of Karp, Miller and Winograd on systems of uniform recurrence equations [34, 35]. This model is particularly useful when analyzing affine loops; loops where all loop bounds and array references are affine functions of the enclosing loop iterators and loop-invariant parameters. A parameter is a symbolic loop nest invariant; the set of parameters often bounds the problem size. In the polytope model, the executions of a statement are represented by a set of integer points, contained in a polyhedron, defined through a conjunction of inequalities. Solutions to an optimization problem are found through linear programming techniques. Thus, it provides a mathematical abstraction of the statements by representing each dynamic instance of a statement by an integer point in a well defined space. Since the model considers each statement instance, it is much closer to the program execution, when compared to other syntactic representations. In the other hand, usual program representations, such as Control Flow Graphs (CFG), Abstract Syntax Trees (AST), are not sufficient to model dependencies. Alias analysis [36] and Iteration Space Graph [37] do not precisely capture the dependencies or rely on exhaustive impractical descriptions.

This section provides a basic overview of the polyhedral model [1, 38, 39]. We start by defining the basic mathematical objects used to characterize polytopes.

Second, we show the utility of scattering functions and access functions to perform polyhedral transformations on the code.

2.2.1 Notations and Definition

In the following, $\mathbb{K}$ denotes an euclidean space.

Definition 2.1 (Affine function). *A function $f : \mathbb{K}^m \to \mathbb{K}^n$ is affine if there exists a matrix $A \in \mathbb{K}^{m \times n}$ and a vector $\vec{b} \in \mathbb{K}^n$, such that:*

$$f(\vec{x}) = A\vec{x} + \vec{b}$$

Definition 2.2 (Affine hyperplane). *An affine hyperplane is an affine $(n-1)$-dimensional subspace of an n-dimensional affine space. For $\vec{c} \in \mathbb{K}^n$ with $\vec{c} \neq \vec{0}$ and a scalar $b \in \mathbb{K}$ an affine hyperplane is the set of all vectors $\vec{x} \in \mathbb{K}^n$, such that:*

$$\vec{c} \cdot \vec{x} = b$$

It generalizes the notion of planes: for instance, a point, a line, a plane are hyperplanes in 1-, 2- and 3- dimensional spaces.

Definition 2.3 (Affine half-space). *A hyperplane divides the space into two half-spaces H_1 and H_2, so that:*

$$H_1 = \{\vec{x} \in \mathbb{K}^n | \vec{c} \cdot \vec{x} \leq b\}$$

and

$$H_2 = \{\vec{x} \in \mathbb{K}^n | \vec{c} \cdot \vec{x} \geq b\}$$

$\vec{c} \in \mathbb{K}^n$ with $\vec{c} \neq \vec{0}$ and $b \in \mathbb{K}$.

Definition 2.4 (Convex polytope or polyhedron). *A convex polytope is the intersection of finite number of half-spaces. We denote $A \in \mathbb{K}^{m \times n}$ a constraints matrix, $b \in \mathbb{K}^n$ a constraints vector and P a convex polytope so that $P \subset \mathbb{K}^n$:*

$$P = \{\vec{x} \in \mathbb{K}^n | A\vec{x} + \vec{b} \geq 0\}$$

Definition 2.5 (Parametric polytope). *A parametric polytope denoted $P(\vec{p})$ is parametrized by a vector of symbols denoted $\vec{p}$. We define $A \in \mathbb{K}^{m \times n}$ a constraints matrix, $B \in \mathbb{K}^{m \times p}$ a coefficient matrix, a vector $\vec{b} \in \mathbb{K}^m$ and P a convex polytope such that $P(\vec{p}) \subset \mathbb{K}^n$:*

$$P(\vec{p}) = \{\vec{x} \in \mathbb{K}^n | A\vec{x} + B\vec{p} + \vec{b} \geq 0\}$$

Definition 2.6 (Polyhedron image). *The image of a polyhedron $\mathcal{P} \in \mathbb{K}^n$ by an affine function f, is another polyhedron $\mathcal{Q} \in \mathbb{K}^m$. Notice that this is true when $\mathbb{K}^n$ is a field, but not if it is a ring like $\mathbb{Z}^n$.*

Definition 2.7 (Perfect loop nest, Imperfect loop nest). *A set of nested loops is called a perfect loop nest iff all statements appearing in the nest appear inside the body of the innermost loop. Otherwise, the loop nest is called an imperfect loop nest. Listing 2.1 and 2.2 show a perfect and an imperfect loop nest respectively.*

```
1 for (i=0 ; i<N ; i++) {
2   for (j=0 ; j<N ; j++) {
3     S (i,j);
4   }
5 }
```

Listing 2.1 – *Perfect Loop Nest*

```
1 for (i=0 ; i<N ; i++) {
2   S1 (i);
3   for (j=0 ; j<N ; j++) {
4     S2 (i,j);
5 } }
```

Listing 2.2 – *Imperfect Loop Nest*

Definition 2.8 (Affine loop nest). *An affine loop nest is a perfect or imperfect loop nest with loop bounds and array accesses that are affine functions of encompassing loop iterators and program parameters.*

2.2.2 Static Control Parts (SCoP)

Definition 2.9 (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.*

In our work we focus on statically analyzable for-loop nests, namely Static Control Parts (SCoPs). A SCoP is defined as a maximal set of consecutive statements, where loop bounds and conditionals are affine functions of the surrounding loop iterators and the parameters (constants whose values are unknown at compilation time). Breaking the control flow with instructions such as `break`, `goto`, `return` is illegal inside a SCoP. The iteration domain of these loops can always be specified thanks to a set of linear inequalities defining a polyhedron [34, 40, 41]. The term polyhedron will be used to denote a set of integer points in a $\mathbb{Z}^n$ vector space bounded by affine inequalities:

$$\mathcal{D}(\vec{p}) = \{\vec{x} | \vec{x} \in \mathbb{Z}^n, A\vec{x} + B\vec{p} + \vec{b} \geq 0\} \tag{2.1}$$

where $\vec{x}$ is the iteration vector (the vector of the loop counter values), A and B are constant matrices, $\vec{p}$ is a vector of parameters and $\vec{b}$ is a constant vector. The iteration domain is a subset of the full possible iteration space: $D(\vec{p}) \subseteq \mathbb{Z}^n$.

A SCoP may be detected automatically [42, 43, 44], or tagged manually with the help of pragmas. Typically, *#pragma SCoP* and *#pragma endSCoP* is used to mark the beginning and end of the SCoP respectively. This annotation is recognized by most of the polyhedral extraction tools such as *Clan* and *Pet*. An example of a valid SCoP is shown in Listing 2.3.

```
#pragma scop
/* C := A*B    E := C*D */
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++) {
    C[i][j] = 0;
    for (k = 0; k < nk; ++k)
      C[i][j] += A[i][k] * B[k][j];
  }
for (i = 0; i < ni; i++)
  for (j = 0; j < nl; j++) {
    E[i][j] = 0;
    for (k = 0; k < nj; ++k)
      E[i][j] += C[i][k] * D[k][j];
  }
#pragma endscop
```

Listing 2.3 – *Example of a Valid Static Control Part (`2mm` Kernel from the Polybench Benchmark Suite [27]).*

2.2.3 Iteration Vector

Definition 2.10 (Iteration vector). *A statement instance coordinates are defined by a vector $\vec{s} \in \mathbb{K}^n$ with n the depth of the enclosing loop nest.*

An iteration vector of a statement is a vector consisting of values of all its enclosing loop iterators. Thus, an occurrence of a statement at loop depth n can be represented by an iteration vector of size n. Each statement is executed once for each possible value of the iteration vector. Thus, the iteration vector represents each dynamic instance of a statement. They can be expressed as follows:

$$\vec{s} = (i_1, i_2, \cdots, i_n)^T$$

where i_k is the k^{th} loop index and n is the loop depth.

2.2.4 Iteration Domain

A compact way to represent all the instances of a given statement is to consider the set of all possible values of its iteration vector. This set is called the iteration domain. It can be conveniently described thanks to all the constraints on the various iterator the statement depends on [42].

For instance, let us consider the statement $S0$ in Figure 2.1. The statement $S0$ is enclosed by a two dimensional loop nest and it is executed only when $j \leq i + 3$. The iteration domain is the set of iteration vectors (i,j). Because of the parameters, we are not able to achieve a precise list of all possible values, it would look similar to this (if $N <= M$):

```
(1,1)   (1,2)   ...   (1,4)
(2,1)   (2,2)   ...   (2,5)
...     ...     ...   ...
(M,1)   (M,2)   ...   (M,N)
```

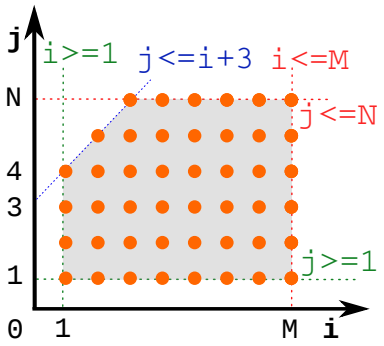
A better way is to express the iteration domain of $S0$ in a concise manner is the following mathematical form:

$$\mathcal{D}_{S0} = \{(i,j) \in \mathbb{Z}^2 | (1 \leq i \leq M) \wedge (1 \leq j \leq N) \wedge (j \leq i+3)\} \tag{2.2}$$

One can see that this iteration domain is a subset of $\mathbb{Z}^2$. It is often illustrated using a graphical representation (see Figure 2.1 (b)). Every integer point (dot in the figure) corresponds to an instance of the statement. M and N are parameters of the loop nest. To express an iteration domain, the bounds are extracted to form a system of inequalities. The resulting constraint matrix encodes a canonical form of the affine inequalities of a loop nest (see Figure 2.2). The polyhedron associated to statement $S0$ has the same dimensionality as the loop nest (2 in this example).

```
1 for (i = 1 ; i <= M ; i++) {
2   for (j = 1 ; j <= N ; j++) {
3     if (j <= i+3) {
4       S0(i,j);
5     }
6   }
7 }
```

(a) *affine* for-*loop nest*



(b) *polyhedron*

Figure 2.1 – *A Loop Nest and its Associated Iteration Domain.*

$$\begin{cases} i-1 & \geq 0 \\ j-1 & \geq 0 \\ -i+M & \geq 0 \\ -j+N & \geq 0 \\ i-j+3 & \geq 0 \end{cases} \qquad \mathcal{D}^{S_0}(M,N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{K}^n \middle| \begin{pmatrix} 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 1 & -1 & 0 & 0 & 3 \end{pmatrix} \begin{pmatrix} i \\ j \\ M \\ N \\ 1 \end{pmatrix} \geq 0 \right\}$$

Figure 2.2 – *Constraint inequalities and the Corresponding Constraint Matrix.*

Definition 2.11 (Iteration domain). *An iteration domain is the set of integer points corresponding to the actual instances of a statement. The iteration domain of a SCoP statement S can be modeled by an n-polytope, $\mathcal{D}^S(\vec{p}) \subset \mathbb{K}^n$, such that:*

$$\mathcal{D}^S(\vec{p}) = \{\vec{x} \in \mathbb{K}^n | A\vec{x} + B\vec{p} + \vec{b} \geq 0\}$$

2.2.5 Scattering Function

The iteration domain does not contain any ordering information: it only describes the set of statement instances but not the order in which they have to be executed relatively to each other. In the past, the lexicographic order of the iteration domain was considered, which is not sufficient (for example when transforming the code). If no ordering information is given, this means that the statement instances may be executed in any order (this is useful, *e.g.*, to specify parallelism), but some statement instances may depend on some others and it may be critical to impose a given order [42]. Hence additional information is needed.

We call "scattering" any kind of ordering information in the Polyhedral Model. There exists many kind of ordering indeed, as allocation, scheduling, chunking etc. Nevertheless they are all expressed in the same way, using logical stamps that can have various semantics.[42]

A very useful example of multi-dimensional scattering functions is the scheduling of the original program. The method to compute it is quite simple [40]. The idea is to build an abstract syntax tree of the program and to read the schedule for each statement. For instance, let us consider the following implementation of the `symm` Kernel benchmark:

```
1 for (i = 0 ; i < ni ; i++)
2   for (j = 0 ; j < nj ; j++) {
3     acc = 0;                                          /* S1 */
4     for (k = 0 ; k < j-1 ; k++) {
5       C[k][j] += alpha*A[k][i]*B[i][j];               /* S2 */
6       acc += B[k][j]*A[k][i];                         /* S3 */
7     }
8     C[i][j] = beta*C[i][j]+alpha*A[i][i]*B[i][j]+alpha*acc; /* S4 */
9   }
```

Listing 2.4 – *Example Schedule (`symm` Kernel).*

$$\begin{cases} \theta_{S_1}(i,j) &= (0,i,0,j,0) \\ \theta_{S_2}(i,j,k) &= (0,i,0,j,1,k,0) \\ \theta_{S_3}(i,j,k) &= (0,i,0,j,1,k,1) \\ \theta_{S_4}(i,j) &= (0,i,0,j,2) \end{cases}$$

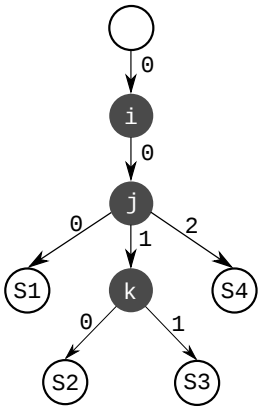


Figure 2.3 – *Abstract Syntax Tree.*

The corresponding abstract syntax tree is given in Figure 2.3. It directly

gives the scattering functions (schedules) for all the statements of the program. The odd dimensions of the schedule represent the textual order, and the even dimensions expresses the relationship with the iterators.

These schedules depend on the iterators and give for each instance of each statement a unique execution date. The scattering of a statement S, denoted by θ_S, can be defined as follows:

$$\theta_S(\vec{x}) = A\vec{x} \tag{2.3}$$

Where $\vec{x}$ is the iteration vector and A is the scattering matrix.

Definition 2.12 (Scheduling Function). *The scheduling function of a statement S, known also as the schedule of S, is a function that maps each dynamic instance of S to a logical date, expressing the execution order between statements:*

$$\forall \vec{x} \in \mathcal{D}^S, \theta_S(\vec{x}) = T\vec{x} + \vec{t}$$

As example let us consider our previous schedule, modifying the schedule of S_3 this way $\theta_{S_3}(i,j,k) = (j+2, 3\times i + j, 2\times k + 3)$ can be expressed by the following scattering matrix:

$$\mathcal{T}_{S_3}\begin{pmatrix} i \\ j \\ k \\ 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 2 \\ 3 & 1 & 0 & 0 \\ 0 & 0 & 2 & 3 \end{pmatrix}\begin{pmatrix} i \\ j \\ k \\ 1 \end{pmatrix}$$

2.2.6 Access Functions

Access functions are functions which map the iteration domain of a statement to the memory locations accessed by the statement. Note that in polyhedral model, the memory accesses are performed through array references. Each statement may access multiple memory locations using multiple arrays. Thus, the data access functions of a statement are the set of all array access functions of the arrays accessed by the statement. They can be represented by:

$$f_{\{R,W\}}(\vec{x}) = F\vec{x} + \vec{f}$$

where F is a matrix $\mathcal{M}^{d\times n}$, d is the dimension of the array, n is the loop depth, $\vec{x}$ is an iteration vector and $\vec{f}$ is a vector of size d. R and W indicate either a Read or a Write from/to the indiced memory location.

```
1 for (i = 0 ; i < N ; i++)
2   for (j = 0 ; j < N ; j++)
3     for (k = 0 ; k < N ; ++k)
4       C[i][j] += alpha * A[i][k] * B[2*k+1];
```

Listing 2.5 – *Affine Loop Nest.*

In Listing 2.5, there are two statements which access five unique memory locations. There are two write accesses (both on $C[i][j]$) and six read accesses (S1 : $C[i][j]$, *beta* and S2: $C[i][j]$, $alpha$, $A[i][k]$, $B[k][j]$). Note that variables such as $alpha$ and *beta*, which are not arrays, are treated as arrays which always operates on the zeroth index. The corresponding access functions are listed below:

$$f_{R^C}(\vec{x}) = f_{W^C}(\vec{x}) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ N \\ 1 \end{pmatrix} \Leftrightarrow C[i][j]$$

$$f_{R^{alpha}}(\vec{x}) = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ N \\ 1 \end{pmatrix} \Leftrightarrow \&alpha[0]$$

$$f_{R^A}(\vec{x}) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & k & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ N \\ 1 \end{pmatrix} \Leftrightarrow A[i][k]$$

$$f_{R^B}(\vec{x}) = \begin{pmatrix} 0 & 0 & 2 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ N \\ 1 \end{pmatrix} \Leftrightarrow B[2 * k + 1]$$

2.2.7 Data Dependence

Definition 2.13 (Data Dependence)**.** *There is a data dependence whenever two statements $S_i(\vec{x_i})$ and $S_j(\vec{x_j})$ access the same memory location, and at least one access is a write.*

If in the original sequential order $S(\vec{x})$ is executed before $R(\vec{x})$, R is said to be dependent on S. Under these circumstances, S is called the source and R is the destination of the dependence. To preserve the semantics, the execution of two dependent statements must be the same in the original sequential and in the transformed parallel order. On the other hand, two independent statements can be executed in arbitrary order. Dependencies are classified in three categories, depending on the order of read and write operations:

- **RAW:** Read After Write, or *Flow dependence*.
- **WAR:** Write After Read, or *Anti-dependence*.

- **WAW:** Write After Write, or *Output dependence*.

Note that **RAR**, read-after-read, is not considered as a dependence, since the memory is not altered, hence the order of execution of the two read operations is arbitrary. However, in some optimizations, RAR dependencies are considered for improving data locality. Anti-dependencies and output dependencies can be alleviated by various algorithms such as privatization [45], renaming, scalar expansion [46] or array expansion [47], such that the constraints are relaxed and more optimizing techniques become legal.

Formally, the existence of dependencies can be expressed using the *Bernstein* Conditions. Let S_1 and S_2 represent two statements. Let $\mathcal{R}(S_i)$ ($\mathcal{W}(S_i)$) represent the set of memory locations read (written) by the statement S_i respectively. Assume that there exists a feasible execution path from S_1 to S_2. The condition for each dependence can be represented as

- Flow dependence: $\mathcal{W}(S_1) \cap \mathcal{R}(S_2) \neq \emptyset$
- Anti-dependence: $\mathcal{R}(S_1) \cap \mathcal{W}(S_2) \neq \emptyset$
- Output dependence: $\mathcal{W}(S_1) \cap \mathcal{W}(S_2) \neq \emptyset$

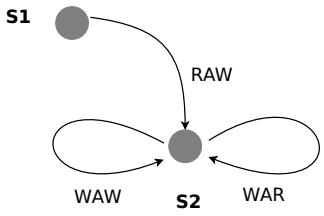
Thus, the statements S_1 and S_2 are dependent when:

$$[\mathcal{W}(S_1) \cap \mathcal{R}(S_2)] \cup [\mathcal{R}(S_1) \cap \mathcal{W}(S_2)] \cup [\mathcal{W}(S_1) \cap \mathcal{W}(S_2)] \neq \emptyset \tag{2.4}$$

A legal schedule combines transformations so that the sequential execution order of dependencies is respected. A loop-carried dependence occurs when two different iterations of the loop access the same memory location. A data dependence graph defines the statement dependencies, so that vertices represent the source and the target statements, and edges the inter- and intra-statement dependencies (see Figure 2.4).

```
1 for (i = 0 ; i < N ; i++) {
2   A[i] += 10;                //S1
3   for (j = 0; i < N; j++)
4     B[i] = A[i]+B[i]+i+j; //S2
5 }
```

(a) *code example*



(b) *dependence graph*

Figure 2.4 – *A Loop Nest and its Corresponding Dependence Graph.*

An edge e can be translated into a *dependence polyhedron* $\mathcal{P}_e$ [48], containing the exact instance dependencies. In other words, a dependence polyhedron expresses the dependence relation between two statements. Note that P_e is a subset of the cartesian product of the iteration domains. Let us denote S_i, the *source* and S_j the *target* of the dependence. A valid formulation of a dependence polyhedron must

respect the lexicographic order, $\theta^{S_i}(\vec{x_i}) \ll \theta^{S_j}(\vec{x_j})$. Also, the instances generating the dependence must exist, $\vec{x_i} \in \mathcal{D}^{S_i}$ and $\vec{x_j} \in \mathcal{D}^{S_j}$. Eventually, the memory accesses must touch the same memory location, $f^{S_i}(\vec{x_i}) = f^{S_j}(\vec{x_j})$. The code presented in Listing 2.4 (a) contains a flow-dependence, between $\mathcal{W}_{S_1}(A[i])$ and $\mathcal{R}_{S_2}(A[i])$. After computing the *access domain*, that is to say the image of the accesses, the following dependence polyhedron describes this dependence:

$$\mathcal{P}_e = \left\{ ((i_1),(i_2,j_2)) \middle| \begin{matrix} 0 \leq i_1, i_2, j_2 \leq N \\ i_1 = i_2 \end{matrix} \right\}$$

2.2.8 Dependence Vectors

Let us consider one loop L containing two statements S and R. If $R(\vec{x_R})$ or simply $(R(\vec{j}))$ depends on $S(\vec{x_S})$ or simply $(S(\vec{i}))$, then $S(\vec{i})$ must be executed before $R(\vec{j})$. In loop dependence analysis [49], it is said that "*the iteration j of loop L depends on iteration i*". Hence, R depend on S with:

- the *distance vector* $\vec{d}$ = $\vec{j}$ - $\vec{i}$
- the *direction vector* σ = **sign**($\vec{d}$)
- at level l = **lev**($\vec{d}$)

Distance Vector. A distance vector indicates the distance of the dependence, i.e. the number of iterations between the source and the target statement. Distance vectors can be directly related to data reuse. A lower distance indicates that the number of iterations between the source and target statements is small, resulting in a better temporal cache reuse. A distance vector between the source statement $S(\vec{x_s})$ and the target statement $T(\vec{x_t})$, in a loop of depth n can be represented by an n dimensional vector as follows:

$$\vec{d} = \vec{x_t} - \vec{x_s}$$

```
1 for (i = 0 ; i < N ; i++)
2   for (j = 0 ; j < N ; j++)
3     A[i][j] = (A[i][j-1] + A[i+2][j]) / 2.0; //So
```

Listing 2.6 – *Example to Illustrate Distance Vectors and Direction Vectors*

The distance vectors representing the *RAW* and *WAR* dependencies of statement S_0 in Listing 2.6 are shown below:

- RAW : $\vec{d_1} = (0,1)$
- WAR : $\vec{d_2} = (2,0)$

Direction Vector. Direction vectors just show the direction of the dependence. They can be obtained just by considering the sign of the direction vectors. The direction vector $\vec{d} = (d_1, d_2, d_3, ..., d_n)$ between the source statement $S(\vec{x}_s)$ and the target statement $T(\vec{x}_t)$, in a loop of depth n can be represented by an n dimensional vector as follows:

$$\textbf{sign}(\vec{d}) = (sign(d_1), sign(d_2), ..., sign(d_n))$$

The sign of an integer i, denoted σ is:

$$sign(i) = \begin{cases} 1, & if \;\; i > 0 \\ -1, & if \;\; i < 0 \\ 0, & if \;\; i = 0 \end{cases}$$

The direction vectors representing the *RAW* and *WAR* dependencies of statement S_0 in Listing 2.6 are shown below:

- RAW : $\sigma_1 = (0, 1)$
- WAR : $\sigma_2 = (1, 0)$

The direction vectors contain less information when compared to distance vectors. However, some transformations such as loop interchange or parallelization only requires direction vectors. Moreover, direction vectors are convenient in case distance vectors cannot summarize the dependencies.

Level. Given that m is the depth of the loop L, for a distance vector $\vec{d} = (d_1, d_2, ..., d_m)$, the leading element is the first non-zero element. If this is d_l, then l represents the level of $\vec{d}$ and $1 \leq l \leq m$. Also, the level of the zero vector is defined to be $m + 1$. The vector $\vec{d}$ is said to be lexicographically positive or negative if its leading element is positive or negative, respectively.

A distance vector or a direction vector of a dependence must always be lexicographically non-negative. Thus, considering the dependence vector $\vec{d}$ at level l, of statements S and T, $l \in 1, 2, ..., m + 1$. If $1 \leq l \leq m$ we say that the dependence of T on S is carried by the loop of depth l. The dependence of T on S is loop independent (not carried by any loop) if $l = m + 1$, equivalent to $\vec{d} = \vec{0}$.

2.2.9 Polyhedral Transformation

Program transformations [38, 50] in the polyhedral model can be specified by well chosen scheduling functions. They modify the source polyhedra into a target polyhedra containing the same integer points but in a new coordinate system, thus with a different lexicographic order.

Definition 2.14 (Transformation). *A transformation maps a statement instance logical date to a date in the transformed space, with respect to the original program semantics.*

Transformations are typically applied to enhance data locality, reduce control code and expose or improve the degree of parallelism. The new schedule $\Theta^S \vec{x}$ might express a new execution order on statement S.

A one-dimensional affine transformation [1] for statement S is an affine function defined by:

$$\begin{aligned} \phi_S(\vec{i}) &= (c_1^S \; c_2^S \; ... \; c_{m_S}^S)(\vec{i_S}) + c_0^S \\ &= (c_1^S \; c_2^S \; ... \; c_{m_S}^S) \begin{pmatrix} \vec{i_S} \\ 1 \end{pmatrix} \end{aligned} \tag{2.5}$$

where $c_1,\ c_2\ ...,\ c_{m_S} \in \mathbb{Z}, \vec{i} \in \mathbb{Z}^{m_S}$
Hence, a one-dimensional affine transform for each statement can be interpreted as a partitioning hyperplane with normal vector $(c_1,\ c_2\ ...,\ c_{m_S})$. A multi-dimensional affine transformation [1] can be represented as a sequence of such ϕ's for each statement. We use a superscript to denote the hyperplane for each level. ϕ_S^k represents the hyperplane at level k for statement S. If $1 \leq k \leq d$, all the ϕ_S^k can be represented by a single d-dimensional affine function $\mathcal{T}_S$ given by:

$$\mathcal{T}_S \vec{i_S} \;=\; M_S \vec{i_S} + \vec{t_S} \tag{2.6}$$

where $M_S \in \mathbb{Z}^{d \times m_S}$, $\vec{t_S} \in \mathbb{Z}^d$

$$\mathcal{T}_S(\vec{i}) = \begin{pmatrix} \phi_S^1(\vec{i}) \\ \phi_S^2(\vec{i}) \\ \vdots \\ \phi_S^d(\vec{i}) \end{pmatrix} = \begin{pmatrix} c_{11}^S & c_{12}^S & ... & c_{1m_S}^S \\ c_{21}^S & c_{22}^S & ... & c_{2m_S}^S \\ \vdots & \vdots & \vdots & \vdots \\ c_{d1}^S & c_{d2}^S & ... & c_{dm_S}^S \end{pmatrix} \vec{i_S} + \begin{pmatrix} c_{10}^S \\ c_{20}^S \\ \vdots \\ c_{d0}^S \end{pmatrix} \tag{2.7}$$

Scalar dimensions. [1] The dimensionality of $\mathcal{T}_S$, d, may be greater than m_S as some rows in $\mathcal{T}_S$ serve the purpose of representing partially fused or unfused dimensions at a level. Such a row has $(c_1,\ c_2\ ...,\ c_{m_S}) = 0$, and a particular constant for c_0 . All statements with the same c_0 value are fused at that level and the unfused sets are placed in the increasing order of their c_{0^S}. We call such a level a **scalar dimension**. Hence, a level is a scalar dimension if the ϕ's for all statements at that level are constant functions. Figure 2.5 shows a sequence of matrix-matrix multiplies and how a transformation captures a legal fusion: the transformation fuses *ji* of $S1$ with *jk* of $S2$; ϕ^3 is a scalar dimension.

Complete scanning order. [1] The number of rows in M_S for each statement should be the same (d) to map all iterations to a global space of dimensionality d. To provide a complete scanning order for each statement, the number of linearly independent ϕ_S's for a statement should be the same as the dimensionality of the statement, m_S , *i.e.*, $\mathcal{T}_S$ should have full column rank. Note that it is always

possible to represent any transformed code (any nesting) with at most $2m_S^* + 1$ rows, where $m_S^* = max_{S \in \mathbf{S}} m_S$.

```
for (i = 0; i < n; i++)
 for (j = 0; j < n; j++)
  for (k = 0; k < n; ++k) {
   S1: C[i][j]+=A[i][k]*B[k][j];
  }
for (i = 0; i < n; i++)
 for (j = 0; j < n; j++)
  for (k = 0; k < n; ++k) {
   S2: E[i][j]+=C[i][k]*D[k][j];
  }
```

(a) *original code*

```
for (to = 0; to < n; to++) {
 for (t1 = 0; t1 < n; t1++) {
  for (t3 = 0; t3 < n; ++t3) {
   S1: C[to][t1]+=A[to][t3]*B[t3][t1];
  }
  for (t3 = 0; t3 < n; ++t3) {
   S2: E[to][t1]+=C[to][t3]*D[t3][t1];
  }
 }
}
```

(b) *transformed code*

$$\mathcal{T}_{S_1}(\vec{i}_{S_1}) = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\mathcal{T}_{S_2}(\vec{i}_{S_2}) = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

i.e.,

$$\begin{array}{ll} \phi^1_{S_1} = j & \phi^1_{S_2} = j \\ \phi^2_{S_1} = i & \phi^2_{S_2} = k \\ \phi^3_{S_1} = 0 & \phi^3_{S_2} = 1 \\ \phi^4_{S_1} = k & \phi^4_{S_2} = i \end{array}$$

Figure 2.5 – *Polyhedral Transformation: an Example [1].*

2.3 Polyhedral Tools

2.3.1 OpenScop (SCoPLib)

OpenScop [7] is an open specification that defines a file format and a set of data structures to represent a static control part (SCoP for short), *i.e.*, a program part that can be represented in the polyhedral model. The goal of OpenScop is to provide a common interface to the different polyhedral compilation tools in order to simplify their interaction. OpenScop builds on previous popular polyhedral file and data structure formats, such as .cloog and CLooG data structures, to provide a unique, extensible format to most polyhedral compilation tools. The OpenScop representation is used in our software tools (IBB, XFOR-WIZARD and XFORGEN). The building of the OpenScop encoding and its use is described in details in the manuscript. It is composed of two parts (see Figure 2.6). The first part, the so-called *core* part, is devoted to the polyhedral representation of a

SCoP. It contains what is strictly necessary to build a complete source-to-source framework in the polyhedral model and to output a semantically equivalent code for the SCoP, from analysis to code generation. The second part of the format, the so-called *extension* part, contains extensions to provide additional information to some tools.

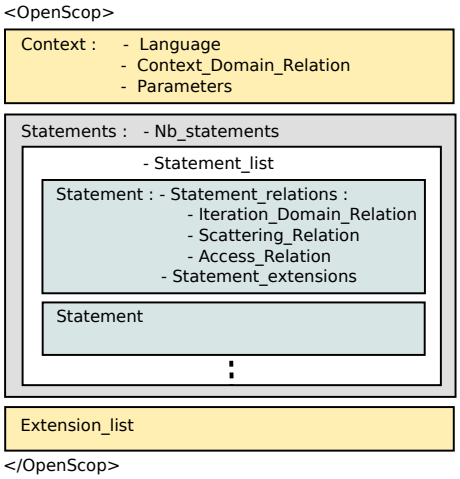


Figure 2.6 – *OpenScop Data Structure.*

As Figure 2.6 shows, the *core* part is composed by the 'Context' and the 'Statements':

- 'Context' represents the global information of the SCoP. It consists on the target language, the global constraints on the parameters (represented as a relation) and optionally the parameter names which may be necessary for the code generation process.
- 'Statements' represents the information about the statements. 'Nb_statements' is the number of statements in the SCoP, i.e. the number of 'Statement' items in the 'Statement_list'. 'Statement' represents the information on a given statement. To each statement is associated a list of relations and, optionaly, a list of statement extensions. The list of relations may include one iteration domain, one scattering relation and several access relations. The statement extensions is an optional information. It starts with a integer which specifies the number of extensions provided.

The 'Extension_list' represents the *extension* part and may contain an arbitrary number of generic informations.

We show in Subsection 2.3.2 an OpenScop description example automatically generated by the software tool Clan from the loop nest represented in Listing 2.7.

2.3.2 Clan

Clan [9, 42] is a free software and library that translates some particular parts of high level programs written in *C*, *C++*, *C#* or Java into a polyhedral representation. This representation may be manipulated by other tools to, e.g., achieve complex program restructuring (for optimization, parallelization or any other kind of manipulation). It has been created to avoid tedious and error-prone input file writing for polyhedral tools (such as CLooG [11, 12, 13, 51], LeTSeE [52], Candl [14] etc.)

Clan stands for *Chunky Loop ANalyzer* : it is a part of the Chunky project, a research tool for data locality improvement [42]. It is designed to be the front-end of any source-to-source automatic optimizers and/or parallelizers. The OpenScop output format has been chosen to be polyhedral library independent, so Clan may integrate any polyhedral compilation framework easily. Clan has been successfully integrated to PoCC [31], Pluto [16, 17, 18] high-level compilers. It is also integrated to our tools XFOR-WIZARD (Chapter 7) and XFORGEN (Chapter 9).

```
1 #pragma scop
2 for (i = 2; i < n - 1; i++)
3   B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
4 for (j = 2; j < n - 1; j++)
5   A[j] = B[j];
6 #pragma endscop
```

Listing 2.7 – *FOR-Loop Nest example* `jacobi-1d-imper` *Kernel.*

We show below the OpenScop representation, automatically generated by Clan, corresponding to the code in Listing 2.7.

```
<OpenScop>
# ============================================== Global
# Language
 C
# Context
CONTEXT
0 3 0 0 0 1
# Parameters are provided
1
<strings>
 n
</strings>
# Number of statements
2
# ============================================== Statement 1
# Number of relations describing the statement:
6
# ----------------------------------------------  1.1 Domain
 DOMAIN
3 4 1 0 0 1
# e/i|  i |  n |  1
   1    1    0   -2    ## i-2 >= 0
```

```
   1   -1    1   -2    ## -i+n-2 >= 0
   1    0    1   -4    ## n-4 >= 0
# ----------------------------------------------  1.2 Scattering
 SCATTERING
3 7 3 1 0 1
# e/i| c1   c2   c3 |  i |  n |  1
   0   -1    0    0    0    0    0    ## c1 == 0
   0    0   -1    0    1    0    0    ## c2 == i
   0    0    0   -1    0    0    0    ## c3 == 0
# ----------------------------------------------  1.3 Access
 WRITE
2 6 2 1 0 1
# e/i| Arr  [1]|  i |  n |  1
   0   -1    0    0    0    3    ## Arr == B
   0    0   -1    1    0    0    ## [1] == i
 READ
2 6 2 1 0 1
# e/i| Arr  [1]|  i |  n |  1
   0   -1    0    0    0    4    ## Arr == A
   0    0   -1    1    0   -1    ## [1] == i-1
 READ
2 6 2 1 0 1
# e/i| Arr  [1]|  i |  n |  1
   0   -1    0    0    0    4    ## Arr == A
   0    0   -1    1    0    0    ## [1] == i
 READ
2 6 2 1 0 1
# e/i| Arr  [1]|  i |  n |  1
   0   -1    0    0    0    4    ## Arr == A
   0    0   -1    1    0    1    ## [1] == i+1
# ------------------------------------  1.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
1
# List of original iterators
i
# Statement body expression
 B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
</body>
# ================================================ Statement 2
# Number of relations describing the statement:
4
# ----------------------------------------------  2.1 Domain
 DOMAIN
3 4 1 0 0 1
# e/i|  j |  n |  1
   1    1    0   -2    ## j-2 >= 0
   1   -1    1   -2    ## -j+n-2 >= 0
   1    0    1   -4    ## n-4 >= 0
# ----------------------------------------------  2.2 Scattering
 SCATTERING
3 7 3 1 0 1
# e/i| c1   c2   c3 |  j |  n |  1
   0   -1    0    0    0    0    1    ## c1 == 1
   0    0   -1    0    1    0    0    ## c2 == j
   0    0    0   -1    0    0    0    ## c3 == 0
# ----------------------------------------------  2.3 Access
```

```
 WRITE
2 6 2 1 0 1
# e/i| Arr  [1]|  j |  n |  1
   0   -1    0    0    0    4    ## Arr == A
   0    0   -1    1    0    0    ## [1] == j
 READ
2 6 2 1 0 1
# e/i| Arr  [1]|  j |  n |  1
   0   -1    0    0    0    3    ## Arr == B
   0    0   -1    1    0    0    ## [1] == j
# ------------------------------------  2.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
1
# List of original iterators
j
# Statement body expression
 A[j] = B[j];
</body>
# ============================================== Extensions
<scatnames>
 b0 i b1
</scatnames>
<arrays>
# Number of arrays
5
# Mapping array-identifiers/array-names
1 i
2 n
3 B
4 A
5 j
</arrays>
<coordinates>
# File name
file.c
# Starting line and column
2 0
# Ending line and column
6 0
# Indentation
0
</coordinates>
</OpenScop>
```

2.3.3 Clay

Clay (*Chunky Loop Alteration wizardrY*) [10] is a free software and library devoted to semi-automatic optimization using the polyhedral model. It can input a high-level program or its polyhedral representation and transform it according to a transformation script. Classic loop transformations primitives are provided (fusion, fission, skewing, strip mining, tiling, unrolling etc.). Clay is able to

check for the legality of the complete sequence of transformation and to suggest corrections to the user if the original semantics is not preserved.

2.3.4 CLooG

CLooG [11, 12, 13, 51] is a free software and library to generate code for scanning Z-polyhedra. That is, it finds a code (e.g. in C, FORTRAN...) that reaches each integral point of one or more parameterized polyhedra. CLooG has been originally written to solve the code generation problem for optimizing compilers based on the polytope model. Nevertheless it is used now in various area e.g. to build control automata for high- level synthesis or to find the best polynomial approximation of a function. CLooG may help in any situation where scanning polyhedra matters. While the user has full control on the generated code quality, CLooG is designed to avoid control overhead and to produce a very effective code.

CLooG stands for *Chunky Loop Generator* : it is a part of the Chunky project, a research tool for data locality improvement [53]. It is designed also to be the back-end of automatic parallelizers like LooPo [54]. Thus it is very compilable code oriented and provides powerful program transformation facilities. Mainly, it allows the user to specify very general schedules where, e.g., unimodularity or invertibility of the transformation doesn't matter. The CLooG library is used in our source-to-source compiler IBB (see Chapter 5 [*XFOR Code Generation : the IBB Compiler*], page 71, for more details).

2.3.5 Candl

Candl (*Chunky ANalyzer for Dependencies in Loops*) [14] is a free software and a library devoted to data dependencies computation. It has been developed to be a basic bloc of our optimizing compilation tool chain in the polyhedral model. From a polyhedral representation of a static control part of a program, it is able to compute exactly the set of statement instances in dependence relations. Hence, its output is useful to build program transformations respecting the original program semantics. This tool has been designed to be robust and precise. It implements some usual techniques for data dependence removal, such as array privatization or array expansion, offers simplified abstractions like dependence vectors and performs violation dependence analysis. Candl library is used in our programming environment XFOR-WIZARD, details are given in Chapter 7 ([*The XFOR Programming Environment XFOR-WIZARD*], page 97).

2.3.6 Clint

Clint (*Chunky Loop INTeraction*) [15] is a direct manipulation interface designed to (i) help programmers with parallelizing compute-intensive programs parts;

(ii) ease the exploration of possible transformations; and (iii) guarantee the correctness of the final code. Clint leverages the geometric nature of the polyhedral model by presenting code statements, their instances, and dependencies in a scatterplot-like visualization of iteration domains similar to those used in the literature on polyhedral compilation. By making the visualization interactive, it reduces parallelism extraction and code transformation tasks to visual pattern recognition and geometrical manipulations, giving a way to manage the complexity of the underlying model.

2.3.7 Pluto

PLUTO (*An automatic parallelizer and locality optimizer for affine loop nests*) [16, 17, 18] is an automatic parallelization tool based on the polyhedral model. Pluto transforms C programs from source to source for coarse-grained parallelism and data locality simultaneously. The core transformation framework mainly works by finding affine transformations for efficient tiling. The scheduling algorithm used by Pluto has been published in [16]. OpenMP parallel code for multicores can be automatically generated from sequential C program sections. Outer (communication-free), inner, or pipelined parallelization is achieved purely with OpenMP parallel for pragmas; the code is also optimized for locality and made amenable to auto-vectorization. Though the tool is fully automatic (C to OpenMP C), a number of options are provided (both command-line and through meta files) to tune aspects like tile sizes, unroll factors, and outer loop fusion structure. Cloog-ISL is used for code generation. The Pluto library is integrated in our software tool XFORGEN, details are given in Chapter 9 ([*XFOR Polyhedral Intermediate Representation and Manual-Automatic Collaborative Approach for Loop Optimization*], page 135).

2.3.8 PLUTO+

Pluto+ [19] is a framework that addresses some limitations of Pluto. In fact, the Pluto algorithm, that include a cost function for modern multicore architectures where coarse-grained parallelism and locality are crucial, consider only a sub-space of transformations to avoid a combinatorial explosion in finding the transformations. The ensuing practical tradeoffs lead to the exclusion of certain useful transformations, in particular, transformation compositions involving loop reversals and loop skewing by negative factors. Acharya and Bondhugula in [19], propose an approach to address this limitation by modeling a much larger space of affine transformations in conjunction with the Pluto algorithm's cost function.

2.3.9 Limits of Automatic Optimizers

The polyhedral model is very powerful and expressive, but it has a limitation; it only deals with affine loop nest. The XFOR construct helps in relaxing this constraint by permitting the user to use array elements or function call in the loop bounds. In addition to that, it is possible for the user to write any instruction in the XFOR-loop body.

Automatic code optimizers and compilers based on the polyhedral model, can automatically extract the polyhedral regions of a code and can produce a fairly good schedule. Several approaches have been tried in the past, however, the search space of the valid transformations is very huge, and thus these automated approaches relies on some heuristics which may result in weak performance of the generated code. One such heuristics is to specify the scheduling in the form of a linear optimization problem and to solve it. The state of the art automated compiler Pluto [16] uses this approach. One other prominent heuristic is to use machine learning, and is studied in [31]. Iterative optimization approaches can be used, to select the best transformation from a set of possibly good ones. Even in this case, the original set of transformations may not contain the best transformation. Thus, the complexity of code optimization, even when handling loops exhibiting seemingly simple shapes, such as loops with linear bounds and linear memory references, in addition to the ever evolving hardware complexity and the nature of the codes generated by back-end compilers, prevent the automatic optimizers of being foolproof. Hence, even today, performance critical applications are often either completely manually written or fine tuned from an automated output.

In Chapter 8 ([*XFOR and Automatic Optimizers*], page 115), we give a detailed study of some performance issues, and we show that automated approaches may suffer from some limitation such as (i) Insufficient data locality optimization. (ii)Excess conditional branches in the generated code. (iii) Too verbose code with too many machine instructions. (iv) Aggressive Data locality optimization resulting in processor stalls. (v) Missed vectorization opportunities.

Apart from these performance issues, one commonly overlooked limitation of automated compilers, especially source-to-source compilers is code maintainability. The original code is written to express the inherent algorithm. After the automated compiler optimized the code, it becomes extremely difficult to trace back the original meaning. If the source code needs to be debugged or improved, the original implementation has to be maintained. The same problem also applies with full manual optimizations. The code containing transformations such as skewing, tiling, alleging for vectorization and optimized for control is extremely difficult to read.

Our proposed solution *XFOR* overcomes all these limitations. The proposed syntax allows the programmer to express the algorithm in an intuitive and a

meaningful way while still exposing the key polyhedral parameters to tune for performance. By directly including the polyhedral parameters, finding a good schedule or changing a schedule is greatly simplified. Additionally, for scheduling, automatic compilers can be used, and the result can then automatically be converted to the XFOR format for easy hand tuning, thus combining the benefits of automated and manual approaches thanks to our software tool XFORGEN (see Chapter 9 [*XFOR Polyhedral Intermediate Representation and Manual-Automatic Collaborative Approach for Loop Optimization*], page 135).

2.4 Conclusion

The polyhedral framework is a very efficient mathematical abstraction for loops. For decades, it has shown its efficiency, despite its limitations, via several automated polyhedral tools that helped the programmer in improving the performance of their program without any engineering cost.

In the next chapter, we present other loop optimizing strategies; which consists in making available for the programmer; (1) new general or *domain specific* optimized languages, or (2) extending existing languages using libraries, in order to produce more efficient code and get better performance.

Related Work

3

3.1 Introduction

In this chapter, we expose the state of the art in proposed languages and frameworks which objectives are related to optimization and performance. In Section 3.2, we address some general-purpose languages proposing specific loop structures. Secondly in Section 3.3, we describe some DSL (*Domain Specific Languages*). Finally in Section 3.4, we present some additional related work dealing with loop optimizations.

3.2 Partitioned Global Address Space Languages

The *Partitioned Global Address Space* (PGAS) model [55] is a parallel programming model designed to improve programmer productivity while aiming for high performance. Before PGAS, *High Performance Computing* (HPC) programming models could be clustered into two main groups: message-passing models such as MPI [56], where isolated processes with isolated memories exchange messages, and shared-memory models, as exemplified by OpenMP [26], where multiple threads can read and write a shared memory. The PGAS model can be situated in-between both models. From the shared-memory model, it inherits the idea that a parallel program operates on one single memory that is conceptually shared among all its processes. From the message-passing model, it inherits the idea that communication between processes is associated with a certain cost.

In this section, we present two representative PGAS languages which are X10 and Chapel since they introduce particular looping structures.

3.2.1 X10

X10 [22, 55] is a programming language being developed by IBM Research [57]. The name X10 refers to *times 10*, the aim of the language to achieve 10 times more productivity in HPC software development. X10 is described as a modern object-oriented programming language providing an asynchronous PGAS programming model with the goal of enabling scalable parallel programming

for high-end computers. X10 extends the PGAS model with asynchronicity by supporting lightweight asynchronous activities and enforcing asynchronous access to non-local state. Its explicit fork/join programming abstractions and a sophisticated type system are meant to guide the programmer to write highly parallel and scalable code, while making the cost of communication explicit. The task parallelism is implemented on top of a work-stealing scheduler.

Loops in X10. X10 [58, 59] has classes and methods, assignments and method invocation, and the usual control constructs: conditionals and loops. X10 supports the usual sort of for loop. The body of a for loop may contain break and continue statements just like while loops. Here is an example of an explicitly enumerated for loop:

```
for (var x:Int=0; x < a.size; ++x)
  result += a(x);
```

X10 also supports enhanced for loops [59]. The for loop may take an index specifier v in r, where r is any value that implements `x10.lang.Iterable[T]` for some type T. The body of the loop is executed once for every value generated by r, with the value bound to v. An example is shown below:

```
for (v in a.values())
  result += v;
```

Of particular interest is `IntRange`. The expression e_1 .. e_2 produces an instance of `IntRange` from l to r if e_1 evaluates to l and e_2 evaluates to r. Used in loops , it allow to enumerate all the values (if any) from l to r (inclusive). Thus we can sum the numbers from 0 to N in the following way:

```
for (v in 0..N)
  result +=v;
```

One can iterate over multiple dimensions at the same time using Region. A Region is a data-structure that compactly represents a set of points. For instance, the region (0..5)*(1..6) is a 2-d region of points (x,y) where x ranges over 0..5 and y over 1..6. (The bounds are inclusive.) The natural iteration order for a region is lexicographic. Thus one can sum the coordinates of all points in a given rectangle in the following way:

```
for ([x,y] in (0..M)*(0..N))
  result += x+y;
```

Here the syntax [x,y] is said to be a destructuring syntax [59]; it destructures a 2-d point into its coordinates x and y. One can write p[x,y] to bind p to the entire point.

On one hand, X10 loops may be re-written using the XFOR construct. However, it would require the programmer to define domains compositions and handle related code modifications, such as indices substitutions and scheduling of the statements. On the other hand, expressing XFOR features (offset and grain, loop transformations) using X10 language is very complicated and almost impossible since this language compiler do not take advantage of polyhedral modeling and optimizations. Moreover, weak performance of the code generated by their compiler has been reported, compared to standard parallel languages as OpenMP.

3.2.2 Chapel

Chapel which stands for *Cascade High-Productivity Language* [55, 60] is developed by Cray as part of the Cray Cascade project. Chapel provides concepts for multithreaded and locality-aware parallel programming. The language also supports many concepts from object-oriented languages and generic programming.

Loops in Chapel. In Chapel, for loops have the following syntax:

```
for index-expression in iterable-expression {
  statements
}
```

The "`iterable-expression`" can either be a predefined collection such as an array or a *tuple*, or a *range* generated on the fly. Here is an example where the two for loops represented in the following code are equivalent:

```
var A: [1..length] string = ("a", "b", "c", "d", "e");

for i in {1..length}
  write(A(i));

for i in A
  write(i);
```

Zipper Iteration. zipper-for-loop has the following Syntax:

```
for index-expression in ( iteratable-exprs ) { stmt-list }
```

Zipper iteration is interpreted as an iterator that iterates over all yielded indices pair-wise. For example, in the following for loop, *i* iterates over (1,5), (2,6) and (3,7).

```
for i in (1..3, 5..7) { ... }
```

Tensor Iteration. Tensor-for-loop has the following Syntax:

```
for index-expr in [ iteratable-exprs ] { stmt-list }
```

Tensor iteration is over all pairs of yielded indices. For example, in the following for loop, *i* iterates over (1,5), (1,6), (1,7), (2,5), (2,6), (2,7), (3,5), (3,6) and (3,7).

```
for i in [1..3, 5..7] { ... }
```

Chapel loops can be re-witten using XFOR loops. However, it would require the programmer to define explicitly the domains and handle indices substitutions. Zipper iteration are similar to nested XFOR loops, but the programmer would have to enumerate all the possible values at each XFOR-level. The XFOR construct enables complicated schedule and transformations that would be difficult to express using the Chapel language. Moreover, the Chapel compiler does not take advantage of polyhedral modeling and optimizations.

3.3 Domain Specific Languages (DSL)

3.3.1 Polymage

Polymage [2] is a DSL for image processing applications based on the polyhedral model. Many image processing can be viewed as pipelines consisting of several interconnected processing stages. Polymage allows the user to express the common image processing computation patterns, such as, point wise operations, stencils, up-sampling and down-sampling, histograms, and time-iterated methods. Functions can be defined by a list of cases where each case construct takes a condition and an expression as arguments. Conditions can be used to specify constraints involving variables, function values, and parameters. The compiler takes as input the program specified in Polymage syntax and the names of the live out functions. The image processing pipelines are represented as a directed acyclic graph (DAG), where each node represents each stage specified by the user and the edges represent the dependencies. The polyhedral representation is then extracted from the specification. A set of criteria are then applied to select a tiling strategy and the code generator generates the corresponding C++ codes. Automatic tuning is then performed to choose the right tile size (see Figure 3.1). When compared to our approach, Polymage is limited and specialized to image processing applications, whereas XFOR is much more general. In addition to that, Polymage specification is very complicated and not very understandable.

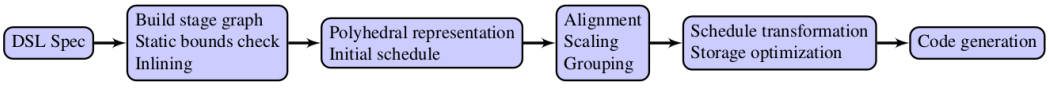


Figure 3.1 – *Phases of the PolyMage compiler [2].*

3.3.2 Halide

Halide [3] is a language and compiler for image processing pipeline and is capable of generating code for x86, ARM and CUDA code. It offers trade-offs between locality, parallelism, and redundant recomputation in stencil pipelines. The user provides the Halide compiler with program written in Halide syntax and a scheduling representation (see Figure 3.2). Reduction operations can be specified by an initial value, the reduction function and the reduction domain. A loop synthesizer based on range analysis is used for data parallel pipelines and the code generator produces the output code. The synthesizer first produces a complete loop nest and allocations from the input specification. Then bounds of each dimension is obtained in a recursive manner, by interval analysis of the expressions in the caller which index that dimension. Then the compiler tries to perform sliding window optimization and storage folding to improve data reuse. After vectorization and unrolling optimizations the final code is generated. An auto tuner based on stochastic search is then applied to automatically find a good schedule.

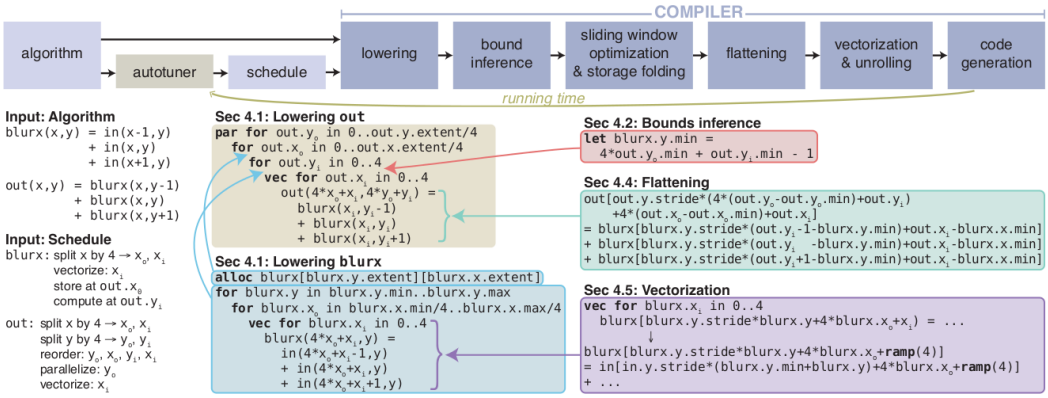


Figure 3.2 – *Halide Image Processing [3].*

When compared to our approach, Halide is limited to image processing applications, whereas XFOR is much more general. The interleaved computations addressed by Halide compiler are the core of the XFOR construct and they are naturally expressed.

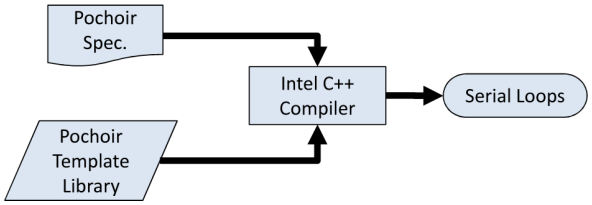
The original program and the schedule representation are separated, and both are needed for the code generation done by Halide. This separation is unpractical and may be error-prone. With the XFOR construct, both information are represented simultaneously; the original program is defined within loop bounds, strides and the XFOR loop body, the schedule and transformations are provided intuitively by the offset and grain parameters. Loop parallelization can be afforded by applying OpenMP *pragma* for convenient XFOR loops and vectorization is implement by setting the appropriate offset value.

In addition to that, the schedule representation of Halide is tedious to write,

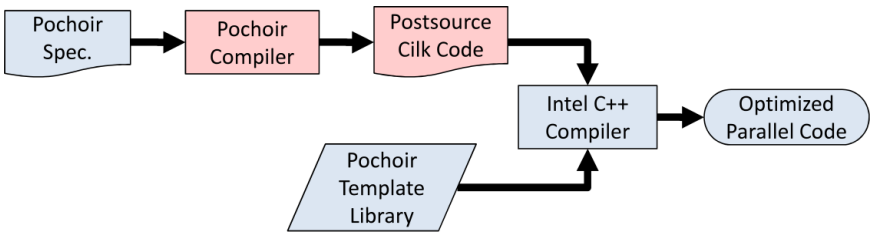
since every single transformation has to be described separately. However, the XFOR representation of transformations is very concise and the transformation composition is done simply by setting the offset values as linear expressions (see Section *Flexible Transformation Composition* in Chapter 9, [*XFOR Polyhedral Intermediate Representation and Manual-Automatic Collaborative Approach for Loop Optimization*], page 151).

3.3.3 Pochoir

The Pochoir compiler [4] transforms code written in the Pochoir specification language into optimized $C++$ code that employs the Intel Cilk multithreading extensions . The Pochoir compiler performs numerous optimizations. The most important ones are code cloning, loop-index calculations, unifying periodic and nonperiodic boundary conditions, and coarsening the base case of recursion.



(a) *phase 1*



(b) *phase 2*

Figure 3.3 – *Pochoir's two-phase compilation strategy [4].*

As shown in Figure 3.3, the Pochoir system operates in two phases. The second one only involves the Pochoir compiler itself. For the first phase, the programmer compiles the source program with the ordinary Intel $C++$ compiler using the Pochoir template library, which implements Pochoir's linguistic constructs using unoptimized but functionally correct algorithms. This phase ensures that the source program is Pochoir-compliant. For the second phase, the programmer runs the source through the Pochoir compiler, which acts as a preprocessor to the Intel $C++$ compiler, performing a source-to-source translation into a postsource $C++$ program that employs the *Cilk* extensions. The postsource is then compiled with the Intel compiler to produce the optimized binary executable [4].

In order to utilize the Pochoir compiler, the user has to write a "pochoir

specification" respecting a specific syntax. We show below the Pochoir stencil source code for a periodic 2D heat equation [4] (Pochoir keywords are boldfaced).

```
# define mod (r,m) ((r)%(m)+((r)<0)?(m):0)

 Pochoir_Boundary_2D(heat_bv,a,t,x,y)
   return a. get (t,mod(x,a.size(1)),mod(y,a.size(0)));
 Pochoir_Boundary_End

int main ( void ) {
    Pochoir_Shape_2D 2D_five_pt[] = {{1,0,0},{0,1,0},
       {0,-1,0},{0,-1,-1},{0,0,-1},{0,0,1}};
    Pochoir_2D heat(2D_five_pt );

    Pochoir_Array_2D(double )u(X,Y);
   u. Register_Boundary( heat_bv );
   heat. Register_Array(u);

    Pochoir_Kernel_2D( heat_fn , t , x , y)
      u(t+1,x,y) = CX*(u(t,x+1,y)-2*u(t,x,y)+u(t,x-1,y))
          +CY*(u(t,x,y+1)-2*u(t,x,y)+u(t,x,y-1))+u(t,x,y);
    Pochoir_Kernel_End

   for ( int x = 0 ; x < X ; ++x)
     for ( int y = 0 ; y < Y ; ++y)
       u(0 , x , y) = rand () ;

   heat. Run(T, heat_fn );

   for ( int x = 0 ; x < X ; ++x)
     for ( int y = 0 ; y < Y ; ++y)
       cout << u(T,x,y);

   return 0;
}
```

In next Chapters, it is shown that the XFOR syntax is more compact, concise, simple and easy to understand compared to the Pochoir specification syntax. In addition to that, Pochoir is limited to stencil computations, while XFOR is much more general. Moreover, the XFOR construct enables more powerful and general polyhedral transformations and allows the programmer to take care of other performance counters, not only cache optimization and parallelism.

3.3.4 Stencil Domain Specific Language (SDSL)

In [61], Henretty et al. we propose a domain-specific language and compiler for stencil computations that allows specification of stencils in a concise manner and automates both locality and short-vector SIMD optimizations, along with effective utilization of multi-core parallelism. Loop transformations to enhance data locality and enable load-balanced parallelism are combined with a data layout transformation to effectively increase the performance of stencil computations.

Henretty et al., in [61], describe an integrated approach and compiler algorithms to perform split-tiling in conjunction with *Dimension-Lifting-Transpose* (DLT) transformation to generate efficient parallel code for stencil computations over large data sets on shared memory multiprocessors. Split-tiling was adopted in order to enhance inter-tile concurrency. A DLT was developed to overcome the fundamental data access inefficiency on current short-vector SIMD architectures with stencil computations, thus, it optimizes vector loads/stores

In general, an SDSL program contains one grid, one or more griddata, one iterate, and one or more stencil definitions, where each stencil may define one or more subdomains and the stencil functions that operate upon them [61]. We show below the general form of an SDSL program:

```
grid g[dimK]...[dim1];
griddata g1,g2...,gM on g;
iterate T {
  stencil s1 {
    [lb_s1_K:ub_s1_K]...[lb_s1_1:ub_s1_1] : f1(...);
  }
  stencil s2 {
    [lb_s2_K:ub_s2_K]...[lb_s2_1:ub_s2_1] : f2(...);
  }
  ...
  stencil sN {
    [lb_sN_K:ub_sN_K]...[lb_sN_1:ub_sN_1] : fN(...);
  }
}
```

The computation shown in Listing 3.1 is a standard Jacobi 2D computation, which averages the value of the 5 neighboring points (up, down, left, right, and center) to compute the new value of the center point.

```
1 grid g[dim1][dim0];
2 double griddata a on g at 0,1;
3 iterate 100 {
4   stencil five_pt {
5     [1:dim1-2][1:dim0-2] : [1]a[0][0] = 0.2*([0]a[-1][0]+
6     [0]a[0][-1]+[0]a[0][0]+[0]a[0][1]+[0]a[1][0]);
7 } }
```

Listing 3.1 – *A simple Jacobi 2D example in SDSL.*

The main limitation of this work is that it automates optimizing transformations of stencil computations related to some specific loop tiling. Obviously, it does not provide a programming flexibility similar to the XFOR structure. In addition to that, the XFOR construct provides more general transformations and gives the user the opportunity to improve other performance issues, not only data locality and parallelism.

3.3.5 ExaStencils

In [5] Christian Lengauer et al. presents a domain specific language targeting stencil computations for `exascale` systems. The platform has multiple levels of DSL, each level representing an abstraction. Modifications to lower layers of DSL can be done to fine tune the code. It uses machine learning to recommend the suitable combinations of configuration options (algorithmic components, data structure alternatives, and parameter values). The polyhedral model is then used for loop parallelization. With the help of domain specific knowledge, the reduction operations can be classified as associative and commutative. This information is also used while constructing the polytope model, which improves performance. The code generator produces the final code in C++ with OpenMP and CUDA.

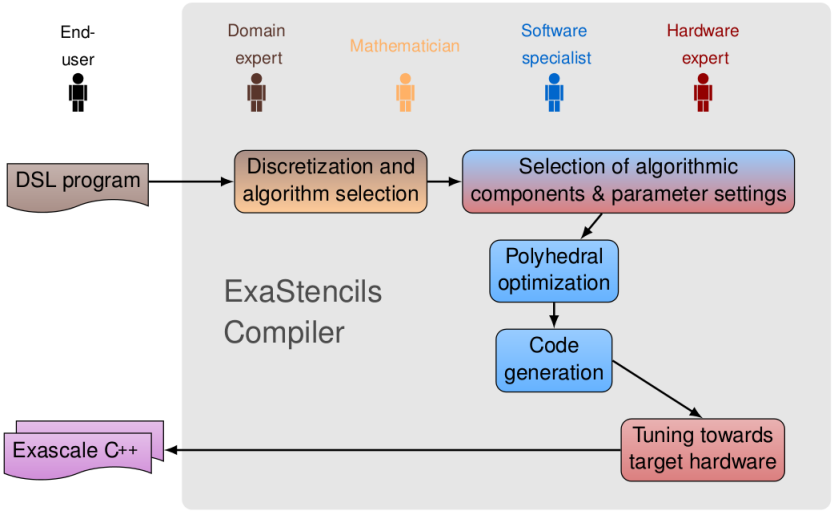


Figure 3.4 – *The workflow of the ExaStencils programming paradigm: the ExaStencils compiler builds on the combined knowledge of domain experts, mathematicians, and software and hardware specialists to generate high-performance target code [5].*

The workflow of a stencil code generation in ExaStencils is illustrated in Figure 3.4. In a first step, a stencil algorithm is defined by a mathematician. The solution is put into a first executable form via a cooperation of the mathematician with a software engineer. In the ExaStencils approach, the software description names a set of algorithmic and platform choices, each made from a number of options and alternatives. Then, an implementation is "woven" automatically.

The weaving algorithm is capable of applying optimizations customized for the specific choices made. One powerful model exploited in ExaStencils is the polyhedral model for automatic loop parallelization. In a final step, some low-level fine-tuning for the platform at hand takes place. The target code can be in any language (or, indeed, mix of languages) that is suitable [5].

The *ExaStencils* compiler relies on the combined knowledge of domain experts, mathematicians, and software and hardware specialists to generate high-performance target code. This restriction makes it out of reach of a usual user. In contrast, our looping structure XFOR may be used by any user who is familiar with programming. And thanks to the assistance of our software tools (IBB, XFOR-WIZARD and XFORGEN), the user is not required to have massive knowledge to optimize his program.

3.4 Additional Related Work

K. Stock et al. present in [62] a framework utilizing the associativity and commutativity of operations in loops in order to reduce excessive data traffic caused by poor register reuse in executing repetitive operations on a multi-dimensional arrays through data locality enhancement. Their transformation framework is particularly useful for high-order stencil computations.

Example. We reproduce here the illustrative example used in [62]. The very first step of the described framework is to convert an input program into an internal representation that is amenable to effective re-timing. For maximal flexibility, it is best to split a single statement, as in Listing 3.2, that contains multiple associative accumulation operations '+' into distinct statements, in a way to have only one accumulation operation per statement, as in Listing 3.3. This enables different re-timing for the operands of each accumulation.

```
1 for (i=1 ; i<N-1 ; ++i)
2   S1 : OUT[i] = W[0]*IN[i-1]+W[1]*IN[i]+W[2]*IN[i+1];
```

Listing 3.2 – *Jacobi 1D using a weight array W [62].*

```
1 for (i=1 ; i<N-1 ; ++i) {
2   S1 : OUT[i]  = W[0]*IN[i-1];
3   S2 : OUT[i] += W[1]*IN[i];
4   S3 : OUT[i] += W[2]*IN[i+1];
5 }
```

Listing 3.3 – *Jacobi 1D after statement splitting [62].*

```
S1 : OUT[1]  = W[0]*IN[0];
S1 : OUT[2]  = W[0]*IN[1];
S2 : OUT[1] += W[1]*IN[1];
for (i=2 ; i<N-2 ; ++i) {
  S1 : OUT[i+1]  = W[0]*IN[i];
  S2 : OUT[i]   += W[1]*IN[i];
  S3 : OUT[i-1] += W[2]*IN[i];
}
S2 : OUT[N-1] += W[1]*IN[N-1];
S3 : OUT[N-2] += W[2]*IN[N-1];
S3 : OUT[N-1] += W[2]*IN[N];
```

Listing 3.4 – *Jacobi 1D after retiming (all-scatter) [62].*

Listing 3.4 shows a transformed version of the code, where multidimensional re-timing has been applied to ”realign“ the accesses to IN so that inside an iteration of loop i, the same element of IN is being accessed.

The XFOR structure allows the users to write codes implementing explicitly and in a concise manner the proposed scatter-gather combinations for stencil computations. Similar strategy was described in Section *Minimizing Intra-Statement Data Reuse Distance*, in Chapter 6 [*XFOR Programming Strategies*], page 87. It consists in splitting a statement into several statements and then reorder the resulting code, using the offset and grain parameters, in order to enhance data locality, while our source-to-source compiler IBB takes care of the equivalent code generation. In addition to that, our programming environment XFOR-WIZARD may be used in order to check the legality of the schedule. The XFOR code is more flexible, more maintainable, and offer to the user more transformations such as loop unrolling, loop skew, etc... (see Chapter 9, [*XFOR Polyhedral Intermediate Representation and Manual-Automatic Collaborative Approach for Loop Optimization*], page 135, for more details). As an example, we show in Listing 3.5 the XFOR code equivalent to the code shown in Listing 3.4 and the corresponding IBB-generated code (Listing 3.6). Details on how this XFOR code has been defined are presented in the next Chapter.

```
xfor ( io=1 , i1=1 , i2=1 ; io<N-1 , i1<N-1 , i2<N-1;++ io ,++ i1 ,++ i2 ;1 ,1 ,1;0 ,1 ,2) {
  0 : OUT[ io ]  = W[0]*IN[ io-1 ];
  1 : OUT[ i1 ] += W[1]*IN[ i1 ];
  2 : OUT[ i2 ] += W[2]*IN[ i2+1 ];
}
```

Listing 3.5 – *Jacobi 1D after retiming (all-scatter) - XFOR Equivalent Code.*

```
int _mfr_refo ;
if (N >=3) {
  OUT [1]=W [0]*IN [0];
  if (N ==3) {
```

```
5     OUT [1]+=W [1]*IN [1];
6   }
7   if (N >=4) {
8     OUT [2]=W [0]*IN [1];
9     OUT [1]+=W [1]*IN [1];
10  }
11  for (_mfr_refo =2; _mfr_refo <=N -3; _mfr_refo ++) {
12    OUT [ _mfr_refo +1]=W [0]*IN [ _mfr_refo ];
13    OUT [ _mfr_refo ]+=W [1]*IN [ _mfr_refo ];
14    OUT [ _mfr_refo -1]+=W [2]*IN [ _mfr_refo ];
15  }
16  if (N >=4) {
17    OUT [N-2]+=W [1]*IN [N-2];
18    OUT [N-3]+=W [2]*IN [N-2];
19  }
20  OUT [N-2]+=W [2]*IN [N-1];
21 }
```

Listing 3.6 – *Jacobi 1D after retiming (all-scatter) - IBB Generated Code.*

Additionally, since the proposed framework focuses only on data reuse and vectorization, and does not address the other optimization effects, it is possible to generate automatically XFOR loops from the code generated by the framework thanks to our software tool XFOR-WIZARD (see Chapter 7, [*The XFOR Programming Environment XFOR-WIZARD*], page 97). After that, reorder the statements in the generated XFOR code using the offset parameter in order to enhance performance, since performance depends on several issues (such as processor stalls due to branch misses, register dependencies and instruction count.), not only on data locality (see Chapter 8, [*XFOR and Automatic Optimizers*], page 115, for more details).

3.5 Conclusion

In this Chapter, we presented some high level programming languages that were designed specifically for improving performance. Some of these languages are general purpose, the other are *domain specific*. Most of the new proposed programming languages imply to change drastically programmers habits and have weak chances to be adopted by the software industry [63]. As an answer to the identified limitations, we propose a new looping structure permitting to define several loops at the same time, and to schedule them and also apply advanced transformations easily thanks to dedicated parameters. In the next chapter, we define the syntax and semantics of our proposed structure: the "XFOR".

XFOR Syntax and Semantics

4

4.1 Introduction

XFOR is a looping structure permitting the user to define several loop indices at the same time and also to schedule their corresponding statements thanks to dedicated parameters: a running frequency (the grain) and a relative position (the offset).

This chapter introduces the XFOR programming control structure and its semantics. It starts by a motivating example that shows its simplicity and its ease of use. Then, its syntax and semantics are detailed in Section 4.3. For better understanding on the usage of XFOR, several illustrative examples are given in Section 4.4. We first present the case of one unique XFOR construct, as the case of nested XFOR-loops present some specificities which are presented afterwards.

4.2 Motivation

Consider the image shown in Figure 4.1. Assume that we intend to apply two successive processes on it; First, we want to apply a function $f1()$ for the whole image. Then, we want to apply a function $f2()$ just for the region inside the white square.

Figure 4.1 – *Motivating Example.*

The easiest way to implement this operation consists of writing a succession of two loop nests; the first one scans the whole image and applies $f1()$ and the

second scans the region inside the square and applies $f2()$. The corresponding algorithm is represented in Listing 4.1. The problem of this algorithm is that it scans the image twice. Thus it may lead to a long execution-time, due to the cache misses generated from loading each pixel twice: the data reuse distances are too large.

```
for (i=0 ; i<=N ; i++)
  for (j=0 ; j<=M ; j++)
    f1(image[i][j]);

for (i=a ; i<=b ; i++)
  for (j=c ; j<=d ; j++)
    f2(image[i][j]);
```

Listing 4.1 – *Functional Implementation.*

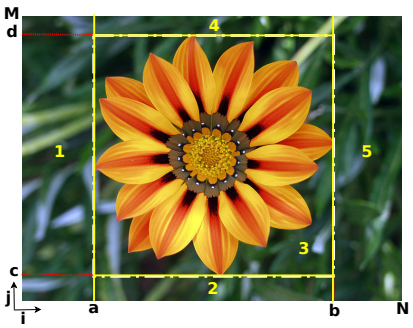


Figure 4.2 – *Motivating Example - 2.*

```
for (i = 0 ; i <= a ; i++)
  for (j = 0 ; j <= M ; j++) //*1*
    f1 (image[i][j]);
for (i = a ; i <= b ; i++) {
  for (j = 0 ; j <= c ; j++) //*2*
    f1 (image[i][j]);
  for (j = c ; j <= d ; j++){ //*3*
    f1 (image[i][j]);
    f2 (image[i][j]);
  }
  for (j = d ; j <= M ; j++) //*4*
    f1 (image[i][j]);
}
for (i = b ; i <= N ; i++)
  for (j = 0 ; j <= M ; j++) //*5*
    f1 (image[i][j]);
```

Listing 4.2 – *Optimized Solution.*

An enhanced solution divides the image into distinct parts according to the applied functions. As shown in Figure 4.2, we distinguish five parts. Let

us write a program that scans them separately, and applies, for each part, the corresponding functions. A possible solution is shown in Listing 4.2. This program exhibits a better temporal data locality since both functions that access the same pixels in the image are invoked in the same loop.

```
xfor(i1=0,i2=a ; i1<N,i2<b ; i1++,i2++ ; 1,1 ; 0,a)
 xfor(j1=0,j2=c ; j1<M,j2<d ; j1++,j2++ ; 1,1 ; 0,c) {
   0: f1(image[i1][j1]);
   1: f2(image[i2][j2]);
 }
```

Listing 4.3 – *XFOR Solution.*

This optimized solution can be written easily in a more compact way using the XFOR looping structure, the corresponding code is represented in Listing 4.3. Note that, when compared to the previous version, the length of the XFOR version, as well as the number of loops, the duplicated instructions and the various image references are significantly less. This simple example shows that XFOR is a looping structure that enables expressing complicated sequence of loop nests in a precise and concise manner.

4.3 Syntax and Semantics

4.3.1 Non-Nested XFOR-Loop

```
xfor ( index = expr, [index = expr, ...] ;
       index relop expr, [index relop expr, ...] ;
       index += incr, [index += incr, ...] ;
       grain, [grain, ...] ; offset, [offset, ...] ) {
       label: {statements}
       [label: {statements} ...]
}
```

Listing 4.4 – *Non-Nested XFOR-Loop Syntax.*

Listing 4.4 defines the syntax of an XFOR loop, where [...] denotes optional arguments, *index* denotes the indices of the loops composing the XFOR, *expr* denotes affine arithmetic expressions of enclosing loop indices, or constants, *incr* denotes an integer constant, *relop* denotes a relational operator ($==$, $<$, $\leq$, ...).

The first three elements in the XFOR header are similar to the initialization, test, and increment parts of a traditional C for-loop, except that all these elements describe two or more loop indices. The last two components define the offset and grain for each index; the offset is an affine expression on encompassing and nested indices, or a constant. The grain is a positive constant ($grain \geq 1$). Every index in the set must be present in all components of the header. In

addition to that, (sequences of) statements must be labeled with the rank of the corresponding index (0 for the first index, 1 for the second, and so on) [23, 24].

The list of indices defines several for-loops whose respective iteration domains are all mapped onto a same global "virtual referential" domain. The way iteration domains of the for-loops overlap is defined only by their respective offsets and grains, and not by the values of their respective indices, which have their own ranges of values. The grain defines the frequency in which the associated loop has to run relatively to the referential domain. For instance, if we consider an XFOR loop with two indices; if the first index has a grain equal to 1 and the second has 2 as grain, then the first index will be unrolled twice , consequently, at each iteration of the referential loop, we have one execution of the second index and two execution of the first one. The offset defines the gap between the first iteration of the referential and the first iteration of the associated loop. For instance, if the offset is equal to 3, then the first iteration of the associated loop will run at the fourth iteration of the referential loop [24].

The size and shape of the referential domain can be deduced from the for-loop domains composing the XFOR-loop. Geometrically, the referential domain is defined as the union of the for-loop domains, where each domain has been shifted according to its offset and unrolled according to the *least common multiple* of all the grains that appear in the XFOR header divided by its grain. That is, the referential domain is not only defined by the bounds of loop indices and the offset, but also, by the number of instance of unrolled instructions by iteration which is equal to the *least common multiple* divided by the corresponding grain. Hence, in the domain presentations in the following Sections, we present the iterations and the instances of instructions.

The relative positions of the iterations of the individual for-loops composing the XFOR-loop depend on how individual domains overlap. Iterations are executed in the lexicographic order of the referential domain. On portions of the referential domain where at least two domains overlap, the corresponding statements are run in the order implied by the order in which they appear in the XFOR body.

The bodies of the for-loops composing the XFOR-loop can be any C-based code. However, their statements can only access their respective loop indices, and not any other loop index whose value may be incoherent in their scope. Moreover, indices can only be modified in the loop header by incrementation, and never in the loop body.

Loop index declarations in the XFOR header are not allowed. However, it is possible to declare variables inside the XFOR-loop body.

4.3.2 Nested XFOR-Loops

```
xfor ( index1 = expr, index2 = expr ;
       index1 < expr, index2 < expr ;
       index1 += cst, index2 += cst ;
       grain1 , grain2 ;
        offset1 , offset2 ) {
    label: {statements}
    xfor ( index3 = expr, index4 = expr ;
           index3 < expr, index4 < expr ;
           index3 += cst, index4 += cst ;
           grain3 , grain4 ;
            offset3 , offset4 ) {
        label: {statements}
    }
    label: {statements}
}
```

Listing 4.5 – *Nested XFOR-Loop Syntax for two loop indices per loop level in a 2-depth XFOR nest.*

Nested XFOR-loops present some particularities and specific semantics that have to be described. Without loss of generality, consider the XFOR-loop nest in Listing 4.5. Such a nest behaves as two for-loop nests, $(index_1, index_3)$ and $(index_2, index_4)$ respectively, running simultaneously in the same way as it is for one unique XFOR-loop. The grain and the offset are applied with the same reasoning as with the non-nested case at each loop depth. The lower and upper bounds are affine functions of the enclosing loop indices.

Nested XFOR-loops behave like several nested for-loops which are synchronized according to the common referential domain. Nested for-loops are defined according to the order in which their respective indices appear in the XFOR headers. For instance, in a 2-level XFOR nest, the first index variable of the outermost loop is linked to the first index variable of the inner loop, the second to the second, and so on. Hence the same number of indices have to be defined at each level of any XFOR nest. This is not a strong restriction. The syntax enables shorter specifications of indices which are not used inside statements. We illustrate this notion with some examples of nested XFOR loops later in subsection 4.4.2 (*Nested XFOR-Loops Examples*, page 65).

4.3.3 XFOR-Loop Execution

Initially, XFOR was designed as a structure that exhibits a straightforward parallelization strategy, that means, at each iteration, running the loop bodies of the defined for-loops in parallel. This was achieved by using *OpenMP sections* (See Section 5.2.4 in chapter 5 ([*XFOR Code Generation: the IBB Compiler*], page 75)). The motivation was to provide new opportunities to the polytope model, since it enables the expression of parallel programming models that were unattainable

before [23]. But, following some experiments, we found that running loop bodies in parallel was not very interesting, regarding the parallelism granularity efficiency on general-purpose multicore processors. In addition to this, the XFOR construct allows OpenMP-like loop parallelization for each XFOR-loop of the nest. Therefore we choose to execute the statements of the for-loops sequentially, *i.e.* in an interleaved fashion. The statements are executed in the order defined by their offsets and their relative order in the XFOR body. For better illustration, the following Section exhibits some illustrative examples of simple and nested XFOR loops.

4.4 Illustrative Examples

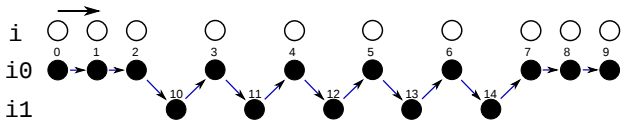
Without loss of generality, we consider in the following that the index step, *cst*, is always equal to one. The general case can be easily deduced.

4.4.1 Non-Nested XFOR-Loops Examples

Example 4.1 Consider the following XFOR loop:

```
1 xfor (io=0,i1=10 ; io<10,i1<15 ; io++,i1++ ; 1,1 ; 0,2) {
2       0: loop_bodyo
3       1: loop_body1 }
```

In this example, the offset of index $i0$ is zero, and that of index $i1$ is 2. Thus, the first iteration of the $i0$-loop will run immediately, while the first iteration of the $i1$-loop will run at the third iteration of the XFOR, but with index value $i1 = 10$. On the sub-domain where both for-loop domains overlap, the loop bodies are run in interleaved fashion starting with loop_bodyo. This behavior is illustrated by the figure below:



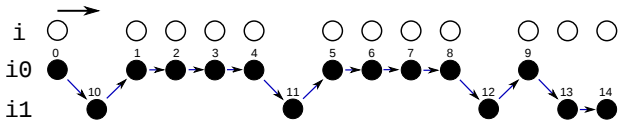
Notice that the index values have no effect on the relative positions of the iteration domains, which are uniquely determined by their respective grains and offsets.

Example 4.2 Another example is the following:

```
1 xfor (io=0,i1=10 ; io<10,i1<15 ; io++,i1++ ; 1,4 ; 0,0) {
2       0: loop_bodyo
3       1: loop_body1 }
```

Now, both offsets are equal to 0, but, the $i0$-grain is 1 and the $i1$-grain is 4. In this case, at each iteration of the referential, four statement instances of the $i0$-loop will be run over one statement of the $i1$-loop on the sub-domain on which

they overlap. The last two iterations of $i1$ occur after the end of the io-loop; their domain can be compressed by a factor of 4, as it is illustrated below:



4.4.2 Nested XFOR-Loops Examples

Example 4.3 Consider the following XFOR-loop nest:

```
xfor (i1=0, i2=0 ; i1<10, i2<5 ; i1++, i2++ ; 1,1 ; 0,2)
 xfor(j1=0, j2=0 ; j1<10, j2<5 ; j1++, j2++ ; 1,1 ; 0,2){
      0: S1 (i1,j1);
      1: S2 (i2,j2); }
```

This XFOR-loop nest is equivalent to these two for-loop nests that have been merged in a specific way:

```
for(i1=0 ; i1<10 ; i1++)
 for(j1=0 ; j1<10 ; j1++){
  S1 (i1,j1);
 }
```

Listing 4.6 – *First FOR-Loop Nest.*

```
for(i2=0 ; i2<5 ; i2++)
 for(j2=0 ; j2<5 ; j2++){
  S2 (i2,j2);
 }
```

Listing 4.7 – *Second FOR-Loop Nest.*

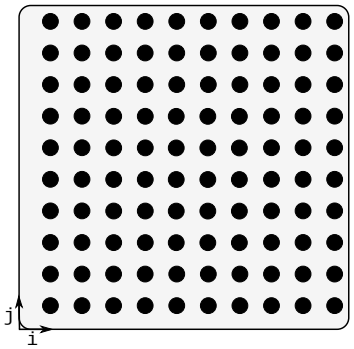


Figure 4.3 – *First Domain.*

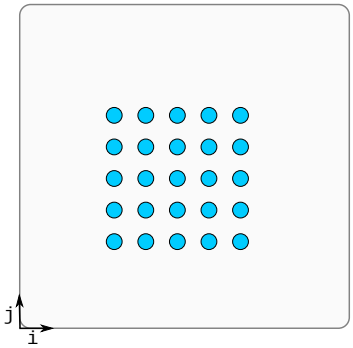


Figure 4.4 – *Second Domain (Shifted).*

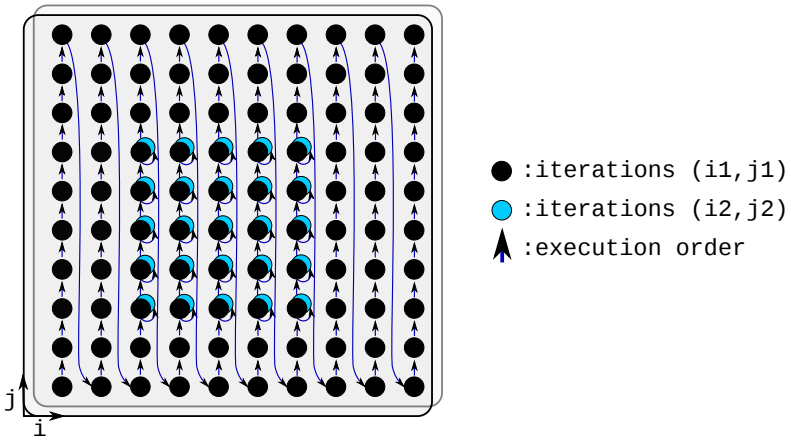


Figure 4.5 – *Example 3.*

The second for-loop nest has an offset of 2 at each loop depth. Hence it is delayed in each dimension of the referential domain. This is shown in Figure 4.5.

Example 4.4 Another example is:

```
xfor (i1=0, i2=0 ; i1<10, i2<3 ; i1++, i2++ ; 1,4 ; 0,0)
 xfor(j1=0, j2=0 ; j1<10, j2<3 ; j1++, j2++ ; 1,4 ; 0,0){
      0: S1 (i1,j1);
      1: S2 (i2,j2);
 }
```

This XFOR-loop nest is equivalent to these two for-loop nests that have been merged in a specific way:

```
for(i1=0 ; i1<10 ; i1++)
 for(j1=0 ; j1<10 ; j1++){
  S1 (i1,j1);
 }
```

Listing 4.8 – *First FOR-Loop Nest.*

```
for(i2=0 ; i2<3 ; i2++)
 for(j2=0 ; j2<3 ; j2++){
  S2 (i2,j2);
 }
```

Listing 4.9 – *Second FOR-Loop Nest.*

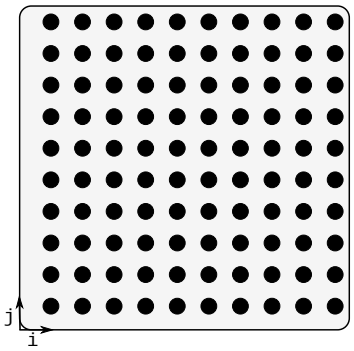


Figure 4.6 – *First Domain.*

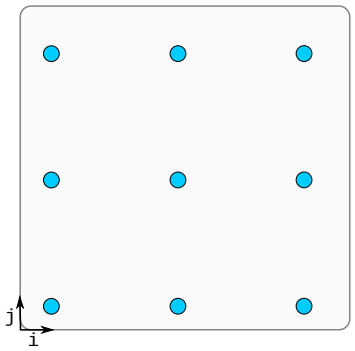


Figure 4.7 – *Second Domain (Dilated).*

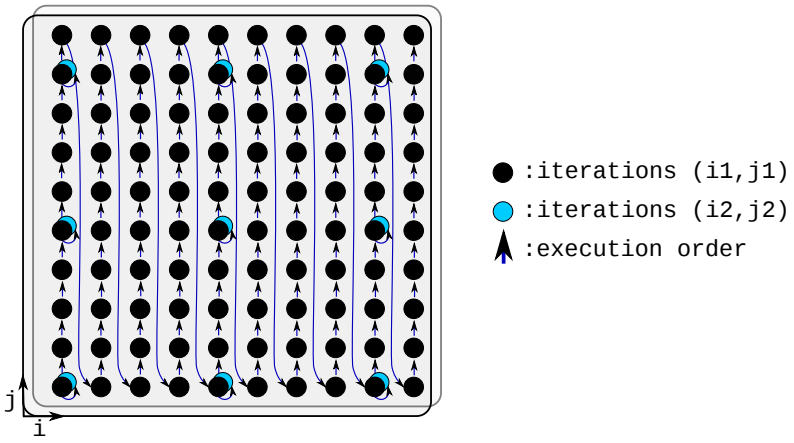


Figure 4.8 – *Example 4.*

In this example, the second for-loop nest has a 4-grain at each loop depth. Hence its iterations are spaced by 3 statement instances of the first loop nest

in dimension j and 3 iterations in dimension i of the referential domain as it is represented in Figures 4.7 and 4.8.

Example 4.5 In this example, the upper bound of the inner loop of the first loop nest is an affine function.

```
xfor (i1=0, i2=0 ; i1<6,    i2<6 ; i1++, i2++ ; 1,1 ; 0,1)
 xfor(j1=0, j2=0 ; j1<6-i1, j2<6 ; j1++, j2++ ; 1,1 ; 0,0){
      0: S1 (i1,j1);
      1: S2 (i2,j2);
 }
```

This XFOR-loop nest is equivalent to the merge of these two for-loop nests:

```
for(i1=0; i1<6; i1++)
 for(j1=0; j1<6-i1; j1++){
  S1 (i1,j1);
 }
```

Listing 4.10 – *First FOR-Loop Nest.*

```
for(i2=0; i2<6; i2++)
 for(j2=0; j2<6; j2++){
  S2 (i2,j2);
 }
```

Listing 4.11 – *Second FOR-Loop Nest.*

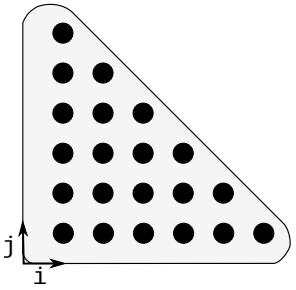


Figure 4.9 – *First Domain.*

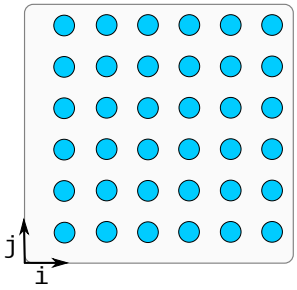


Figure 4.10 – *Second Domain.*

The second for-loop nest has a 1-offset at the outer loop. Hence it is delayed in the i-dimension of the referential domain as it is represented in Figure 4.11.

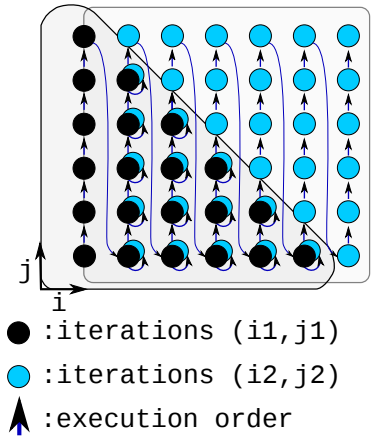


Figure 4.11 – *Example 5.*

Example 4.6 In this example, the offset of the inner loop of the first loop nest is an affine function. It is equal to $i1$. Hence the first loop is delayed in the j-dimension of the referential domain as it is represented in Figure 4.14.

```
xfor (i1=0, i2=0 ; i1<6  i2<6 ; i1++, i2++ ; 1,1 ; 0,0)
 xfor(j1=0, j2=0 ; j1<6, j2<6 ; j1++, j2++ ; 1,1 ; i1,0){
      0: S1 (i1,j1);
      1: S2 (i2,j2);
 }
```

This XFOR-loop nest is equivalent to the merge of these two for-loop nests:

```
for(i1=0 ; i1<6 ; i1++)
 for(j1=0 ; j1<6 ; j1++){
  S1 (i1,j1);
 }
```

Listing 4.12 – *First FOR-Loop Nest.*

```
for(i2=0 ; i2<6 ; i2++)
 for(j2=0 ; j2<6 ; j2++){
  S2 (i2,j2);
 }
```

Listing 4.13 – *Second FOR-Loop Nest.*

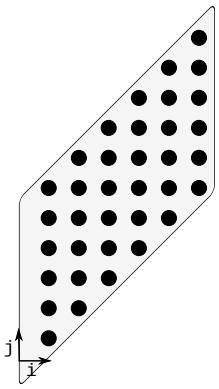


Figure 4.12 – *First Domain (Delayed).*

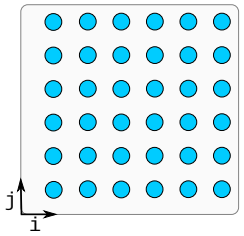


Figure 4.13 – *Second Domain.*

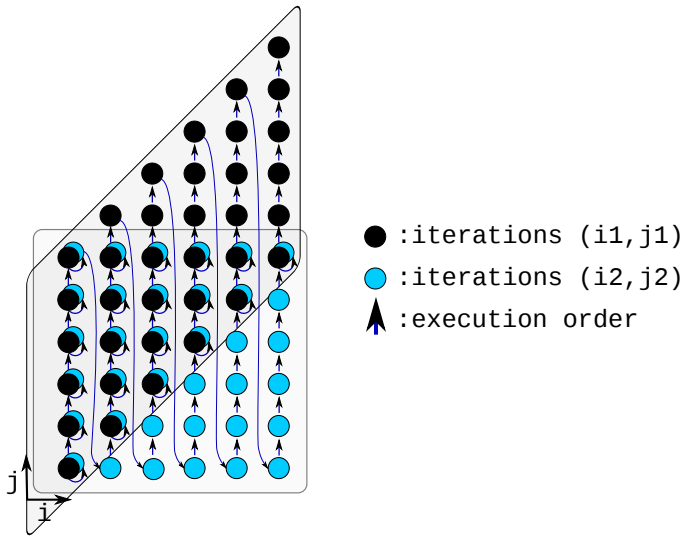


Figure 4.14 – *Example 6.*

In chapter 9 (*XFOR Polyhedral Intermediate Representation and Manual-Automatic Collaborative Approach for Loop Optimization*, page 135), we show that even more complex transformations can be expressed in a concise way using XFOR loops, and in particular any polyhedral transformation.

4.5 Conclusion

In this chapter, we presented the XFOR Structure, we detailed its syntax and explained the semantics using several examples. In order to prepare the XFOR code for general compilation and execution, it has, first to be converted into *for loops*. The validation and translation of the XFOR code is done by our source-to-source compiler, IBB, which is presented in the next chapter.

5 XFOR Code Generation : the IBB Compiler

5.1 Introduction

IBB [25] stands for "'*Iterate, But Better !*". It is a source-to-source compiler that takes as input any C-based program using the XFOR construct and generates as output, a file containing an equivalent program, in which all XFOR-loop nests have been replaced by semantically equivalent for-loop nests. In addition to the automatic code generation, IBB also performs a syntax validation of the input XFOR code.

This chapter is devoted to the IBB-compiler. First, in Section 5.2, we explain how to use the IBB software. Second, we highlight the computation of the global reference domain in Section 5.3. Then, in Section 5.4, we detail how IBB translates the XFOR loops to final for-loops. Each IBB module and its contribution to the XFOR translation is presented in a dedicated subsection.

5.2 Using the IBB Software

5.2.1 A First Example

```
1  xfor( i0=1, i1=1, i2=1, i3=1, i4=1, i5=1;  i0<N, i1<N, i2<N, i3<N, i4<N, i5<N;
2        i0++, i1++, i2++, i3++, i4++, i5++; 1,1,1,1,1,1 ; 0,0,2,1,1,2) {
3  xfor( j0=1, j1=1, j2=1, j3=1, j4=1, j5=1;  j0<N, j1<N, j2<N, j3<N, j4<N, j5<N;
4        j0++, j1++, j2++, j3++, j4++, j5++; 1,1,1,1,1,1 ; 1,1,1,0,2,1) {
5        0: b[i0][j0] = 0    ;
6        1: b[i1][j1] += a[i1-1][j1] ;
7        2: b[i2][j2] += a[i2+1][j2] ;
8        3: b[i3][j3] += a[i3][j3-1] ;
9        4: b[i4][j4] += a[i4][j4+1] ;
10       5: b[i5][j5] =  b[i5][j5]/4 ; }}
```

Listing 5.1 – *XFOR Code Example* (`xfor.c`).

The IBB compiler takes as input a 'C' file that contains XFOR-loops. IBB can automatically identify the XFOR-loop nests and then translates them to semantically equivalent for-loops. Additionally, it supports some options (command line arguments) to customize the target code, as detailed in Subsection 5.2.4 ([*IBB Options*], page 74). However, the basic use of IBB is easy and intuitive. In this Section, we explain how to generate the code corresponding to an XFOR-loop nest example.

The XFOR loop nest in Listing 5.1 implements an optimized version of a stencil computation. It calculates the average of the four neighboring elements of an array *a*, and writes the result in array *b*. Let us assume that the code is saved in a file named '`xfor.c`'. We can invoke IBB to process it and to generate an equivalent code using for-loops by a simple call with this filename as input: `IBB xfor.c`

By default, IBB will save the generated code in a file named '`a.out.c`'.

```
/*- Generated from xfor.c by IBB -*/
if (N >= 2) {
  for (_mfr_ref1 = 1; _mfr_ref1 <= N-1; _mfr_ref1++) {
    b[1][_mfr_ref1]=0;
    b[1][_mfr_ref1]+=a[0][_mfr_ref1]; }
  if (N == 2) {
    b[1][1]+=a[1][0];
    b[1][1]+=a[1][2]; }
  if (N >= 3) {
    b[1][1]+=a[1][0];
    b[2][1]=0;
    b[2][1]+=a[1][1];
    b[1][2]+=a[1][1];
    for (_mfr_ref1 = 2; _mfr_ref1 <= N-2; _mfr_ref1++) {
      b[2][_mfr_ref1]=0;
      b[2][_mfr_ref1]+=a[1][_mfr_ref1];
      b[1][_mfr_ref1+1]+=a[1][_mfr_ref1];
      b[1][_mfr_ref1-1]+=a[1][_mfr_ref1]; }
    b[2][N-1]=0;
    b[2][N-1]+=a[1][N-1];
    b[1][N-2]+=a[1][N-1];
    b[1][N-1]+=a[1][N]; }
  for (_mfr_refo = 2; _mfr_refo <= N-2; _mfr_refo++) {
    b[_mfr_refo][1]+=a[_mfr_refo][0];
    b[_mfr_refo+1][1]=0;
    b[_mfr_refo+1][1]+=a[_mfr_refo][1];
    b[_mfr_refo-1][1]+=a[_mfr_refo][1];
    b[_mfr_refo][2]+=a[_mfr_refo][1];
    b[_mfr_refo-1][1]=b[_mfr_refo-1][1]/4;
    for (_mfr_ref1 = 2; _mfr_ref1 <= N-2; _mfr_ref1++) {
      b[_mfr_refo+1][_mfr_ref1]=0;
      b[_mfr_refo+1][_mfr_ref1]+=a[_mfr_refo][_mfr_ref1];
      b[_mfr_refo-1][_mfr_ref1]+=a[_mfr_refo][_mfr_ref1];
      b[_mfr_refo][_mfr_ref1+1]+=a[_mfr_refo][_mfr_ref1];
```

```
35            b[ _mfr_refo ][ _mfr_ref1-1]+=a[ _mfr_refo ][ _mfr_ref1 ];
36            b[ _mfr_refo-1 ][ _mfr_ref1 ]=b[ _mfr_refo-1 ][ _mfr_ref1 ]/4; }
37        b[ _mfr_refo+1 ][N-1]=0;
38        b[ _mfr_refo+1 ][N-1]+=a[ _mfr_refo ][N-1 ];
39        b[ _mfr_refo-1 ][N-1]+=a[ _mfr_refo ][N-1 ];
40        b[ _mfr_refo ][N-2]+=a[ _mfr_refo ][N-1 ];
41        b[ _mfr_refo-1 ][N-1]=b[ _mfr_refo-1 ][N-1]/4;
42        b[ _mfr_refo ][N-1]+=a[ _mfr_refo ][N]; }
43    if (N >= 3) {
44        b[N-1 ][ 1]+=a[N-1 ][ 0 ];
45        b[N-2 ][ 1]+=a[N-1 ][ 1 ];
46        b[N-1 ][ 2]+=a[N-1 ][ 1 ];
47        b[N-2 ][ 1]=b[N-2 ][ 1]/4;
48        for ( _mfr_ref1 = 2; _mfr_ref1 <= N-2; _mfr_ref1 ++) {
49            b[N-2 ][ _mfr_ref1 ]+=a[N-1 ][ _mfr_ref1 ];
50            b[N-1 ][ _mfr_ref1 +1]+=a[N-1 ][ _mfr_ref1 ];
51            b[N-1 ][ _mfr_ref1 -1]+=a[N-1 ][ _mfr_ref1 ];
52            b[N-2 ][ _mfr_ref1 ]=b[N-2 ][ _mfr_ref1 ]/4; }
53        b[N-2 ][N-1]+=a[N-1 ][N-1 ];
54        b[N-1 ][N-2]+=a[N-1 ][N-1 ];
55        b[N-2 ][N-1]=b[N-2 ][N-1]/4;
56        b[N-1 ][N-1]+=a[N-1 ][N]; }
57    for ( _mfr_ref1 = 1; _mfr_ref1 <= N-1; _mfr_ref1 ++) {
58        b[N-1 ][ _mfr_ref1 ]+=a[N][ _mfr_ref1 ];
59        b[N-1 ][ _mfr_ref1 ]=b[N-1 ][ _mfr_ref1 ]/4; }
60 }
```

Listing 5.2 – *Code Example Automatically Generated by IBB* (`a.out.c`).

The code generated by IBB from the '`xfor.c`' file is shown in Listing 5.2. Note the length of this code, as well as the number of loops, the duplicated instructions and the various array references which are more numerous than in the XFOR Version. Even this simple example shows the improvement in terms of productivity that the XFOR structure and the IBB compiler may offer for writing efficient code.

5.2.2 Writing The Input File

The input file of IBB is a source code written using any language among 'C', 'C++', 'Java' and 'C#', as long as the code parts to translate are syntactically valid in 'C'. IBB will only translate XFOR-loop nests according to the syntax described previously in chapter 4 ([*XFOR Syntax and Semantics*], page 59).

The IBB compiler also handles OpenMP [26] pragmas to parallelize loops. This means that the user can apply "`#pragma omp [parallel] for`" pragmas to XFOR-loops. In addition to this, IBB can process XFOR-loops with bounds expressed using *min* and *max* functions. Furthermore, bounds that are functions of array elements or function calls are also handled.

5.2.3 Calling IBB

IBB can be invoked using the following command:

```
IBB [ options ] input_file
```

The default behavior of IBB is to read the input code from a file and to write the generated code in an output file. If the user does not specify a name for the output file, the generated file will be named by default as 'a.out.c'. IBB's behavior and the shape of the output code can be controlled by the user, thanks to some options which are detailed in the following subsection.

5.2.4 IBB Options

Parallel XFOR Loops: It is possible to exhibit parallelism in XFOR loops by using OpenMP [26] pragmas. IBB copies the pragma from a parallelized XFOR loop to every for-loop resulting from its translation while preserving the convenient OpenMP clauses, as "shared" or "private". The new referential loop indices introduced by IBB are inserted inside the "private" clause.

Example 5.1 Let us consider the parallel XFOR nest in Listing 5.3. The corresponding IBB-generated code is displayed in Listing 5.4. Note the number of the parallel for-loops generated by IBB in the resulting code.

```
#pragma omp parallel for private (a)
xfor (i1=0,i2=1 ; i1<M,i2<M-1 ; i1++,i2++ ; 1,1 ; 0,1)
xfor (j1=2,j2=1 ; j1<M,j2<N ; j1++,j2++ ; 1,1 ; 2,0) {
  0: X[i1] = X[i1] - Y[j1];
  1: B[i2][j2] = a * B[i2][j2] + Y[i2]; }
```

Listing 5.3 – *Example of a Parallel XFOR Loop.*

```
if (M>=3) {
  if (N>=2)
    for (j=2 ; j<=M-1 ; j++)
      X[0]=X[0]-Y[j];
  if (N<=1) {
    #pragma omp parallel for private (a,i,j)
    for (i=0 ; i<=M-1 ; i++)
      for (j=2 ; j<=M-1 ; j++)
        X[i]=X[i]-Y[j];
  }
  if (N>=2) {
    #pragma omp parallel for private (a,i,j)
    for (i=1 ; i<=M-2 ; i++) {
      for (j=0 ; j<=min (1,N-2) ; j++)
        B[i][j+1]=a*B[i][j+1]+Y[i];
      for (j=2 ; j<=min (M-1,N-2) ; j++) {
```

```
17          X[i]=X[i]-Y[j];
18          B[i][j+1]=a*B[i][j+1]+Y[i];
19        }
20        for (j=max(2,N-1) ; j<=M-1 ; j++)
21          X[i]=X[i]-Y[j];
22        for (j=M ; j<=N-2 ; j++)
23          B[i][j+1]=a*B[i][j+1]+Y[i];
24      }
25    }
26    if (N>=2)
27      for (j=2 ; j<=M-1 ; j++)
28        X[M-1]=X[M-1]-Y[j];
29  }
```

Listing 5.4 – *IBB Generated Code from a Parallel XFOR Loop.*

Parallel Code '-omp': This option informs IBB to generate a parallel code using OpenMP sections[1]. If this option is not specified, the generated code will be sequential, unless "omp for" pragmas are present in the code (which are automatically handled). For instance, a user can generate two different versions of code from the same input file, by using or not the '`-omp`' flag. To illustrate this, consider the following example.

Example 5.2 Calling IBB with '`-omp`' option on an input file containing the code in Listing 5.5 will generate the code represented in Listing 5.7.

```
1 xfor (io=0,i1=10 ; io<10,i1<20 ; io++,i1++ ; 1,1 ; 0,0) {
2   0: S0(io);
3   1: S1(i1);
4 }
```

Listing 5.5 – *XFOR Example (*`input_file.c`*).*

Note that the code in Listing 5.7 is parallel, and it uses OpenMP sections. The code generated by default (sequential code) is represented in Listing 5.6.

```
1 for (_mfr_refo=0 ; _mfr_refo<=9 ; _mfr_refo++) {
2   S0(_mfr_refo);
3   S1(_mfr_refo+10);
4 }
```

Listing 5.6 – *Code Generated from* `input_file.c` *Without Using Option '*`-omp`*'.*

[1]IBB has a restriction regarding parallel XFOR loops. The '`-omp`' option shouldn't be used if the XFOR code contains "`#pragma omp [parallel] for`" pragmas.

```
for ( _mfr_refo =0 ; _mfr_refo <=9 ; _mfr_refo ++) {
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      So ( _mfr_refo ) ;
    }
    #pragma omp section
    {
      S1 ( _mfr_refo +10) ;
    }
  }
}
```

Listing 5.7 – *Code Generated from* `input_file.c` *Using Option '*`-omp`*'.*

Adding Constraints on Parameters '-positive_parameters': This option tells IBB that all the parameters that appear in the XFOR-loops are positive. It is useful to simplify the tests generated by CLooG.

Debugging Mode '-debug': This option asks IBB to display on the standard output the built OpenScop (This OpenScop is the input for the code generator CLooG [11], see Section 5.4, page 80).

Output '-o <output>': This option sets the output file name. Default name is '`a.out.c`'.

Help '–help' or '-h': This option asks IBB to Display a short help.

5.3 Reference Domain

As mentioned in chapter 4 ([*XFOR Syntax and Semantics*], page 62), the XFOR-loop indices are expressed relatively to a global referential domain. Therefore, referential for loops generated in the final code iterate over this domain. In this section, we explain in details how IBB computes the domains of these loops.

Consider an XFOR-loop of depth one. The referential for-loops cadencing the XFOR execution must scan a sufficient number of iterations. Let us denote by f the number of for-loops defined in the XFOR header (*i.e.* XFOR indices). By computing the disjoint union of all for-loops iteration domains, we obtain a set of disjoint domains D_i on which some of the f loops overlap. Let us denote by lb_i, ub_i, $grain_i$ and $offset_i$, $i = 1..f$ the parameters characterizing each for-loop in the XFOR header. Let us set:

$$nlb_i = offset_i$$
$$nub_i = (ub_i - lb_i + 1)/(lcm(grain_k)/grain_i) + offset_i, k = 1..f$$

which define the lower and upper bounds of each loop in the referential domain, since nlb_i consists in shifting the domain and nub_i consists in shrinking the domain by a factor equal to the *least common Multiple* of all grains divided by the current grain. In fact, the grain is translated by loop unrolling in order to have an efficient equivalent code. Each index j will be unrolled by a factor equal to the $lcm(grain_i)/grain_j, i = 1..f$.

The disjoint union of the D_i's is computed using these latter bounds, resulting in disjoint domains R_j. The initial index value of the referential domain is $MIN_{i=1..f}\,(nlb_i)$. Hence, the total number of iterations in the referential domain, before compression, is:

$$MAX_{i=1..f}\,(nub_i) - MIN_{i=1..f}\,(nlb_i) + 1 \tag{5.1}$$

More generally for any XFOR-loop nest, the computation of the referential domain is performed in four steps. First, each iteration domain associated to one for-loop nest composing an XFOR-loop nest is normalized (*i.e.* the lower bounds are shifted to 0). Second, it is translated from the origin according to its offsets. Notice that values actually taken by the indices of the for-loop nests are not defining their positions in the referential domain. Third, each domain D_i is unrolled by a factor equal to $lcm(grain_k)/grain_i, k = 1..f$. Finally, a disjoint union is computed and resulting in a union of convex domains R_j.

5.4 Code Generation

Figure 5.1 illustrates the functioning of the XFOR-compiler. While parsing the input source file, IBB copies the code to the output file till it identifies an XFOR-loop nest. The identified XFOR nest is sent to the XFOR-Translator. The latter translates it into equivalent for-loop nests. Then, IBB copies the generated code into the output file. This process is repeated until the end of the input file. Once the parsing is finished, the generated code is formatted.

The XFOR-Translator is responsible for generating a *C* code made of "regular" for-loops that are semantically equivalent to a given XFOR-loop nest. This is done in three steps (see Figure 5.2); First, the parser analyzes the XFOR-loop nest and generates an abstract syntax tree (AST) that contains all the information about the XFOR indices. Second, the OpenScop generator turns the index domains into polytopes over a common referential domain, and finally, the code generator CLooG [11] generates the scanning code for the union of these polytopes.

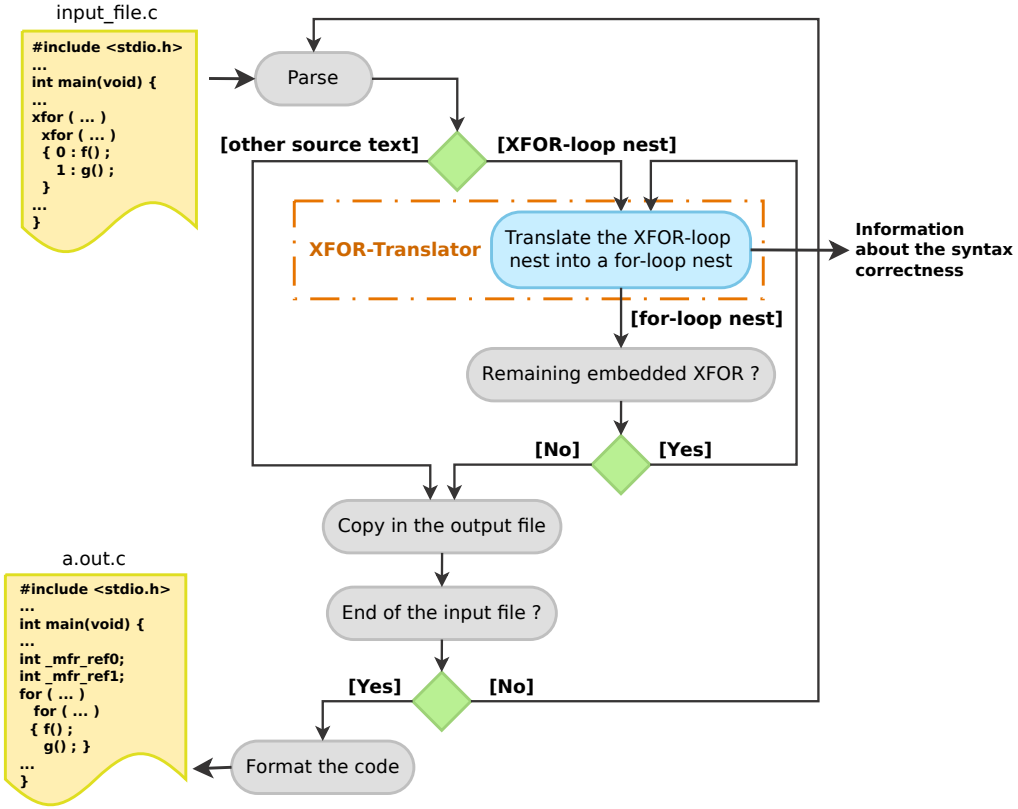


Figure 5.1 – *IBB: XFOR-Compiler.*

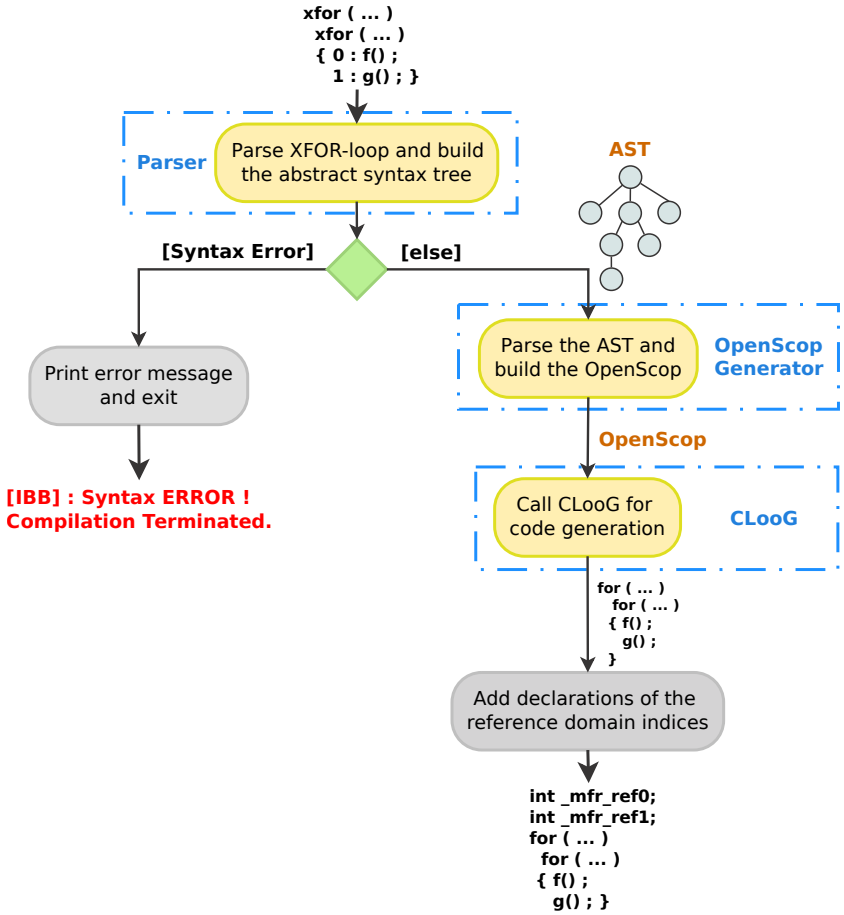


Figure 5.2 – *XFOR-Translator.*

5.4.1 Parser

The parser module, or syntactic analyzer, performs the syntax validation and also the generation of the abstract syntax tree, that contains all the information about the XFOR-loop nests. The parser works in coordination with the lexer.

The XFOR-loop nest data are saved into an N-ary Tree provided by glib [64]. Nodes of the tree have the following structure:

```
struct GNode {
  gpointer data;
  GNode   *next;
  GNode   *prev;
  GNode   *parent;
  GNode   *children;
};
```

The data field points to an array of pointers (*i.e.* GPtrArray, glib [64]). Each element of the array points to a given index of the XFOR-loop. The indices information is saved in a structure called `InfoFor` given below:

```
typedef struct InfoFor InfoFor;
struct InfoFor
{   char *indice;
    char *opRel;
    char *bInf;
    char *bSup;
    int  incr;
    int  grain;
    char *offset;
    GSList * instr;
    GSList * bInfCloog;
    GSList * bSupCloog;
    InputClooG * offsetCloog;
    InfoPragma * ompPragma;
};
```

where:

- **indice** is the name of the index.
- **opRel** is the relational operator.
- **bInf** is the lower bound.
- **bSup** is the upper bound.
- **incr** is the stride.
- **grain** is the grain.
- **offset** is the offset.

- **instr** is a linked list used to save the list of instructions.
- **bInfCLooG** is a representation of the lower bound that will be used in the construction of the DOMAIN relation of the OpenScop.
- **bSupCLooG** is a representation of the upper bound useful for the construction of the DOMAIN relation of the OpenScop.
- **offsetCLooG** is a dedicated representation of the offset that will be used in the construction of the SCATTERING relation of the OpenScop.
- **ompPragma** indicates if the loop is parallel or not.

The instructions are saved in a singly-linked list (*i.e.* GSList, glib [64]). Each element of the linked list points to the following structure:

```
typedef struct Instruc Instruc;
struct Instruc
{   char* instr;
    int   index;
};
```

where:

- **instr** is a string containing the statement body.
- **index** defines the relative position of the statement regarding other XFOR loops in the nest. It is particularly useful when dealing with imperfectly nested XFOR loops. The statements that appear at the beginning of the loop body will have an **index** equal to "-1". The one that appear after the first XFOR loop, will have "0" as **index** and so on.

5.4.2 OpenScop Generator

IBB builds a specific polyhedral, matrix-based, representation of XFOR-loop nests using the OpenScop format [7]. The latter will be the input to the code generator CLooG. This format makes it easy to replace the original indices with their referential domain equivalent. If the *'-debug'* option is specified, IBB will print the generated OpenScop to the standard output.

```
1 xfor (i0=5,i1=10 ; i0<50,i1<100 ; i0++,i1+=2 ; 1,3 ; 0,1) {
2   0: S0(i0);
3   1: S1(i1); }
```

Listing 5.8 – *XFOR Code Example.*

In this subsection, we give more details about the OpenScop generation. For instance, let us consider the xfor-loop nest code in Listing 5.8. Each iteration domain of each for-loop nest composing an XFOR nest defines a Z-polytope, *i.e.*, a lattice of integer points delimited by a finite polyhedra:

1. $5 \leq i0 \leq 49$
2. $10 \leq i1 \leq 99$

First, these iteration domains are normalized (*i.e.* shifted to 0). These two equations are then expressed in the DOMAIN relation of the OpenScop:

1. $0 \leq i0 \leq 44$
2. $0 \leq i1 \leq 89$

In order to preserve the semantics of the original program, each index i appearing in the statements should be substituted by $i + lower\_bound\_i$. In the case of the previous example, the statements will be transformed as shown below:

1. $S0((i0+5))$;
2. $S1((i1+10))$;

The grain is translated to a loop unrolling. Thus every statement i will be duplicated $lcm(grain_j)/grain_i$, times. Where $lcm(grain_j)$ is the *least common multiple* of all the indices present in the XFOR header. Second, the referential domain is implicitly created by expressing each created statement k into a common basis of referential indices, *ref_index*, according to the relation 5.2.

$$\frac{lcm(grain_j)}{grain_k} \times incr_k \times ref\_index = original\_index_k + \frac{offset_k}{lcm(grain_j)} - k \qquad (5.2)$$

for every k = 0 .. ($lcm(grain_j)/grain_i$-1)

where $incr_i$ represents the corresponding increment. The respective grains and offsets must be considered in order to define the union of Z-polytopes, corresponding to the overlapping of the domains that are defined by the XFOR structure. These relations define the scheduling of the statements. They are defined in the OpenScop within the SCATTERING relations. In order to have a complete scheduling, the order of the statement in the XFOR body must be taken into account. This is done using an additional scattering dimension. In the case of our example, by adding an additional scattering dimensions $c2$ and $c3$. We show below the OpenScop corresponding to the previous example, which is automatically generated by IBB.

```
<OpenScop>
# ============================================== Global
# Language
 C
```

```
# Context
 CONTEXT
1 2 0 0 0 0
# e/i|  1
   1    0    ## 0 >= 0
# Parameters are provided
0
# Number of statements
4
# ================================================ Statement 1
# Number of relations describing the statement:
2
# ------------------------------------------------  1.1 Domain
 DOMAIN
2 3 1 0 0 0
# e/i| i0 |  1
   1    1    0    ## i0 >= 0
   1   -1   44    ## -i0+44 >= 0
# ------------------------------------------------  1.2 Scattering
 SCATTERING
3 6 3 1 0 0
# e/i| c1   c2   c3 | i0 |  1
   0    3    0    0   -1    0    ## 3*c1-i0 == 0
   0    0    1    0    0    0    ## c2 == 0
   0    0    0    1    0    0    ## c3 == 0
# ------------------------------------------------  1.3 Access
# NULL relation list
# -----------------------------------  1.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
1
# List of original iterators
 i0
# Statement body expression
 S0((i0+5));
</body>
# ================================================ Statement 2
# Number of relations describing the statement:
2
# ------------------------------------------------  2.1 Domain
 DOMAIN
2 3 1 0 0 0
# e/i| i0 |  1
   1    1    0    ## i0 >= 0
   1   -1   44    ## -i0+44 >= 0
# ------------------------------------------------  2.2 Scattering
 SCATTERING
3 6 3 1 0 0
# e/i| c1   c2   c3 | i0 |  1
```

```
   0    3    0    0   -1    1    ## 3*c1-i0+1 == 0
   0    0    1    0    0   -1    ## c2 == 1
   0    0    0    1    0    0    ## c3 == 0
# ------------------------------------------  2.3 Access
# NULL relation list
# ----------------------------------  2.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
1
# List of original iterators
 i0
# Statement body expression
 S0((i0+5));
</body>
# ============================================== Statement 3
# Number of relations describing the statement:
2
# ------------------------------------------  3.1 Domain
 DOMAIN
2 3 1 0 0 0
# e/i| i0 |  1
   1    1    0    ## i0 >= 0
   1   -1   44    ## -i0+44 >= 0
# ------------------------------------------  3.2 Scattering
 SCATTERING
3 6 3 1 0 0
# e/i| c1   c2   c3 | i0 |  1
   0    3    0    0   -1    2    ## 3*c1-i0+2 == 0
   0    0    1    0    0   -2    ## c2 == 2
   0    0    0    1    0    0    ## c3 == 0
# ------------------------------------------  3.3 Access
# NULL relation list
# ----------------------------------  3.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
1
# List of original iterators
 i0
# Statement body expression
 S0((i0+5));
</body>
# ============================================== Statement 4
# Number of relations describing the statement:
2
# ------------------------------------------  4.1 Domain
 DOMAIN
2 3 1 0 0 0
```

```
# e/i| i1 |  1
   1    1    0    ## i1 >= 0
   1   -1   88    ## -i1+88 >= 0
# ----------------------------------------------  4.2 Scattering
 SCATTERING
3 6 3 1 0 0
# e/i| c1   c2   c3 | i1 |  1
   0    2    0    0   -1    0    ## 2*c1-i1 == 0
   0    0    1    0    0   -1    ## c2 == 1
   0    0    0    1    0   -1    ## c3 == 1
# ----------------------------------------------  4.3 Access
# NULL relation list
# -------------------------------------  4.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
1
# List of original iterators
 i1
# Statement body expression
 S1((i1+10));
</body>
</OpenScop>
```

5.4.3 CLooG

The third step is performed using the CLooG library [11, 12, 13, 51] devoted to generate for-loops that reaches each integral point of one or several parametrized polyhedra. CLooG is designed to avoid control overhead and to produce very effective code (see Subsection 2.3.4, Chapter 2, page 42).

5.5 Conclusion

In this chapter, we presented the source-to-source XFOR-Compiler: "IBB". We described the different parts composing IBB, and for each one we detailed its functioning and its contribution in the code generation process. The IBB compiler generates both sequential and parallel code (using OpenMP sections) and it also handles OpenMP pragmas for loop parallelization.

In the next chapter, we will focus on major programming strategies allowing programmers to take advantages of XFOR to produce very efficient code.

XFOR Programming Strategies 6

6.1 Introduction

Writing efficient and optimized XFOR codes can become easy and intuitive. The programmer can determine whether the XFOR construct is beneficial for his target loop nest. Actually, The XFOR construct is well suited for writing data-locality aware programs. For instance, it allows the programmer to explicitly control the data reuse distances between the different statements and tune various parameters to optimize the code, which in-turn control the performance of the code.

In this chapter we introduce some techniques to write efficient XFOR codes and to improve the execution time of a given code. We first define the data reuse distance in section 6.2. Then, three different programming strategies are described; the programmer may choose to minimize the inter-statement data reuse distance (subsection 6.2.1), or the intra-statement data reuse distance (subsection 6.2.2) along with loop parallelization (section 6.3). We illustrate each described strategy with a step by step example. In section 6.4, we evaluate XFOR codes performance versus sequential-vectorised and parallel executions of non-XFOR original codes.

6.2 Data Reuse Distance Minimization

The basic architecture of memory hierarchies promotes the minimization of data reuse distances to reduce the execution times. Efficient use of the cache hierarchy is a key factor of the performance of a program. In the following, we consider reuse distance as being the number of iterations between two successive accesses (read or write) to the same memory location [65]. The XFOR structure allows to explicitly bring closer instances of the statements that share common data, thanks to the grain and offset parameters.

When handling several loop statements inside a loop nest, the first step is to identify data dependencies that occur between them, *i.e.*, relations between iteration points of possibly different domains, characterizing dependencies of

types Read-After-Write, Write-After-Read or Write-After-Write. Read-After-Read relations dependencies are also considered, since it also implies data reuse. Convenient offset and grain values regarding two dependent statements can be determined with the support of distance vectors.

Let $a[f(I_0)]$ and $a[g(I_1)]$ be two array references appearing in two dependent statements S_0 and S_1 inside an XFOR loop nest, where I_k denotes the iteration vector of the loop indices enclosing S_k. Let $f()$ and $g()$ denote affine functions. Let O_0 and O_1 be their respective vectors of offset values. The distance vector d from S_0 to S_1 is defined by:

$$d = g(I_1) + O_1 - (f(I_0) + O_0) \tag{6.1}$$

The offsets has to be set in order to get a lexicographically positive vector in order to ensure the semantic correctness of the schedule. Null vectors are allowed if the dependent statements are written in a correct dependence-aware order inside the loop body. Data reuse distance minimization is performed by minimizing the components of d, and by minimizing primarily the outermost indices, since it defines the longest reuse distances which are carried by the outermost loops.

However, even if minimizing data reuse distances is beneficial in general, one must pay attention at too short reuse distances that may result in performance loss due to stalls in the processor pipeline. Accesses to common data where at least one access is a write, have to be slightly spaced to avoid pipeline hazards. This can be achieved either by inserting in between at least one statement referencing other data, or in avoiding too close common accesses by slightly increasing the offset value which is associated with one of the accesses [24, 65, 66].

In the following, we describe programming strategies for inter and intra-statement data reuse distance minimization, *i.e.*, reuse between statements belonging originally to separated loop bodies, and reuse between statements belonging originally to a single loop body, respectively. Finally, we show how to exhibit parallelism with the XFOR construct.

6.2.1 Minimizing Inter-Statement Data Reuse Distance

A typical case for minimizing inter-statement data reuse distance is when handling several iteration domains scanned by successive loop nests. The minimization is done by overlapping the domains through shifting (offset) and unrolling (grain), while respecting data dependencies, and according to the lexicographic order of the loop indices. The final schedule can then be described by a XFOR-loop nest.

Illustrative Example. Consider the loop kernel of code *jacobi-2d-imper* in Listing 6.1, extracted from the Polybench benchmark suite [27]. In the first loop nest, array B is updated (one write access) using array A (five read accesses). And in the second one, an element of array A is updated using the same element of array B. $A[i][j]$ can be updated only if $B[i-1][j]$ and $B[i][j-1]$ have been updated, since they are computed using the initial values of matrix A. So, the minimal value of the offset that guarantees the respect of the dependencies is equal to (1, 1) for the second loop nest. This means that, this loop nest must be shifted by 1 in the i-direction and by 1 in the j-direction. The XFOR-loop permitting to minimize data-reuse distances and respecting dependencies at the same time is represented in Listing 6.2.

```
for (i = 2; i < n - 1; i++)
  for (j = 2; j < n - 1; j++)
    B[i][j] = 0.2*(A[i][j]+A[i][j-1]+A[i][1+j]+A[1+i][j]+A[i-1][j]);
for (i = 2; i < n-1; i++)
  for (j = 2; j < n-1; j++)
    A[i][j] = B[i][j];
```

Listing 6.1 – *2D Jacobi Stencil Computation.*

```
xfor(i1=2,i2=2; i1<n-1,i2<n-1; i1++,i2++; 1,1; 0,1)
xfor(j1=2,j2=2; j1<n-1,j2<n-1; j1++,j2++; 1,1; 0,1) {
  0: B[i1][j1] = 0.2*(A[i1][j1]+A[i1][j1-1]+A[i1][1+j1]
                                +A[1+i1][j1]+A[i1-1][j1]);
  1: A[i2][j2] = B[i2][j2]; }
```

Listing 6.2 – *XFOR Implementation of 2D Jacobi Stencil Computation.*

6.2.2 Minimizing Intra-Statement Data Reuse Distance

The second programming strategy is devoted to the minimization of intra-statement data reuse distances between statements of a single loop body. In this strategy, an iteration domain is split into several domains, each one being associated to a subset of statements of the original loop body, or a partial computation of an original arithmetic statement, if such a decomposition is allowed regarding mathematical properties and arithmetic precision. This approach may also be useful to optimize codes containing conditionals with modulo expressions of the loop indices, as it is also illustrated with the Red-Black example in the following (see Subsection 6.2.3). Thus, statements can be re-scheduled by overlapping these domains as described in the previous strategy.

Illustrative Example. Consider the loop nest in Listing 6.3 which was extracted from the *seidel* program of the polybench benchmark suite [27]. This code is

a classic stencil computation where the eight neighbors of each grid point are accessed to update the point with their average. Thus, each array element is reused eight times to compute eight different averages. One important issue is, that following the canonical lexicographic order of the 2-level loop nest, each point is updated using four neighbors that are already updated, while the four others are still assigned their initial values (see Figure 6.1).

```
1 for (t = 0; t <= tsteps - 1; t++)
2  for (i =1; i<= n-2; i++)
3   for (j=1; j<=n-2; j++)
4    A[i][j] = (A[i-1][j-1]+A[i-1][j]+A[i-1][j+1]+A[i][j-1]+A[i][j]
5               +A[i][j+1]+A[i+1][j-1]+A[i+1][j]+A[i+1][j+1])/9.0;
```

Listing 6.3 – *Main For-Loop Nest of the Seidel Code.*

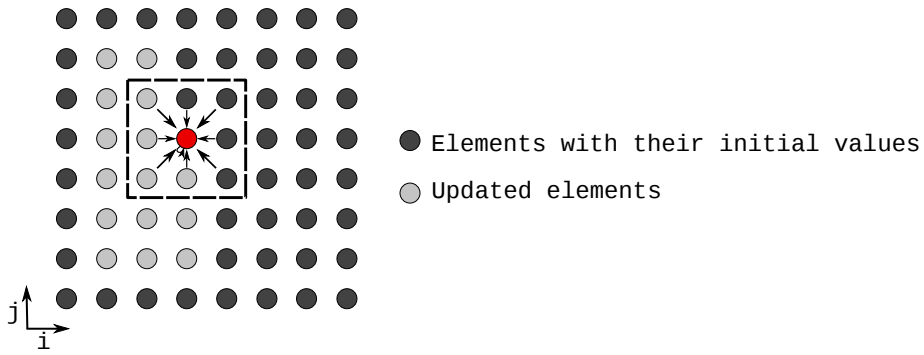


Figure 6.1 – *Seidel Domain.*

To minimize the data reuse distances, we split the statement into five elementary statements: four statements each consisting of adding one neighbor that has not yet been updated, and one last statement consisting of adding all the remaining updated neighbors and of computing the average. The resulting loop body is shown in Listing 6.4 as the body of an XFOR loop structure nest. In this version of the loop body, read accesses to a given element of array `A` made by statements 0 to 3 are all made once and for all at each iteration, while successively taking part of the computation of four stencil computations in which they appear. The code required to implement such a schedule by using standard for-loops would be tedious and complex to program.

Let us focus on the offset parameters of the example, appearing at the very end of the XFOR headers. For instance, the first 0 in the outer loop list and the first 2 of the inner loop list – noted as (0,2) in the following – tell that the first statement is shifted by 0 in the outer loop direction and by 2 in the inner loop direction, thus resulting in an execution behavior similar to the one of the corresponding statement in comments in Listing 6.4. Note that array elements that are actually accessed inside an XFOR-loop iteration are given by subtracting

the offset values to the indices inside the array reference functions, so resulting in the code in comments.

```
for (t = 0; t <= tsteps-1; t++)
xfor (i0=1,i1=1,i2=1,i3=1,i4=1;
      i0<=n-2,i1<=n-2,i2<=n-2,i3<=n-2,i4<=n-2;
      i0++,i1++,i2++,i3++,i4++;
      1,1,1,1,1; /* grains */
      0,1,1,1,1 ) /* offsets */ {
xfor (j0=1,j1=1,j2=1,j3=1,j4=1;
      j0<=n-2,j1<=n-2,j2<=n-2,j3<=n-2,j4<=n-2;
      j0++,j1++,j2++,j3++,j4++;
      1,1,1,1,1; /* grains */
      2,0,1,2,2 ) /* offsets */ {

      0: A[i0][j0] += A[i0][j0+1];
      1: A[i1][j1] += A[i1+1][j1-1];
      2: A[i2][j2] += A[i2+1][j2];
      3: A[i3][j3] += A[i3+1][j3+1];
      4: A[i4][j4] = (A[i4][j4]+A[i4-1][j4-1]+A[i4-1][j4]
                      +A[i4-1][j4+1]+A[i4][j4-1])/9.0;

/*    0: A[i][j-2]   += A[i][j-1];
      1: A[i-1][j]   += A[i][j-1];
      2: A[i-1][j-1] += A[i][j-1];
      3: A[i-1][j-2] += A[i][j-1];
      4: A[i-1][j-2] = (A[i-1][j-2]+A[i-2][j-3]+A[i-2][j-2]
                      +A[i-2][j-1]+A[i-1][j-3])/9.0; */ }}
```

Listing 6.4 – *XFOR Implementation of Seidel Stencil Computation.*

Dependent statements must be conveniently scheduled thanks to their respective offset values. The final computation of the average (statement 4), using array elements that were already updated, has to be performed after the last element update (statement 3). Thus statement 4 has been assigned the greatest couple of offset values (1,2), which is the same as for statement 3. Since statement 3 is appearing before statement 4 in the loop body, they are executed in this order and dependencies are respected.

6.2.3 Red-Black Gauss-Seidel

Actually, in order to produce more efficient code, the two precedent strategies may be applied together. We illustrate this using the *Red-Black Gauss-Seidel* algorithm. This code is composed of two phases. The first phase updates the red elements of a grid, which are every second point in the i and j directions, starting from the first point at the bottom left corner, by using their North-South-East-West (NSEW) neighbors, which are black elements (see Figure 6.2, left). The second phase updates the black elements from the red ones using the

same stencil f [24]. For a 2D $N \times N$ problem, the standard code is of the form shown in Listing 6.5 (the border elements initialization has been omitted). This code is not handled by the automatic loop optimizer Pluto [6, 16] which is unable to handle non-linear conditionals nor dilated domains.

```
1  // Red phase
2  for (i=1; i < N-1; i++)
3   for (j=1; j < N-1; j++)
4    if ((i+j) % 2 == 0)
5     u[i][j] = f(u[i][j+1], u[i][j-1], u[i-1][j],u[i+1][j]);
6  // Black phase
7  for (i=1; i < N-1; i++)
8   for (j=1; j < N-1; j++)
9    if ((i+j) % 2 == 1)
10    u[i][j] = f(u[i][j+1], u[i][j-1], u[i-1][j],u[i+1][j]);
```

Listing 6.5 – *Red-Black Gauss-Seidel Code.*

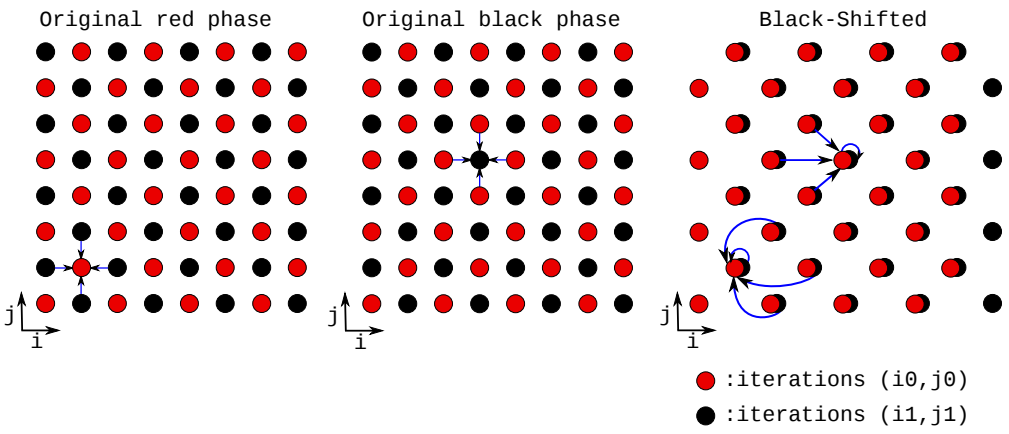


Figure 6.2 – *Gauss-Seidel Red and Black Domains, Original (Left) and Black-Shifted (Right).*

This example obviously defines two dependent iteration domains: the red one which includes points (i, j) such that $(i + j)$ *modulo* $2 = 0$, and the black one with points such that $(i + j)$ *modulo* $2 = 1$. Each black point depends on its four NSEW red neighbors. Both domains can be scheduled such that any black point is computed as soon as all four red points from which it depends have been computed. This means that according to the lexicographic order, any black point can be computed as soon as its eastern neighbor is available, since it is the last computed point of the stencil. Hence, a shift of the black domain of one unit in direction east, *i.e.*, along the i axis, overlaps black points with their respective eastern red points (see Figure 6.2, right). Both red and black points define the body of the XFOR-loop nest where the red statement precedes the black statement, in order to respect their dependencies. The resulting XFOR nest is shown in Listing 6.6.

```
xfor (io=1,i1=1; io<N-1,i1<N-1; io++,i1++; 1,1; 0,1)
xfor (jo=1,j1=1; jo<N-1,j1<N-1; jo++,j1++; 1,1; 0,0) {
 0: {if ((io+jo) % 2 == 0)
       u[io][jo] = f(u[io][jo+1], u[io][jo-1],u[io-1][jo],u[io+1][jo]);}
 1: {if ((i1+j1) % 2 == 1)
       u[i1][j1] = f(u[i1][j1+1], u[i1][j1-1],u[i1-1][j1],u[i1+1][j1]);}}
```

Listing 6.6 – *First Red-Black Gauss-Seidel XFOR Code.*

More formally, this XFOR code can be deduced from the distance vectors. Accesses to array elements occurring during the black phase are all dependent on updates performed during the red phase. Let O_{0i} and O_{0j} (respectively O_{1i} and O_{1j}) be the offset values assigned to statement 0 (respectively 1) at loop levels i and j. The four reads performed by statement 1 are all defining a distance vector regarding the write of statement 0 (Read-After-Write):

$$d_1 = \begin{pmatrix} O_{1i} - O_{0i} \\ O_{1j} - O_{0j} + 1 \end{pmatrix} \quad d_2 = \begin{pmatrix} O_{1i} - O_{0i} \\ O_{1j} - O_{0j} - 1 \end{pmatrix}$$

$$d_3 = \begin{pmatrix} O_{1i} - O_{0i} - 1 \\ O_{1j} - O_{0j} \end{pmatrix} \quad d_4 = \begin{pmatrix} O_{1i} - O_{0i} + 1 \\ O_{1j} - O_{0j} \end{pmatrix}$$

The write of statement 1 is also defining four distance vectors regarding the reads of statement 0 (Write-After-Read):

$$d_5 = \begin{pmatrix} O_{1i} - O_{0i} \\ O_{1j} - O_{0j} - 1 \end{pmatrix} \quad d_6 = \begin{pmatrix} O_{1i} - O_{0i} \\ O_{1j} - O_{0j} + 1 \end{pmatrix}$$

$$d_7 = \begin{pmatrix} O_{1i} - O_{0i} + 1 \\ O_{1j} - O_{0j} \end{pmatrix} \quad d_8 = \begin{pmatrix} O_{1i} - O_{0i} - 1 \\ O_{1j} - O_{0j} \end{pmatrix}$$

To minimize the components of these vectors while ensuring their lexicographic non-negativeness, the best choice is obviously to set O_{1i} to 1 and the other values to 0, resulting in the following valid distance vectors:

$$d_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad d_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \quad d_3 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad d_4 = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

$$d_5 = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \quad d_6 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad d_7 = \begin{pmatrix} 2 \\ 0 \end{pmatrix} \quad d_8 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

In addition to inter-statement data reuse distance minimization, we can opt to improve the intra-statement data locality. In fact, the XFOR program of Listing 6.6 contains guards testing the parity of $(i + j)$. However, these conditionals

yield empty iterations that can be removed by splitting each of the red and black domains into two red and two black domains according to the parity of the indices (*i.e.* defined respectively by i *modulo* $2 = 0$ and i *modulo* $2 = 1$) and by expressing the conditionals using 2-grain parameters, 2-increments and convenient offsets. The resulting XFOR code is shown in Listing 6.7, where statements 0 and 1 are associated with the red domain and statements 2 and 3 are associated with the black domain.

```
xfor (i0=1,i1=2,i2=1,i3=2; i0<N-1,i1<N-1,i2<N-1,i3<N-1;
      i0+=2,i1+=2,i2+=2,i3+=2; 2,2,2,2; 0,1,1,2)
xfor (j0=1,j1=2,j2=2,j3=1; j0<N-1,j1<N-1,j2<N-1,j3<N-1;
      j0+=2,j1+=2,j2+=2,j3+=2; 2,2,2,2; 0,1,1,0) {
 0: u[i0][j0] = f(u[i0][j0+1], u[i0][j0-1], u[i0-1][j0],u[i0+1][j0]);
 1: u[i1][j1] = f(u[i1][j1+1], u[i1][j1-1], u[i1-1][j1],u[i1+1][j1]);
 2: u[i2][j2] = f(u[i2][j2+1], u[i2][j2-1], u[i2-1][j2],u[i2+1][j2]);
 3: u[i3][j3] = f(u[i3][j3+1], u[i3][j3-1], u[i3-1][j3],u[i3+1][j3]); }
```

Listing 6.7 – *Second Red-Black Gauss-Seidel XFOR Code.*

6.3 Parallel XFOR

Beside data locality, parallelization is obviously an important issue with current multicore processors. In this section, we address two main parallelization opportunities considered by most compilers: vectorization and loop parallelization. We show that XFOR structures promotes better effectiveness of the parallel codes.

Since XFOR structures are used to minimize data reuse distances, successive iterations are probably strongly data-dependent, thus preventing loop parallelization. However, some offsets may be increased at a given XFOR-loop level to enlarge reuse distances and exhibit slices of independent iterations, as soon as dependent memory references are performed by domain-separated instructions. Each slice can then be parallelized and all the slices being run serially by using an enclosing for-loop. This approach results in a software pipelining where data reuse distances are still sufficiently small to take advantage of memory data locality, and sufficiently large for efficient parallelization.

6.3.1 Parallel XFOR Loop

The IBB compiler handles OpenMP pragmas for XFOR-loop parallelization (`#pragma omp [parallel] for`). IBB copies them above every for-loop resulting from a parallelized XFOR-loop in the output source code, while preserving the convenient OpenMP clauses "`shared`" or "`private`". The new referential loop indices introduced by IBB are inserted inside the "`private`" clauses.

This feature provides the opportunity of parallelizing XFOR-loops bodies. Obviously, the parallelization of an XFOR-loop must take care of data dependencies in order to respect the semantics. Here also, the offset parameter provides an easy way to generate several valid solutions, as it is illustrated by the example below.

Let us go back to the *Red-Black Gauss-Seidel* example. As it can be observed on the right of Figure 6.2, for a fixed value of the i-index, all j-iterations may be performed in parallel, since no dependence occurs along the j-axis, and the dependence between body statements are not violated thanks to their preserved order. However, parallelization of the outer XFOR-loop may provide better performance thanks to a larger parallelism grain and fewer synchronizations. In order to respect data dependencies, a larger offset must be applied to the black domain. So, it increases dependence distances and enables the parallel execution of several successive iterations of the outer loop. More precisely, an offset incremented by $2 \times k$ allows k successive iterations of the outer XFOR-loop to run in parallel. To implement this solution, an enclosing for-loop is inserted in order to scan iterations per groups of k iterations, while the outer XFOR-loop scans the parallel iterations inside each group. This newly introduced for-loop requires some related modifications of the XFOR-loop bounds and offsets. The resulting code is shown in Listing 6.8.

```
1  #define k NUMBER_OF_THREADS
2  for(i=1; i < (N-1)/2*k; i+=2*k)
3  #pragma omp parallel for private (io,i1,i2,i3) \
4                 private (jo,j1,j2,j3)   firstprivate  (i)  shared (u)
5  xfor (io=i, i1=i+1, i2=i, i3=i+1 ;
6        io < min(i+2*k,N-1), i1 < min(i+1+2*k,N-1),
7        i2 < min(i+2*k,N-1), i3 < min(i+1+2*k,N-1) ;
8        io+=2, i1+=2, i2+=2, i3+=2 ; 2,2,2,2 ; i-1,i,1+i+2*k,2+i+2*k)
9  xfor (jo=1, j1=2, j2=1, j3=2;
10       jo < N-1, j1 < N-1, j2 < N-1, j3 < N-1;
11       jo+=2, j1+=2, j2+=2, j3+=2 ; 2,2,2,2 ; 0,1,0,1) {
12   0: u[io][jo] = f(u[io][jo+1], u[io][jo-1], u[io-1][jo],u[io+1][jo]);
13   1: u[i1][j1] = f(u[i1][j1+1], u[i1][j1-1], u[i1-1][j1],u[i1+1][j1]);
14   2: u[i2][j2] = f(u[i2][j2+1], u[i2][j2-1], u[i2-1][j2],u[i2+1][j2]);
15   3: u[i3][j3] = f(u[i3][j3+1], u[i3][j3-1], u[i3-1][j3],u[i3+1][j3]); }
```

Listing 6.8 – *Parallelized Red-Black Gauss-Seidel XFOR Code.*

6.3.2 Vectorized XFOR

Vectorization depends on two main parameters: data dependence and alignment. Processors' SIMD units require fixed-size vectors, say sv, of equally spaced data , *i.e.*, spaced by constant memory strides. Thus, sv iterations are run in parallel thanks to the SIMD unit. Mainstream compilers featuring automatic

vectorization also require straightforward memory access patterns. Thus, the XFOR programming strategy promoting vectorization builds bodies of statements whose inter data reuse distance is strictly greater than the SIMD vector size, and the alignment of accessed data complies with the processor requirements. A convenient adjustment of the offset values allows easy compliance with these requirements. We illustrate this programming strategy using an XFOR implementation of a 2D Jacobi stencil.

Illustrative Example. Consider the XFOR-loop nest in Listing 6.9 which implements a 2D Jacobi stencil computation. This nest includes a body of two statements carrying data dependencies regarding their accesses to arrays A and B. Offsets set to 1 for the second statement, at both loop levels, are the minimum values ensuring simultaneously minimized reuse distances and respected dependencies. Current general-purpose processors generally contain 128 or 256 bits vector registers. Assuming registers of 256 bits and array elements of type double float (64 bits), dependence distances should be greater or equal to 5, in order to build vectors of 4 independent elements. Thus, automatic vectorization is made possible by setting the innermost XFOR-loop with a second offset greater or equal to 5. Experiments show that an offset equal to 6 provides the best vectorization performance using the GCC and ICC compilers and running the code on an Intel Xeon x86-64 processor.

```
1 xfor (io=1,i1=1; io<N-1,i1<N-1; io++,i1++; 1,1; 0,1)
2 xfor (jo=1,j1=1; jo<N-1,j1<N-1; jo++,j1++; 1,1; 0,6) {
3   0: B[io][jo] = 0.2*(A[io][jo]+A[io][jo-1]
4                       +A[io][jo+1]+A[io+1][jo]+A[io-1][jo]);
5   1: A[i1][j1] = B[i1][j1]; }
```

Listing 6.9 – *Vectorizable XFOR Implementation of 2D Jacobi Stencil Computation.*

6.4 XFOR Performance Evaluation

6.4.1 Experimental Setup

Experiments have been conducted on an Intel Xeon X5650 6-core processor 2.67 GHz running Linux 3.16.0. Our set of benchmarks is derived from the Polyhedral Benchmark suite [27].

6.4.2 Sequential-Vectorized Code Measurements

Every code has been rewritten using the XFOR structure. Original and XFOR versions have been compiled using GCC 4.8.2 and ICC 14.0.3 with options `O3` and `march=native`. Automatic vectorization is enabled to take advantage of it

when possible. Execution times of the main loop kernels, original and rewritten as XFOR loops, are given in seconds. Table 6.1, gives the resulting speed-up (*original time / XFOR time*) for the sequential and automatically vectorized code versions.

XFOR programs yield impressive speedups. For instance, the XFOR version of the *correlation* code is more than 6 times faster than the original one. Also, we can mention the case of the *covariance* code: the XFOR version is 4 times faster.

Code	Orig. time (gcc)	XFOR time (gcc)	Speed -up (gcc)	Orig. time (icc)	XFOR time (icc)	Speed -up (icc)
Red-Black	3.19	1.99	**1.60**	3.95	2.00	**1.98**
jacobi-2d	0.96	0.72	**1.33**	0.96	0.74	**1.30**
jacobi-1d	0.65	0.41	**1.59**	0.64	0.41	**1.56**
fdtd-2d	0.48	0.36	**1.33**	0.38	0.29	**1.31**
fdtd-apml	0.86	0.45	**1.91**	0.72	0.51	**1.41**
gauss-filter	0.40	0.43	**0.93**	0.67	0.55	**1.22**
seidel	4.49	2.57	**1.75**	5.06	2.67	**1.90**
2mm	2.33	1.86	**1.25**	0.67	0.67	**1.00**
3mm	3.39	2.45	**1.38**	0.97	2.03	**0.48**
correlation	0.57	0.10	**5.70**	0.56	0.09	**6.22**
covariance	0.56	0.12	**4.67**	0.57	0.13	**4.38**
mvt	0.34	0.17	**2.00**	0.18	0.13	**1.38**
gemver	0.47	0.36	**1.31**	0.27	0.27	**1.00**
syr2k	2.53	2.01	**1.26**	1.51	1.11	**1.36**

Table 6.1 – *Sequential-Vectorized Code Measurements.*

6.4.3 OpenMP Parallel Code Measurements

OpenMP parallelization has been turned on in GCC using option `-fopenmp`, and in ICC using option `-openmp`. Automatic vectorization is enabled to take advantage of both parallelizations when possible. According to the dependences, the outermost possible XFOR-loops and original for-loops have been parallelized. Codes have been run using 6 parallel threads mapped on the 6 cores of the Xeon X5650 processor. Speed-ups in Table 6.2 are given by comparison between the parallel original and the parallel XFOR code versions. As shown, XFOR codes can be 6 times faster than the original one (case of *correlation* code compiled with ICC).

Code	Orig. time (gcc)	XFOR time (gcc)	Speed -up (gcc)	Orig. time (icc)	XFOR time (icc)	Speed -up (icc)
Red-Black	1.60	1.52	**1.05**	1.43	1.40	**1.02**
jacobi-2d	0.61	0.42	**1.45**	0.58	0.42	**1.38**
jacobi-1d	0.48	0.39	**1.23**	0.41	0.36	**1.14**
fdtd-2d	0.32	0.22	**1.45**	0.32	0.22	**1.45**
fdtd-apml	0.15	0.08	**1.88**	0.15	0.09	**1.67**
gauss-filter	4.14	0.18	**23**	0.31	0.19	**1.63**
2mm	0.86	0.34	**2.53**	0.13	0.20	**0.65**
3mm	1.31	0.35	**3.74**	1.39	0.35	**3.97**
correlation	0.18	0.04	**4.50**	0.18	0.03	**6.00**
covariance	0.17	0.04	**4.25**	0.18	0.04	**4.50**
mvt	0.12	0.12	**1.00**	0.12	0.12	**1.00**
gemver	0.20	0.17	**1.18**	0.18	0.14	**1.29**
syr2k	0.50	0.30	**1.67**	0.40	0.20	**2.00**

Table 6.2 – *OpenMP Parallel Code Measurements.*

6.5 Conclusion

XFOR lets users express complex loop fusion while saving them the burden of writing prologues and epilogues loops, complex bounds, etc. When handling several iteration domains scanned by a sequence of loop nests exhibiting some data reuse, these nests may be carefully fused to be scheduled more efficiently. This is achieved by overlapping accurately their respective iteration domains through shifting (offset) and unrolling (grain). The offset and grain values must yield data reuse distances among the statements which promote simultaneously short data reuse distances and vectorization, while paying attention to data dependencies.

In the next chapter, we introduce our software tool "XFOR-WIZARD" which is an assistant that guides the programmer in implementing the previously described strategies. XFOR-WIZARD generates automatically from any for loop nest a perfectly nested XFOR loops. Then, the programmer can set explicitly the reuse distances between the statements. At each step, the programmer may refer to XFOR-WIZARD to check the legality of the applied schedule.

7 The XFOR Programming Environment XFOR-WIZARD

7.1 Introduction

XFOR-WIZARD is a software environment that guides the programmer in rewriting a reference program using the XFOR construct. It generates from any for-loop nests, perfectly nested XFOR loops. Then, the wizard lets the control for to the user to apply polyhedral transformations on the generated loops. At each step, the programmer may refer to XFOR-WIZARD to check the legality of the implemented transformation.

In this chapter, we start by explaining the need that drives us to develop this wizard; we first present the very first version of our schedule validator XFORSCAN in section 7.2. Then in section 7.3, we present the current version of XFOR-WIZARD. We describe the different offered functionalities together with a presentation of the developed visual interface that eases significantly the use of the software. Finally, in section 7.4, we give a detailed description on how the wizard generates the XFOR code and how dependence verification has been implemented.

7.2 XFORSCAN

When rewriting a program using the XFOR construct, one needs to ensure that the XFOR-program is semantically equivalent to the initial program, *i.e.*, that it does not violate any data dependence.

Our first idea was to write an XFOR SCheduling ANalyser (XFORSCAN) which is, basically, a dependence analyzer that checks if the XFOR-program respects the dependencies expressed in the reference program. XFORSCAN was expected to take as input two files; the original program and the XFOR-program[1]. Then, each loop nest has to be described using the OpenScop structure

[1]The respective for-loop and XFOR loop nests must be placed between `#pragma scop` end `#pragma endscop`.

[7]. To do this, we extended Clan [42], which is a tool for automatically extracting a polyhedral representation from input for-loop nests, in order to parse XFOR-loops and generate the corresponding OpenScop[2]. Finally, we invoke the dependence analyzer Candl [14] to compare the two OpenScop inputs and see if they are equivalent (see Figure 7.1).

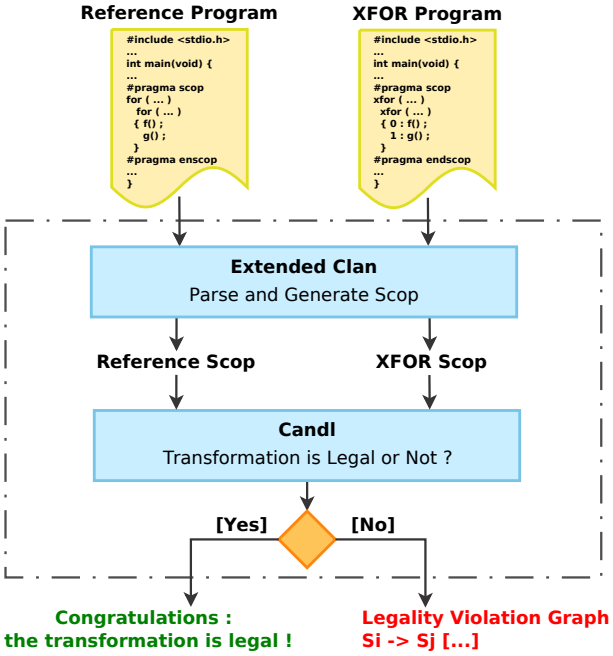


Figure 7.1 – *XFORSCAN Architecture.*

However, this first implementation of XFORSCAN did not work as expected. The information regarding the legality of the transformation was random. In fact, in order to be able to compare two OpenScop files, Candl has some restrictions:

1. The two OpenScop files must have the same number of statements.
2. The statements must appear in the same order in each OpenScop file.
3. The structure of relations `Domain`, `Scattering` and `Acces Array` must be similar. This means:
   - They must have the same number of parameters.
   - The parameters must appear in the same order in all relations in both OpenScop files.
   - Array identifiers must be the same (appear in the same order and have the same references in both OpenScop files).
4. The structure of relations `Domain` and `Acces Array` must be identical (*i.e.* same contents).

[2]We cannot use IBB for generating the OpenScop file corresponding to XFOR loops, because IBB does not analyze the memory accesses of the loop body statements.

We have no control on the manner in which the XFOR code was written. For example, the user may change the order of the instructions in the XFOR code, or change the order of appearance of the parameters which will affect their order in the OpenScop encoding. Hence we decided to adapt XFORSCAN in the following way. XFOR-WIZARD, it is an assistant that guides the programmer in the process of re-writing a for-loop based program using XFOR-loops. The XFOR loops generation is performed by XFOR-WIZARD. Thus the wizard guarantees that both generated OpenScop files, from the reference program and from the XFOR program, fulfill the expectations of Candl. In this context, Candl provides a valid answer regarding the correctness of the XFOR program.

7.3 XFOR-WIZARD

7.3.1 License

XFOR-WIZARD is released under the GNU General Public License, version 2 (GPL v2) [67].

7.3.2 Using the XFOR-WIZARD Software

XFOR-WIZARD, on its current release, offers many functionalities in addition to its main role as XFOR loop generator and dependencies checker. The latter routines are described in details in section 7.4. Here, we enumerate and describe the different uses of the wizard software. The XFOR-WIZARD is composed of two main interfaces; the file editor interface and the XFOR editor interface.

7.3.2.1 File Editor Interface

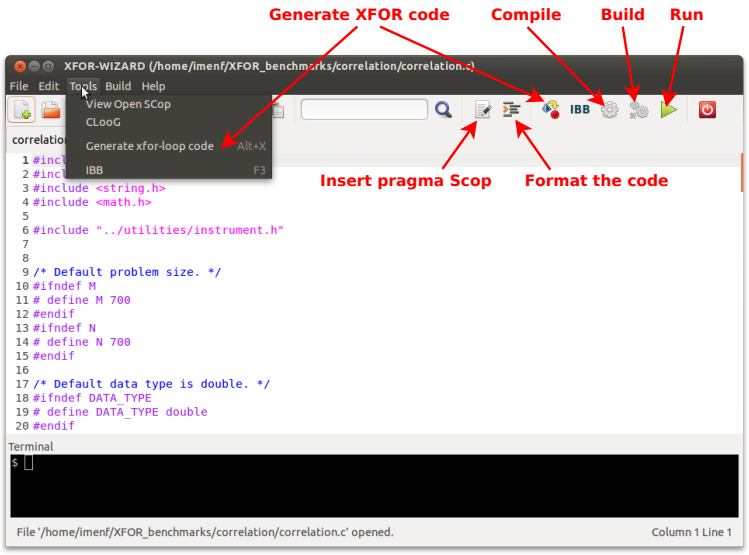


Figure 7.2 – *XFOR-WIZARD: File Editor.*

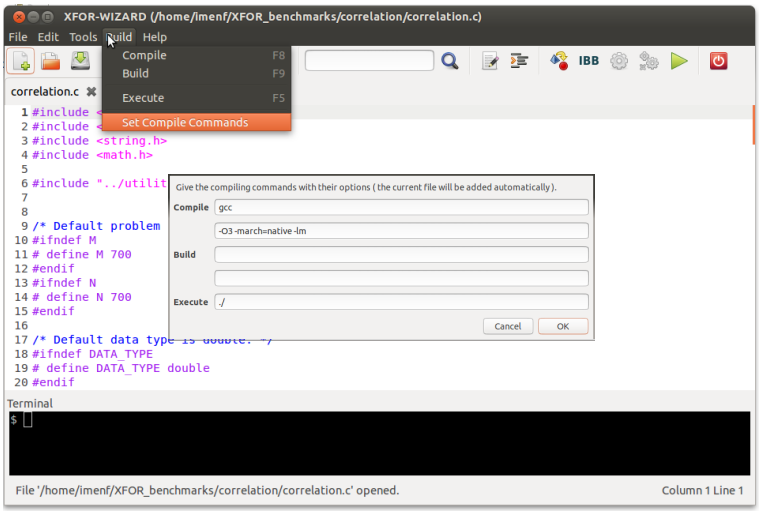


Figure 7.3 – *XFOR-WIZARD: Setting Compile Commands.*

This interface is displayed in Figures 7.2 and 7.3. Basically, it is a file editor permitting code visualization and modification. In addition to that, several operations are possible:

1. Displaying, in a new tab, the OpenScop description of a code (to understand the domains and the schedules).
2. Calling CLooG on an OpenScop file and displaying the generated code in a new tab.
3. Calling IBB on a XFOR code file and displaying the generated code in a new tab.
4. Setting compile commands, to compile a code and execute it, in order to evaluate performance and compare the XFOR-loop program to the referential program (see Figure 7.3).

The user may also exploit the terminal in order to execute some commands.

7.3.2.2 XFOR Editor Interface

This interface is devoted to apply transformations on the automatically generated XFOR-loop nest (see Figure 7.4) and the reference loop nests also. XFOR-WIZARD assists and helps the user in applying transformations on his code and allows him to verify at each step the legality of the modifications. If the wizard is unable to verify the correctness of the transformation, it although gives the opportunity to the user to define the modified nest as a reference. This may be useful for advanced programmers that have some knowledge about loop transformations and dependencies, since they may apply some incorrect transformations in order to reach at the end a semantically correct XFOR-loop.

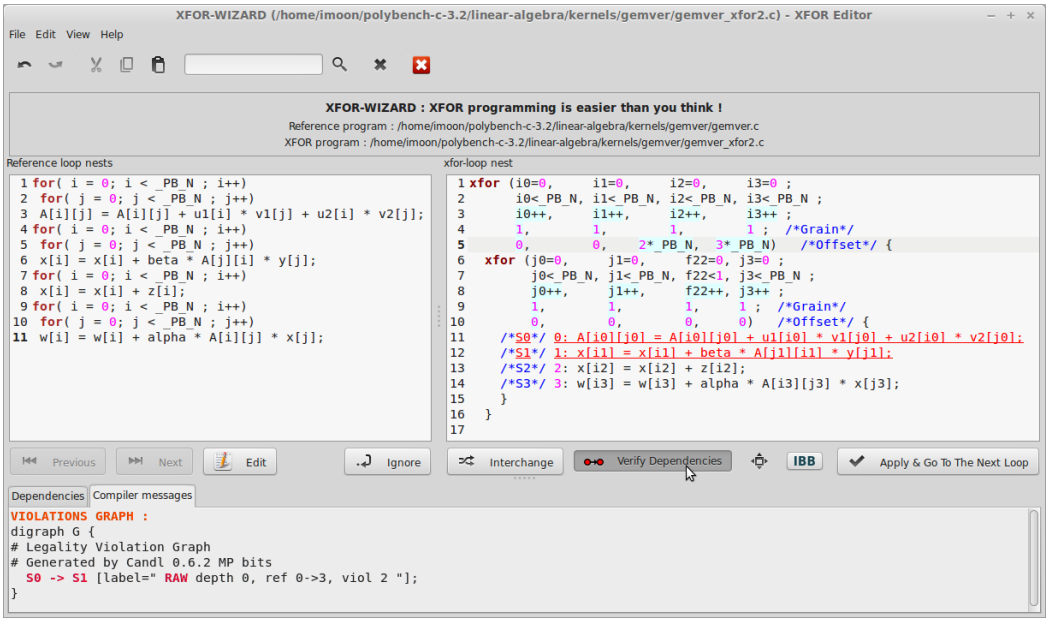


Figure 7.4 – *XFOR-WIZARD: XFOR Editor.*

The XFOR editor is composed of three parts; the first part deals with the original for-loops, the second one is devoted to the XFOR loop nest, and the third part is dedicated for displaying compiler messages and information about dependencies.

Regarding the reference loop part, four buttons are available:

- **Previous/Next**: Those buttons allow the user to go to the previous or the next for-loop nests.

- **Edit**: In some cases, the user may need to modify the reference loop nest in order to apply some transformations such as loop unrolling or dividing an instruction into several statements in order to minimize the intra-statement data-reuse distance (See section 6.2.2 in chapter 6 ([*XFOR Programming Strategies*], page 87)). Once the edited for-loop has been saved, a new XFOR loop is generated.

- **Ignore**: This button is useful when the user changes his mind and decides not to substitute a given for-loop nest with an XFOR nest.

Concerning the XFOR loop part, the programmer is allowed to modify the offsets and the grains of the XFOR loop nest. He may also apply a loop interchange. This can be directly afforded by the offset as shown in subsection 9.2.8 in chapter 9 ([*XFOR Polyhedral Intermediate Representation and Manual-Automatic Collaborative Approach for Loop Optimization*], page 145) or, for beginners, the **interchange** button may be helpful. The programmer can also invoke IBB for the current XFOR-loop nest only. And from this interface, it is still possible also to display the OpenScop description of the reference loops or the XFOR nest.

7.3.3 Polyhedral Toolkit

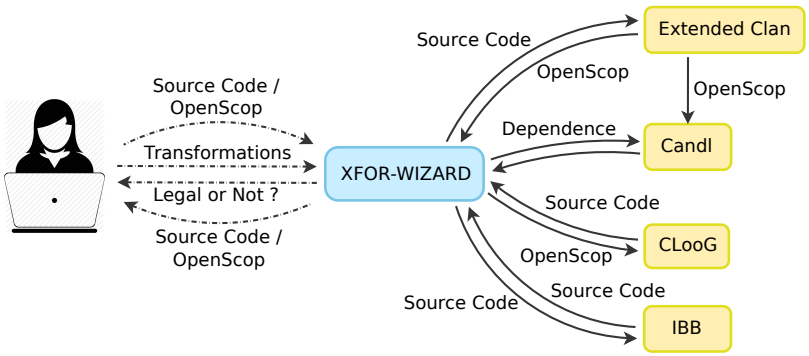


Figure 7.5 – *XFOR-WIZARD Architecture and Interactions.*

As Figure 7.5 shows, the user interacts only with the wizard. He/She performs loop manipulations and transformations. Then, XFOR-WIZARD notifies him about the transformation's legality and correctness. During this process, XFOR-WIZARD invokes the two polyhedral tools Clan (the XFOR extended version) and Candl.

7.4 XFOR Code Generation and Loop Transformation Verification

Figure 7.6 illustrates the functioning and the general architecture of XFOR-WIZARD. Essentially, the wizard takes as input a for-loop based program, but it is also able to analyze XFOR loops, this is particularly useful when the programmer wants to resume a project that he/she started before. Thus, one can continue transforming the XFOR loops and verifying step-by-step the correctness of the transformations. In order to help the programmer to generate an equivalent but more efficient XFOR-loop program, XFOR-WIZARD starts by parsing the reference program and identifies parts of code placed between `#pragma scop` and `#pragma endscop`. Then, for each identified part, it generates automatically a perfectly nested XFOR-loop which is semantically equivalent to the initial set of loop nests, where all the statements are identically scheduled. This is achieved by fusing all loop nests into one unique XFOR-loop nest, where the XFOR-loop has as many indices as the number of statements in the original loop nests, and the offsets of the outermost XFOR-loops are set to the maximum values (typically the original loop number of iterations) that guarantee a sequential execution similar to the sequence of the original nests. Once the XFOR-loop nest has been generated, it is considered as the reference loop. XFOR-WIZARD provides to the user a copy of the reference XFOR on which he may apply transformations and verify, step-by-step, if they are legal regarding data dependencies.

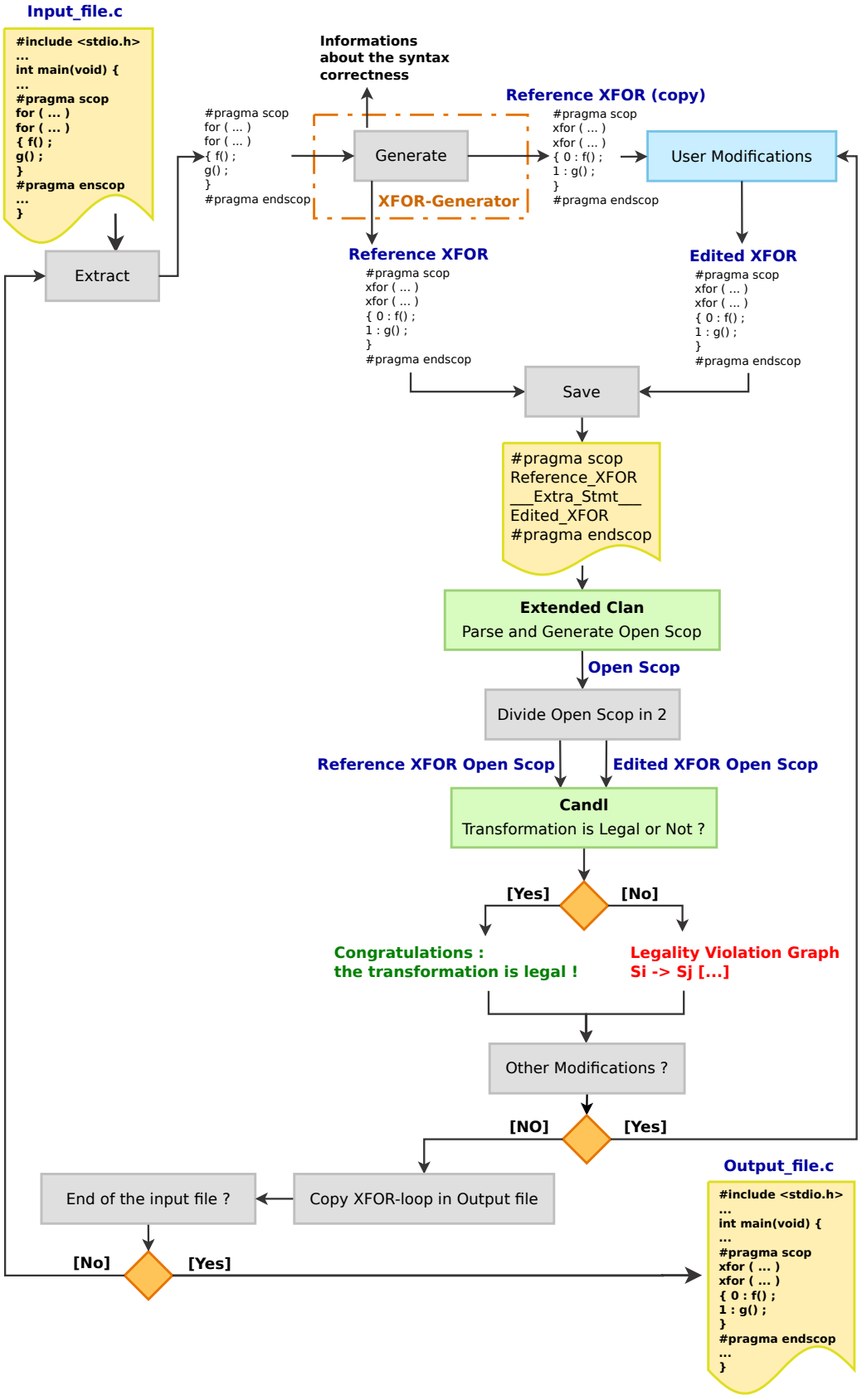


Figure 7.6 – *XFOR Code Generation and Loop Transformation Verification.*

In order to be able to use the dependency analyzer Candl [14] for verifying the dependencies, we proceed as follows ; we save the reference loop and the edited one in the same file[3]. This guarantees that when calling the extended version of Clan, both OpenScop representations have the same order of parameters and the same identifiers for arrays. After that, we call Clan to generate the OpenScop encoding. Then, the result is divided in two parts, since each loop nest has his own OpenScop representation. Finally, Candl is invoked. The latter starts by computing the dependencies of the reference, then compares it to the set of dependencies of the transformed XFOR, and finally informs the user about the correctness of his modifications.

7.4.1 Example

The code in Listing 7.1 represents two matrix multiplications ($E := A \times B \times D$) extracted from the `2mm.c` benchmark of the Polyhedral Benchmark suite [27]. The first loop nest computes a matrix $C := A \times B$. And the second loop nest computes the matrix $E := C \times D$.

```
#pragma scop
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++) {
    C[i][j] = 0;
    for (k = 0; k < nk; k++)
      C[i][j] += A[i][k] * B[k][j];
  }
for (i = 0; i < ni; i++)
  for (j = 0; j < nl; j++) {
    E[i][j] = 0;
    for (k = 0; k < nj; k++)
      E[i][j] += C[i][k] * D[k][j];
  }
#pragma endscop
```

Listing 7.1 – *2mm Kernel.*

The equivalent XFOR-loop which is automatically generated by XFOR-WIZARD is displayed in Listing 7.2. Note that this XFOR loop is perfectly nested thanks to the additional indices ($f21$ and $f23$) inserted by the wizard. Note also that the offsets of the outermost indices corresponding to the second for-loop nest are equal to the number of iterations of the first loop nest, which guarantees exactly the same scheduling as the original program.

[3]We insert an extra statement between the two loops to separate the corresponding statements easily after the OpenScop generation.

```
#pragma scop
xfor (i1=0,i2=0,i3=0,i4=0 ; i1<ni,i2<ni,i3<ni,i4<ni ;
      i1++,i2++,i3++,i4++ ; 1,1,1,1 ; 0,0,ni,ni) {
  xfor (j1=0,j2=0,j3=0,j4=0 ; j1<nj,j2<nj,j3<nl,j4<nl ;
        j1++,j2++,j3++,j4++ ; 1,1,1,1 ; 0,0,0,0) {
    xfor (f21=0,k2=0,f23=0,k4=0 ; f21<1,k2<nk,f23<1,k4<nj ;
          f21++,k2++,f23++,k4++ ; 1,1,1,1 ; 0,0,0,1) {
      /*S0*/ 0: C[i1][j1] = 0;
      /*S1*/ 1: C[i2][j2] += A[i2][k2] * B[k2][j2];
      /*S2*/ 2: E[i3][j3] = 0;
      /*S3*/ 3: E[i4][j4] += C[i4][k4] * D[k4][j4];
  } } }
#pragma endscop
```

Listing 7.2 – *2mm XFOR Code Automatically Generated by XFOR-WIZARD.*

7.4.2 XFOR Code Generation

The goal is to generate a perfectly nested XFOR loop from any set of for-loop nests, by associating to each statement of the for-loop bodies an index in the XFOR loop. Thus, the final XFOR loop has as many indices as the total number of instructions in the initial code. To ensure this, each time a new index (*i.e.* instruction) is added, we must have the same number of indices in all XFOR headers, and all for-loop nests composing the XFOR nest must have the same depth. In the following, we describe in details the algorithm of XFOR generation together with the used data structures.

7.4.2.1 Data Structures

During the process of XFOR code generation, the parameters of an XFOR are saved in an intermediate structure. An XFOR-loop nest is represented by a single-linked list. One node of the list has the following structure:

```
typedef struct xfor_loop * xfor_loop_p;
struct xfor_loop
{   index_p     indices;
    xfor_loop_p inner_loop;
};
```

Where **Indices** is a linked list that involves the indices that belong to the same XFOR (*i.e.* appear in the same XFOR header). One node of this list is called `index`. It is defined below:

```
typedef struct index * index_p;
struct index
{   char*   name;
    char*   lower_b;
    char*   relop;
    char*   cond;
    char*   stride;
    int     grain;
    char*   offset;
    index_p next;
};
```

The instructions are saved in a single-linked list in the same order in which they appear in the original for-loop nests (which is the same order of the corresponding indices). Each element of the linked list points to the following structure:

```
typedef struct instruction * instruction_p;
struct instruction
{   char*         stmt;
    instruction_p next;
};
```

7.4.2.2 XFOR Generation

The XFOR generation process is described in Figure 7.6. Here, the input is a code composed of a set of for-loop nests and the output is an `xfor_loop_p` that represents the equivalent perfectly nested XFOR. In order generate at each step a *sane* XFOR nest, we use the following variables:

- **xfor_nest**: of type `xfor_loop_p`. It saves the output XFOR nest.
- **all_stmt**: of type `instruction_p`. It is used to store the list of statements of the final XFOR.
- **for_loop**: of type `index_p`. It includes the information about the current read loop of the input code.
- **xfor_loop_depth**: an integer that indicates the depth of the current instruction.
- **xfor_loop_depth_max**: an integer that represents the maximum depth of the XFOR loops.
- **xfor_nb_indices**: an integer, it gives the number of indices in the XFOR.

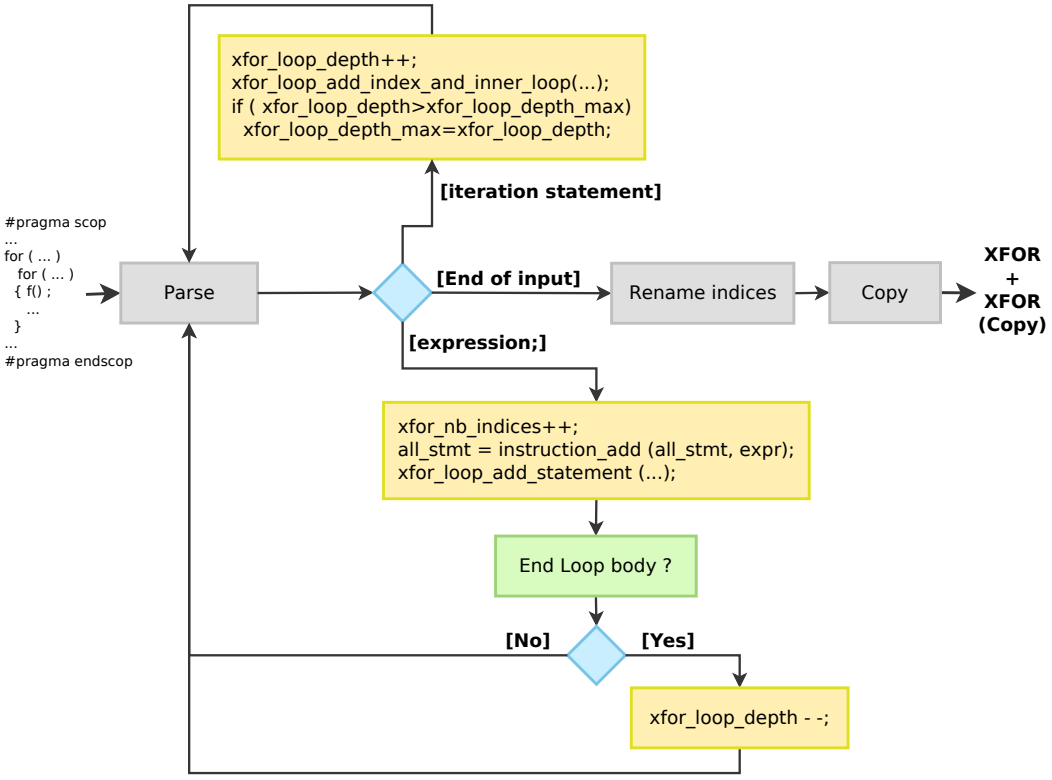


Figure 7.7 – *XFOR Loop Generation.*

As shown in Figure 7.7, the parser analyzes the input code. Once an `iteration statement` (*i.e.* a for loop) is recognized, the `xfor_loop_depth` is incremented, and an inner loop is added to the `xfor_nest` structure. This is done thanks to function `xfor_loop_add_index_and_inner_loop` which is described in details below. Then, if the current depth is greater than the maximum depth of the XFOR loops, the `xfor_loop_depth_max` variable is updated. This procedure is repeated for every `iteration statement`.

When the parser identifies an `expression;` (*i.e.* a simple instruction, a computation for example), a new index is inserted, and variable `xfor_nb_indices` is incremented. Then, the new instruction is added at the end of the instructions list `all_stmt`. After that, a new index is added to all the levels of the `xfor_nest` structure. This is done thanks to function `xfor_loop_add_statement` which is described in details below. Finally, if the end of the loop body has been reached, the variable `xfor_loop_depth` is decremented. These operations are recured for each encountered statement.

When the end of the input has been reached, the XFOR indices are renamed as follows; for every index, the new name is built from the concatenation of the old name + the index depth + the position of the index in the XFOR header. In this way, we guarantee that each index in the XFOR has a different name. Finally, the obtained XFOR is copied in order to be used by the programmer for applying transformations.

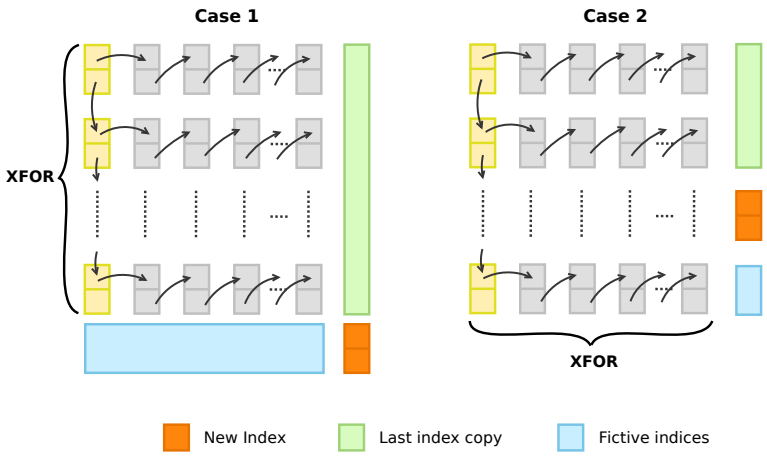


Figure 7.8 – *XFOR-WIZARD: New Loop Insertion.*

xfor_loop_add_index_and_inner_loop

Essentially, this routine adds an inner loop to the XFOR Structure (`xfor_nest`). Two cases are possibles:

1. The current depth of the loop is higher than the maximum depth (*i.e.* `xfor_loop_depth` > `xfor_loop_depth_max`). As illustrated in Figure 7.8, Case 1, the new loop insertion is achieved in four steps:
   (a) Copy the last index of all the XFOR loops where depth <= `xfor_loop_depth_max`.
   (b) Create a new XFOR loop with `xfor_nb_indices - 1` fictive indices doing just one iteration.
   (c) Insert the new index at the end of the newly created XFOR loop. Here the offset of the new index is equal to 0.
   (d) Add the newly created XFOR loop at depth `xfor_loop_depth` (*i.e.* `xfor_loop_depth_max+1`).
2. Else (*i.e.* `xfor_loop_depth` <= `xfor_loop_depth_max`). This case is illustrated in Figure 7.8, Case 2. Here, the new loop insertion is done in four steps:
   (a) Copy the last index of the XFOR loops of depth < `xfor_loop_depth`.
   (b) Insert the new index at the XFOR of depth `xfor_loop_depth`.
   (c) Compute the offset of the new index. Its value is equal to the offset of the previous index + the number of iterations of that index.
   (d) Insert a fictive index doing just one iteration for all XFOR loops from depth `xfor_loop_depth+1` to `xfor_loop_depth_max`.

xfor_loop_add_statement
Basically, this routine adds an index at the XFOR of depth `xfor_loop_depth`. This is achieved in three steps (see Figure 7.8, Case 2):

1. Copy the last index of the XFOR loops of depth $<$ `xfor_loop_depth`.
2. Insert the new index at the XFOR of depth `xfor_loop_depth`.
3. Insert a fictive index doing just one iteration for all XFOR loops from depth `xfor_loop_depth+1` to `xfor_loop_depth_max`.

7.4.3 Clan Extension

Clan translates parts of the program that are placed between `#pragma scop` and `#pragma endscop` into a specific polyhedral, matrix-based, representation of XFOR-loop nests called OpenScop [7]. We have extended the Clan grammar in order to be able to parse XFOR-loops and to generate the corresponding OpenScop description. Hence, the XFOR construct may be manipulated by any polyhedral compilation framework easily. In addition to that, we added the "`-normalize`" option to Clan, which permits to generated domains shifted to zero. This option is mandatory when analyzing XFOR-loops. Extended Clan allows a more general syntax. In addition to what was described in chapter 4 ([*XFOR Syntax and Semantics*], page 59), it permits the use of some non labeled instructions ; (*i.e*) `for` and `if` statements. The non labeled statements are factorized, this means that they are executed for all XFOR indices.

Example. Let us consider this XFOR-loop nest:

```
xfor (i=0,j=0 ; i<=n,j<=m ; i++,j++ ; 1,1 ; a,b) {
  for (q=0 ; q<Q ; q++)
    xfor (k=100,l=100 ; k<=n,l<=m ; k++,j++ ; 1,1 ; e,f) {
      1: c[i][l] = 0 ;
      0: c[i][k] = 0 ;
    }
}
```

Here, the q-loop is applied to both indices of the XFOR loop. Thus, this nest is equivalent to the following XFOR nest:

```
xfor (i=0,j=0 ; i<=n,j<=m ; i++,j++ ; 1,1 ; a,b) {
  xfor (q1=0,q2=0 ; q1<Q,q2<Q ; q1++,q2++ ; 1,1 ; 0,0)
    xfor (k=100,l=100 ; k<=n,l<=m ; k++,j++ ; 1,1 ; e,f) {
      1: c[i][l] = 0 ;
      0: c[i][k] = 0 ;
    }
}
```

Extended Clan transforms the index domains into polytopes over a common referential domain. Each iteration domain of each for-loop nest composing an

XFOR nest defines a Z-polytope, *i.e.*, a lattice of integer points delimited by a finite polyhedron. Respective grains and offsets must also be considered in order to define the union of Z-polytopes corresponding to the overlapping of the domains that are defined by the XFOR structure. The referential domain is implicitly created by expressing the original indices of the XFOR structure into a common basis of referential indices, *ref_index*, according to the relation:

$$incr_i \times ref\_index = grain_i \times (index_i - lower\_bound_i) + offset_i \times incr_i \tag{7.1}$$

If the stride is a multiple of the grain (or the grain is a multiple of the stride), an additional constraint is added in order to prevent CLooG from simplifying the previous equation:

$$index_i - lower\_bound_i = incr_i \times k, \text{ for every integer } k.$$

7.4.3.1 Illustrative Example

```
#pragma scop
xfor (i0=5,i1=10 ; i0<N,i1<N ; i0++,i1+=2 ; 1,4 ; a,b) {
  0: T[i0] = Y[i0+1]+Y[i0-1];
  1: Y[i1] += T[i1]; }
#pragma endscop
```

Listing 7.3 – *XFOR Code Example.*

In this subsection, we give more details about the OpenScop generation performed by the XFOR-extended version of Clan. For instance, consider the XFOR-loop nest code in Listing 7.3. Each iteration domain of each for-loop nest composing an XFOR nest defines a lattice of integer points delimited by a finite polyhedron:

1. $\begin{cases} 5 \leq i0 \\ i0 \leq N-1 \\ 0 \leq N-6 \end{cases}$

2. $\begin{cases} 10 \leq i1 \\ i1 \leq N-1 \\ 0 \leq N-11 \end{cases}$

The referential domain is implicitly created by expressing the original indices of the XFOR structure into a common basis of referential indices $c1$, $c2$ and $c3$, according to the relation:

1. $\begin{cases} c1 = 0 \\ c2 = i0 + a - 5 \\ c3 = 0 \end{cases}$

2. $$\begin{cases} c1 = 0 \\ 2 \times c2 = 4 \times i1 + 2 \times b - 40 \\ i1 - 10 = 2 \times l1 \\ c3 = 1 \end{cases}$$

These relations define the scheduling of the statements. They are defined in the OpenScop encoding within the SCATTERING relations. In order to have a complete scheduling, the order of the statements in the XFOR body must be taken into account. This is done by using an additional scattering dimension. In the case of our example, we add the scattering dimensions $c1$ and $c3$. We show below the OpenScop representation corresponding to the previous example, automatically generated by the XFOR-extended Clan version.

```
<OpenScop>
# ============================================== Global
# Language
 C
# Context
 CONTEXT
0 5 0 0 0 3
# Parameters are provided
1
<strings>
 N a b
</strings>
# Number of statements
2
# ============================================== Statement 1
# Number of relations describing the statement:
5
# ----------------------------------------------  1.1 Domain
 DOMAIN
3 6 1 0 0 3
# e/i| i0 |  N    a    b |  1
   1    1    0    0    0   -5  # i0-5 >= 0
   1   -1    1    0    0   -1  # -i0+N-1 >= 0
   1    0    1    0    0   -6  # N-6 >= 0
# ----------------------------------------------  1.2 Scattering
 SCATTERING
3 10 3 1 1 3
# e/i| c1   c2   c3 | i0 | l1 |  N    a    b |  1
   0   -1    0    0    0    0    0    0    0    0  # c1 == 0
   0    0   -1    0    1    0    0    1    0   -5  # c2 == i0+a-5
   0    0    0   -1    0    0    0    0    0    0  # c3 == 0
# ----------------------------------------------  1.3 Access
WRITE
2 8 2 1 0 3
# e/i| Arr  [1]| i0 |  N    a    b |  1
   0   -1    0    0    0    0    0    6  # Arr == T
   0    0   -1    1    0    0    0    0  # [1] == i0
```

```
READ
2 8 2 1 0 3
# e/i| Arr  [1]| i0 |  N    a    b |  1
   0   -1    0    0    0    0    0    7  # Arr == Y
   0    0   -1    1    0    0    0    1  # [1] == i0+1
READ
2 8 2 1 0 3
# e/i| Arr  [1]| i0 |  N    a    b |  1
   0   -1    0    0    0    0    0    7  # Arr == Y
   0    0   -1    1    0    0    0   -1  # [1] == i0-1
# ------------------------------------  1.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
1
# List of original iterators
 i0
# Statement body expression
 T[i0] = Y[i0+1]+Y[i0-1];
</body>
# ================================================ Statement 2
# Number of relations describing the statement:
5
# ----------------------------------------------  2.1 Domain
 DOMAIN
3 6 1 0 0 3
# e/i| i1 |  N    a    b |  1
   1    1    0    0    0  -10  # i1-10 >= 0
   1   -1    1    0    0   -1  # -i1+N-1 >= 0
   1    0    1    0    0  -11  # N-11 >= 0
# ----------------------------------------------  2.2 Scattering
 SCATTERING
4 10 3 1 1 3
# e/i| c1   c2   c3 | i1 | l1 |  N    a    b |  1
   0   -1    0    0    0    0    0    0    0    0  # c1 == 0
   0    0   -2    0    4    0    0    0    2  -40  # -2*c2+4*i1+2*b-40 == 0
   0    0    0    0    1   -2    0    0    0  -10  # i1-2*l1-10 == 0
   0    0    0   -1    0    0    0    0    0    1  # c3 == 1
# ----------------------------------------------  2.3 Access
READ
2 8 2 1 0 3
# e/i| Arr  [1]| i1 |  N    a    b |  1
   0   -1    0    0    0    0    0    7  # Arr == Y
   0    0   -1    1    0    0    0    0  # [1] == i1
WRITE
2 8 2 1 0 3
# e/i| Arr  [1]| i1 |  N    a    b |  1
   0   -1    0    0    0    0    0    7  # Arr == Y
   0    0   -1    1    0    0    0    0  # [1] == i1
READ
```

```
2 8 2 1 0 3
# e/i| Arr  [1]| i1 |  N    a    b |  1
   0   -1    0    0    0    0    0    6  # Arr == T
   0    0   -1    1    0    0    0    0  # [1] == i1
# ------------------------------------  2.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
1
# List of original iterators
 i1
# Statement body expression
 Y[i1] += T[i1];
</body>
# ================================================ Extensions
<scatnames>
 b0 _mfr_ref0 b1
</scatnames>
<arrays>
# Number of arrays
7
# Mapping array-identifiers/array-names
1 i0
2 i1
3 N
4 a
5 b
6 T
7 Y
</arrays>
<coordinates>
# File name
 zzz.c
# Starting line and column
2 0
# Ending line and column
5 0
# Indentation
0
</coordinates>
</OpenScop>
```

7.5 Conclusion

XFOR-WIZARD is a software tool that helps the programmer in rewriting programs using the XFOR looping construct. This tool generates automatically from a reference for-loop nest a semantically equivalent XFOR nest, and then offers to the user the opportunity of modifying the scheduling of the XFOR

loops and verifying the legality of the applied transformations. XFOR-WIZARD is helpful in program optimization as it allows to write efficient codes with a low engineering cost. It helps also to avoid syntax errors while writing XFOR codes.

The next chapter is dedicated to the comparison between the performance of XFOR codes (semi-automatic optimization) and the performance of codes that were generated by fully automatic optimizers. In addition to that, we show how XFOR allows to bridge the gap between both code optimization strategies.

8 XFOR and Automatic Optimizers

8.1 Introduction

In this chapter, we show that the XFOR programming structure allows to fill important performance gaps remaining with automatic loop optimizers. We consider the well-known polyhedral optimizer *Pluto* [6, 16] which implements some of the most advanced loop optimizing strategies and transformations. We demonstrate that the XFOR construct helps in highlighting important performance issues, that could not be clearly identified without it. This is done by comparing XFOR-generated codes to Pluto-generated ones, and to other XFOR-codes. We highlight five important gaps in the currently adopted and well-established code optimization strategies: insufficient data locality optimization, excess of conditional branches in the generated code, verbose code with too many machine instructions, data locality optimization resulting in processor stalls, and finally missed vectorization opportunities. We illustrate the importance of these issues in program optimization using eleven representative codes selected from the Polybench benchmark suite [27]. In Section 8.2, we focus on these five important performance issues by relating them as being jointly the cause of wasted processor cycles. Then, each highlighted issue is addressed in a dedicated subsection using illustrative benchmark codes. Section 8.3 is dedicated to experiments where Pluto-optimized and XFOR versions are compared regarding sequential/vectorized, and loop-parallelized executions.

8.2 Wasted Processor Cycles

The execution time of a program is obviously directly related to the total number of cycles spent by the CPU for running all its instructions. Among these consumed cycles, some of them may be stalled, and some others may be spent uselessly in executing instructions performing computations that could either have been achieved using a significantly smaller number of

instructions, or by taking advantage of some accelerator processor units using dedicated instructions. The latter issue is detailed in subsection 8.2.5 regarding vectorization, while the previous one is addressed in subsection 8.2.3. It is shown that codes exhibiting a good data locality may be even slower than codes with weaker locality, just because of one of these issues.

Stalled processor cycles are cycles spent by the processor in waiting for the completion of some event on which the continuation of the current instruction sequence depends. Thus these cycles are wasted since they uselessly consume time and energy. Although such processor stalls can never be completely avoided, or may be partially hidden by simultaneous instruction executions, their amount should be minimized. For this purpose, their cause have to be handled specifically when optimizing programs. They can be classified into four main categories:

1. *Stalls due to insufficient computing resources:* for example, the processor core does not have enough floating-point units to perform all floating point operations that are ready to execute.

2. *Stalls due to memory latency:* this issue is one of the most frequently handled issues in program optimization techniques, with goals like data locality improvement and minimization of cache misses.

3. *Stalls due to dependencies between instructions:* this occurs when the executed code contains many sequences of dependent instructions, *i.e.*, instructions for which at least one operand is reused in some closely following instructions in a Read-After-Write fashion. Such a situation prevents superscalar microprocessors to launch simultaneously several instructions due to the unavailability of operands. This may potentially occur with codes resulting from aggressive data locality optimization, since data reuse distances are traditionally minimized by bringing as close as possible instructions referencing common data which may be dependent.

4. *Stalls due to branch mispredictions:* When a branch prediction made by the CPU is incorrect, all the speculatively executed instructions are discarded as soon as the correct branch is determined, and the processor execution pipeline restarts with instructions from the correct branch destination. This halt while the new instructions work their way down the execution pipeline causes a processor stall. It is a major drain on performance.

While point 1 can be solved using more hardware, point 2 is handled by most compilers which implement data locality optimization techniques that are more or less efficient. Regarding linear loop nests, Pluto, the source-to-source compiler [6, 16] implements some of the most advanced data locality optimization strategies based on the polyhedral model, *e.g.* some advanced tiling

techniques, loop interchange, skewing, etc. However, the heuristics that are used necessarily miss some optimization opportunities that may be handled by an expert programmer, particularly when using the XFOR structure. Altogether, the strategies used are not conscious of the other performance issues described below, and may have such a negative impact that they annihilate the gain provided by data locality improvement, as it will be shown in the following subsections.

Regarding points 3, this issue is never addressed explicitly by automatic optimizers since data locality optimization is always considered as a final goal. However, we show in subsection 8.2.4 that code versions that are all exhibiting similar and "optimized" memory performance (and similar performance regarding all the other points) may show very different execution times because of this issue. Additionally, the minimization of data reuse distances among instructions may prevent vectorization of these instructions (subsection 8.2.5).

Regarding point 4, while branch predictors cannot be controlled by software, the potential risk with loop transformations regarding branch mispredictions is related to the kind of optimizing transformation that has been applied and the number of branches resulting from it in the executable code. The classic tiling transformation may present such a potential risk due to the complicated control it requires, particularly when it involves non-rectangular shapes. This point is addressed in subsection 8.2.2.

In the following subsections, we illustrate the importance of these issues in program optimization using eleven representative codes extracted from the Polybench benchmark suite [27]. Every code has been rewritten using the XFOR structure and also optimized by the most recent version of the source-to-source Pluto polyhedral compiler [6] with the combination of options generating the best performing code among `-tile` (with the default tile size of 32 in each tilable dimension), `-l2tile`, `-smartfuse`, `-maxfuse` and `-rar`. XFOR and Pluto versions are compared regarding several relevant processor performance counters whose values were collected using the `perf` linux tool and the `libpfm` library [68]. The collected CPU events are detailed in Table 8.1. Notice that the origins of stalls are generally difficult to classify using CPU events. Each performance counter related to stalled cycles monitors a particular hardware unit that may stall for many reasons, and several units may stall for a common reason. Thus the reported counters in the following subsections provide some hints about the origins of some stalls, but can never be exhaustive. Experiments have been conducted on an Intel Xeon X5650 6-core processor 2.67GHz (Westmere) running Linux 3.2.0.

Event	Definition
CPU cycles	Total number of CPU cycles, halted and unhalted.
L1 data loads	Total number of data references to the L1 cache.
Li misses	Total number of loads that miss the Li cache.
TLB misses	Total number of load misses in the TLB that cause a page walk.
Branches	Total number of retired branch instructions.
Branch misses	Total number of branch mispredictions.
Stalled cycles	Total number of cycles in which no micro-operations are executed on any port.
Resource related stalls	Total number of allocator resource related stalls.
Reservation Station stalls	Number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly instructions dependent upon instructions further down the pipeline that have yet to retire). Regarding program analysis, a high count of this event most probably exhibits the effect of long chains of dependencies between close instructions.
Re-Order Buffer stalls	Number of cycles when the number of instructions in the pipeline waiting for retirement reaches the limit. A high count for this event indicates that there are long latency operations in the pipe (possibly, load and store operations that miss the L2 cache, and other instructions that depend on these cannot execute until the former instructions complete execution). Regarding program analysis, a high count of this event most probably exhibits the effect of long latency memory operations and TLB or cache misses.
Instructions	Total number of retired instructions.

Table 8.1 – *CPU Events Collected Using* `libpfm`

Among the eleven benchmark codes, we identified the ones whose runtime behavior is more significantly impacted by one single performance issue among the five highlighted ones, even if in general, performance is a question of balance among the provided gains and overheads. Thus these eleven codes have been selected because they enable such discrimination for pedagogical purposes. Notice also that the highlighted issues are independent of the compiler. We have observed similar runtime behaviors with codes compiled with the Intel compiler ICC, excepting for automatic vectorization which is generally better handled by ICC.

8.2.1 Gap 1: Insufficient Data Locality Optimization

Tables 8.2, 8.3, 8.4 and 8.5 show four codes whose best Pluto-optimized versions are compared to better performing XFOR-optimized versions. By comparing their respective performance counters, one can observe that the number of stalled cycles and the number of TLB and data cache misses are showing important

differences, while the other values do not show such significant disparity. From more than 25% up to 99% more TLB misses, and more than 15% up to 98% more L3 misses, have been observed with Pluto codes, obviously yielding more stalled cycles associated to larger memory access latencies. The origin of this higher amount of stalls is specifically highlighted by the high count of re-order buffer stalls which are symptomatic of long latency memory operations.

mvt	**Pluto**	**XFOR**	**Ratios**
#CPU cycles	**3,824M**	**2,425M**	**-36.58%**
#L1 data loads	748M	451M	-39.71%
#L1 misses	45M	50M	+10.71%
#L2 misses	**29M**	**5.8M**	**-80.09%**
#L3 misses	**38M**	**14M**	**-63.77%**
#TLB misses	**3.8M**	**0.7M**	**-82.62%**
#Branches	224M	212M	-4.89%
#Branch misses	470K	439K	-6.58%
#Stalled Cycles	**2,742M**	**1,582M**	**-42.29%**
#Resource related stalls	**2,544M**	**1,347M**	**-47.05%**
#Reservation Station stalls	431M	447M	+3.63%
#Re-Order Buffer stalls	**2,008M**	**771M**	**-61.62%**
#Instructions	2,469M	2,010M	-18.58%

Table 8.2 – *XFOR Speedups and Decreased Stalls Attributable to Decreased TLB and Cache Misses Regarding mvt Code*

syr2k	**Pluto**	**XFOR**	**Ratios**
#CPU cycles	**7,005M**	**5,671M**	**-19.05%**
#L1 data loads	4,322M	2,158M	-50.06%
#L1 misses	**299M**	**137M**	**-54.18%**
#L2 misses	**8.4M**	**3.6M**	**-55.94%**
#L3 misses	**10M**	**5.1M**	**-48.57%**
#TLB misses	**4.3M**	**3.2M**	**-25.78%**
#Branches	1,072M	1,078M	+0.58%
#Branch misses	1,072K	1,084K	+1.03%
#Stalled Cycles	**1,570M**	**1,346M**	**-14.27%**
#Resource related stalls	**1,495M**	**1,332M**	**-10.91%**
#Reservation Station stalls	327M	1,199M	+266.50%
#Re-Order Buffer stalls	**1,182M**	**132M**	**-88.80%**
#Instructions	11,890M	13,946M	+17.29%

Table 8.3 – *XFOR Speedups and Decreased Stalls Attributable to Decreased TLB and Cache Misses Regarding syr2k Code*

3mm	Pluto	XFOR	Ratios
#CPU cycles	**17,557M**	**4,358M**	**-75.18%**
#L1 data loads	4,226M	2,440M	-24.36%
#L1 misses	**815M**	**206M**	**-74.67%**
#L2 misses	**554M**	**5.4M**	**-99.02%**
#L3 misses	**174M**	**3M**	**-98.25%**
#TLB misses	**541M**	**3.2M**	**-99.41%**
#Branches	1,625M	813M	-49.96%
#Branch misses	2,704K	1,630K	-39.73%
#Stalled Cycles	**12,695M**	**524M**	**-95.87%**
#Resource related stalls	**12,392M**	**387M**	**-96.87%**
#Reservation Station stalls	**10,667M**	**379M**	**-96.44%**
#Re-Order Buffer stalls	**2,606M**	**38M**	**-98.52%**
#Instructions	11,331M	8,941M	-21.09%

Table 8.4 – *XFOR Speedups and Decreased Stalls Attributable to Decreased TLB and Cache Misses Regarding 3mm Code*

gauss-filter	Pluto	XFOR	Ratios
#CPU cycles	**3,457M**	**2,963M**	**-14.28%**
#L1 data loads	873M	843M	-3.45%
#L1 misses	**75M**	**46M**	**-38.97%**
#L2 misses	**4.2M**	**2.4M**	**-42.33%**
#L3 misses	**29.5M**	**24.8M**	**-15.91%**
#TLB misses	**1.5M**	**0.7M**	**-49.78%**
#Branches	724M	572M	-20.92%
#Branch misses	622K	689K	+10.78%
#Stalled Cycles	**1,351M**	**1,196M**	**-11.45%**
#Resource related stalls	**924M**	**824M**	**-10.82%**
#Reservation Station stalls	174M	150M	-13.88%
#Re-Order Buffer stalls	**171M**	**134M**	**-21.25%**
#Instructions	5,026M	4,652M	-7.44%

Table 8.5 – *XFOR Speedups and Decreased Stalls Attributable to Decreased TLB and Cache Misses Regarding gauss-filter Code*

The main loop kernel of the original `mvt` code is shown in Listing 8.1. As one can see, it consists of two loop nests computing respectively vectors `x1` and `x2`. There is no data dependence between both computations. Therefore, both loop nests may be fused into one unique loop nest. However, both loop nests are reading array `A`, respectively in row-major order and in column-major order. Since column-major is exhibiting a very poor data locality, the second loop nest may take advantage of a loop interchange which is correct regarding data dependencies. Thus, the XFOR code shown in Listing 8.2 is built by interchanging loops of the second nest, and then fusing both loops into one unique loop nest. The resulting code exhibits a good data locality regarding accesses to array `A`: row-major order for both read accesses and very short reuse distance between both.

The Pluto compiler applies a different transformation. As shown in Listing 8.3, it fuses both loop nests, and then tiles both loops to improve data locality. However, the second access to array `A` is still in column-major order since no loop interchange has been applied. An explanation of Pluto's strategy is that Pluto always tries to exhibit a parallel loop, even when no automatic parallelization has been requested by the user. Indeed, the outermost loop of Pluto's code may be parallelized, while the XFOR code does not exhibit any parallel loop. But there is no way with Pluto to generate a better sequential code.

```
for (i=0 ; i<n ; i++)
  for (j=0 ; j<n ; j++)
    x1[i]=x1[i]+A[i][j]*y_1[j] ;
for (i=0 ; i<n ; i++)
  for (j=0 ; j<n ; j++)
    x2[i]=x2[i]+A[j][i]*y_2[j] ;
```

Listing 8.1 – `mvt`*: Original Code.*

```
xfor (io=0,j1=0 ; io<n,j1<n ; io++,j1++ ; 1,1 ; 0,0)
xfor (jo=0,i1=0 ; jo<n,i1<n ; jo++,i1++ ; 1,1 ; 0,0) {
  0: x1[io]=x1[io]+A[io][jo]*y_1[jo] ;
  1: x2[i1]=x2[i1]+A[j1][i1]*y_2[j1] ; }
```

Listing 8.2 – `mvt`*: XFOR Code: Interchange + Fusion.*

```
for (t1=0 ; t1<=floord(n-1,32) ; t1++) {
  for (t2=0 ; t2<=floord(n-1,32) ; t2++) {
    for (t3=32*t1 ; t3<=min(n-1,32*t1+31) ; t3++) {
      for (t4=32*t2 ; t4<=min(n-1,32*t2+31) ; t4++) {
        x1[t3]=x1[t3]+A[t3][t4]*y_1[t4] ;
        x2[t3]=x2[t3]+A[t4][t3]*y_2[t4] ; }}}}
```

Listing 8.3 – `mvt`*: Pluto Optimized Version.*

```
for (i=0 ; i<n ; i++)
  for (j=0 ; j<n ; j++)
    for (k=0 ; k<m ; k++) {
      C[i][j]+=alpha*A[i][k]*B[j][k] ;
      C[i][j]+=alpha*B[i][k]*A[j][k] ;
    }
```

Listing 8.4 – `syr2k`*: Original & Pluto Code.*

With `syr2k`, the XFOR code in Listing 8.5 is built by splitting the iteration domain into two iteration domains respectively associated with one of the two statements, and by exchanging the loops of the second domain from i-j-k to j-i-k. Thus, each couple of accesses to elements of arrays A and B is reused

consecutively by both statements. Temporal reuse was also promoted by using a temporary variable instead of `C[i][j]` for the second statement. Pluto just kept intact the original code which seemingly was evaluated exhibiting a good data locality (see Listing 8.4).

```
xfor (io=0, j1=0 ; io<n, j1<n ; io++, j1++ ; 1,1 ; 0,0 ) {
 xfor (jo=0, i1=0 ; jo<n, i1<n ; jo++, i1++ ; 1,1 ; 0,0 ) {
  0: tempo=0.0 ;
  1: temp1=0.0 ;
  xfor (ko=0,k1=0 ; ko<m,k1<m ; ko++,k1++ ; 1,1 ; 0,0 ) {
   0: tempo+=alpha*A[io][ko]*B[jo][ko] ;
   1: temp1+=alpha*B[i1][k1]*A[j1][k1] ; }
  0: C[io][jo]+=tempo ;
  1: C[i1][j1]+=temp1 ;
 }
}
```

Listing 8.5 – `syr2k`*: XFOR Code: Splitting + Interchange + Fusion.*

Pluto's heuristics do not seem to promote temporal data reuse among different statements at all, despite the `-rar` and `-maxfuse` options. For example, with `mvt`, Pluto did not detect the opportunity of interchanging loops of the second loop nest before merging them. With `syr2k`, the XFOR code promotes the inter-statement data reuse of elements of matrices A and B, while the Pluto code prioritizes only intra-statement data locality for each single access to the matrices. Similar situations occur with `3mm` and `gauss-filter`.

8.2.2 Gap 2: Excess of Conditional Branches

Codes `seidel`, `correlation` and `covariance`, are symptomatic cases where loop tiling is more penalizing than advantageous, despite the fact that it may provide a significantly better cache performance. Pluto's best performing versions for these three codes are tiled versions embedding many additional loop levels and complex loop bounds made with combinations of `min, max, floor` and `ceiling` functions invocations (see Listing 8.8). This additional control yields many more branches in the final generated code than in a version built without tiling, and thus more machine instructions. Consequently, Pluto's codes are more exposed to branch misses as exhibited by the performance counters (see Tables 8.6, 8.7 and 8.8). Moreover, branches resulting from complex combinations of `min, max, floor` and `ceiling` may be hardly predictable. Thus, the larger amount of instructions and the related branch misses completely annihilate the gain expected from the significantly lower number of TLB misses generated with `seidel` and `covariance` Pluto's versions. No tiling has been applied in the XFOR codes. Notice that for `covariance`, the XFOR code is even exhibiting more stalled cycles than the Pluto code, although it is still globally faster.

The original kernel code of `seidel` benchmark is represented in Listing 8.6. The corresponding XFOR optimized version is displayed in Listing 8.7 and the Pluto-optimized code is shown in Listing 8.6. One can observe the complex loop bounds in the code generated by Pluto, which results from skewing and tiling transformations. Such loop bounds yield many conditional branches and machine instructions in the final executable code. Alternatively, The XFOR code has been built by first splitting the original statement into several simpler statements which are grouped in several distinct loop nests. Then, the outermost XFOR-loop is unrolled 2 times. And finally, the offset values are assigned in a manner that promotes minimizing data reuse distances and that is compliant regarding data dependencies. Notice that splitting the original statements into simpler statements enables a finer schedule of memory accesses better promoting short data reuse distances. The XFOR code yields simple loop bounds in the efficient code generated by the XFOR compiler IBB which is shown in Listing 8.9.

```
for (t=0 ; t<=tsteps-1 ; t++)
  for (i=1 ; i<=n-2 ; i++)
    for (j=1 ; j<=n-2 ; j++)
      A[i][j]=(A[i-1][j-1]+A[i-1][j]+A[i-1][j+1]+A[i][j-1]+A[i][j]
              +A[i][j+1]+A[i+1][j-1]+A[i+1][j]+A[i+1][j+1])/9.0 ;
```

Listing 8.6 – *Original Code of* `seidel`.

```
for (t=0 ; t<=tsteps-1 ; t++) {
xfor (i0=1,i1=1,i2=1,i3=1 ; i0<=n-2,i1<=n-2,i2<=n-2,i3<=n-2 ;
      i0+=2,i1+=2,i2+=2,i3+=2 ; 1,1,1,1 ; 0,0,0,0 ) {
xfor (j0=1,j1=1,j2=1,j3=1 ; j0<=n-2,j1<=n-2,j2<=n-2,j3<=n-2 ;
      j0++,j1++,j2++,j3++ ; 1,1,1,1 ; 0,1,1,2 ) {
0: { A[i0][j0]+=A[i0][j0+1] ;
     A[i0][j0]+=A[i0+1][j0-1] ; }
1: { A[i1][j1]+=A[i1+1][j1] ;
     A[i1][j1]+=A[i1+1][j1+1] ;
     A[i1+1][j1]+=A[i1+2][j1] ;
     A[i1+1][j1]+=A[i1+1][j1+1] ;
     A[i1+1][j1]+=A[i1+2][j1-1] ;
     A[i1+1][j1]+=A[i1+2][j1+1] ; }
2: { A[i2][j2]=(A[i2][j2]+A[i2-1][j2-1]+A[i2-1][j2]
               +A[i2-1][j2+1]+A[i2][j2-1])/9.0 ; }
3: { A[i3+1][j3]=(A[i3+1][j3]+A[i3][j3-1]+A[i3][j3]
                  +A[i3][j3+1]+A[i3+1][j3-1])/9.0 ; } } } }
```

Listing 8.7 – `seidel`*: XFOR Code: Statement Split + Shifts.*

```
for (t1=0; t1<=floord(tsteps-1,32); t1++)
 for (t2=t1; t2<=min(floord(32*t1+n+29,32),floord(tsteps+n-3,32)); t2++)
  for (t3=max(ceild(64*t2-n-28,32),t1+t2);
       t3<=min(min(min(min(floord(32*t1+n+29,16),
       floord(tsteps+n-3,16)), floord(64*t2+n+59,32)),
       floord(32*t1+32*t2+n+60,32)),floord(32*t2+tsteps+n+28,32)); t3++)
   for (t4=max(max(max(32*t1,32*t2-n+2),16*t3-n+2),-32*t2+32*t3-n-29);
        t4<=min(min(min(min(32*t1+31,32*t2+30),16*t3+14),tsteps-1),
        -32*t2+32*t3+30); t4++)
    for (t5=max(max(32*t2,t4+1),32*t3-t4-n+2);
         t5<=min(min(32*t2+31,32*t3-t4+30),t4+n-2); t5++)
     for (t6=max(32*t3,t4+t5+1); t6<=min(32*t3+31,t4+t5+n-2); t6++) {
      A[-t4+t5][-t4-t5+t6] = ...; }
```

Listing 8.8 – `seidel`: *Pluto Code: Skewing + Tiling.*

```
for (t=0;t<=tsteps-1;t++) {
  if (n>=3) {
    for (i=0;i<=floord (n-3,2);i++) {
      A[2*i+1][1]+=A[2*i+1][2];
      A[2*i+1][1]+=A[2*i+2][0];
      if ((i==0)&&(n==3)) {
        A[1][1]+=A[2][1];
        A[1][1]+=A[2][2];
        A[2][1]+=A[3][1];
        A[2][1]+=A[2][2];
        A[2][1]+=A[3][0];
        A[2][1]+=A[3][2];
        A[1][1]=(A[1][1]+A[0][0]+A[0][1]+A[0][2]+A[1][0])/9.0;
      }
      if (n>=4) {
        A[2*i+1][2]+=A[2*i+1][3];
        A[2*i+1][2]+=A[2*i+2][1];
        A[2*i+1][1]+=A[2*i+2][1];
        A[2*i+1][1]+=A[2*i+2][2];
        A[2*i+2][1]+=A[2*i+3][1];
        A[2*i+2][1]+=A[2*i+2][2];
        A[2*i+2][1]+=A[2*i+3][0];
        A[2*i+2][1]+=A[2*i+3][2];
        A[2*i+1][1]=(A[2*i+1][1]+A[2*i][0]+A[2*i][1]+A[2*i][2]+A[2*i+1][0])/9.0;
      }
      for (j=2;j<=n-3;j++) {
        A[2*i+1][j+1]+=A[2*i+1][j+2];
        A[2*i+1][j+1]+=A[2*i+2][j];
        A[2*i+1][j]+=A[2*i+2][j];
        A[2*i+1][j]+=A[2*i+2][j+1];
        A[2*i+2][j]+=A[2*i+3][j];
        A[2*i+2][j]+=A[2*i+2][j+1];
        A[2*i+2][j]+=A[2*i+3][j-1];
        A[2*i+2][j]+=A[2*i+3][j+1];
        A[2*i+1][j]=(A[2*i+1][j]+A[2*i][j-1]+A[2*i][j]
                     +A[2*i][j+1]+A[2*i+1][j-1])/9.0;
        A[2*i+2][j-1]=(A[2*i+2][j-1]+A[2*i+1][j-2]+A[2*i+1][j-1]
                       +A[2*i+1][j]+A[2*i+2][j-2])/9.0;
      }
      if (n>=4) {
        A[2*i+1][n-2]+=A[2*i+2][n-2];
        A[2*i+1][n-2]+=A[2*i+2][n-1];
```

```
43          A[2*i+2][n-2]+=A[2*i+3][n-2];
44          A[2*i+2][n-2]+=A[2*i+2][n-1];
45          A[2*i+2][n-2]+=A[2*i+3][n-3];
46          A[2*i+2][n-2]+=A[2*i+3][n-1];
47          A[2*i+1][n-2]=(A[2*i+1][n-2]+A[2*i][n-3]+A[2*i][n-2]
48                         +A[2*i][n-1]+A[2*i+1][n-3])/9.0;
49          A[2*i+2][n-3]=(A[2*i+2][n-3]+A[2*i+1][n-4]+A[2*i+1][n-3]
50                         +A[2*i+1][n-2]+A[2*i+2][n-4])/9.0;
51        }
52        A[2*i+2][n-2]=(A[2*i+2][n-2]+A[2*i+1][n-3]+A[2*i+1][n-2]
53                       +A[2*i+1][n-1]+A[2*i+2][n-3])/9.0;
54      }
55    }
56  }
```

Listing 8.9 – `seidel`: *IBB-Generated Code.*

Seidel	Pluto	XFOR	Ratios
#CPU cycles	**15,721M**	**7,476M**	**-52.45%**
#L1 data loads	3,099M	672M	-78.31%
#L1 misses	12M	83M	+569.40%
#L2 misses	3.7M	1.2M	-65.64%
#L3 misses	3.9M	3.4M	-12.69%
#TLB misses	78K	688K	+783.18%
#Branches	**387M**	**179M**	**-53.88%**
#Branch misses	**456K**	**132K**	**-70.97%**
#Stalled Cycles	**11,297M**	**4,499M**	**-60.18%**
#Resource related stalls	**11,030M**	**4,4281M**	**-59.85%**
#Reservation Station stalls	**3,017M**	**440M**	**-85.39%**
#Re-Order Buffer stalls	**9,466M**	**3,982M**	**-57.93%**
#Instructions	**10,015M**	**7,857M**	**-21.55%**

Table 8.6 – *XFOR Speedups Partially Attributable to Decreased Branch Mispredictions Regarding* `Seidel` *Code*

Correlation	Pluto	XFOR	Ratios
#CPU cycles	**425M**	**426M**	**+0.22%**
#L1 data loads	224M	186M	-17.10%
#L1 misses	3.7M	12M	+223.95%
#L2 misses	2.2M	1M	-50.77%
#L3 misses	635K	395K	-37.83%
#TLB misses	294K	306K	+4.27%
#Branches	**120M**	**78M**	**-34.39%**
#Branch misses	**549K**	**231K**	**-58.01%**
#Stalled Cycles	**115M**	**47M**	**-58.79%**
#Resource related stalls	**81M**	**24M**	**-69.49%**
#Reservation Station stalls	**47M**	**3.7M**	**-92.10%**
#Re-Order Buffer stalls	**16M**	**14M**	**-13.31%**
#Instructions	**906M**	**934M**	**+3.03%**

Table 8.7 – *XFOR Speedups Partially Attributable to Decreased Branch Mispredictions Regarding* `Correlation` *Code*

Covariance	Pluto	XFOR	Ratios
#CPU cycles	**419M**	**320M**	**-23.71%**
#L1 data loads	217M	117M	-46.19%
#L1 misses	3.5M	22M	+539%
#L2 misses	1.9M	9M	+366.65%
#L3 misses	744K	496K	-33.42%
#TLB misses	247K	501K	+102.87%
#Branches	**119M**	**35M**	**-70.40%**
#Branch misses	**721K**	**199K**	**-72.37%**
#Stalled Cycles	**61M**	**123M**	**+100.75%**
#Resource related stalls	**59M**	**117M**	**+98.54%**
#Reservation Station stalls	**44M**	**43M**	**-1.40%**
#Re-Order Buffer stalls	**17M**	**75M**	**+344.54%**
#Instructions	**1,050M**	**506M**	**-51.86%**

Table 8.8 – *XFOR Speedups Partially Attributable to Decreased Branch Mispredictions Regarding* `Covariance` *Code*

Complex loop control yields also many more instructions of various kinds in the final executable than with simpler control, as it is clearly highlighted by the number of retired instructions for `seidel` and `covariance`. This issue, which is specifically addressed in the next subsection, impacts also solely performance significantly.

8.2.3 Gap 3: Number of Instructions

Jacobi-2d	Pluto	XFOR1	XFOR2
#CPU cycles	**12,136M**	**13,700M**	**12,641M**
#L1 data loads	1,400M	1,530M	1,529M
#L1 misses	236M	206M	205M
#L2 misses	**44M**	**6M**	**11M**
#L3 misses	**76M**	**68M**	**68M**
#TLB misses	2.7M	2.8M	3M
#Branches	657M	564M	650M
#Branch misses	1,560K	1,448K	1,329K
#Stalled Cycles	9,265M	9,463M	8,673M
#Resource related stalls	8,317M	8,433M	7,606M
#Reservation Station stalls	1,123M	1,088	930M
#Re-Order Buffer stalls	5,435M	4,775M	4,740M
#Instructions	**6,950M**	**9,370M**	**10,469M**

Table 8.9 – *XFOR slowdowns attributable to higher instruction counts*

Both Pluto and `XFOR1` codes of Table 8.9 are implementing a similar transformation of the original `jacobi-2d` code which consists in fusing both original loop nests in order to promote inter-statement data reuse and minimize loop control cost. Even if `XFOR1` and `XFOR2` exhibit a better data locality than Pluto's code (fewer caches misses), they also execute a significantly greater number of instructions making them slower. The small differences in

the reservation station and re-order buffer stalls show that the execution times differences are not significantly influenced by differences regarding memory operations or dependencies between instructions.

8.2.4 Gap 4: Unaware Data Locality Optimization

We have written three XFOR code versions of the polybench `seidel` code which just differ by their offset values. The XFOR code is shown in Listing 8.10 while the offset values are shown in Table 8.11. Notice that these codes have a different shape than the XFOR `seidel` code addressed in subsection 8.2.2, which explains the different counter values. One can observe from the performance counters that these three codes are behaving mostly similarly at runtime, while showing important execution time differences. The only performance counters showing significant differences are those related to stalled cycles. However, neither the amount of branch misses, instructions, nor cache misses can explain these differences. Some of these numbers seem even slightly more favorable for the slowest code.

```
for (t = 0 ; t <= tsteps-1 ; t++)
xfor (i0=1,i1=1,i2=1,i3=1,i4=1 ;
      i0<=n-2,i1<=n-2,i2<=n-2,i3<=n-2,i4<=n-2 ;
      i0+=2,i1+=2,i2+=2,i3+=2,i4+=2 ;
      1,1,1,1,1 ; /* grains */
      ?,?,?,?,? ) /* offsets */ {
xfor (j0=1,j1=1,j2=1,j3=1,j4=1 ;
      j0<=n-2,j1<=n-2,j2<=n-2,j3<=n-2,j4<=n-2 ;
      j0++,j1++,j2++,j3++,j4++ ;
      1,1,1,1,1 ; /* grains */
      ?,?,?,?,? ) /* offsets */ {

      0: { A[i0][j0] += A[i0][j0+1] ;
           A[i0+1][j0] += A[i0+1][j0+1] ; }
      1: { A[i1][j1] += A[i1+1][j1-1] ;
           A[i1+1][j1] += A[i1+2][j1-1] ; }
      2: { A[i2][j2] += A[i2+1][j2] ;
           A[i2+1][j2] += A[i2+2][j2] ; }
      3: { A[i3][j3] += A[i3+1][j3+1] ;
           A[i3+1][j3] += A[i3+2][j3+1] ; }
      4: { A[i4][j4] = (A[i4][j4]+A[i4-1][j4-1]+A[i4-1][j4]
                        +A[i4-1][j4+1]+A[i4][j4-1])/9.0 ;
           A[i4+1][j4] = (A[i4+1][j4]+A[i4][j4-1]+A[i4][j4]
                          +A[i4][j4+1]+A[i4+1][j4-1])/9.0 ; } }}
```

Listing 8.10 – *The XFOR seidel code used for register dependence analysis*

XFOR1	ms
addsd %xmm7, %xmm0	
addsd %xmm1, %xmm0	44
divsd %xmm3, %xmm0	
movsdq %xmm0, -0x8(%r8)	8
movsdq -0x8(%rcx), %xmm2	
movsdq (%r9), %xmm13	72
addsd %xmm1, %xmm2	
movapd %xmm13, %xmm1	
addsd %xmm9, %xmm1	
addsd %xmm0, %xmm2	12
addsd %xmm7, %xmm1	
addsd %xmm13, %xmm2	
addsd %xmm5, %xmm2	
divsd %xmm3, %xmm2	20
movsdq %xmm2, -0x8(%rcx)	796
movsdq (%rax), %xmm11	28

XFOR2	ms
addsd %xmm11, %xmm2	
addsd %xmm0, %xmm2	70
addsd %xmm4, %xmm0	108
divsd %xmm3, %xmm2	
movsdq %xmm2, (%rdi)	542
addsd %xmm2, %xmm0	48
movsdq 0x8(%r9), %xmm9	64
addsd %xmm9, %xmm0	
addsd %xmm1, %xmm0	40
movapd %xmm10, %xmm1	78
divsd %xmm3, %xmm0	
movsdq %xmm0, (%rax)	526
movsdq 0x8(%rcx), %xmm4	40

XFOR3	ms
addsd %xmm9, %xmm0	28
addsd %xmm7, %xmm0	
addsd %xmm8, %xmm0	60
divsd %xmm3, %xmm0	48
movsdq %xmm0, -0x8(%rcx)	602
addsd %xmm0, %xmm1	20
movsdq (%r9), %xmm2	124
addsd %xmm2, %xmm1	
addsd %xmm13, %xmm1	96
divsd %xmm3, %xmm1	42
addsd %xmm1, %xmm2	824
movsdq %xmm1, -0x8(%rdx)	74

Table 8.10 – *Total Aggregated CPU Time per Instructions (ms) – Source: Intel VTune*

This performance issue is probably the most surprising one among the five issues highlighted in this chapter. It is generally difficult to identify since it is usually hidden by other performance issues. The XFOR structure allows to isolate it, thanks to its explicit control of the data reuse distances, which enables the generation of several code versions which are all exhibiting a similar well-optimized data locality.

Thanks to the Intel Vtune profiling tool, a precise view of the CPU time spent by the respective groups of most time-consuming assembly instructions of `XFOR1`, `XFOR2` and `XFOR3` is shown in Table 8.10. It clearly shows excessive times spent by some instructions. Instructions spending up to hundreds of milliseconds are exhibiting dependencies due to accesses to common registers that could not be resolved through register renaming. These dependencies are typically Read-After-Write (RAW) dependencies. These excessive latencies are particularly exacerbated by the use of the x86 `divsd` floating-point division instruction which is costly: its latency is about 24 CPU cycles on Westmere microprocessors as reported in the related documentations. Thus, any delay regarding its execution

has a significant impact on depending instructions, and any delay regarding instructions on which it depends extends significantly its latency.

Typically, in this example in Table 8.10, each instruction following immediately instruction `divsd` exhibits a high latency due to its RAW register dependence with instruction `divsd`: `movsdq` and register `xmm2` for `XFOR1`, `movsdq` and register `xmm0` for `XFOR2`, `addsd` and register `xmm1` for `XFOR3`.

Seidel	XFOR1	XFOR2	XFOR3
Offsets-i	0,0,0,0,1	0,1,0,0,1	0,1,1,1,1
Offsets-j	0,0,0,0,0	0,0,0,0,0	0,0,0,0,0
#CPU cycles	**7,392M**	**11,393M**	**12,283M**
#L1 data loads	986M	997M	837M
#L1 misses	123M	123M	103M
#L2 misses	1.9M	1.9M	1.6M
#L3 misses	3.5M	3.5M	3.5M
#TLB misses	725K	694K	693K
#Branches	97M	94M	96M
#Branch misses	74K	78K	78K
#Stalled Cycles	**5,100M**	**8,002M**	**9,367M**
#Resource related stalls	**5,076M**	**7,969M**	**9,334M**
#Reservation Station stalls	**1,543M**	**7,765M**	**9,130M**
#Re-Order Buffer stalls	**3,537M**	**170M**	**157M**
#Instructions	6,131M	7,146M	6,503M

Table 8.11 – *Increased Stalls Attributable to Increased Register Dependencies for Three XFOR Versions of* `Seidel`

These code examples show that a "too good" data locality may introduce long chains of many short dependencies making instructions so tightly coupled that despite register renaming, and despite out-of-order execution, the microprocessor cannot find any independent instructions to launch simultaneously. This issue is particularly highlighted by the higher counts of the reservation station stalls in Table 8.11.

8.2.5 Gap 5: Insufficient Handling of Vectorization Opportunities

Tables 8.12, 8.13 and 8.14 show three codes whose XFOR versions are significantly faster than Pluto's versions, although their respective performance counters are not exhibiting great differences. Some counters are even in contradiction with the execution times (number of TLB and cache misses). In contrast to the previous issue regarding short dependencies between instructions, these codes are representative of another issue related to vectorization: the compiler automatically vectorize kernel loops of the XFOR codes, while it did not for Pluto's codes. This has been clearly observed thanks to the `-ftree-vectorizer-verbose` GCC option.

Jacobi-1d	Pluto	XFOR	Ratios
#CPU cycles	**9,711M**	**9,063M**	**-6.67%**
#L1 data loads	895M	885M	-0.03%
#L1 misses	110M	110M	-0.53%
#L2 misses	4M	4.7M	+16.78%
#L3 misses	54M	57M	+5.34%
#TLB misses	2.3M	2M	-15.51%
#Branches	508M	505M	-0.48%
#Branch misses	1,031K	1,174K	+13.91%
#Stalled Cycles	7,465M	6,844M	-8.32%
#Instructions	4,891M	4,924M	+0.69%

Table 8.12 – *Not Vectorized/Vectorized Codes* - `Jacobi-1d`

Fdtd-2d	Pluto	XFOR	Ratios
#CPU cycles	**7,631M**	**5,679M**	**-25.58%**
#L1 data loads	950M	962M	1.25%
#L1 misses	130M	114M	-12.29%
#L2 misses	5.6M	11.3M	+103.02%
#L3 misses	39M	32M	-18.81%
#TLB misses	1.8M	1.4M	-25.64%
#Branches	345M	249M	-27.85%
#Branch misses	755K	636K	-15.79%
#Stalled Cycles	5,844M	3,871M	-33.77%
#Instructions	3,936M	4,427M	+12.46%

Table 8.13 – *Not Vectorized/Vectorized Codes* - `fdtd-2d`

fdtd-apml	Pluto	XFOR	Ratios
#CPU cycles	**2,969M**	**1,871M**	**-36.96%**
#L1 data loads	360M	333M	-7.56%
#L1 misses	27M	30M	+10.85%
#L2 misses	971K	1,127K	+16.11%
#L3 misses	9.6M	9.2M	-3.55%
#TLB misses	710K	925K	+30.31%
#Branches	97M	81M	-17%
#Branch misses	476K	572K	+20.31%
#Stalled Cycles	2,196M	1,190M	-45.81%
#Instructions	1,581M	1,448M	-8.46%

Table 8.14 – *Not Vectorized/Vectorized Codes* - `fdtd-apml`

Vectorization is subject to two main parameters: data dependence and alignment. Processors' SIMD units require fixed-size vectors, say *sv*, of equally spaced data, *i.e.*, spaced by constant memory strides. Thus, *sv* iterations are run in parallel thanks to the SIMD unit. Mainstream compilers featuring automatic vectorization also require contiguous and regular memory access patterns. Thus, the XFOR programming strategy promoting vectorization is to build bodies of statements whose inter-statement data reuse distance is strictly greater than the SIMD vector size, and whose alignment of accessed data complies with

the straightforward processor requirements. A convenient adjustment of the offset values allows easy compliance with these requirements. In the following, we illustrate this programming strategy using the XFOR implementation of `jacobi-1d` (see Listing 8.11).

As done in the XFOR code, Pluto fuses appropriately both original loops into one unique loop where the second statement is shifted by one iteration in order to respect the Write-After-Read dependence regarding accesses to array A (see Listing 8.12). However, such a program construction does not promote vectorization since the CPU cannot run simultaneously both statements because of the simultaneous write and read of array element `A[t1-1]`. On the other hand, the XFOR structure allows to set the reuse distance precisely such that the final generated loops are in favor of automatic vectorization. For `jacobi-1d`, a reuse distance set to 9 provides the best performance as shown in Listing 8.13.

```
for (i=2 ; i<n-1 ; i++)
  B[i]=0.33333*(A[i-1]+A[i]+A[i+1]) ;
for (j=2 ; j<n-1 ; j++)
  A[j]=B[j] ;
```

Listing 8.11 – *Original Code of* `jacobi-1d`.

```
B[2]=0.33333*(A[1]+A[2]+A[3]) ;
for (t1=3 ; t1<=n-2 ; t1++) {
  B[t1]=0.33333*(A[t1-1]+A[t1]+A[t1+1]) ;
  A[t1-1]=B[t1-1] ; }
A[n-2]=B[n-2] ;
```

Listing 8.12 – *Pluto Version of* `jacobi-1d`*: Reuse Distance = 1.*

```
xfor (jo=2,j1=2 ; jo<n-1,j1<n-1 ; jo++,j1++ ; 1,1 ; 0,9) {
  0: B[jo]=0.33333*(A[jo-1]+A[jo]+A[jo+1]) ;
  1: A[j1]=B[j1] ; }
```

Listing 8.13 – *XFOR Version of* `jacobi-1d`*: Reuse Distance = 9.*

8.3 Experiments

Although several experiments have already been presented in Section 8.2, we show the execution times that were collected from running the best performing Pluto codes and XFOR codes in some other scenarios. The Red-Black Gauss-Seidel XFOR code is compared to its original version since it is not handled by Pluto, while every other XFOR code is compared to the best Pluto-code (speed-up = (Pluto time)/(XFOR time)). Pluto and XFOR versions have been compiled using *GCC*4.8.1 and *ICC*14.0.3 with options `O3` and `march=native`, and their

outputs have been compared to ensure correctness of the XFOR codes. Execution times of the main loop kernels are given in seconds in the tables. The XFOR grains are always set to 1, except for the Red-Black code whose grain is 2.

8.3.1 Sequential and vectorized codes

ICC was successful in vectorizing some codes while GCC was not. Typical examples are those exhibited in subsection 8.2.5: for `jacobi-1d`, `fdtd-2d` and `fdtd-apml`, ICC is able to vectorize the Pluto codes, while GCC only handles the XFOR codes. Measurements are shown in Table 8.15.

Code	Pluto time (gcc)	XFOR time (gcc)	Speed -up (gcc)	Pluto time (icc)	XFOR time (icc)	Speed -up (icc)
Red-Black	N/A	1.92	**1.66** *over orig.*	N/A	1.92	**2** *over orig.*
mvt	0.71	0.18	**3.94**	0.44	0.15	**2.93**
syr2k	2.54	2.12	**1.20**	1.43	1.32	**1.08**
3mm	5.93	1.61	**3.68**	0.93	1.60	**0.58**
gauss-filter	1.14	0.94	**1.21**	0.91	0.83	**1.10**
seidel	5.28	2.56	**2.06**	4.71	3.17	**1.49**
correlation	0.15	0.10	**1.50**	0.17	0.09	**1.88**
covariance	0.15	0.12	**1.25**	0.15	0.12	**1.25**
jacobi-2d	0.71	0.74	**0.95**	0.71	0.75	**0.95**
jacobi-1d	0.66	0.44	**1.50**	0.44	0.44	**1.00**
fdtd-2d	0.61	0.42	**1.45**	0.33	0.33	**1.00**
fdtd-apml	0.91	0.50	**1.82**	0.76	0.55	**1.38**

Table 8.15 – *Vectorized Code Measurements*

8.3.2 OpenMP loop parallelization

Code	Pluto time (gcc)	XFOR time (gcc)	Speed -up (gcc)	Pluto time (icc)	XFOR time (icc)	Speed -up (icc)
Red-Black	N/A	0.88	**1.18** *over orig.*	N/A	0.84	**1.09** *over orig.*
mvt	0.12	0.10	**1.2**	0.13	0.11	**1.18**
syr2k	0.28	0.17	**1.65**	0.25	0.16	**1.56**
3mm	1.75	*0.20*	**8.75**	0.27	*0.48*	**0.56**
gauss-filter	1.13	0.11	**10.27**	0.91	0.11	**8.27**
correlation	0.12	0.04	**3.00**	0.05	0.02	**2.50**
covariance	0.03	0.03	**1.00**	0.03	0.03	**1.00**
jacobi-2d	1.41	0.41	**3.44**	1.34	0.43	**3.12**
fdtd-2d	0.30	0.19	**1.58**	0.30	0.19	**1.58**
fdtd-apml	0.11	0.07	**1.57**	0.15	0.08	**1.88**

Table 8.16 – *OpenMP Parallel Code Measurements (12 threads)*

For the second set of measurements, OpenMP parallelization has been turned on in Pluto using option `-parallel`, in GCC using option `-fopenmp`, and in ICC using option `-openmp`. Codes have been run using 12 parallel threads mapped on the 6 hyperthreaded processor cores of the Xeon X5650 processor. `Seidel` does not appear in Table 8.16 because it requires skewing to allow parallelization. Parallelization of `jacobi-1d` does not provide any speed-up for neither Pluto nor XFOR codes. Measurements are given in Table 8.16 where speed-ups are given by comparing the parallel Pluto and parallel XFOR code versions.

8.4 Conclusion

In this chapter, we showed typical examples of codes from the Polybench benchmark suite [27] where XFOR enables more efficient codes than Pluto, thanks to the provided flexibility and expressiveness. Without loss of generality, we compared the best codes optimized using the polyhedral compiler Pluto [6] against corresponding XFOR codes. We particularly highlighted five important issues that may occur with automatically optimized codes: *insufficient data locality optimization, excess of conditional branches in the generated code, verbose code with too many machine instructions, data locality optimization resulting in processor stalls*, and finally, *missed vectorization opportunities.*

Although automatic optimizers will always improve regarding their effectiveness, there will also always be some performance gaps that any advanced user may want to fill. For this purpose, one may want to modify codes generated by automatic optimizers. Unfortunately, this option is not sustainable since automatically optimized codes are very complex and unreadable, and thus impossible to modify safely. In the next chapter, we show another interesting and related feature of the XFOR structure: using XFOR as an intermediate representation language for polyhedral optimizations, making it possible to share automatic and manual code transformations.

9 XFOR Polyhedral Intermediate Representation and Manual-Automatic Collaborative Approach for Loop Optimization

9.1 Introduction

Automatic polyhedral transformations, performed by dedicated compilers as Pluto, are based on heuristics that cannot , by nature, be always optimal. We have shown in the previous chapter that XFOR codes can often outperform Pluto codes. However, writing directly the best performing codes with XFOR may be very difficult. On the other hand, polyhedral source-to-source compilers' outputs are generally very complex to be understood and modified by a programmer. Our solution is to use XFOR codes as an intermediate representation form which can be more easily handled by a programmer to optimize further polyhedral compilers' output codes.

In this chapter, we show that the XFOR structure allows to express directly traditional loop transformations (Section 9.2), and more generally any composition of transformations (Section 9.3). In Section 9.4, we present our implementation of this concept as a tool named XFORGEN, which automatically generates an XFOR representation from any OpenScop representation of transformed loop nests. We use this tool in Section 9.5 to further improve outputs of the polyhedral compiler Pluto, through manual modifications made with ease thanks to the XFOR representation.

9.2 Loop Transformations

The main goals of loop transformations are data locality optimization and parallelization. Loop transformations are various and can be carried out in a very flexible way. In the following, we show how major loop transformations can be expressed easily using XFOR.

9.2.1 Loop Shifting

Loop shifting is applied by delaying a statement by a constant number of iterations. It is beneficial for increasing *Instruction Level parallelism* (ILP) by allowing pipelining. This transformation is not always legal (it must enforce data dependencies). Loop shifting is directly controlled by the offset. Actually, in order to shift a statement by a factor f, the corresponding offset value must be set simply to f. In terms of statement scheduling, consider the schedule θ_S of a statement S. Shifting index i by f will modify the schedule as shown below:

$$\theta_S(i,j) = \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \quad \begin{matrix} Shift(S,i,f) \\ \Longrightarrow \end{matrix} \quad \theta'_S(i,j) = \left\{ \begin{pmatrix} \beta'_0 \\ \alpha'_1 \\ \beta'_1 \\ \alpha'_2 \\ \beta'_2 \end{pmatrix} \middle| \begin{matrix} \beta'_0 = \beta_0 \\ \alpha'_1 = \alpha_1 + f \\ \beta'_1 = \beta_1 \\ \alpha'_2 = \alpha_2 \\ \beta'_2 = \beta_2 \end{matrix} \right\}$$

As an example, consider the initial XFOR code in Listing 9.1. The original iteration domain is represented on the left in Figure 9.1. In Listing 9.2, we give the shifted XFOR code. Here, index $i1$ was shifted by a factor equal to 5 and index $j1$ was shifted by a factor equal to 2. The iteration domain after transformation is shown on the right in Figure 9.1. The IBB-generated code (Listing 9.3) shows the code that should have been written by a programmer with traditional for-loops instead of XFOR-loops if he had applied manually this transformation. The schedule of $S1$, θ_{S1}, is transformed into θ'_{S1}:

$$\theta_{S1}(i_1,j_1) = \begin{pmatrix} 0 \\ i_1 \\ 0 \\ j_1 \\ 1 \end{pmatrix} \quad \Longrightarrow \quad \theta'_{S1}(i_1,j_1) = \begin{pmatrix} 0 \\ i_1 + \mathbf{5} \\ 0 \\ j_1 + \mathbf{2} \\ 1 \end{pmatrix}$$

```
1 xfor (i0=0,i1=0 ; i0<10,i1<5 ; i0++,i1++ ; 1,1 ; 0,0)
2 xfor (j0=0,j1=0 ; j0<10,j1<5 ; j0++,j1++ ; 1,1 ; 0,0) {
3   0: S0(i0,j0);
4   1: S1(i1,j1); }
```

Listing 9.1 – *Loop Shifting - Initial XFOR Code.*

```
xfor (io=0,i1=0 ; io<10,i1<5 ; io++,i1++ ; 1,1 ; 0,5)
xfor (jo=0,j1=0 ; jo<10,j1<5 ; jo++,j1++ ; 1,1 ; 0,2) {
  0: So(io,jo);
  1: S1(i1,j1); }
```

Listing 9.2 – *Loop Shifting - (i1,j1) Shifted XFOR Code.*

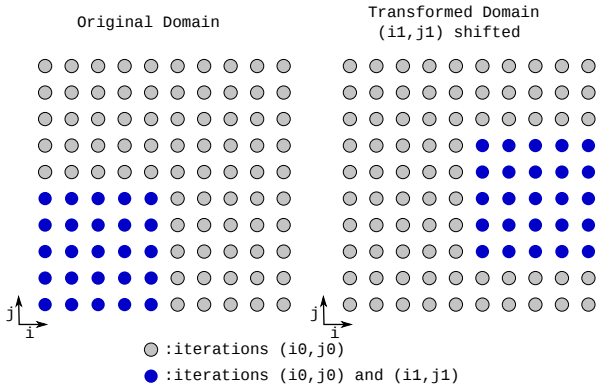


Figure 9.1 – *Example of Loop Shifting with XFOR.*

```
for (i=0;i<=4;i++)
  for (j=0;j<=9;j++)
    So(i,j);
for (i=5;i<=9;i++) {
  for (j=0;j<=1;j++)
    So(i,j);
  for (j=2;j<=6;j++) {
    So(i,j);
    S1(i-5,j-2);
  }
  for (j=7;j<=9;j++)
    So(i,j);
}
```

Listing 9.3 – *Loop Shifting - IBB Generated Code.*

9.2.2 Loop Distribution

This transformation is also called loop fission or splitting. It consists in dividing the loop control over different statements in the loop body. Loop distribution may be applied to any loop as long as the dependencies are preserved. All statements belonging to the same dependence cycles must be placed in the same sub-loop. If, for example, a statement S_2 depends on S_1 in the original loop, then the sub-loop containing S_1 must precede the sub-loop containing S_2.

Listing 9.4 shows a code example for which loop fission is invalid.

```
for (i=0 ; i<100 ; i++) {
  A[i] = B[i] + 2 * C[i] ;
  D[i] = A[i+1] ;
}
```

Listing 9.4 – *Code Example for which Loop Distribution is Not Valid.*

Loop distribution is beneficial in different ways. It is useful in isolating data dependencies cycles in preparation for loop vectorization. In addition to that, it can expose parallelism in one of the new loops. Also, loop splitting may enable other transformations, such as loop interchange. Moreover, this transformation helps in improving cache locality by reducing the total amount of data that is referenced during the entire execution of each loop. Finally, it separates different data streams in disjoint loops to promote hardware prefetching.

Loop fission can be achieved using the offset. Consider the initial for-loop code in Listing 9.5. An equivalent XFOR code is shown in Listing 9.6. In order to split this loop, all what is required is to choose as offset of the second index a value grater than the number of iterations of the previous index. The resulting XFOR-loop is shown in Listing 9.7. The IBB-generated code in Listing 9.8 shows the actual loop splitting. We show below the impact of this transformation on the scheduling of statement $S1$.

$$\theta_{S0}(i_0) = \begin{pmatrix} 0 \\ i_0 \\ 0 \end{pmatrix} \implies \theta'_{S0}(i_0) = \begin{pmatrix} 0 \\ i_0 \\ 0 \end{pmatrix}$$

$$\theta_{S1}(i_1) = \begin{pmatrix} 0 \\ i_1 \\ 1 \end{pmatrix} \implies \theta'_{S1}(i_1) = \begin{pmatrix} 0 \\ i_1 + (\mathbf{n} - \mathbf{1}) \\ 1 \end{pmatrix}$$

```
for (i=1 ; i<n ; i++) {
  S0(i);
  S1(i); }
```

Listing 9.5 – *Loop Distribution - Initial Code.*

```
xfor (io=1,i1=1 ; io<n,i1<n ; io++,i1++ ; 1,1 ; 0,0) {
  0: S0(io);
  1: S1(i1); }
```

Listing 9.6 – *Loop Distribution - XFOR Equivalent Code.*

```
xfor (io=1,i1=1 ; io<n,i1<n ; io++,i1++ ; 1,1 ; 0,n-1) {
  0: S0(io);
  1: S1(i1); }
```

Listing 9.7 – *Loop Distribution - XFOR Code Distributed Version.*

```
for (i=0 ; i<=n-2 ; i++)
  S0(i+1);
for (i=n-1 ; i<=2*n-3 ; i++)
  S1(i - n+2);
```

Listing 9.8 – *Loop Distribution - IBB Generated Code.*

9.2.3 Loop Fusion

This transformation merges adjacent loops into one unique loop. It is valid if it does not introduce any lexicographically negative data dependence. Fusion is useful for reducing loop overhead, increasing the granularity of work done in a loop and reducing data reuse distances.

Loop Fusion is very easily achieved using the offset. Consider the initial for-loop code in Listing 9.9. The equivalent XFOR-loop is shown in Listing 9.10. In order to merge the loops, all what is required is to set the offset of the second index to the minimum value where no lexicographically negative data dependence occurs. Consider 10 as a valid minimum value, then the resulting XFOR-loop is the one in Listing 9.11. The IBB-generated code in Listing 9.12 shows the actual loop fusion. We show below the impact of this transformation on the scheduling of statement $S1$.

$$\theta_{S0}(i_0) = \begin{pmatrix} 0 \\ i_0 \\ 0 \end{pmatrix} \quad \Longrightarrow \quad \theta'_{S0}(i_0) = \begin{pmatrix} 0 \\ i_0 \\ 0 \end{pmatrix}$$

$$\theta_{S1}(i_1) = \begin{pmatrix} 0 \\ i_1 + N - 1 \\ 1 \end{pmatrix} \quad \Longrightarrow \quad \theta'_{S1}(i_1) = \begin{pmatrix} 0 \\ i_1 + \mathbf{10} \\ 1 \end{pmatrix}$$

Note that with the XFOR construct, the user does not need to worry about any prologue or epilogue code and about the loop bounds. An XFOR-loop can contain indices with different bounds. The IBB compiler generates the convenient scanning code for the domains, where the loop nests are fused only on the sub-domains in which they overlap.

```
for (i=0 ; i<N ; i++)
  S0(i);
for (i=0 ; i<M ; i++)
  S1(i);
```

Listing 9.9 – *Loop Fusion - Initial Code.*

```
xfor (io=0,i1=0 ; io<N,i1<M ; io++,i1++ ; 1,1 ; 0,N-1) {
  0: S0(io);
  1: S1(i1); }
```

Listing 9.10 – *Loop Fusion - XFOR Equivalent Initial Code.*

```
xfor (io=0,i1=0 ; io<N,i1<M ; io++,i1++ ; 1,1 ; 0,10) {
  0: S0(io);
  1: S1(i1); }
```

Listing 9.11 – *Loop Fusion - Fused XFOR Code.*

```
for (i = 0 ; i <= min(9,N-1) ; i++)
  S0(i);
for (i = 10 ; i <= min(M+9,N-1) ; i++) {
  S0(i);
  S1(i - 10); }
for (i = max(10,M+10) ; i <= N-1 ; i++)
  S0(i);
for (i = max(10,N) ; i <= M+9 ; i++)
  S1(i - 10);
```

Listing 9.12 – *Loop Fusion - IBB Generated Code.*

9.2.4 Loop Reversal

Loop reversal consists of changing the order in which a loop iterates. The main advantage of this transformation is that data dependencies change, which may enable other optimizations. In addition to that, when loops iterate down to 0, the special "`jump if zero`" (`JMPZ`) machine instruction can be used. Loop reversal is admissible for any loop of depth p if all data dependencies are carried by outer loops, *i.e.*

$\forall$ dependence distance vector d: $d_p = 0 \wedge \exists k < p : d_k \neq 0$

In order to isolate and better highlight how the direction of the dependency would be reversed by loop reversal, consider, without any loss of generality, this simple example of an XFOR loop composed by one index (see Listing 9.14). The equivalent initial for-loop nest is shown in Listing 9.13. The dependence distance in the initial version is: $d = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$.

```
for ( i1 = 1 ; i1 <= 4 ; i1++)
  for ( i2 = 1 ; i2 <= 4 ; i2++)
    A[i1][i2] = A[i1-1][i2+1] + 16;
```

Listing 9.13 – *Loop Reversal Example - Initial Code.*

```
xfor ( i1=1 ; i1<=4 ; i1++ ; 1 ; 0)
  xfor ( i2=1 ; i2<=4 ; i2++ ; 1 ; 0) {
    0: A[i1][i2] = A[i1-1][i2+1] + 16; }
```

Listing 9.14 – *Loop Reversal Example - XFOR Equivalent Code.*

The outermost loop cannot be reversed, because that would lead to a negative dependence distance $d = \begin{pmatrix} -1 \\ -1 \end{pmatrix}$. However, the second index $i2$ can be reversed. This is done by setting the corresponding offset to $-2 \times i2$, since $-i2 = i2 - 2 \times i2$ (see Listing 9.15). The IBB generated code from this XFOR loop is represented in Listing 9.16. For this version, the new dependence distance is equal to $d = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$. We show below the impact of this transformation on the scheduling of the statement.

$$\theta_S(i_1, i_2) = \begin{pmatrix} 0 \\ i_1 \\ 0 \\ i_2 \\ 1 \end{pmatrix} \implies \theta'_S(i_1, i_2) = \begin{pmatrix} 0 \\ i_1 \\ 0 \\ -i_2 \\ 1 \end{pmatrix}$$

```
xfor ( i1=1 ; i1<=4 ; i1++ ; 1 ; 0)
  xfor ( i2=1 ; i2<=4 ; i2++ ; 1 ; -2*i2 ) {
    0: A[i1][i2] = A[i1-1][i2+1] + 16; }
```

Listing 9.15 – *Loop Reversal Example - Reversed XFOR Version.*

```
for (i = 0 ; i <= 3 ; i++)
  for (j = -3 ; j <= 0 ; j++)
    A[i+1][-j+1]=A[i][-j+2]+16;
```

Listing 9.16 – *Loop Reversal Example - IBB Generated Code.*

9.2.5 Loop Skewing

Loop skewing consists of shifting a loop index by an affine function of other loop indices. It is particularly used to exhibit parallel loops by removing cross-iteration dependencies for some loops. Loop Skewing is enabled by setting offsets that are affine functions of the encompassing and nested loop indices. For better clarity, we give two different illustrative examples.

9.2.5.1 Example 1

Consider the initial XFOR-loop nest in Listing 9.17. The corresponding iteration domain is represented in Figure 9.2 (on the left). Loop skewing of inner loop $j1$ by a factor f adds $f \times i1$ to the upper and lower bounds of $j1$ and subtracts $f \times i1$ from all array references of loop $j1$ (see Listing 9.19). In our example, the inner loop is skewed by a factor $f = 1$. We show below the impact of this skew on the scheduling of statement $S1$.

$$\theta_{S1}(i_1, j_1) = \begin{pmatrix} 0 \\ i_1 \\ 0 \\ j_1 \\ 1 \end{pmatrix} \quad \Longrightarrow \quad \theta'_{S1}(i_1, j_1) = \begin{pmatrix} 0 \\ i_1 \\ 0 \\ j_1 + \mathbf{i_1} \\ 1 \end{pmatrix}$$

The skewed XFOR code is shown in Listing 9.18 and the resulting domain is represented on the right in Figure 9.2.

```
xfor (io=0,i1=0 ; io<2*N,i1<N ; io++,i1++ ; 1,1 ; 0,0)
xfor (jo=0,j1=0 ; jo<2*N,j1<N ; jo++,j1++ ; 1,1 ; 0,0) {
  0: So(io,jo);
  1: S1(i1,j1); }
```

Listing 9.17 – *Example 1: Loop Skewing - Initial XFOR Code.*

```
xfor (io=0,i1=0 ; io<2*N,i1<N ; io++,i1++ ; 1,1 ; 0,0)
xfor (jo=0,j1=0 ; jo<2*N,j1<N ; jo++,j1++ ; 1,1 ; 0,i1) {
  0: So(io,jo);
  1: S1(i1,j1); }
```

Listing 9.18 – *Example 1: Loop Skewing - (i1,j1) Skewed XFOR Code.*

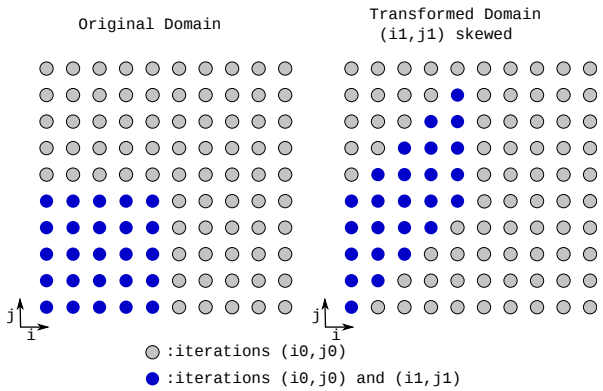


Figure 9.2 – *Example 1: Skewing with XFOR.*

```
for (i=0;i<=N-1;i++) {
  for (j=0;j<=i-1;j++)
    So(i,j);
  for (j=i ;j<=i+N-1;j++) { //skew
    So(i,j);
    S1(i,-i+j); //skew
  }
  for (j=i+N ;j<=2*N-1;j++)
    So(i,j);
}
for (i=N ;i<=2*N-1;i++)
  for (j=0;j<=2*N-1;j++)
    So(i,j);
```

Listing 9.19 – *Example 1: Loop Skewing - IBB Generated Code.*

9.2.5.2 Example 2

Consider again the initial XFOR-loop nest in Listing 9.17. Now, the outermost loop is skewed by inner index $j1$ as shown in the skewed XFOR code in Listing 9.20. The resulting domain is illustrated on the right in Figure 9.3. The IBB generated skewed code is displayed in Listing 9.21. We show below the impact of this skew on the scheduling of statement $S1$.

$$\theta_{S1}(i_1, j_1) = \begin{pmatrix} 0 \\ i_1 \\ 0 \\ j_1 \\ 1 \end{pmatrix} \quad \Longrightarrow \quad \theta'_{S1}(i_1, j_1) = \begin{pmatrix} 0 \\ i_1 + \mathbf{j_1} \\ 0 \\ j_1 \\ 1 \end{pmatrix}$$

```
xfor (io=0,i1=0 ; io<2*N,i1<N ; io++,i1++ ; 1,1 ; 0,j1)
xfor (jo=0,j1=0 ; jo<2*N,j1<N ; jo++,j1++ ; 1,1 ; 0,0) {
  0: So(io,jo);
  1: S1(i1,j1); }
```

Listing 9.20 – *Example 2: Loop Skewing - (i1,j1) Skewed XFOR Version.*

```
for (i=0 ; i<=2*N-2 ; i++) {
  for (j=0 ; j<=i-N ; j++)
    So(i,j);
  for (j=max(0,i-N+1) ; j<=min(i,N-1) ; j++) { //skew
    So(i,j);
    S1(i-j,j); //skew
  }
  for (j=N ; j<=i ; j++)
    So(i,j);
  for (j=i+1 ; j<=2*N-1 ; j++)
    So(i,j);
}
for (j=0 ; j<=2*N-1 ; j++)
  So(2*N-1,j);
```

Listing 9.21 – *Example 2: Loop Skewing - IBB Generated Code.*

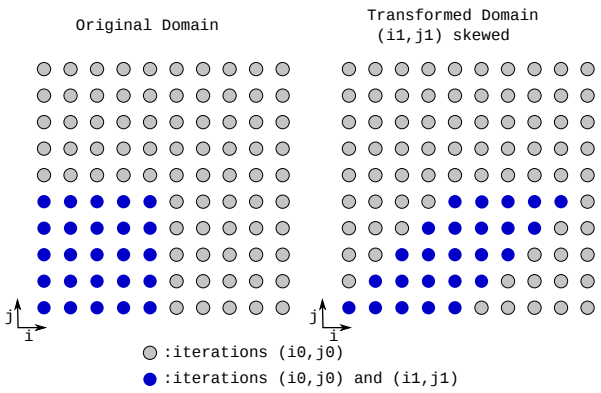


Figure 9.3 – *Example 2: Loop Skewing with XFOR.*

9.2.6 Loop Peeling

Loop peeling consists of removing some first or last iterations of a loop into a separate code outside the loop. It is always legal, provided that no additional iterations are introduced. This transformation is useful to enforce a particular initial memory alignment on array references prior to loop vectorization.

Loop Peeling can be performed using the offset. Consider the initial for-loop code in Listing 9.22. In order to extract the first and last iterations, we duplicate the loop statement into three different iteration domains defined in the XFOR header as shown in Listing 9.23. The IBB-generated code int Listing 9.24 shows the actual loop peeling. Note the additional runtime tests that guarantee the same number of iterations as in the initial code.

```
for (i=0 ; i<N ; i++)
  S(i);
```

Listing 9.22 – *Loop Peeling - Initial Code.*

```
xfor (io=0,i1=1,i2=max(1,N-1) ; io<min(1,N),i1<N-1,i2<N ;
      io++,i1++,i2++ ; 1,1,1 ; 0,1,N-1) {
  0: S(io);
  1: S(i1);
  2: S(i2); }
```

Listing 9.23 – *Loop Peeling - XFOR Peeled Code.*

```
if (N >= 1) {
  S(0);
  for (i=1 ; i<=N-2 ; i++) {
    S(i);
  }
  if (N >= 2)
    S(max(1,-1+N));
}
```

Listing 9.24 – *Loop Peeling - IBB Generated Code.*

9.2.7 Statement Reordering

Statements belonging to the same index are reordered in the same manner as in classical for-loops. However, statements associated to different indices are scheduled according to their offset and grain and the statement order in the XFOR body is relevant in the sub-domains on which they overlap.

9.2.8 Loop Interchange

This transformation switches the depths of two loops in a loop nest. It may be used to expose parallelism or to improve data locality. Loop interchange is legal if the outermost loop does not carry any data dependence going from one statement instance executed for i and j to another statement instance executed for i' and j' where $i < i'$ and $j > j'$.

Example. The code in Listing 9.25 shows an example of a loop nest for which loop interchange is not legal. But the loops in Listing 9.26 can be interchanged.

```
for (i=1 ; i<10 ; i++)
  for (j=1 ; j<10 ; j++)
    B[i][j] = B[i-1][j+1];
```

Listing 9.25 – *Code Example for which Loop Interchange is Not Valid.*

```
for (i=1 ; i<10 ; i++)
  for (j=1 ; j<10 ; j++)
    B[i][j] = B[i-1][j-1];
```

Listing 9.26 – *Code Example for which Loop Interchange is Valid.*

With XFOR, interchange of XFOR-loops may impact more than two indices. One index is interchanged with another index having the same rank in the XFOR header, *i.e.*, one of the encompassing or nested indices[1]. Interchanging indices i and j in an XFOR-loop nest can be done by setting the offset of index i to $j - i$ and by setting the offset of j to $i - j$.

To illustrate this, consider the loop nest in Listing 9.26. The loop interchange $((i, j) \rightarrow (j, i))$, is expressed as shown in Listing 9.27. Listing 9.28 represents the equivalent code generated by IBB. We show below the impact of this interchange on the scheduling of the statement.

$$\theta_S(i,j) = \begin{pmatrix} 0 \\ i-1 \\ 0 \\ j-1 \\ 1 \end{pmatrix} \quad \Longrightarrow \quad \theta'_S(i,j) = \begin{pmatrix} 0 \\ \mathbf{j-1} \\ 0 \\ \mathbf{i-1} \\ 1 \end{pmatrix}$$

```
xfor (i=1 ; i<10 ; i++ ; 1 ; j-i)
  xfor (j=1 ; j<10 ; j++ ; 1 ; i-j) {
    0: B[i][j] = B[i-1][j-1]; }
```

Listing 9.27 – *Loop Interchange - Interchanged XFOR Code.*

[1] In chapter 7 ([*XFOR-WIZARD*], page 97), we introduce our software tool "XFOR-WIZARD" which enables index interchange and verifies if it is legal regarding dependencies.

```
for (_mfr_ref0 =0; _mfr_ref0 <=8; _mfr_ref0 ++) {
  for (_mfr_ref1 =0; _mfr_ref1 <=8; _mfr_ref1 ++) {
    B[_mfr_ref1 +1][_mfr_ref0 +1]=B[_mfr_ref1][_mfr_ref0]; } }
```

Listing 9.28 – *Loop Interchange - IBB Generated Code.*

9.2.9 Loop Unrolling and Dilatation

9.2.9.1 Loop Unrolling

Loop unrolling is the combination of two or more loop iterations together with a corresponding reduction of the trip count. It is particularly useful for exposing more *Instruction Level parallelism* (ILP). This transformation is applicable with the XFOR construct using the grain parameter. Loop unrolling can be applied to one or more indices in the XFOR loop. In the IBB generated code, the duplicated statements of different indices will be interlaced according to the lexicographic order.

As an example, consider the initial XFOR code in Listing 9.29. We intend to unroll 4 times the second loop (*i.e.* index $i1$). To do this, the grain of $i0$ (not $i1$) must be set to 4. This means for every 4 iterations of $i1$, $i0$ will be executed once (see Figure 9.4). The unrolled XFOR is represented in Listing 9.30 and the equivalent IBB generated code is given in Listing 9.31.

```
xfor (i0=0,i1=0 ; i0<10,i1<10 ; i0++,i1++ ; 1,1 ; 0,0) {
  0: S0(i0);
  1: S1(i1); }
```

Listing 9.29 – *Loop Unrolling - Initial XFOR Code.*

```
xfor (i0=0,i1=0 ; i0<10,i1<10 ; i0++,i1++ ; 4,1 ; 0,0) {
  0: S0(i0);
  1: S1(i1); }
```

Listing 9.30 – *Loop Unrolling - i1 Unrolled XFOR Code.*

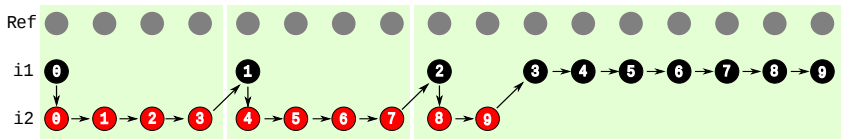


Figure 9.4 – *Loop Unrolling (Example 1) - XFOR Domain.*

```
for (i=0 ; i<=1 ; i++) {
  So (i);
  S1 (4*i);
  S1 (4*i+1);
  S1 (4*i+2);
  S1 (4*i+3);
}
So (2);
S1 (8);
S1 (9);
for (i=3 ; i<=9 ; i++)
  So (i);
```

Listing 9.31 – *Loop Unrolling - IBB Generated Code.*

Let us now consider the XFOR code example in Listing 9.32. Here, the grain of $i0$ is equal to 4 and the grain of $i1$ is equal to 3 . This means $i0$ is unrolled 3 times and $i1$ is unrolled 4 times. Consequently, the final loop body in the code generated by IBB will contain $3 + 4 = 7$ instructions as shown in Listing 9.33. For more understanding, we represent the XFOR uncompressed domain in Figure 9.5. The compressed domain in shown in Figure 9.6.

```
xfor (io=0,i1=0 ; io<10,i1<10 ; io++,i1++ ; 4,3 ; 0,0) {
  0: So(io);
  1: S1(i1); }
```

Listing 9.32 – *Loop Unrolling - Unrolled XFOR Code (Example 2).*

```
for (i=0 ; i<=1 ; i++) {
  So(3*i);
  S1(4*i);
  S1(4*i+1);
  So(3*i+1);
  S1(4*i+2);
  So(3*i+2);
  S1(4*i+3);
}
So(6);
S1(8);
S1(9);
So(7);
So(8);
So(9);
```

Listing 9.33 – *Loop Unrolling - IBB Generated Code (Example 2).*

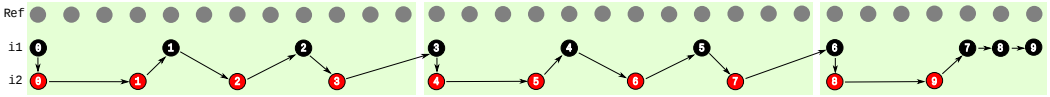


Figure 9.5 – *Loop Unrolling (Example 2) - Uncompressed XFOR Domain.*

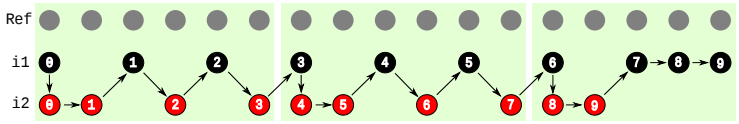


Figure 9.6 – *Loop Unrolling (Example 2) - Compressed XFOR Domain.*

More generally, for an XFOR loop:
`xfor(`i_1`,` i_2`, ...,` i_n `; ... ;` g_1`,` g_2`, ...,` g_n `; ...)`
each index i_m $(1 \leq m \leq n)$ is unrolled $lcm(g_k)/g_m$ $(k = 1..n)$times.

9.2.9.2 Loop Dilatation

Loop dilatation consists in stretching an index by a positive constant number. It can be used in order to improve the data locality of the program. This transformation is not always legal, since it must respect the data dependencies. Loop dilatation is directly linked to the grain. Actually, in order to stretch an index by a factor g, the corresponding grain value must be set simply to g. A loop dilatation is translated by IBB into a loop unrolling in order to have an efficient equivalent code, avoiding modulo conditionals.

As an example, consider the initial XFOR code in Listing 9.34. The original iteration domain is represented on the left in Figure 9.7. In Listing 9.35, we present the dilated XFOR loop. In this code, both indices $i1$ and $j1$ were stretched by a factor equal to 2. The transformed iteration domain is represented on the right in Figure 9.7. The IBB-generated code (Listing 9.36) shows the code that should have been written by a programmer with traditional for-loops instead of XFOR-loops if he had applied manually this transformation.

```
1 xfor (io=0,i1=0 ; io<10,i1<5 ; io++,i1++ ; 1,1 ; 0,0)
2 xfor (jo=0,j1=0 ; jo<10,j1<5 ; jo++,j1++ ; 1,1 ; 0,0) {
3   0: S0(io,jo);
4   1: S1(i1,j1); }
```

Listing 9.34 – *Loop Dilatation - Initial XFOR Code.*

```
1 xfor (io=0,i1=0 ; io<10,i1<5 ; io++,i1++ ; 1,2 ; 0,0)
2 xfor (jo=0,j1=0 ; jo<10,j1<5 ; jo++,j1++ ; 1,2 ; 0,0) {
3   0: S0(io,jo);
4   1: S1(i1,j1); }
```

Listing 9.35 – *Loop Dilatation - (i1,j1) Dilated XFOR Code.*

```
for (i=0 ; i<=4 ; i++) {
  for (j=0 ; j<=4 ; j++) {
    S0(2*i,2*j);
    S1(i,j);
    S0(2*i,2*j+1); }
  for (j=0 ; j<=4 ; j++) {
    S0(2*i+1,2*j);
    S0(2*i+1,2*j+1); }
}
```

Listing 9.36 – *Loop Dilatation - IBB Generated Code.*

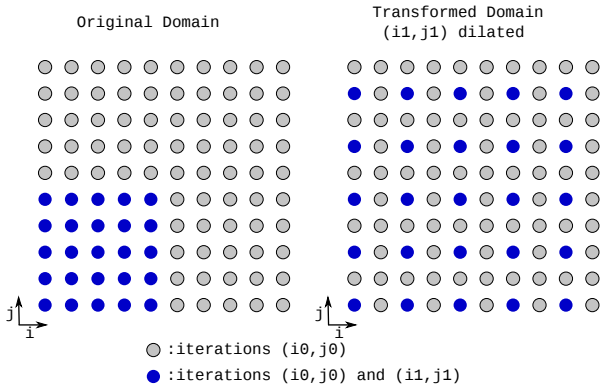


Figure 9.7 – *Example of Loop Dilatation with XFOR.*

9.2.10 Unroll and Jam

Unroll and jam implies partially unrolling one or more loops higher in the nest than the innermost loop, and fusing ("jamming") the resulting loops back together. This transformation increases the size of the loop body and hence available *Instruction Level parallelism* (ILP). It can be used also to improve data locality. An application example of unroll-and-jam is shown with the code in Listing 9.37. This code presents two successive loop nests. The first is updating array A and the second is updating even rows of array C, using the elements of array A that belong to the two previous rows. In order to improve data reuse among the statements, a way is to unroll the first loop nest and then fuse it with the second nest. This transformation can be applied using the XFOR construct in two steps, first the index $i0$ is unrolled by a factor equal to 2. This is done thanks to the XFOR nest represented in Listing 9.38. The equivalent unrolled XFOR nest generated by XFOR-WIZARD is represented in Listing 9.39. Finally, the jam is performed by fusing the generated loops by setting the corresponding offsets to 0 (see Listing 9.40). Listing 9.41 shows the equivalent code generated by IBB.

```
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    A[i][j] = B[i][j];
for (i=2;i<N;i+=2)
  for (j=0;j<N;j++)
    C[i][j] = A[i-1][j] + A[i-2][j];
```

Listing 9.37 – *Unroll and Jam - Initial Code.*

```
xfor (io=0,i1=2 ; io<N,i1<N ; io++, i1+=2 ; 1,2 ; 0,1)
xfor (jo=0,j1=0 ; jo<N,j1<N ; jo++, j1++  ; 1,1 ; 0,0) {
  0: A[io][jo]    = B[io][jo];
  1: C[i1][j1] =  A[i1-1][j1] + A[i1-2][j1]; }
```

Listing 9.38 – *Unroll and Jam - XFOR First Version (io,jo) Unrolled.*

```
xfor (io=0,i1=0,i2=0 ; io<=N-1,i1<=N-1,i2<=N-3 ;
      io+=2,i1+=2,i2+=2 ; 1,1,1 ; 0,0,0) {
xfor (jo=0,j1=0,j2=0 ; jo<=N-1,j1<=N-1,j2<=N-1 ;
      jo++,j1++,j2++ ; 1,1,1 ; 0,N,N) {
    0: A[io][jo]=B[io][jo];
    1: A[i1+1][j1]=B[i1+1][j1];
    2: C[i2+2][j2]=A[i2+2-1][j2]+A[i2+2-2][j2]; } }
```

Listing 9.39 – *Unroll and Jam - XFOR Second Version (io,jo) Unrolled.*

```
xfor (io=0,i1=0,i2=0 ; io<=N-1,i1<=N-1,i2<=N-3 ;
      io+=2,i1+=2,i2+=2 ; 1,1,1 ; 0,0,0) {
xfor (jo=0,j1=0,j2=0 ; jo<=N-1,j1<=N-1,j2<=N-1 ;
      jo++,j1++,j2++ ; 1,1,1 ; 0,0,0) {
    0: A[io][jo]=B[io][jo];
    1: A[i1+1][j1]=B[i1+1][j1];
    2: C[i2+2][j2]=A[i2+2-1][j2]+A[i2+2-2][j2]; } }
```

Listing 9.40 – *Unroll and Jam - XFOR Third Version (io,jo) Unrolled + Jammed and Fused with (i1,j1).*

```
if (N>=1) {
  for (i=0 ; i<=floord(N-3,2) ; i++) {
    for (j=0; j<=N-1 ; j++) {
      A[2*i][j]   = B[2*i][j];
      A[2*i+1][j] = B[2*i+1][j];
      C[2*i+2][j] = A[2*i+1][j]+A[2*i][j];
  } }
  i =floord (N-1,2);
  for (j=0 ; j<=N-1 ; j++) {
    A[2*i][j]   = B[2*i][j];
    A[2*i+1][j] = B[2*i+1][j];
} }
```

Listing 9.41 – *Unroll and Jam - IBB Generated Code.*

9.2.11 Loop Strip-Mining

Strip-mining is a method for adjusting the granularity of an operation by splitting a single loop (generally the innermost loop) into a nested loop. The resulting inner loop iterates over a chunks of constant size of the original loop, and the new outer loop runs the inner loop enough times to cover all the strips, achieving the necessary total number of iterations. The number of iterations of the inner loop is known as the loop's strip length. Strip mining helps to ensure that the data used in a loop stays in the cache until it is reused, and promotes vectorization. The partitioning of the loop iteration space leads to the partitioning of a large array into smaller blocks, thus fitting accessed array elements into the cache size, enhancing cache reuse and eliminating cache size requirements. This transformation is still applicable with the XFOR construct and it is applied exactly in the same manner like in the case of for-loops.

9.2.12 Loop Tiling

This transformation is also known as loop blocking. It is a loop optimization used by compilers to make the execution of certain types of loops more efficient. Loop tiling is the multidimensional generalization of strip mining. It consists in breaking the loops into "tiles" or blocks. Loop tiling is often used in conjunction with loop skewing, loop reversal and loop interchange, and used for program parallelization and for improving spacial and temporal locality. Loop tiling is still applicable with the XFOR construct and it is applied exactly as to for-loops.

9.3 Flexible Transformation Composition

In addition to the ease of applying loop transformations, XFOR loops makes the composition of transformations easier and intuitive. A loop dilatation can be implemented simply by setting the corresponding grain to the stretching coefficient. This will unroll the other indices i_k by a factor equal to : $lcm(grain_j)/grain_k$. Where $lcm(grain_j)$ is the *least common multiple* of all the grains that appear in the XFOR header and $grain_k$ is the grain corresponding to index i_k. The rest of the transformations can be applied by setting the offset expression as described below.

9.3.1 Generalization

Without any loss of generality, consider the case of 2-depth XFOR loop. An affine transformation: $((i,j) \rightarrow (ai+bj+c, a'i+b'j+c'))$ may be expressed using the following offsets:

```
1 xfor (i=... ; ... ; (a-1)*i+b*j+c, ...)
2 xfor (j=... ; ... ; a'*i+(b'-1)*j+c', ...)
```

More generally, consider an iteration vector I of dimension n. A transformation t of an index $i_j, 0 \leq j < n$:

$$t(i_j) = \sum_{k=0}^{j-1} \alpha_k i_k + \alpha_j i_j + \sum_{k=j+1}^{n-1} \alpha_k i_k$$

is applied by setting the offset of index i_j to this expression:

$$\sum_{k=0}^{j-1} \alpha_k i_k + (\alpha_j - 1) i_j + \sum_{k=j+1}^{n-1} \alpha_k i_k$$

9.4 XFORGEN: XFOR Generator from OpenScop Description

In order to generalize the use of XFOR loops as a Polyhedral Intermediate Representation of loop transformations, we have developed a software tool that generates equivalent XFOR loops from the resulting OpenScop generated by automatic loop optimizers. This software is called XFORGEN. As automatic locality optimizer, we consider Pluto [6, 16] which is the most well-known source-to-source polyhedral compiler. Pluto implements some of the most advanced loop optimizing strategies and transformations.

In this section, we start by introducing XFORGEN and give a detailed description of the XFOR loops generation. Then, we show that XFOR loops is a well-suited Intermediate Representation (IR) for loop transformations by comparing Pluto-generated codes to the corresponding XFORGEN-generated codes.

License: XFORGEN is released under the GNU General Public License, version 2 (GPL v2) [67].

9.4.1 Using the XFORGEN Software

Calling XFORGEN: XFORGEN can be invoked by the following command:

```
xforgen [ options ] input_file
```

The default behavior of XFORGEN is to read the input file and to print the generated XFOR code on the standard output. The input file may be a for/xfor loop based source code or an OpenScop file. Depending on the type of the input, a specific option should be used.

XFORGEN Options:

1. -o <output>: This option sets the output file name.

2. -openscop: This option must be used in the case of an OpenScop file as input.

3. **-pluto:** This option asks XFORGEN to invoke the locality optimizer Pluto for loop optimizations. Then, from the Pluto-generated OpenScop, it generates the XFOR-loop nests. In addition to that, the user can still set Pluto options.

4. **-normalize:** This option asks XFORGEN to normalize the loop bounds. It is mandatory when scanning a code with xfor loop nests.

4. **-h:** This option asks XFORGEN to Display a short help.

9.4.2 XFOR Code Generation

The functioning of XFORGEN is described in Figure 9.8. XFORGEN takes as input a file; the file may be an OpenScop description, or a for/xfor-loop based program. First, the input file is parsed and the loop nests to optimize are represented using the Scop data structure. After that, the user can chose to call the automatic optimizer Pluto or not. If the user wants to invoke Pluto, the first step is to set Pluto options, then Pluto is invoked for rescheduling the initial Scop. Third, the modified Scop is scanned in order to generate an abstract syntax tree (AST) for the equivalent XFOR loop nest. Finally, the XFOR code is generated from this intermediate structure.

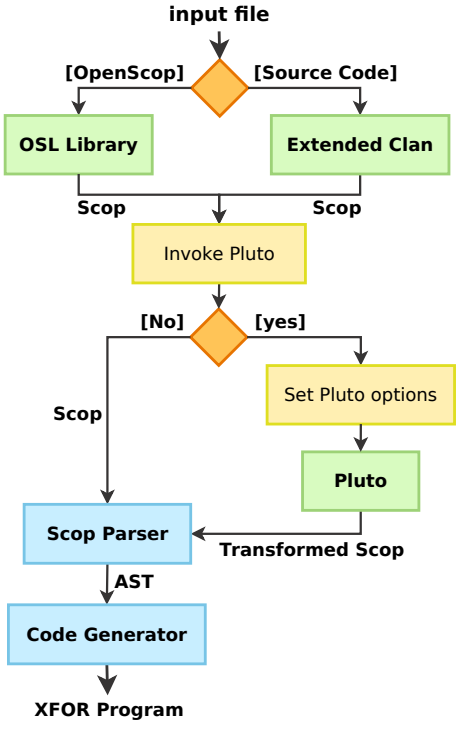


Figure 9.8 – *XFORGEN Functioning.*

Data Structures. During the process of XFOR code generation, the parameters of an XFOR are saved in an intermediate structure. An XFOR-loop nest is represented by a single-linked list. One node of the list has the following structure:

```
typedef struct xfor_loop * xfor_loop_p;
struct xfor_loop
{   index_p     indices;
    xfor_loop_p inner_loop;
};
```

Where **indices** is a linked list that involves the indices that belong to the same XFOR (*i.e.* appear in the same XFOR header). One node of this list is called `index`. It holds all the information relative to an index. The `index` structure is defined below:

```
typedef struct index * index_p;
struct index
{   char*   name;
    char*   lower_b;
    char*   relop;
    char*   cond;
    char*   stride;
    int     grain;
    char*   offset;
    index_p next;
};
```

The instructions are saved in a single-linked list in the order defined by their schedule. Each element of the linked list points to the following structure:

```
typedef struct instruction * instruction_p;
struct instruction
{   char*         stmt;
    int           weight;
    int           rank;
    instruction_p next;
};
```

Where, `stmt` is the statement expression, `rank` is the rank of the corresponding index in the XFOR header and `weight` is an integer that implies the scheduling of the corresponding statement. It is computed using the β-vector (the constant dimensions) of the Scattering relation.

Scop Parser. Figure 9.9 details the XFOR structure generation. The first step consists in eliminating the union domain from the Scop structure; each statement composed of a union of relations is decomposed in as much statements as the number of relations in the union. Second, the new Scop structure is parsed. For each statement, the following operations are performed:

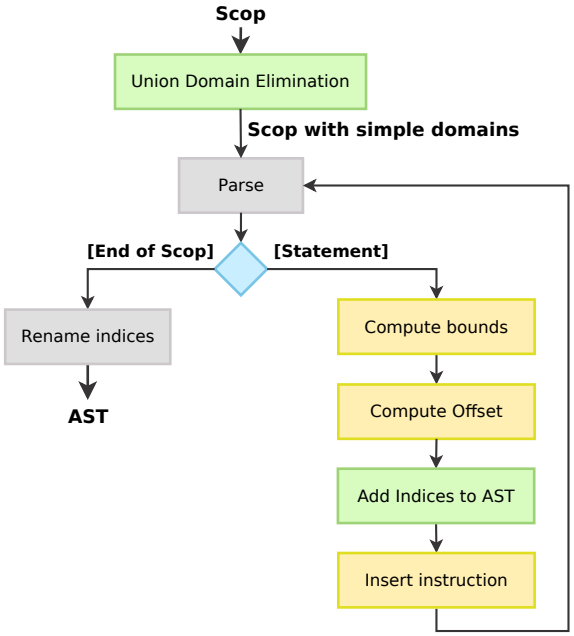


Figure 9.9 – *Scop Parser and XFOR Loops Generation.*

1. Read the `Domain` relation and compute the loop nest bounds. Here, not just one index is computed, but the index and all the nested indices. This means that at each step, one column is added to the `xfor_loop` structure (*i.e* list of indices having the same rank in the XFOR nested headers).

2. Read the `Scattering` relation and compute the offset for each index of the nest.

3. Add the new indices to the AST representation of the XFOR loop nest. In order to have at the end a perfectly nested XFOR loop, fictive indices with just one iteration are inserted. Two cases are possible:

   - If the depth of the extracted nest is lower than the current depth of the XFOR structure, the new indices are inserted at the end of the index list, then fictive indices are added in a way that the current column has the same depth as the previous one (see Figure 9.10, Case 1).
   - When the depth of the extracted nest is higher than the current depth of the XFOR structure, first, the new loop nest is inserted. Suppose that the difference between both depths is equal to d. Then, for each level l ($current\_depth + 1 \leq l \leq current\_depth + d$), we create a new XFOR loop having $nb\_indices - 1$ fictive indices performing just one iteration and add it to the XFOR structure (see Figure 9.10, Case 2).

4. Insert the corresponding instruction to the instruction list. The position of the instruction in the final XFOR nest body is defined by the constant output dimensions of the `Scattering` relation (their weighted average is computed and saved in the `weight` variable).

5. Once all statements have been scanned, the indices are renamed in the final AST, so, there will be no duplicated index in the XFOR header.

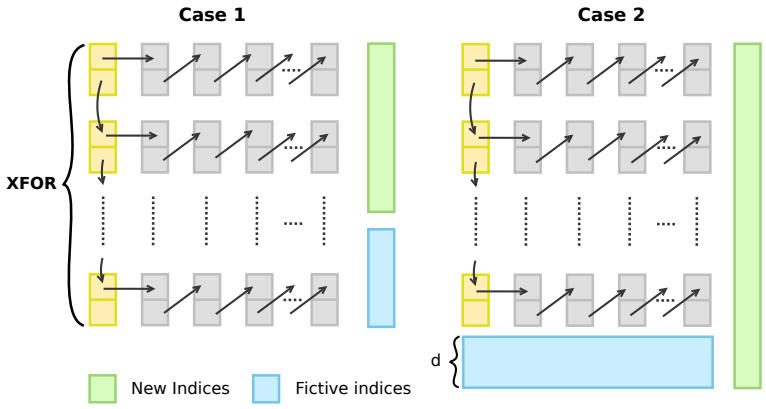


Figure 9.10 – *XFORGEN: New Loop Insertion.*

9.4.3 Qualitative Comparison between Pluto Generated and XFORGEN Generated Codes

In this section, we show the improvements in term of expressiveness and readability that XFOR loops provide when compared to the Pluto generated codes. XFORGEN generated loops are much easier to understand and by the way easier to maintain and modify. Additionally, they help the programmer to understand the transformations applied by Pluto. On the other hand, Pluto generated codes are very long and complicated and it is impossible for a programmer to modify them.

Example 1. Listing 9.42 represents the loop kernel of the code `fdtd-2d` from the Polybench benchmarks suite [27]. It consists of a succession of four loop nests; the first nest computes the first column of matrix *ey*. The second updates the rest of the columns of matrix *ey* using matrix *hz*. Then, the third nest computes the elements of matrix *ex* using elements of matrix *hz*. Finally, the last nest updates matrix *hz* using matrices *ey* and *ex*.

```
#pragma scop
for (j = 0 ; j < ny ; j++)
  ey[0][j] = _fict_[t];
for (i = 1 ; i < nx ; i++)
  for (j = 0 ; j < ny ; j++)
    ey[i][j]=ey[i][j]-0.5*(hz[i][j]-hz[i-1][j]);
for (i = 0 ; i < nx ; i++)
  for (j = 1 ; j < ny ; j++)
    ex[i][j]=ex[i][j]-0.5*(hz[i][j]-hz[i][j-1]);
for (i = 0 ; i < nx-1 ; i++)
  for (j = 0 ; j < ny-1 ; j++)
    hz[i][j] = hz[i][j]-0.7*(ex[i][j+1]-ex[i][j]+ey[i+1][j]-ey[i][j]);
#pragma endscop
```

Listing 9.42 – `fdtd-2d`*: Original Code.*

We invoke Pluto for optimizing this code using `-noprevector -maxfuse` options. The Pluto generated code in represented in Listing 9.43 and the XFOR equivalent code generated by XFORGEN is displayed in Listing 9.44. As one can see, editing Pluto generated code is almost like rewriting it by hand. But, in the XFOR version, the programmer just has to set the convenient values for the offset and our source-to-source compiler IBB takes care of the code generation.

```
if (ny >= 1) {
  if (nx >= 2) {
    ey[0][0]=_fict_[t];
    for (t2=1;t2<=nx-1;t2++) {
      ey[t2][0]=ey[t2][0]-0.5*(hz[t2][0]-hz[t2-1][0]);
    }
  }
  if (nx == 1) {
    ey[0][0] = _fict_[t];
  }
  if (nx <= 0) {
    for (t1=0;t1<=ny-1;t1++) {
      ey[0][t1]=_fict_[t];
    }
  }
  if (nx >= 2) {
    for (t1=1;t1<=ny-1;t1++) {
      ex[0][t1]=ex[0][t1]-0.5*(hz[0][t1]-hz[0][t1-1]);
      ey[0][t1]=_fict_[t];
      for (t2=1;t2<=nx-1;t2++) {
        ey[t2][t1]=ey[t2][t1]-0.5*(hz[t2][t1]-hz[t2-1][t1]);
        ex[t2][t1]=ex[t2][t1]-0.5*(hz[t2][t1]-hz[t2][t1-1]);
        hz[(t2-1)][(t1-1)]=hz[(t2-1)][(t1-1)]-0.7*(ex[(t2-1)][(t1-1)+1]
        -ex[(t2-1)][(t1-1)]+ey[(t2-1)+1][(t1-1)]-ey[(t2-1)][(t1-1)]);
      }
    }
  }
  if (nx == 1) {
    for (t1=1;t1<=ny-1;t1++) {
      ey[0][t1]=_fict_[t];
      ex[0][t1]=ex[0][t1]-0.5*(hz[0][t1]-hz[0][t1-1]);
    }
  }
}
```

Listing 9.43 – `fdtd-2d`: *Pluto Generated Code Using* `-noprevector -maxfuse` *Options.*

```
xfor (jo=0,i1=1,i2=0,i3=0 ; jo<=ny-1,i1<=nx-1,i2<=nx-1,i3<=nx-2 ;
      jo++,i1++,i2++,i3++ ; 1,1,1,1 ; 0,-i1+j1,-i2+j2,-i3+j3+1)
xfor (f20=0,j1=0,j2=1,j3=0 ; f20<1,j1<=ny-1,j2<=ny-1,j3<=ny-2 ;
      f20++,j1++,j2++,j3++ ; 1,1,1,1 ; 0,i1-j1,i2-j2,i3-j3+1) {
  0: ey[0][jo]=_fict_[t];
  1: ey[i1][j1]=ey[i1][j1]-0.5*(hz[i1][j1]-hz[i1-1][j1]);
  2: ex[i2][j2]=ex[i2][j2]-0.5*(hz[i2][j2]-hz[i2][j2-1]);
  3: hz[i3][j3]=hz[i3][j3]-0.7*(ex[i3][j3+1]-ex[i3][j3]
                +ey[i3+1][j3]-ey[i3][j3]);
}
```

Listing 9.44 – `fdtd-2d`: *XFOR Equivalent Code.*

The Pluto generated code is **34**-lines long, while the XFOR code is composed of **10** lines only, which is about **70%** less. In addition to that, the XFOR construct gives the programmer a clear idea about the transformations applied by Pluto. One can see easily that all loops were fused. In order to respect dependencies, the last loop nest was shifted by 1. In addition to that, all loops, except the first one, were interchanged for minimizing the data reuse distances.

Example 2. Consider the `2mm` kernel from the Polybench benchmarks [27] in Listing 9.45. It is a succession of two loop nests computing two matrix multiplications ($E = A \times B \times D$). We invoked Pluto for optimizing this code using these options: `-noprevector -tile -intratileopt -maxfuse -smartfuse`. The Pluto generated code is shown in Listing 9.46 and the XFOR equivalent code generated by XFORGEN is displayed in Listing 9.47.

```
#pragma scop
for (i = 0; i < ni; i++)
  for (j = 0; j < nj; j++) {
    C[i][j] = 0;
    for (k = 0; k < nk; ++k)
      C[i][j] += A[i][k] * B[k][j];
  }
for (i = 0; i < ni; i++)
  for (j = 0; j < nl; j++) {
    E[i][j] = 0;
    for (k = 0; k < nj; ++k)
      E[i][j] += C[i][k] * D[k][j];
  }
#pragma endscop
```

Listing 9.45 – `2mm`: *Original Code.*

```
if (ni >= 1) {
  for (t2=0;t2<=floord(ni-1,32);t2++) {
    if ((nj >= 0) && (nl >= 0)) {
      for (t3=0;t3<=floord(nj+nl-1,32);t3++) {
        if ((nj >= nl+1) && (t3 <= floord(nl-1,32)) && (t3 >= ceild(nl-31,32))) {
          for (t4=32*t2;t4<=min(ni-1,32*t2+31);t4++) {
            for (t5=32*t3;t5<=nl-1;t5++) {
              E[t4][t5] = 0;
              C[t4][t5] = 0;
            }
            for (t5=nl;t5<=min(nj-1,32*t3+31);t5++) {
              C[t4][t5] = 0;
            }
          }
        }
        if ((nj >= nl+1) && (t3 <= floord(nl-32,32))) {
          for (t4=32*t2;t4<=min(ni-1,32*t2+31);t4++) {
            for (t5=32*t3;t5<=32*t3+31;t5++) {
              E[t4][t5] = 0;
              C[t4][t5] = 0;
            }
          }
        }
        if ((nj <= nl-1) && (t3 <= floord(nj-1,32)) && (t3 >= ceild(nj-31,32))) {
          for (t4=32*t2;t4<=min(ni-1,32*t2+31);t4++) {
            for (t5=32*t3;t5<=nj-1;t5++) {
              E[t4][t5] = 0;
              C[t4][t5] = 0;
            }
```

```
30            for (t5=nj;t5<=min(nl-1,32*t3+31);t5++) {
31              E[t4][t5] = 0;
32            }
33          }
34        }
35        if ((nj <= nl-1) && (t3 <= floord(nj-32,32))) {
36          for (t4=32*t2;t4<=min(ni-1,32*t2+31);t4++) {
37            for (t5=32*t3;t5<=32*t3+31;t5++) {
38              E[t4][t5] = 0;
39              C[t4][t5] = 0;
40            }
41          }
42        }
43        if ((nj == nl) && (t3 <= floord(nj-1,32))) {
44          for (t4=32*t2;t4<=min(ni-1,32*t2+31);t4++) {
45            for (t5=32*t3;t5<=min(nj-1,32*t3+31);t5++) {
46              E[t4][t5] = 0;
47              C[t4][t5] = 0;
48            }
49          }
50        }
51        if ((t3 <= floord(nj-1,32)) && (t3 >= ceild(nl,32))) {
52          for (t4=32*t2;t4<=min(ni-1,32*t2+31);t4++) {
53            for (t5=32*t3;t5<=min(nj-1,32*t3+31);t5++) {
54              C[t4][t5] = 0;
55            }
56          }
57        }
58        if ((t3 <= floord(nl-1,32)) && (t3 >= ceild(nj,32))) {
59          for (t4=32*t2;t4<=min(ni-1,32*t2+31);t4++) {
60            for (t5=32*t3;t5<=min(nl-1,32*t3+31);t5++) {
61              E[t4][t5] = 0;
62            }
63          }
64        }
65      }
66    }
67    if (nl <= -1) {
68      for (t3=0;t3<=floord(nj-1,32);t3++) {
69        for (t4=32*t2;t4<=min(ni-1,32*t2+31);t4++) {
70          for (t5=32*t3;t5<=min(nj-1,32*t3+31);t5++) {
71            C[t4][t5] = 0;
72          }
73        }
74      }
75    }
76    if (nj <= -1) {
77      for (t3=0;t3<=floord(nl-1,32);t3++) {
78        for (t4=32*t2;t4<=min(ni-1,32*t2+31);t4++) {
79          for (t5=32*t3;t5<=min(nl-1,32*t3+31);t5++) {
80            E[t4][t5] = 0;
81          }
82        }
83      }
84    }
85  }
86  if (nj >= 1) {
87    for (t2=0;t2<=floord(ni-1,32);t2++) {
88      for (t3=0;t3<=floord(nj-1,32);t3++) {
89        if (nk >= 1) {
90          for (t5=0;t5<=floord(nk-1,32);t5++) {
91            for (t6=32*t2;t6<=min(ni-1,32*t2+31);t6++) {
92              for (t7=32*t3;t7<=min(nj-1,32*t3+31);t7++) {
93                for (t9=32*t5;t9<=min(nk-1,32*t5+31);t9++) {
94                  C[t6][t7] += A[t6][t9] * B[t9][t7];
95                }
96              }
97            }
98          }
99        }
100       if (nl >= 1) {
101         for (t5=0;t5<=floord(nl-1,32);t5++) {
102           for (t6=32*t2;t6<=min(ni-1,32*t2+31);t6++) {
103             for (t7=32*t3;t7<=min(nj-1,32*t3+31);t7++) {
104               for (t9=32*t5;t9<=min(nl-1,32*t5+31);t9++) {
105                 E[t6][t9] += C[t6][t7] * D[t7][t9];
106               }
107             }
108           }
109         }
```

```
110         }
111       }
112     }
113   }
114 }
```

Listing 9.46 – 2mm: *Pluto Generated Code Using* `-noprevector -tile -intratileopt -maxfuse -smartfuse` *Options.*

The Pluto generated code is **114**-lines long, while the XFOR code is composed of **39** lines, which is about **65%** less. In addition to that, the XFOR construct gives the programmer a clear idea about the transformations applied by Pluto. One can see that all loops were tiled and, in the second loop nest, *j* and *k* indices were interchanged.

```
1  xfor(fk00 = 0,              fk01 = 0,              fk02 = 0,              fk03 = 0;
2       fk00 <=floord(ni-1,32), fk01 <=floord(ni-1,32), fk02 <=floord(ni-1,32), fk03 <=floord(ni-1,32);
3       fk00 ++,               fk01 ++,               fk02 ++,               fk03 ++;
4       1,                     1,                     1,                     1;
5       0,                     1+floord(ni-132),      0,                     1+floord(ni-132)){
6  xfor(fk10 = 0,              fk11 = 0,              fk12 = 0,              fk13 = 0;
7       fk10 <=floord(nj-1,32), fk11 <=floord(nj-1,32), fk12 <=floord(nl-1,32), fk13 <=floord(nj-1,32);
8       fk10 ++,               fk11 ++,               fk12 ++,               fk13 ++;
9       1,                     1,                     1,                     1;
10      0,                     0,                     0,                     0){
11 xfor(io = 32*fk00,             fk21 = 0,              i2 = 32*fk02,             fk23 = 0;
12      io<=min(32*fk00+31,ni-1), fk21 <=floord(nk-1,32), i2<=min(32*fk02+31,ni-1), fk23 <=floord(nl-1,32);
13      io ++,                    fk21 ++,               i2 ++,                    fk23 ++;
14      1,                        1,                     1,                        1;
15      0,                        0,                     0,                        1+floord(nk-132)){
16 xfor(jo = 32*fk10,             i1 = 32*fk01,             j2 = 32*fk12,             i3 = 32*fk03;
17      jo<=min(32*fk10+31,nj-1), i1<=min(32*fk01+31,ni-1), j2<=min(32*fk12+31,nl-1), i3<=min(32*fk03+31,ni-1);
18      jo ++,                    i1 ++,                    j2 ++,                    i3 ++;
19      1,                        1,                        1,                        1;
20      0,                        0,                        0,                        0){
21 xfor(f50 = 0,j1 = 32*fk11,             f52 = 0,j3 = 32*fk23;
22      f50<1,  j1<=min(32*fk11+31,nj-1), f52<1,  j3<=min(32*fk23+31,nl-1);
23      f50++,  j1++,                     f52++,  j3++;
24      1,      1,                        1,      1;
25      1,      0,                        0,      -j3+k3){
26 xfor(f60 = 0,k1 = 32*fk21,             f62 = 0,k3 = 32*fk13;
27      f60<1,  k1<=min(32*fk21+31,nk-1), f62<1,  k3<=min(32*fk13+31,nj-1);
28      f60++,  k1++,                     f62++,  k3++;
29      1,      1,                        1,      1;
30      0,      0,                        0,      j3-k3){
31              2: E[i2][j2] = 0;
32              0: C[io][jo] = 0;
33              1: C[i1][j1] += A[i1][k1] * B[k1][j1];
34              3: E[i3][j3] += C[i3][k3] * D[k3][j3];
35          }
36        }
37      }
38    }
39  }
40 }
```

Listing 9.47 – 2mm: *XFOR Code Equivalent to Pluto Code.*

As one can see in Listing 9.46, editing Pluto generated code is out of reach and it consists almost in rewriting the totality of the code by hand. But, in the XFOR version, the programmer just has to set the convenient values for the offset and our source-to-source compiler IBB will take care of the code generation. The programmer can, also, refer to our software tool XFOR-WIZARD (see chapter

7 ([*The XFOR Programming Environment XFOR-WIZARD*], page 97).) in order to verify the validity of the new offset expressions.

9.5 XFORGEN: Enhancing Pluto Optimized Codes

In this Section, we show how XFORGEN helps the programmer to improve Pluto generated codes. We illustrate this strategy using some examples selected from the Polybench benchmarks suite [27].

9.5.1 Example 1

Listing 9.48 displays `jacobi-1d` code. It is composed of a succession of two loop nests; the first one performs a stencil computation on matrix A and stores the result on matrix B and the second nest updates A matrix using B. The Pluto optimized version of this code (using `-maxfuse` option) is represented on Listing 9.49. In this code, the two loop nests were fused and the first nest was shifted by 1 in order to respect dependencies. The equivalent XFOR version is shown in Listing 9.50.

```
for (i=2 ; i<n-1 ; i++)
  B[i]=0.33333*(A[i-1]+A[i]+A[i+1]) ;
for (j=2 ; j<n-1 ; j++)
  A[j]=B[j] ;
```

Listing 9.48 – *Original Code of* `jacobi-1d`.

```
B[2]=0.33333*(A[1]+A[2]+A[3]) ;
for (t1=3 ; t1<=n-2 ; t1++) {
  B[t1]=0.33333*(A[t1-1]+A[t1]+A[t1+1]) ;
  A[t1-1]=B[t1-1] ; }
A[n-2]=B[n-2] ;
```

Listing 9.49 – *Pluto Version of* `jacobi-1d`*: Reuse Distance = 1.*

```
xfor (io=2,j1=2 ; io<=n-2,j1<=n-2 ; io++,j1++ ; 1,1 ; 0,1) {
  0: B[io] = 0.33333 * (A[io-1] + A[io] + A[io + 1]);
  1: A[j1] = B[j1];
}
```

Listing 9.50 – *XFORGEN Generated code: XFOR Equivalent Version of* `jacobi-1d`*: Reuse Distance = 1.*

The very short inter-statement reuse distance prevents the processor from vectorizing the code. Hence, it is recommended to increase the offset value in order to relax the dependencies. It was shown in Chapter 8 ([*XFOR and Automatic Optimizers*], page 115), that fixing the offset value to 9 allows the best performance for our target processor. The optimized XFOR code is shown in Listing 9.51 and the corresponding IBB generated code is displayed in Listing 9.52. This

modification yields a speedup of 1.5× over the Pluto code, when compiled using *GCC4.8.1*.

```
xfor (io=2,j1=2 ; io<=n-2,j1<=n-2 ; io++,j1++ ; 1,1 ; 0,9) {
  0: B[io] = 0.33333 * (A[io-1] + A[io] + A[io + 1]);
  1: A[j1] = B[j1];
}
```

Listing 9.51 – *XFOR Improved Version of* `jacobi-1d`*: Reuse Distance = 9.*

```
if (n>=4) {
  for (i=0 ; i<=min(8,n-4) ; i++)
    B[i+2] = 0.33333*(A[i+1]+A[i+2]+A[i+3]);
  for (i=9 ; i<=n-4 ; i++) {
    B[i+2] = 0.33333*(A[i+1]+A[i+2]+A[i+3]);
    A[i-7] = B[i-7];
  }
  for (i=max(9,n-3) ; i<=n+5 ; i++)
    A[i-7] = B[i-7];
}
```

Listing 9.52 – *IBB Generated Code of* `jacobi-1d`*: Reuse Distance = 9.*

9.5.2 Example 2

In `mvt` (see Listing 9.53), from the first to the second loop nest, only array A is reused. Data-reuse distances between both iteration domains are large, since array A is accessed in row-major order in the first nest, and in column-major order in the second nest, which is also an unfavorable access order regarding inter-statement data locality. Since no data dependencies prevent loop transformations, interchanging loops i and j in the second nest is obviously beneficial. Finally, overlapping both resulting iteration domains optimizes inter-statement data reuse distances and provides a significant speed-up.

```
for (i=0 ; i<n ; i++)
  for (j=0 ; j<n ; j++)
    x1[i]=x1[i]+A[i][j]*y_1[j] ;
for (i=0 ; i<n ; i++)
  for (j=0 ; j<n ; j++)
    x2[i]=x2[i]+A[j][i]*y_2[j] ;
```

Listing 9.53 – `mvt`*: Original Code.*

```
if (n >= 1) {
  for (t1=0;t1<=n-1;t1++) {
    for (t2=0;t2<=n-1;t2++) {
      x1[t1]=x1[t1]+A[t1][t2]*y_1[t2] ;;
      x2[t1]=x2[t1]+A[t2][t1]*y_2[t2] ;;
} } }
```

Listing 9.54 – `mvt`*: Pluto Optimized Version.*

Listing 9.54 shows the best Pluto version resulting from merging both loop nests (using `-maxfuse` options), however without interchanging loops of the second nest. The corresponding XFOR equivalent code is represented in Listing 9.55. An improvement of this version consists in interchanging indices $i1$ and $j1$. This modification yields a speedup of 3.94× over the Pluto code, when compiled using *GCC*4.8.1.

```
xfor (io = 0,  i1 = 0 ;
      io<=n-1, i1<=n-1 ;
      io++,    i1++ ;
      1,       1 ;  /* grain */
      0,       0)   /* offset */ {
  xfor (jo = 0,  j1 = 0 ;
        jo<=n-1, j1<=n-1 ;
        jo++,    j1++ ;
        1,       1 ;  /* grain */
        0,       0)   /* offset */ {
    0: x1[io]=x1[io]+A[io][jo]*y_1[jo] ;
    1: x2[i1]=x2[i1]+A[j1][i1]*y_2[j1] ;
  }
}
```

Listing 9.55 – `mvt`: *XFOR Equivalent Code.*

```
xfor (io = 0,  i1 = 0 ;
      io<=n-1, i1<=n-1 ;
      io++,    i1++ ;
      1,       1 ;  /* grain */
      0,       j1-i1)   /* offset */ {
  xfor (jo = 0,  j1 = 0 ;
        jo<=n-1, j1<=n-1 ;
        jo++,    j1++ ;
        1,       1 ;  /* grain */
        0,       i1-j1)   /* offset */ {
    0: x1[io]=x1[io]+A[io][jo]*y_1[jo] ;
    1: x2[i1]=x2[i1]+A[j1][i1]*y_2[j1] ;
  }
}
```

Listing 9.56 – `mvt`: *XFOR Enhaced Version ($i1$ and $j1$ interchanged).*

```
if (n>=1) {
  for (_mfr_refo=0; _mfr_refo<=n-1; _mfr_refo++) {
    for (_mfr_ref1=0; _mfr_ref1<=n-1; _mfr_ref1++) {
      x1[_mfr_refo]=x1[_mfr_refo]+A[_mfr_refo][_mfr_ref1]*y_1[_mfr_ref1];
      x2[_mfr_ref1]=x2[_mfr_ref1]+A[_mfr_refo][_mfr_ref1]*y_2[_mfr_refo];
    }
  }
}
```

Listing 9.57 – `mvt`: *IBB Generated Code.*

9.5.3 Example 3

Let us consider the `3mm` loop kernel represented in listing 9.58. This code computes three matrices multiplications $G = (A \times B) \times (C \times D)$.

```
for (i = 0; i < ni; i++)    //* E := A*B *
  for (j = 0; j < nj; j++) {
    E[i][j] = 0;
    for (k = 0; k < nk; ++k)
      E[i][j] += A[i][k] * B[k][j];
  }
for (i = 0; i < nj; i++)    //* F := C*D *
  for (j = 0; j < nl; j++) {
    F[i][j] = 0;
    for (k = 0; k < nm; ++k)
      F[i][j] += C[i][k] * D[k][j];
  }
for (i = 0; i < ni; i++)    //* G := E*F *
  for (j = 0; j < nl; j++) {
    G[i][j] = 0;
    for (k = 0; k < nj; ++k)
      G[i][j] += E[i][k] * F[k][j];
  }
```

Listing 9.58 – `3mm`: *Original Code.*

The optimized code version generated by Pluto (using option `-smartfuse`) is displayed in Listing 9.59, and the XFOR equivalent code in shown in Listing 9.59.

```
if (nl >= 1) {
  for (t2=0;t2<=min(ni-1,nj-1);t2++) {
    for (t3=0;t3<=nl-1;t3++) {
      F[t2][t3] = 0;
      G[t2][t3] = 0;
    }
  }
}
if (nl >= 1) {
  for (t2=max(0,ni);t2<=nj-1;t2++) {
    for (t3=0;t3<=nl-1;t3++) {
      F[t2][t3] = 0;
    }
  }
}
if (nl >= 1) {
  for (t2=max(0,nj);t2<=ni-1;t2++) {
    for (t3=0;t3<=nl-1;t3++) {
      G[t2][t3] = 0;
    }
  }
}
if ((nl >= 1) && (nm >= 1)) {
  for (t2=0;t2<=nj-1;t2++) {
    for (t3=0;t3<=nl-1;t3++) {
      for (t5=0;t5<=nm-1;t5++) {
        F[t2][t3] += C[t2][t5] * D[t5][t3];
```

```
      }
    }
  }
}
if (nj >= 1) {
  for (t2=0; t2<=ni-1; t2++) {
    for (t3=0; t3<=nj-1; t3++) {
      E[t2][t3] = 0;
    }
  }
}
if ((nj >= 1) && (nk >= 1) && (nl >= 1)) {
  for (t2=0; t2<=ni-1; t2++) {
    for (t3=0; t3<=nj-1; t3++) {
      for (t5=0; t5<=nk-1; t5++) {
        E[t2][t3] += A[t2][t5] * B[t5][t3];
      }
      for (t5=0; t5<=nl-1; t5++) {
        G[t2][t5] += E[t2][t3] * F[t3][t5];
      }
    }
  }
}
if ((nj >= 1) && (nk >= 1) && (nl <= 0)) {
  for (t2=0; t2<=ni-1; t2++) {
    for (t3=0; t3<=nj-1; t3++) {
      for (t5=0; t5<=nk-1; t5++) {
        E[t2][t3] += A[t2][t5] * B[t5][t3];
      }
    }
  }
}
if ((nj >= 1) && (nk <= 0) && (nl >= 1)) {
  for (t2=0; t2<=ni-1; t2++) {
    for (t3=0; t3<=nj-1; t3++) {
      for (t5=0; t5<=nl-1; t5++) {
        G[t2][t5] += E[t2][t3] * F[t3][t5];
      }
    }
} }
```

Listing 9.59 – `3mm`: *Pluto Optimized Version.*

```
xfor (i0 = 0,    i1 = 0,    i2 = 0,    i3 = 0,    i4 = 0,    i5 = 0 ;
      i0<=ni-1, i1<=ni-1, i2<=nj-1, i3<=nj-1, i4<=ni-1, i5<=ni-1 ;
      i0++,      i1++,      i2++,      i3++,      i4++,      i5++ ;
      1,         1,         1,         1,         1,         1 ;
      2*nj,      2*nj+ni,   0,         nj,        0,         2*nj+ni) {
xfor (j0 = 0,    j1 = 0,    j2 = 0,    j3 = 0,    j4 = 0,    j5 = 0 ;
      j0<=nj-1, j1<=nj-1, j2<=nl-1, j3<=nl-1, j4<=nl-1, j5<=nl-1 ;
      j0++,      j1++,      j2++,      j3++,      j4++,      j5++ ;
      1,         1,         1,         1,         1,         1 ;
      0,         0,         0,         0,         0,         -j5+k5) {
xfor (f30 = 0, k1 = 0,    f32 = 0, k3 = 0,    f34 = 0, k5 = 0 ;
      f30<1,    k1<=nk-1, f32<1,    k3<=nm-1, f34<1,    k5<=nj-1 ;
      f30++,    k1++,      f32++,    k3++,      f34++,    k5++ ;
       1,         1,         1,         1,         1,         1 ;
       0,         0,         0,         0,         0,         j5-k5+nk) {
    2: F[i2][j2] = 0;
```

```
17      4: G[i4][j4] = 0;
18      3: F[i3][j3] += C[i3][k3] * D[k3][j3];
19      0: E[io][jo] = 0;
20      1: E[i1][j1] += A[i1][k1] * B[k1][j1];
21      5: G[i5][j5] += E[i5][k5] * F[k5][j5];
22  } } }
```

Listing 9.60 – 3mm: *XFOR Equivalent Code.*

As one can notice in the XFOR equivalent code, Pluto prefers to fuse the statement 0 and 2. In order to minimize the inter-statement data reuse distances between statements 1 and 5, indices $j5$ and $k5$ were interchanged the way to have the same access manner to matrix F. As it was shown in Chapter 8 ([*XFOR and Automatic Optimizers*], page 115), a better performance can be obtained by fusing statements 2 and 3 in addition to fusing 0, 1 and 4. In addition to that, each outermost index is unrolled twice in order to improve the data reuse between statements. the corresponding code is represented in Listing 9.61. This modification yields a speedup of 3.68× over the Pluto code, when compiled using *GCC*4.8.1.

```
1  xfor (io = 0,   i1 = 0,   i2 = 0,   i3 = 0,   i4 = 0,   i5 = 0 ;
2        io<=ni-1, i1<=ni-1, i2<=nj-1, i3<=nj-1, i4<=ni-1, i5<=ni-1 ;
3        io+=2,    i1+=2,    i2+=2,    i3+=2,    i4+=2,    i5+=2  ;
4        1,        1,        1,        1,        1,        1  ;
5        nj,       nj,       0,        0,        nj,       2*nj) {
6  xfor (jo = 0,   j1 = 0,   j2 = 0,   j3 = 0,   j4 = 0,   j5 = 0 ;
7        jo<=nj-1, j1<=nj-1, j2<=nl-1, j3<=nl-1, j4<=nl-1, j5<=nl-1 ;
8        jo++,     j1++,     j2++,     j3++,     j4++,     j5++ ;
9        1,        1,        1,        1,        1,        1 ;
10       0,        0,        0,        0,        0,        0) {
11 xfor (f30 = 0, k1 = 0,   f32 = 0, k3 = 0,   f34 = 0, k5 = 0 ;
12       f30<1,   k1<=nk-1, f32<1,   k3<=nm-1, f34<1,   k5<=nj-1 ;
13       f30++,   k1++,     f32++,   k3++,     f34++,   k5++ ;
14        1,       1,        1,       1,        1,       1 ;
15        0,       0,        0,       0,        0,       0) {
16    2: { F[i2][j2] = 0;
17         F[i2+1][j2] = 0; }
18    4: { G[i4][j4] = 0;
19         G[i4+1][j4] = 0; }
20    3: { F[i3][j3] += C[i3][k3] * D[k3][j3];
21         F[i3+1][j3] += C[i3+1][k3] * D[k3][j3]; }
22    0: { E[io][jo] = 0;
23         E[io+1][jo] = 0; }
24    1: { E[i1][j1] += A[i1][k1] * B[k1][j1];
25         E[i1+1][j1] += A[i1+1][k1] * B[k1][j1]; }
26    5: { G[i5][j5] += E[i5][k5] * F[k5][j5];
27         G[i5+1][j5] += E[i5+1][k5] * F[k5][j5]; }
28 } } }
```

Listing 9.61 – 3mm: *XFOR Enhaced Version.*

9.6 Conclusion

As it was described in this chapter, the XFOR construct offers to the programmer a simple way to apply and compose loop transformations such as loop shifting, loop dilatation, loop fusion, loop interchange, loop skewing, etc. Thanks to XFOR, loop transformations can be applied simultaneously to several for loops. Thanks to our software tool XFORGEN, the programmer may improve the code generated by automatic loop optimizers such as Pluto. This opportunity was not attainable before since automatically generated codes are not even readable nor modifiable.

Conclusion 10

10.1 Contributions

We have proposed a new programming control structure called "XFOR" or "multifor", allowing to define several for-loops at the same time. Respective iteration domains are mapped onto each other according to a running frequency (*the grain*) and a relative position (*the offset*). The XFOR structure takes its roots in the polyhedral model [8]. Many studies in the field target automatic parallelization [16]. However, fundamental complexity limits, difficult syntactic and semantic analysis, and the variety of possible optimization criteria make it difficult to automate program transformations [30]. The goal of our work is to bring sophisticated and efficient polyhedral techniques at the programming-language level.

Improving Data Locality. XFOR is well suited for data locality aware programming. It allows the programmer to explicitly control the inter-statement and/or intra-statement reuse distances. It has been shown in Chapter 6, when rewriting programs from the Polybench benchmark suite [27], using the XFOR construct, that XFOR programs outperform the original ones. XFOR eases significantly the adaptation of a code to vectorization, and it also supports OpenMP pragmas [26] for XFOR-loop parallelization (`#pragma omp [parallel] for`). Thus, it offers to the programmer more opportunities to write a good optimized program.

IBB. In order to use the XFOR construct, we developed a source-to-source compiler called IBB. Source code containing XFOR loop-structures is translated by the IBB source-to-source compiler into a semantically equivalent C code made of for-loop structures. This is done in two steps. First, the index domains are turned into polyhedra over a common referential domain, and second, the scanning code is generated for their union. The second step is performed using the CLooG library [11] which generates code for scanning unions of polytopes.

XFOR-WIZARD. In addition to that, we developed an XFOR-WIZARD that helps the programmer in re-writing a program with classical for-loop into a program using XFOR-loops. To implement this tool, we extended the polyhedral tool Clan [42] to support XFOR. Thus we let our construct interact with polyhedral tools (via OpenSCop [7]). XFOR-WIZARD takes as input a for-loop program and helps the programmer to generate an equivalent but more efficient XFOR-loop program. First, XFOR-WIZARD parses the reference program and identifies parts of code placed between *#pragma scop* and *#pragma endscop*. Then, for each identified part, it generates automatically a perfectly nested XFOR-loop which is semantically equivalent to the initial set of loop nests, where all the statements are identically scheduled. This is achieved by fusing all loop nests into one unique XFOR-loop nest, where the outermost XFOR-loop has as many indices as the number of statements in the original loop nests, and the offsets of the outermost XFOR-loop are set to the maximum values that guarantee a sequential execution similar to the sequence of the original nests. Once the XFOR-loop nest has been generated, the user may apply transformations to it and verify, step-by-step, if they are legal regarding data dependencies. For verifying the dependencies, XFOR-WIZARD calls the dependency analyzer Candl [14] to compare the set of dependencies of the original for-loop program against the dependencies of the XFOR-loop program, and thus informs the user about the correctness of their modifications.

Bridging Performance Gaps. We show that the XFOR structure helps in highlighting important performance issues, that could not have been clearly identified before for some of them. By comparing XFOR-generated codes to Pluto-generated codes [6], and also XFOR-codes among each other, we highlighted five important gaps in the currently adopted and well-established code optimization strategies: *insufficient data locality optimization, excess of conditional branches in the generated code, verbose code with too many machine instructions, data locality optimization resulting in processor stalls*, and finally *missed vectorization opportunities.* We illustrated the importance of these issues in program optimization using eleven representative codes extracted from the Polybench benchmark suite [27]. Every code has been rewritten using the XFOR structure and also optimized by the most recent version of Pluto with the combination of options generating the best performing code.

Polyhedral IR and XFORGEN. We show that the XFOR loop structure extends programming expressiveness toward optimizing loop transformations. It allows to schedule loop statements precisely by setting conveniently the offset and grain expressions. In addition of being a powerful programming structure, XFOR is also an intermediate representation language for polyhedral loop

transformations. Actually, we developed a tool named XFORGEN, that generates XFOR-loops from the OpenScop definition modified by automatic optimizers as Pluto. This tool helps the programmer to understand the transformations applied by the optimizer and also allows him to enhance the generated code. Last but not least, the XFOR construct may be used as a didactic tool for teaching loop transformations and the polyhedral model.

10.2 Perspectives

Non Linear XFOR. As perspectives, it is possible to extend the XFOR construct to more general codes embedding indirect and non-linear memory accesses. And then, it would be interesting to test some non-polyhedral benchmarks like, for example, SPARKoo [28] which is a benchmark specialized on computations on sparse matrices that are stored in a compressed way. A possible approach is also to perform dynamic analysis using the Apollo platform [69] for example.

Parallel Execution of XFOR Indices. Another possible path is to try to test the parallel version of XFOR (using OMP sections) on some (non-general purpose) processor architecture, and identify how the current implementation may be improved to reach better performance.

XFOR for GPU. In addition to that, it may be interesting to make the XFOR construct available for GPU programming. It would be very efficient, specially in the phase of data loading where the grain and offset will help the programmer significantly in expressing complex array accesses.

Appendix

A

Contents

A.1 IBB Installation

License: IBB is released under the GNU General Public License, version 2 (GPL v2) [67].

Requirements: To successfully install the IBB compiler, you should install the following:

1. Flex or Lex

Flex [70] is a well known lexical analyzer generator. It is a tool for generating programs that perform pattern-matching on text. There are many applications for Flex, including writing compilers in conjunction with GNU Bison [71]. Flex is a free implementation of the well known Lex program. It features a Lex compatibility mode, and also provides several new features such as exclusive start conditions.
On Debian based systems, Flex can be installed using the following command:

```
$ sudo apt-get install flex
```

2. Bison or Yacc

Bison [71] is a general-purpose parser generator that converts a grammar description for an LALR context-free grammar into a C parser program. Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change.

On Debian based systems, Bison can be installed using the following command:

```
$ sudo apt-get install bison
```

3. Glib

GLib [64] is a general-purpose utility library, which provides many useful data types, macros, type conversions, string utilities, file utilities, a main loop abstraction, and so on.
On Debian based systems, GLib can be installed using the following command:

```
$ sudo apt-get install libglib2.0-dev
```

4. OpenScop

OpenScop [7] is an open specification that defines a file format and a set of data structures to represent a SCoP (for more details, refer to Subsection 2.3.1, Chapter 2, page 37).
To successfully install IBB, it is recommended to install a specific version of OpenScop. The OpenScop version that is compatible with IBB may be installed using the following commands:

```
$ git clone https://github.com/periscop/OpenScop.git
$ cd OpenScop
$ git checkout fe126235d9
```

```
$ ./autogen.sh
$ ./configure --with-gmp=no
$ make
$ sudo make install
```

5. CLooG

CLooG [11] is a free software and library to generate code for scanning Z-polyhedra (more details are given in Subsection 2.3.4, Chapter 2, page 42). CLooG depends on the GNU Multiple Precision Library (GMP) [72] version 4.1.4 or above. To install it, the user may use this command:

```
$ sudo apt-get install libgmp-dev
```

To successfully install IBB, it is recommended to install the version 0.18.3 of CLooG. This version may be installed using the following commands:

```
$ git clone https://github.com/periscop/cloog.git
$ cd cloog
$ git checkout cloog-0.18.3
$ ./get_submodules.sh
$ ./autogen.sh
$ ./configure --with-osl=system --with-osl-prefix=/usr/local
$ make
$ sudo make install
```

IBB Installation: Once downloaded and unpacked, one can compile IBB by typing the following command on the IBB's root directory: `$ make`.

Bibliography

[1] U. Bondhugula, *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, The Ohio State University, 2008. pages viii, 25, 36 et 37.

[2] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic Optimization for Image Processing Pipelines," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 429–443, ACM, 2015. pages viii, 3, 22 et 50.

[3] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), pp. 519–530, ACM, 2013. pages viii, 3, 22 et 51.

[4] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The Pochoir Stencil Compiler," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, (New York, NY, USA), pp. 117–128, ACM, 2011. pages viii, 3, 22, 52 et 53.

[5] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rude, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt, "ExaStencils: Advanced Stencil-Code Engineering," in *Euro-Par 2014: Parallel Processing Workshops* (L. Lopes, J. Žilinskas, A. Costan, R. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander, eds.), vol. 8806 of *Lecture Notes in Computer Science*, pp. 553–564, Springer International Publishing, 2014. pages viii, 55 et 56.

[6] "PLUTO - An automatic parallelizer and locality optimizer for multicores." `http://pluto-compiler.sourceforge.net`. [Online; accessed 14-October-2015]. pages 2, 13, 19, 90, 115, 116, 117, 133, 152 et 170.

[7] C. Bastoul, "OpenScop: A Specification and a Library for Data Exchange in Polyhedral Compilation Tools," tech. rep., Paris-Sud University, France, September 2011. pages 2, 5, 37, 80, 98, 109, 170 et 174.

[8] P. Feautrier and C. Lengauer, "Polyhedron model," in *Encyclopedia of Parallel Computing* (D. Padua, ed.), pp. 1581–1592, Springer US, 2011. pages 2, 22 et 169.

[9] C. Bastoul, "Extracting polyhedral representation from high level languages," tech. rep., LRI, Paris-Sud University, France, 2008. pages 2 et 39.

[10] C. Bastoul, "Clay: the chunky loop alteration wizardry," tech. rep., LRI, Paris-Sud University, France, 2012. pages 2 et 41.

[11] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, (Juan-les-Pins, France), pp. 7–16, September 2004. pages 2, 5, 8, 19, 23, 39, 42, 76, 77, 84, 169 et 175.

[12] N. Vasilache, C. Bastoul, and A. Cohen, "Polyhedral Code Generation in the Real World," in *Compiler Construction* (A. Mycroft and A. Zeller, eds.), vol. 3923 of *Lecture Notes in Computer Science*, pp. 185–201, Springer Berlin Heidelberg, 2006. pages 2, 5, 39, 42 et 84.

[13] C. Bastoul, "Generating loops for scanning polyhedra," tech. rep., PRiSM, Versailles University, 2002. pages 2, 5, 39, 42 et 84.

[14] C. Bastoul and L.-N. Pouchet, "Candl: the chunky analyzer for dependences in loops," tech. rep., LRI, Paris-Sud University, France, 2012. pages 2, 39, 42, 98, 104 et 170.

[15] O. Zinenko, S. Huot, and C. Bastoul, "Clint: A direct manipulation tool for parallelizing compute-intensive program parts," in *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, pp. 109–112, July 2014. pages 2 et 42.

[16] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 101–113, ACM, 2008. pages 2, 3, 13, 19, 22, 39, 43, 44, 90, 115, 116, 152 et 169.

[17] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic Transformations for Communication-minimized Parallelization and Locality Optimization in the Polyhedral Model," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, (Berlin, Heidelberg), pp. 132–146, Springer-Verlag, 2008. pages 2, 39 et 43.

[18] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan, "A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 343–352, ACM, 2010. pages 2, 39 et 43.

[19] A. Acharya and U. Bondhugula, "PLUTO+: Near-complete Modeling of Affine Transformations for Parallelism and Locality," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, (New York, NY, USA), pp. 54–64, ACM, 2015. pages 2 et 43.

[20] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A High-performance Java Dialect," *Concurrency: Practice and Experience*, vol. 10, no. 11-13, pp. 825–836, 1998. pages 3 et 21.

[21] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007. pages 3 et 21.

[22] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, "X10 Language Specification Version 2.2," 2012. pages 3, 21 et 47.

[23] I. Fassi, P. Clauss, M. Kuhn, and Y. Slama, "Multifor for Multicore," in *IMPACT 2013, Third International Workshop on Polyhedral Compilation Techniques*, (Berlin, Germany), pp. 37–44, Epubli, Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques, January 2013. pages 4, 62 et 64.

[24] I. Fassi and P. Clauss, "XFOR: Filling the Gap between Automatic Loop Optimization and Peak Performance," in *ISPDC 2015, IEEE 14th International Symposium on Parallel and Distributed Computing*, (Limassol, Cyprus), 2015. pages 4, 5, 9, 62, 86 et 90.

[25] I. Fassi and P. Clauss, "IBB : The Multifor Compiler." `https://team.inria.fr/camus/ibb/`, 2013 – 2014. [Online; accessed 14-October-2015]. pages 5 et 71.

[26] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Comput. Sci. Eng.*, vol. 5, pp. 46–55, Jan. 1998. pages 6, 47, 73, 74 et 169.

[27] "The Polyhedral Benchmark suite." `http://sourceforge.net/projects/polybench`. [Online; accessed 14-October-2015]. pages 9, 15, 28, 87, 94, 104, 115, 117, 133, 156, 158, 161, 169 et 170.

[28] H. L. A. van der Spek, E. M. Bakker, and H. A. G. Wijshoff, "SPARKoo: A Benchmark Package for the Compiler Evaluation of Irregular/Sparse Codes," *CoRR*, vol. abs/0805.3897, 2008. pages 20 et 171.

[29] E. W. Dijkstra, "A preliminary investigation into Computer Assisted Programming." circulated privately, n.d. page 21.

[30] L. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, "Iterative optimization in the polyhedral model: part ii, multidimensional time," in *Proc. of the ACM SIGPLAN 2008 Conf. on Programming Language Design and Implementation*, pp. 90–100, 2008. pages 23 et 169.

[31] E. Park, J. Cavazos, L. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan, "Predictive Modeling in a Polyhedral Optimization Space," *International Journal of Parallel Programming*, vol. 41, no. 5, pp. 704–750, 2013. pages 23, 39 et 44.

[32] Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O'Boyle, "Integrating profile-driven parallelism detection and machine-learning-based mapping," *TACO*, vol. 11, no. 1, p. 2, 2014. page 23.

[33] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. V. Bonilla, J. Thomson, C. K. I. Williams, and M. F. P. O'Boyle, "Milepost GCC: Machine Learning Enabled Self-tuning Compiler," *Int. J. of Parallel Programming*, vol. 39, no. 3, pp. 296–327, 2011. page 23.

[34] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The Polyhedral Model Is More Widely Applicable Than You Think," in *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, LNCS, (Paphos, Cyprus), Springer-Verlag, Mar. 2010. pages 25 et 27.

[35] R. M. Karp, R. E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *J. ACM*, vol. 14, pp. 563–590, July 1967. page 25.

[36] A. W. Appel, *Modern Compiler Implementation in C: Basic Techniques*. New York, NY, USA: Cambridge University Press, 1997. page 25.

[37] S. P. Midkiff, *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2012. page 25.

[38] C. Bastoul, *Improving data locality in static control programs*. PhD thesis, Université Paris 6, 2004. pages 25 et 35.

[39] C. Bastoul, *ontributions to High-Level Program Optimization*. PhD thesis, Université Paris-Sud, 2012. page 25.

[40] P. Feautrier, "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, 1992. pages 27 et 30.

[41] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache, "Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, (Washington, DC, USA), pp. 144–156, IEEE Computer Society, 2007. page 27.

[42] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam, "Putting Polyhedral Loop Transformations to Work," in *LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, (College Station, Texas), pp. 209–225, october 2003. pages 27, 28, 30, 39, 98 et 170.

[43] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *Second International Workshop on Polyhedral Compilation Techniques, Paris, France*, 2012. page 27.

[44] T. Grosser, A. Größlinger, and C. Lengauer, "Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation," *Parallel Processing Letters*, vol. 22, no. 4, 2012. page 27.

[45] L. Rauchwerger and D. Padua, "The LRPD Test: Speculative Run-time Parallelization of Loops with Privatization and Reduction Parallelization,"

in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, (New York, NY, USA), pp. 218–232, ACM, 1995. page 33.

[46] D. A. Padua and M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Commun. ACM*, vol. 29, pp. 1184–1201, Dec. 1986. page 33.

[47] P. Feautrier, "Array Expansion," in *In ACM Int. Conf. on Supercomputing*, pp. 429–441, 1988. page 33.

[48] P. Feautrier, "Dataflow Analysis of Array and Scalar References," *International Journal of Parallel Programming*, vol. 20, 1991. page 33.

[49] U. K. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations*. Norwell, MA, USA: Kluwer Academic Publishers, 1993. page 34.

[50] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache, "Loop Transformations: Convexity, Pruning and Optimization," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, (New York, NY, USA), pp. 549–562, ACM, 2011. page 35.

[51] C. Bastoul, "Efficient Code Generation for Automatic Parallelization and Optimization," in *ISPDC*, vol. 2, pp. 23–30, 2003. pages 39, 42 et 84.

[52] L.-N. Pouchet, C. Bastoul, and A. Cohen, "LetSee: the LEgal Transformation SpacE Explorator." Third International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES'07), L'Aquila, Italia, July 2007. Extended abstract, pp 247–251. page 39.

[53] C. Bastoul and P. Feautrier, "Improving Data Locality by Chunking," in *Compiler Construction* (G. Hedin, ed.), vol. 2622 of *Lecture Notes in Computer Science*, pp. 320–334, Springer Berlin Heidelberg, 2003. page 42.

[54] M. Griebl, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. PhD thesis, 2004. page 42.

[55] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter, "Partitioned Global Address Space Languages," *ACM Comput. Surv.*, vol. 47, pp. 62:1–62:27, may 2015. pages 47 et 49.

[56] M. P. Forum, "MPI: A Message-Passing Interface Standard," tech. rep., Knoxville, TN, USA, 1994. page 47.

[57] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-oriented Approach to Non-uniform Cluster Computing," *SIGPLAN Not.*, vol. 40, pp. 519–538, Oct. 2005. page 47.

[58] P. Feautrier, Ã. Violard, and A. Ketterlin, "Improving the performance of x10 programs by clock removal," in *Compiler Construction* (A. Cohen, ed.), vol. 8409 of *Lecture Notes in Computer Science*, pp. 113–132, Springer Berlin Heidelberg, 2014. page 48.

[59] V. A. Saraswat, O. Tardieu, D. Grove, D. Cunningham, M. Takeuchi, and B. Herta, "A Brief Introduction To X10 (For the High Performance Programmer)." The IBM Corporation, September 2012. page 48.

[60] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and A. Navarro, "User-Defined Parallel Zippered Iterators in Chapel," in *PGAS 2011: Fifth Conf. on Partitioned Global Address Space Programming Models*, October 2011. page 49.

[61] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, (New York, NY, USA), pp. 13–24, ACM, 2013. page 54.

[62] K. Stock, M. Kong, T. Grosser, L. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan, "A framework for enhancing data reuse via associative reordering," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '14*, pp. 65–76, 2014. pages 56 et 57.

[63] I. Christadler, G. Erbacci, and A. D. Simpson, "Facing the Multicore-Challenge II," ch. Performance and productivity of new programming languages, pp. 24–35, Berlin, Heidelberg: Springer-Verlag, 2012. page 58.

[64] "GLib Reference Manual." `https://developer.gnome.org/glib/stable/index.html`. [Online; accessed 14-October-2015]. pages 79, 80 et 174.

[65] P. Clauss, I. Fassi, and A. Jimborean, "Software-controlled Processor Stalls for Time and Energy Efficient Data Locality Optimization," in *SAMOS XIV 2014,International Conference on Embedded Computer Systems: Architectures, Modeling,and Simulation*, pp. 199–206, 2014. pages 85 et 86.

[66] P. Clauss, "Mind The Gap! A study of some pitfalls preventing peak performance in polyhedral compilation using a polyhedral antidote," in *IMPACT 2015, 5th International Workshop on Polyhedral Compilation Techniques*, (Amsterdam, The Netherlands), January 2015. page 86.

[67] Free Software Foundation, "GNU General Public License." `http://www.gnu.org/licenses/gpl-2.0.html`, 1991. [Online; accessed 14-October-2015]. pages 99, 152 et 174.

[68] "perfmon2: improving performance monitoring on Linux." `http://perfmon2.sourceforge.net`. page 117.

[69] A. Sukumaran-Rajam, J. M. Martinez, W. Wolff, A. Jimborean, and P. Clauss, "Speculative Program Parallelization with Scalable and Decentralized Runtime Verification," in *Runtime Verification* (B. Bonakdarpour and S. A. Smolka, eds.), vol. 8734, (Toronto, Canada), pp. 124–139, Springer, Sept. 2014. page 171.

[70] Free Software Foundation, "The Flex Homepage." `http://www.gnu.org/software/flex/`. [Online; accessed 14-October-2015]. page 174.

[71] Free Software Foundation, "The Bison Homepage." `http://www.gnu.org/software/bison/`. [Online; accessed 14-October-2015]. page 174.

[72] Free Software Foundation, "The GNU Multiple Precision Arithmetic Library." https://gmplib.org/. [Online; accessed 14-October-2015]. page 175.

Index

Imen FASSI

XFOR (Multifor) : A New Programming Structure to Ease the Formulation of Efficient Loop Optimizations

Résumé

Nous proposons une nouvelle structure de programmation appelée “XFOR” ou “Multifor”, dédiée à la programmation orientée réutilisation de données. XFOR permet de gérer simultanément plusieurs boucles "*for*" ainsi que d'appliquer des transformations (simples ou composées) de boucles d'une façon aisée et intuitive. Les expérimentations ont montré des accélérations significatives des codes XFOR par rapport aux codes originaux, mais aussi par rapport au codes générés automatiquement par l'optimiseur polyédrique de boucles Pluto. Nous avons mis en œuvre la structure XFOR par le développement de trois outils logiciels : (1) un compilateur source-à-source nommé IBB (*Iterate-But-Better !*), qui traduit automatiquement tout code basé sur le langage C contenant des boucles XFOR en un code équivalent où les boucles XFOR ont été remplacées par des boucles *for* sémantiquement équivalentes. L'outil IBB bénéficie également des optimisations implémentées dans le générateur de code polyédrique CLooG qui est invoqué par IBB pour générer des boucles *for* à partir d'une description OpenScop ; (2) un environnement de programmation XFOR nommé XFOR-WIZARD qui aide le programmeur dans la ré-écriture d'un programme utilisant des boucles *for* classiques en un programme équivalent, mais plus efficace, utilisant des boucles XFOR ; (3) un outil appelé XFORGEN, qui génère automatiquement des boucles XFOR à partir de toute représentation OpenScop de nids de boucles transformées générées automatiquement par un optimiseur automatique.

Summary

We propose a new programming structure named “XFOR” or “Multifor”, dedicated to data-reuse aware programming. It allows to handle several for-loops simultaneously and map their respective iteration domains onto each other according to a running frequency (*the grain*) and a relative position (*the offset*). Additionally, XFOR eases loop transformations application and composition. Experiments show that XFOR codes provides significant speed-ups when compared to the original code versions, but also to the Pluto optimized versions. We implemented the XFOR structure through the development of three software tools: (1) a source-to-source compiler named IBB for Iterate-But-Better!, which automatically translates any C/C++ code containing XFOR-loops into an equivalent code where XFOR-loops have been translated into for-loops. IBB takes also benefit of optimizations implemented in the polyhedral code generator CLooG which is invoked by IBB to generate for-loops from an OpenScop specification; (2) an XFOR programming environment named XFOR-WIZARD that assists the programmer in re-writing a program with classical for-loops into an equivalent but more efficient program using XFOR-loops; (3) a tool named XFORGEN, which automatically generates XFOR-loops from any OpenScop representation of transformed loop nests automatically generated by an automatic optimizer.