U. F. R. de mathématique et d'informatique

Master 2 de Sciences, mention Informatique
spécialité Réseaux Informatiques et Systèmes Embarqués

Cursus Master en Ingénierie
spécialité Informatique, Systèmes et Réseaux

Mémoire de stage présenté par

Marek Felsoci
marek.felsoci@etu.unistra.fr

6 septembre 2019

Integration of the XFOR structure into the Clang/LLVM compiler and extension to parallel programming

Encadré par

Prof. Philippe Clauss
philippe.clauss@inria.fr

Du 21 janvier au 31 juillet 2019

ICube UMR 7357
Laboratoire des sciences d'ingénieur, de l'informatique et de l'imagerie
300 Boulevard Sébastien Brant, CS 10413, F-67412 Illkirch Cedex

The document was typeset with LaTeX, using the *kpfonts* and the *DejaVuSansMono* font packages.

Released on 06/09/2019 at 20:08.

Acknowledgements

In the first place, I would like to thank my supervisor, Prof. Philippe Clauss. Our cooperation is the main origin of my interest and motivation for the domain of scientific research.

I would also like to thank all the members of the ICPS team who made of my second experience at ICube wonderful 6 months and helped me with some important decisions. Special thanks go to Salwa Kobeissi, Raquel Lazcano López, Aurélien Rausch, Maxime Schmitt, Luke Bertot, Harenome Razanajato and Bérenger Bramas.

My appreciation goes also to all of our professors who guided us throuhout our studies at the University of Strasbourg and, especially to Stéphane Cateloin for taking care of us, the CMI students, for five years.

Contents

Chapitre 0

Résumé en français

Mon stage de fin d'études s'est déroulé au laboratoire ICube, dans l'équipe « Informatique et Calcul Parallèle Scientifique ». Formant le dernier semestre du Cursus Master en Ingénierie [12] que je suis à l'Université de Strasbourg depuis 2014, ce stage a pour but de mettre en pratique les connaissances et valider les compétences que j'ai acquises dans le cadre de la formation. Mon travail s'inscrit dans le domaine de l'optimisation de programmes et en particulier de boucles FOR en utilisant le modèle polyèdrique [19].

0.1 Motivation

À l'occasion de mon premier stage à ICube en 2018, j'ai commencé à travailler sur la structure de programmation XFOR [14, 16]. Sa syntaxe est très similaire à celle des boucles FOR standards du langage C. Elle permet de regrouper et de gérer plusieurs boucles FOR en même temps. Grâce à ses deux paramètres spécifiques, que sont le *grain* et l'*offset*, le programmeur peut influer de manière simple et intuitive sur la façon dont ces boucles doivent s'exécuter. Un exemple de boucle XFOR est proposé dans le Listage 1.

```
xfor(i0 = 1, i1 = 1; i0 < N - 1, i1 < N - 1; i0++, i1++; 1, 1; 0, 1)
  xfor(j0 = 1, j1 = 1; j0 < N - 1, j1 < N - 1; j0++, j1++; 1, 1; 0, 6) {
    0 : B[i0][j0] = A[i0][j0] + A[i0][j0 - 1] + A[i0][j0 + 1] + A[i0 + 1][j0] + A[i0 - 1][j0];
    1 : A[i1][j1] = B[i1][j1];
  }
```

Listage 1 – Un nid XFOR gérant deux nids de boucles FOR aux indices respectifs (i_0, j_0) et (i_1, j_1)

L'objectif est d'ajuster l'ordre d'exécution des instructions à l'intérieur des boucles gérées, de manière à mettre en valeur les possibilités qu'offrent les architectures d'ordinateur modernes. Il s'agit principalement d'optimiser l'utilisation de la mémoire cache et d'exploiter au mieux les capacités de parallélisation des processeurs. Un programme re-écrit en utilisant XFOR peut être jusqu'à 6 fois plus rapide par rapport à sa version originale, même sans parallélisation [14].

0.2 Intégration dans le compilateur Clang/LLVM

La structure XFOR n'est pas standardisée dans la spécification officielle du langage C [2]. Par conséquent, les compilateurs de production habituels tels que GCC n'implémentent pas la traduction des codes sources XFOR.

Un des principaux outils dédiés à cette structure est le compilateur « Iterate, But Better ! » (IBB) [15] permettant de traduire des boucles XFOR en boucles FOR équivalentes. Ainsi, le programme traduit peut être compilé par un compilateur quelconque du langage C afin d'obtenir l'exécutable final. Cette nécessité de compiler un programme XFOR deux fois a en partie motivé ma première mission. L'intégration de la structure XFOR dans un compilateur de production tel que Clang/LLVM [11, 22] permettra à l'avenir la compilation directe de programmes XFOR et pourra aider à promouvoir la structure au sein de la communauté de programmeurs.

Dans le cadre de cette mission, j'ai été amené à étendre les analyseurs lexical et syntaxique du compilateur Clang/LLVM, pour que ce dernier soit en mesure de reconnaître et de correctement traduire des boucles XFOR. Aussi, j'ai implémenté la transformation de celles-ci en représentation intermédiaire utilisée par le compilateur [22] pour produire des fichiers exécutables.

0.3 Extension à la programmation parallèle

Afin de rendre les programmes XFOR encore plus performants, nous nous sommes également concentrés sur les moyens de parallélisation de boucles XFOR. De mon côté, il s'agissait notamment d'étudier l'utilisation de la bibliothèque de parallélisation OpenMP [13] dans les programmes XFOR. Aussi, nous nous sommes intéressés à l'assistance au programmeur dans le processus de parallélisation à l'aide du logiciel XFOR-WIZARD [14].

Cet outil est un environnement de développement intégré et dédié à assister le développeur à l'optimisation des programmes existants en les re-écrivant avec la structure XFOR. Il permet de transformer automatiquement un nid de boucles FOR en un nid équivalent de boucles XFOR. Le programmeur peut ensuite ajuster les paramètres spécifiques des boucles XFOR obtenues afin d'optimiser son programme au maximum. À chaque intervention du programmeur, le logiciel est capable de vérifier si sa modification est correcte (voir Figure 0.1). C'est-à-dire, si le programme XFOR transformé est sémantiquement équivalent à l'original.

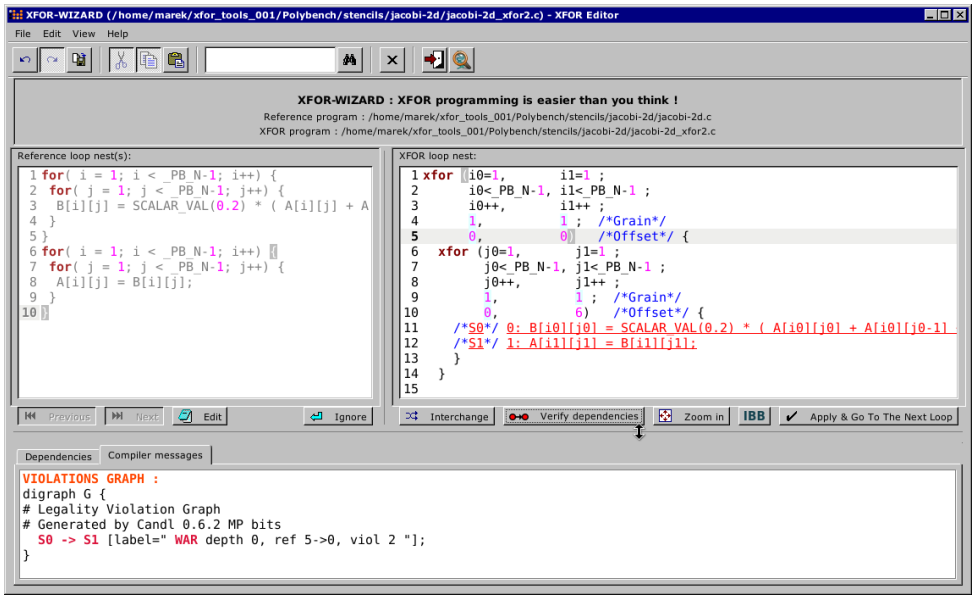


Figure 0.1 – Modification assistée d'un programme XFOR à l'aide de XFOR-WIZARD

0.4 Conclusion et perspectives

Même si un petit nombre de fonctionnalités de XFOR reste encore à implémenter, notre version de Clang/LLVM est désormais capable de compiler des programmes avec des boucles XFOR qu'elles soient simples ou imbriquées.

D'autre part, j'ai étudié et expérimenté plusieurs stratégies de parallélisation de la structure XFOR. La piste d'évolution à suivre dorénavant serait d'approfondir les expérimentations des différentes approches étudiées et d'implémenter les modifications logicielles nécessaires dans les outils dédiés à XFOR.

En outre, d'avantage de perspectives d'optimisation existent pour la structure XFOR. Il s'agit, par exemple, de porter la structure vers la programmation GPU, ce qui permettrait de rendre plus transparente et faciliter l'écriture de programmes optimisés pour les processeurs graphiques.

Chapter 1

Introduction

For my final Master's degree internship, I was hosted by the Scientific and Parallel Computing (ICPS) team at ICube laboratory of engineering science, computer science and imaging. Representing the last semester of my Master's degree in Computer science and Engineering [12] at the University of Strasbourg, the aim of this internship was to turn into practice the theoretical and make use of the practical knowledge I have acquired thanks to my studies.

The work I have done is related to the domain of program optimization and in particular the optimization of for-loops using the polyhedral model [19].

In summer 2018, I had the privilege to join the ICPS team for the first time. In collaboration with Prof. Philippe Clauss, I participated on the development of software tools dedicated to the XFOR programming structure [14, 16] designed within the ICPS team.

XFOR allows to handle multiple for-loops at the same time. Thanks to a couple of parameters specific to the structure, the *grain* and the *offset*, programmers can act on the way the handled loops are executed. The goal is to adjust the order of instructions within the loops in such a way as to improve the usage of cache memory and take advantage of simultaneous instruction execution, namely instruction vectorization capabilities of the target processing unit(s).

This opportunity made me work on new issues and search for solutions to them essentially by my own means. Also, I had the possibility to discover more about the compilation and learn a lot of new, powerful concepts and tools for optimizing and parallelizing programs.

The experience kindled in me a profound interest for scientific research, to the point I returned to ICube for my final Master's degree internship with a tenacious motivation for doctoral studies.

This time, I had two main missions: integrate the XFOR structure into the Clang/LLVM [11, 22] compiler and study the possibilities of extension of the structure to parallel programming.

One of the key software tools related to the XFOR structure is the source-to-source compiler 'Iterate, But Better!' (IBB) [15] which enables to translate XFOR programs into equivalent for-programs. The output of IBB can be then compiled by any production compiler of C language such as gcc, icc, Clang/LLVM, etc. This means that, XFOR programs have to be compiled twice before generating the final executable binary file. Hence the motivation was to extend the Clang/LLVM compiler to be able to compile XFOR programs directly.

Finally, in the second part of the internship, I was exploring different strategies of parallelization of XFOR loops using the OpenMP library [13] with the aim of further increasing the performance of XFOR programs. In addition, we were studying how the capabilities of existing XFOR-related tools should be extended relatively to the explored parallelization methods.

The document is organized as follows; In Chapter 2, we present ICube laboratory in the context of the research sector. In Chapter 3, we remind the polyhedral model which is the core theoretical concept behind our work. In Chapter 4, we follow up with a detailed explanation of the principles and the contributions of the XFOR structure and associated tools in the framework of current loop optimization techniques. Relatively to this, we describe our missions, achievements and discuss our experiments in Chapters 5 and 6. Eventually, we conclude in Chapter 7.

Chapter 2

Host institution

As a research facility, the ICube laboratory obviously belongs to the sector of research. The ultimate goal is to enlarge the knowledge of a specific scientific domain and directly or indirectly contribute to the advancement of the society. Research scientists try to come up with an innovative concept or idea and prove or disprove it depending on the facts they discover during their research activity. The latter must take place in an appropriate environment, generally within a research or an educational facility. In our case, it is a computer science laboratory.

2.1 Research sector

In France, research institutions are either private or public, like ICube. The results published by public institutions are mostly available to everyone, even though they remain the property of the French state. On the other hand, private establishments work mainly for particular clients. As such, they often have to respect the confidentiality of their discoveries. Nevertheless, exceptions can occur in both cases.

There is no unique way of doing research. However, typically we start either with a brand new concept or an existing one with the goal to improve it. An appropriate scientific method should allow us to validate the initial hypothesis. Although, it is not guaranteed. On the other hand, the research activity can, in some cases, lead us to an unexpected discovery.

Also, the policy as well as the financial situation of the research facility may have an important impact on the outcome of our working efforts. In the private sector, the priority is put on the commercialization of the results as fast as possible. Consequently, the freedom of choice of the scientists is significantly more restricted compared to the public sector. However, if a private institution have enough funds on its disposal, it can afford to support long term research projects without an immediate plan for commercialization. Establishments with such possibilities are often multinational companies with worldwide presence such as Google, Amazon or Microsoft.

Moreover, for the results of a research project to be considered trustworthy, they must be based on solid and reproducible proofs approved by the international scientific community. Paradoxically, the ones who are supposed to provide critical reviews and evaluations of our research are not always experts from the domain in question.

2.2 ICube

Founded in 2013, the ICube laboratory has multiple sites in Strasbourg. It has been created by the fusion of five different research facilities. The project was conducted by several institutions engaged both in research and education including the University of Strasbourg [26, 21].

Currently, more than 650 members in 16 research groups and 4 departments of ICube work in the domains of engineering science, computer science and imaging [26, 27].

For my internship, I joined the Scientific and Parallel Computing research team [1] led by Dr. Jens Gustedt and attached to the Computer science department [2] [27]. My position at ICube is illustrated in Figure 2.1.

1. 'Informatique et Calcul Parallèle Scientifique' in French, abbrev. ICPS
2. 'Département Informatique-Recherche' in French, abbrev. D-IR

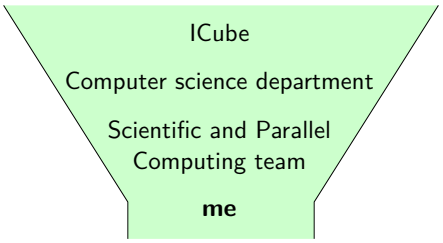


FIGURE 2.1 – My position at ICube

2.3 Scientific and Parallel Computing team

Members of this team work in the domains of parallelization and optimization of programs. Their main research themes include compilation, high performance applications and distributed systems [28].

Some of the members also belong to the common team Camus [3] [28] of the French Institute for Research in Computer science and Automation [4] and the French National Center for Scientific Research [5] [1].

The Camus team works especially on the development of methods and interactive tools for automatic and assisted program parallelization and optimization [23]. As Prof. Philippe Clauss is the head of Camus, the XFOR structure is naturally one of the team's occupations [24].

Another important contribution of the Camus team is APOLLO [24], an automatic and speculative polyhedral optimizer which allows to dynamically perform loop optimizations during execution [25].

3. ' Compilation pour les architectures multicœurs' in French (compilation for multi-core architectures)
4. 'Institut National de Recherche en Informatique et Automatique' in French, abbrev. INRIA
5. 'Centre National de Recherche Scientifique' in French, abbrev. CNRS

Chapter 3

Polyhedral model

Programmers often produce inefficient code. Although, writing a perfectly optimized program capable of taking advantage of all the capabilities of modern computer architectures such as multi-threading, vectorization or multi-level cache memory can be a considerably complex matter especially when attempted to be done manually.

There are many code optimization tools allowing to perform different kinds of optimization automatically and transparently to programmers. The goal is to produce source code capable of making better usage of underlying computer architecture and make programs run faster. In some cases we might also need to achieve the opposite if, for example, some energy constraints need to be taken into account.

Current compilers apply some of these optimizations by default but, fully automated code optimization is not always possible. In some cases, the source code needs to be tagged or partially re-written in a specific way prior to applying an actual optimization technique. This tells the compiler or an optimization tool how and where to optimize the source code.

For example, the OpenMP parallelization library provides a set of special code annotations [13] called `#pragma`, allowing to define how and which portion of the source code should be transformed in order to be executed in parallel (see example in Listing 3.1).

```
#pragma omp parallel for
for(int i = 0; i < 4096; i++)
  output[i] = 0.2 * sqrt(input[i]);
```

Listing 3.1 – Automatic parallelization of a FOR loop using OpenMP

On the other hand, XFOR [14] is a custom made programming language control structure, such as for or while-loops, but designed to allow easy and intuitive expression of complex loop transformations for improving the usage of cache memory and vectorization units.

3.1 Program representation

Transforming directly a program's source code written in a high-level language such as C/C++ would be too complex if not impossible. Code optimization techniques have to rely on some program representation structure which is expressive enough and easy to manipulate.

Traditional representations such as Abstract Syntax Trees (AST) or Control Flow Graphs (CFG) either do not consider all the information necessary to perform optimizations on them or are excessively exhaustive and too complex to deal with.

Here comes the polyhedral model [19], a powerful mathematical abstraction that allows to represent portions of programs having specific properties, so we can apply different code transformations on them and optimize the way they are executed.

3.2 Static control part

Static Control Part (SCoP) is a portion of a program that can be represented using the polyhedral model. It matches the longest sequence of simple or nested for-loops where lower and upper bounds as well as potential conditional statements and memory accesses within the loops are affine functions of surrounding iterators, invariable parameters (variables the values of which are unknown at compilation time but do not change within the loops) and constants. Also, the iteration count of the loops in a SCoP must be known and remain constant during the execution of the program. Therefore, instructions such as `break`, `continue` or `goto` can not be part of it [9, 8]. Figure 3.1 features a couple of SCoP examples.

SCoP detection in source code can be either automatic [8, 20] or manual. In this case, it should be enclosed between a couple of pre-processor directives `#pragma scop` and `#pragma endscop` recognized by the majority of polyhedral code manipulation tools.

```
#pragma scop
for(i = _PB_N - 1; i >= 0; i--)
  for(j = i + 1; j < _PB_N; j++) {
    if(j - 1 >= 0)
      table[i][j] = max_score(table[i][j], table[i][j - 1]);

    ...

    for(k = i + 1; k < j; k++)
      table[i][j] = max_score(table[i][j],
                      table[i][k] + table[k + 1][j]);
  }
#pragma endscop
```

(a) A valid SCoP (extract from an implementation of the Nussinov algorithm)

```
for(i = 1; i < N; i++) {
  for(j = p; j < N; j++)
    A[i][j] = foo(1024);

  if(A[i][0] < 1)
    break;
  else
    p++;
}
```

(b) An invalid SCoP containing a `break` instruction and a loop bound expressed in function of a variable the value of which chages within the loop

Figure 3.1 – Static control part examples

3.3 Iteration domain

Let us consider the two dimensional for-loop nest in Listing 3.2. Inside of both innermost loops, there is a statement. Let us note these statements S_0 and S_1. S_0 represents a simple write access (value assignment) to array A and S_1 a write access to array B combined with a read access to array B and a read access to array A.

```
for(i = 1; i < N; i++) {
  for(j = 1; j < N; j++)
    A[i][j] = 1024; // S0
  for(j = 1; j < N; j++)
    B[i][j] = (B[i][j - 1] + A[i][j]) / 2; // S1
}
```

Listing 3.2 – Example 2-dimensional for-loop nest

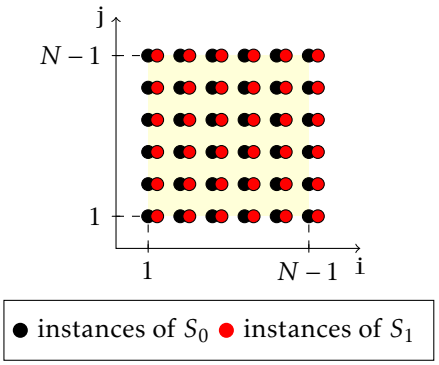
Each execution of a statement represents an instance of the latter. Our loop nest executes, for each value of i, $N-1$ iterations of statement S_0, followed by $N-1$ iterations of S_1. Therefore, S_0 as well as S_1 have $(N-1)\times(N-1)$ instances. Both statements are executed for couples of iterator values $(i,\ j)$ such as $(1,\ 1)$, $(1,\ 2)$, … $(2,\ 1)$, $(2,\ 2)$, … $(N-1,\ N-1)$. We call them iteration vectors. In this case, the vectors are of dimension 2 so as the loop nest depth.

The iteration domain of a statement corresponds to the set of iteration vectors associated with the statement. In other words, each iteration vector of the domain stands for one instance of the related statement. Our example counts two iteration domain sets, $\mathcal{D}_{S_0}$ for statement S_0 and $\mathcal{D}_{S_1}$ for S_1.

According to Listing 3.2, both S_0 and S_1 have the same domain of iteration. Mathematically, they are defined in Figure 3.2a. Graphically, the iteration domains of S_0 and S_1 correspond to a two dimensional polyhedron represented in Figure 3.2b where each point denotes one statement instance.

$$\mathcal{D}_{S_0} = \{\, (i,\ j) \in \mathbb{Z}^2 \mid 1 \leq i \leq N-1 \ \wedge\ 1 \leq j \leq N-1 \,\}$$
$$\mathcal{D}_{S_1} = \mathcal{D}_{S_0}$$

(a) Mathematical representation



(b) Graphical representation

Figure 3.2 – Iteration domains of the statements S_0 and S_1

3.4 Scheduling function

The iteration domain allows us to define how many times and for which iterator values, e. g. i and j, the statements are executed. Although, it does not provide any information on their order of execution which means that the statements might be executed in any order [8].

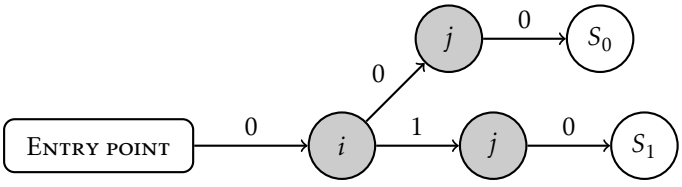
Using a scheduling function we can associate a logical date to each instance of a given statement. The function takes as parameter an iteration vector corresponding to a particular instance of the statement and maps it to a scheduling vector. As such, it allows to express the execution order of one statement in a program relatively to the others. It can be easily determined from the abstract syntax tree (AST) of the program [18]. The AST corresponding to our example from Listing 3.2 is represented in Figure 3.3b.

Let us note θ_{S_0} the scheduling function of the statement S_0 and similarly, θ_{S_1} the scheduling function of S_1. They are defined in Figure 3.3a. The odd dimensions of the resulting scheduling vectors determine the order of the statements relatively to the other statements in the program (lexicographic order) and the even dimensions bound the statements to the associated iterators.

The scheduling vector of S_0 tells us that, couting from zero, S_0 is the first statement in the **first** innermost loop on j in the outermost loop on i comming first after the entry point of the program. Accordingly, we can see, that S_1 is the first statement in the **second** innermost loop on j in the outermost loop on i comming first after the entry point of the program.

$$\theta_{S_0}(i,\ j) = (0,\ i,\ \mathbf{0},\ j,\ 0)$$
$$\theta_{S_1}(i,\ j) = (0,\ i,\ \mathbf{1},\ j,\ 0)$$

(a) Mathematical representation



(b) Graphical representation

Figure 3.3 – Scheduling of the statements S_0 and S_1

3.5 Access function

Access function determines the relation between the iteration domain of the statement and a potential memory access occuring within the latter. Like scheduling function, access function also takes as parameter an iteration vector and maps it to a given memory location accessed by the statement. The output vector has the same dimension as the input iteration vector. One statement may contain multiple memory accesses. So, a set of one or multiple access functions may be associated to one single statement.

In Listing 3.2, we distinguish 4 memory accesses, a write access to $A(i,\ j)$ in S_0, a read access to $B(i,\ j-1)$, a read access to $A(i,\ j)$ and a write access to $B(i,\ j)$ in S_1. We denote them $\mathcal{W}^A_{S_0}$, $\mathcal{R}^B_{S_1}$, $\mathcal{R}^A_{S_1}$ and $\mathcal{W}^B_{S_1}$ respectively. They are defined in Figure 3.4.

$$\mathcal{W}^A_{S_0}(i,\ j) = (i,\ j)$$
$$\mathcal{R}^B_{S_1}(i,\ j) = (i,\ j-1)$$
$$\mathcal{R}^A_{S_1}(i,\ j) = (i,\ j)$$
$$\mathcal{W}^B_{S_1}(i,\ j) = (i,\ j)$$

Figure 3.4 – Access functions of the statements S_0 and S_1

3.6 Program optimization

Once we have constructed the polyhedral representation of the program, we can modify it and transform the original source code in order to improve its performance. The innermost loops on j in Listing 3.2 can be merged into one in such a way as to keep only one innermost loop on j containing both statements S_0 and S_1. The fusion will optimize the loop nest for data locality and improve the usage of cache memory.

In the original version, the two accesses to $A(i,\ j)$ are not close enough. They are separated from each other by N other access to the array A. This means that, the value of $A(i,\ j)$ the program stores to the cache when accessing it in S_0 would likely be overwritten by another value at the time of the second access to $A(i,\ j)$ in S_1, especially when the size of the arrays A and B exceeds the limited capacity of the cache. The optimized version in Listing 3.3 allows to reuse the value at $A(i,\ j)$ in S_1 while it is still in the cache since the first access to $A(i,\ j)$ in S_0.

```
for(i = 1; i < N; i++)
  for(j = 1; j < N; j++) {
    A[i][j] = 1024; /* S0 */
    B[i][j] = (B[i][j - 1] + A[i][j]) / 2; /* S1 */
  }
```

Listing 3.3 – An optimized version of the for loop nest from Listing 3.2

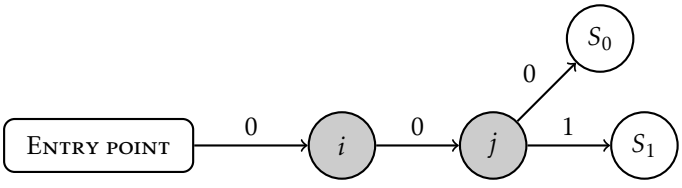
3.7 Code transformation

To apply such kind of transformation to the original source without re-writing it manually is pretty straightforward using the polyhedral representation. A transformation results into a modification of the original program schedule yet without altering its semantics.

In this case, we have to modify the scheduling function of the statement S_1 as proposed in Figure 3.5. After the transformation, S_1 becomes the **second** statement in the **first** innermost loop on j in the outermost loop on i coming first after the entry point of the program.

$$\theta_{S_0}(i,\ j) = (0,\ i,\ \mathbf{0},\ j,\ 0)$$
$$\theta_{S_1}(i,\ j) = (0,\ i,\ \mathbf{0},\ j,\ \mathbf{1})$$

(a) Mathematical representation



(b) Graphical representation

Figure 3.5 – Scheduling of the statements S_0 and S_1 after optimization

3.8 Data dependency

While performing code transformations, we have to be aware of potential dependencies between different memory accesses present in the statements of the program.

Let us consider the source code in Listing 3.3. The read access to $A(i,\ j)$ in statement S_1 depends on the value written to $A(i,\ j)$ by S_0. An example of an illegal code transformation would be to invert the order of these two statements. It would violate the sequential execution order of the statements enforced by the data dependency between the two memory accesses. Eventually, the functionality of the program would be altered. Another data dependency bounds the read access to the location $B(i,\ j-1)$ in the statement S_1 with the write access to the same location in the previous loop iteration.

Both of the data dependencies are of type **Read-After-Write** (RAW). Based on the order of operations, we distinguish two other types of data dependencies, **Write-After-Read** (WAR) and **Write-After-Write** (WAW). **Read-After-Read** (RAR) is not considered to be a data dependency as it does not imply memory modification.

We note $\mathcal{RAW}_0$, the dependency relation between statements S_0 and S_1 involving array A and $\mathcal{RAW}_1$, the dependency relation between the accesses to array B inside statement S_1. The dependencies are graphically represented by a dependency graph in Figure 3.6.

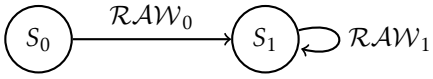


Figure 3.6 – Dependency graph of the source code in Listing 3.3

3.8.1 Dependency polyhedron

There is a data dependency between two memory accesses, if both of them refer to the same location in memory. The pair of involved memory accesses can be either in two different statements as it is in the case of $\mathcal{RAW}_0$, or be part of a single statement like for $\mathcal{RAW}_1$.

A data dependency relation can be described using a dependency polyhedron [17]. Thanks to the latter, we are able to link the iteration vectors corresponding to the pairs of memory access involved in the dependency relation. The polyhedral representation of $\mathcal{RAW}_0$ and $\mathcal{RAW}_1$ is proposed in Figure 3.7.

$$\mathcal{RAW}_0 = \left\{ ((i_0,\ j_0),\ (i_1,\ j_1)) \;\middle|\; \begin{array}{c} (i_0,\ j_0) \in \mathcal{D}_{S_0} \\ (i_1,\ j_1) \in \mathcal{D}_{S_1} \\ \mathcal{W}^A_{S_0}(i_0,\ j_0) = \mathcal{R}^A_{S_1}(i_1,\ j_1) \\ \theta_{S_0}(i_0,\ j_0) \ll \theta_{S_1}(i_1,\ j_1) \end{array} \right\}$$

$$\mathcal{RAW}_1 = \left\{ ((i_0,\ j_0),\ (i_1,\ j_1)) \;\middle|\; \begin{array}{c} (i_0,\ j_0) \in \mathcal{D}_{S_1} \\ (i_1,\ j_1) \in \mathcal{D}_{S_1} \\ \mathcal{W}^B_{S_1}(i_0,\ j_0) = \mathcal{R}^B_{S_1}(i_1,\ j_1) \\ \theta_{S_1}(i_0,\ j_0) \ll \theta_{S_1}(i_1,\ j_1) \end{array} \right\}$$

Figure 3.7 – Dependency polyhedra corresponding to Listing 3.3

In $\mathcal{RAW}_0$, the read access to $A(1,\ 1)$ in iteration $(1,\ 1)$ of statement S_1 is bound to the write access to $A(1,\ 1)$ in iteration $(1,\ 1)$ of statement S_0 and similarly for all couples of iteration vectors of S_0 and S_1 such as $((1,\ 2),\ (1,\ 2))$, $((1,\ 3),\ (1,\ 3))$ … $((N-1,\ N-1))$, $((N-1,\ N-1))$.

In other words, a couple of iteration vectors associated to the statements involved in a dependency relation is bound by the latter, if the access functions of the related memory accesses return the same memory location for these iteration vectors.

In $\mathcal{RAW}_1$, the read access to B in iteration $(1,\ 2)$ of statement S_1 is bound to the write access to B in iteration $(1,\ 1)$. Indeed, the access function $\mathcal{R}^B_{S_1}$ associated to the read access gives $(1,\ 1)$ for the iteration vector $(1,\ 2)$. The access function $\mathcal{W}^B_{S_1}$ associated to the write access returns $(1,\ 1)$ for the corresponding iteration vector which is $(1,\ 1)$. Eventually, we know that the iteration (instance) $(1,\ 2)$ of statement S_1

depends on the iteration (1, 1) of the same statement. As such, the pair of iteration vectors ((1, 1), (1, 2)) is one of those defining the dependency polyhedron $\mathcal{RAW}_1$ in the same way as all couples of iteration vectors of S_1 such as ((1, 2), (1, 3)), ((1, 3), (1, 4))…((N − 3, N − 2)), ((N − 2, N − 1)).

By definition, the order of the memory accesses bound by a dependency relation must be preserved. Therefore, the scheduling vectors of the associated statement instances has to be ordered accordingly. For example, in both $\mathcal{RAW}_0$ and $\mathcal{RAW}_1$, the write access is performed before the read access in the dependency relation.

Another possible way of expressing a dependency relation is to use dependency vectors.

3.8.2 Distance vector

It is often useful to consider the distance in number of statement instances that may separate the couple of memory accesses involved in a dependency relation.

For example, if we intend to improve the usage of cache memory, we would want to minimize the distance between memory accesses in order to increase the probability that a value stored into the cache during the first memory access will remain in the cache until the second access to the same memory location. This prevents redundant accesses to RAM having considerably longer access times compared to cache memory.

The distance vector of a data dependency relation can be determined using associated memory access functions by subtracting the memory access function representing the source of the dependecy from the access function representing the target of the dependency (which depends on the source).

In Figure 3.8, we illustrate the computation of distance vectors for the dependency relations $\mathcal{RAW}_0$ and $\mathcal{RAW}_1$ from the previous section. Definitions of the access functions involved in the computation are given in Figure 3.4.

$$\vec{d}_{\mathcal{RAW}_0} = \mathcal{R}^A_{S_1} - \mathcal{W}^A_{S_0} = (0,0)$$
$$\vec{d}_{\mathcal{RAW}_1} = \mathcal{R}^B_{S_1} - \mathcal{W}^B_{S_1} = (0,1)$$

Figure 3.8 – Distance vectors associated with the dependency relations $\mathcal{RAW}_0$ and $\mathcal{RAW}_1$

3.8.3 Direction vector

A direction vector simply describes the direction of the dependency between a couple of memory accesses. It is based on the sign, either positive or negative, of the elements of the distance vector. Given the definition of the *sign* function in Figure 3.9a, the direction vectors associated to the distance vectors $\vec{d}_{\mathcal{RAW}_0}$ and $\vec{d}_{\mathcal{RAW}_1}$ are defined in Figure 3.9b, where $e_i \in \mathbb{Z}$ and $\vec{v} = (e_1, e_2, \ldots, e_i, \ldots, e_n)$.

$$sign(e_i) = \begin{cases} 1 & \text{if } e_i > 0 \\ -1 & \text{if } e_i < 0 \\ 0 & \text{if } e_i = 0 \end{cases}$$
$$sign(\vec{v}) = (sign(e_1), sign(e_2), \ldots, sign(e_i), \ldots, sign(e_n))$$

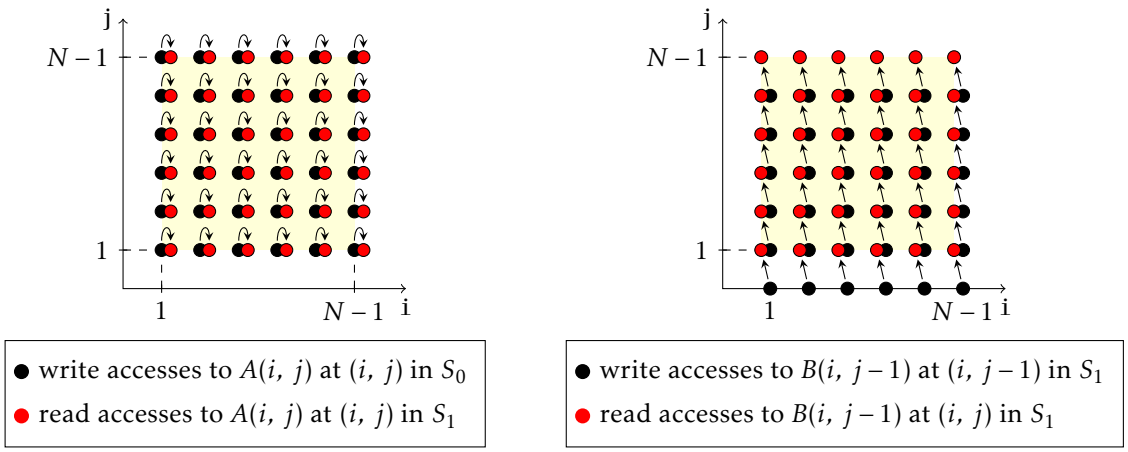
(a) Scalar and vector definitions of the sign function

$$\vec{\sigma}_{\mathcal{RAW}_0} = sign(\vec{d}_{\mathcal{RAW}_0}) = (0,0)$$
$$\vec{\sigma}_{\mathcal{RAW}_1} = sign(\vec{d}_{\mathcal{RAW}_1}) = (0,1)$$

(b) Direction vector definitions

Figure 3.9 – Direction vectors corresponding to the distance vectors $\vec{d}_{\mathcal{RAW}_0}$ and $\vec{d}_{\mathcal{RAW}_1}$

The dependency relations can also be represented graphically as proposed in Figure 3.10, where the arrows represent the dependency relation between different statement instances. They point from the source to the target of the dependency and thus, reflect the direction vectors determined in Figure3.9.



(a) Dependency relation $\mathcal{RAW}_0$

(b) Dependency relation $\mathcal{RAW}_1$

Figure 3.10 – Dependency polyhedrons of $\mathcal{RAW}_0$ and $\mathcal{RAW}_1$

3.8.4 Dependency level

The dependency level determines which loop in a given loop nest carries a particular dependency relation. This information is essential, for example, when parallelizing loops.

It corresponds to the position of the first non-zero element of the related distance vector. If the latter is null, the dependency relation is considered to be loop independent (not carried by any loop within the loop nest). Note that, the first non-zero element of a distance vector must always be positive.

Consequently, $\mathcal{RAW}_0$ is a loop independent dependency relation. On the other hand, $\mathcal{RAW}_1$ is carried by the j-loop based on the associated distance vectors from Figure 3.8. Although, the i-loop does not carry any dependency and can be parallelized, for example, using OpenMP (see Listing 3.4). As such, the iterations of the i-loop can be split among multiple threads and executed in parallel.

```
#pragma omp parallel for
for(i = 1; i < N; i++) {
  for(j = 1; j < N; j++) {
    A[i][j] = 1024; /* S0 */
    B[i][j] = (B[i][j - 1] + A[i][j]) / 2; /* S1 */
  }
}
```

Listing 3.4 – Loop nest from Listing 3.3 automatically parallelized using OpenMP

3.9 Software tools

During the internship, I was manipulating different software tools related to the polyhedral model such as OpenScop [6], an open-source specification defining a common file format and data structures for representing static control parts of a program with the polyhedral model.

OpenScop is used by various polyhedral code manipulation tools as an intermediate representation enabling them to interact. For example, Clan [4, 8] allows to generate OpenScop format from an input source code. The output can be modified using a code optimizer and then re-translated back to the high-level language. This can be done using ClooG [5, 3]. This couple of tools may act as a front-end and a back-end to any polyhedral code manipulation tool.

Candl [7] is another important open-source tools. It is capable of computing data dependencies based on an OpenScop file. Forthermore, this tool can be used to compare two OpenScop representations and determine whether the program portions they depict are semantically equivalent, e. g. having the same functionality.

Chapter 4

XFOR structure

XFOR [14, 16] is a new looping control structure for the C programming language with syntax similar to standard for-loops. It allows to handle multiple for-loops simultaneously and to adjust the scheduling of the associated statements. This is done thanks to two special parameters, the grain and the offset. Using the latter, one can easily and intuitively apply complex code transformations to the loops inside of an XFOR construct. The goal is to improve the usage of cache memory as well as to expose the possibilities of instruction vectorization and parallelization in the loop statements.

4.1 Running example

Let us consider the example picture in Figure 4.1 as well as two filters to be applied successively on the latter. We name them *filter1* and *filter2*. The first filter should cover the entire picture and the second one, only the portion delimited by the black square.

Figure 4.1 – Example picture

Programmatically, the most straightforward way to accomplish this would be to use two successive for-loops iterating over the pixels of the picture stored in a two dimensional array. The first loop would apply *filter1*, and the second one *filter2* (see Listing 4.1). The issue with this solution is that, the picture is unnecessarily scanned twice. A more optimized implementation is proposed in Listing 4.2, where each pixel is loaded from main memory to cache only once when calling the function *filter1*. The value can be rapidly reloaded from the cache and be reused straight away by *filter2*.

```
for(i = 0; i < N; i++)
  for(j = 0; j < M; j++)
    filter1(picture[i][j]);

for(i = a; i < b; i++)
  for(j = c; j < d; j++)
    filter2(picture[i][j]);
```

Listing 4.1 – Trivial solution

```
for(i = 0; i < N; i++) {
  for(j = 0; j < c; j++)
    filter1(image[i][j]); // Area #1

  for(j = c; j < d; j++) {
    filter1(image[i][j]); // Areas #2, #3, #4
    if(i >= a && i < b)
      filter2(image[i][j]); // Area #3 only
  }

  for(j = d; j < M; j++)
    filter1(image[i][j]); // Area #5
}
```

Listing 4.2 – Optimized solution expressed with for-loops

The trivial implementation would not allow such behavior. The capacity of cache memory is generally not high enough to store an entire array of values in it. Thus, before it could be reused by *filter2* in the second loop, the cached pixel value would probably have been already overridden by other pixel values stored into cache because of subsequent iterations of *filter1*. In other words, the accesses to the same memory location by *filter1* at first and then by *filter2* are too far from each other for the value to remain in cache and prevent the reload of the latter from significantly slower main memory.

This is why the optimized version presents a significantly improved cache memory usage compared to the trivial solution. Formally, it presents a better temporal data locality. Although, writing such a source code manually may be tedious and error-prone, especially in case of complex algorithms. As Listing 4.3 shows us, using XFOR we can express this kind of optimization in a much simpler and shorter way.

```
xfor(i0 = 0, i1 = a; i0 < N, i1 < b; i0++, i1++; 1, 1; 0, a)
  xfor(j0 = 0, j1 = c; j0 < M, j1 < d; j0++, j1++; 1, 1; 0, c) {
    0: filter1(image[i0][j0]);
    1: filter2(image[i1][j1]);
  }
```

Listing 4.3 – Optimized solution expressed with XFOR

4.2 Syntax and semantics

The XFOR in Listing 4.3 handles two for-loop nests, where the loop nest on $(i_0,\ j_0)$ such that $(0 \leq i_0 < N,\ 0 \leq j_0 < M)$ applies *filter1* and the nest on $(i_1,\ j_1)$ such that $(a \leq i_1 < b,\ c \leq j_1 < d)$ applies *filter2*. The lower and upper bounds of these loops are the same as for the trivial implementation, yet the execution order of the associated statements has been changed. Using the offset parameter, we have adjusted the scheduling of the latter in such a way as to match the optimized implementation in Figure 4.2.

In the general case, for each for-loop handled within a given XFOR, we need to initialize the iterator value, define the upper bound of the loop using a conditional expression and the associated increment value like in C for-loops. The values at the end of the XFOR specifiers represent the grain and offset parameters.

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

Listing 4.4 – General syntax of XFOR

Listing 4.4 defines the general syntax of the XFOR structure, where *index* denotes a loop iterator, e. g. i or i_0; *expr* as well as *offset* must represent valid affine expressions - functions of enclosing indices, invariable parameters or constants (see Section 3.2); *relop* denotes a relational operator such as $<$, $\leq$, $\geq$ or $>$; *incr* and *grain* are integer constants, where $grain \geq 1$.

4.2.1 Grain

The grain parameter defines the execution frequency of one for-loop within an XFOR relatively to the others. Consider the XFOR in Listing 4.5 and let us say that we want to reduce the execution frequency of the second for-loop associated to index *i1* by 2. To do this, we change the grain value of this loop to 2. Actually, by reducing the execution frequency of the second loop, we increase the frequency of the first one. Thus, for each iteration of the loop on *i1*, 2 iterations of the loop on *i0* are executed.

```
xfor(i0 = 0,  i1 = 0;
     i0 < 10, i1 < 5;
     i0++,    i1++;
     1,       2; // Grains
     0,       0 // Offsets
) {
  0: foo(i0); // S0
  1: bar(i1); // S1
}
```

Listing 4.5 – Example of the grain parameter application

For better understanding, in Figure 4.2 we analyze the iteration domain of the XFOR before and after the frequency reduction. Figure 4.2a shows us that, without reducing the frequency of the second for-loop, the XFOR executes at first, 5 iterations of both statements followed by 5 remaining iterations of S_0 only. Nevertheless, when the frequency of the loop on i_1 is reduced, each iteration of S_1 is followed by 2 iterations of S_0. The result is represented in Figure 4.2b.

(a) Without frequency reduction (b) With frequency reduction

Figure 4.2 – Iteration domain of the XFOR in Listing 4.5

4.2.2 Offset

The offset parameter allows to shift one for-loop within an XFOR relatively to the others. Consider the XFOR in Listing 4.6 which handles two for-loops, one iterating on i_0 and the other on i_1. We want the second loop to start executing only after 5 iterations of the first one. To do this, we set the offset value of the second loop to 4.

```
xfor(i0 = 0,  i1 = 0;
     i0 < 10, i1 < 5;
     i0++,    i1++;
     1,       1; // Grains
     0,       4 // Offsets
) {
  0: foo(i0); // S0
  1: bar(i1); // S1
}
```

Listing 4.6 – Example of the offset parameter application

For better understanding, in Figure 4.3 we analyze the iteration domain of the XFOR before and after the shifting. Figure 4.3a shows us that, without shifting the second for-loop, the XFOR executes at first, 5 iterations of both statements followed by 5 remaining iterations of S_0. On the contrary, when the loop on i_1 is shifted by 4 iterations, at first we execute 5 iterations of only S_0 followed by 5 iterations of both S_0 and S_1 ending with the remaining iteration of S_0. The result is represented in Figure 4.3b.

(a) Without shifting (b) With shifting

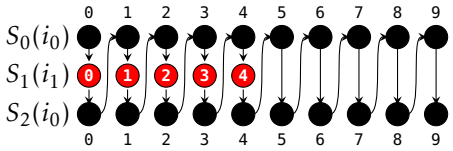
Figure 4.3 – Iteration domain of the XFOR loop in Listing 4.6

4.2.3 Body

Each statement inside of an XFOR body must be associated to one of the for-loops handled by the XFOR. This is done by means of the *label* value. Counting from zero, it corresponds to the position of the loop iterator in the XFOR specifier. So, statements labeled with 0 are associated to the first loop, statements labeled with 1 to the second loop and so on. Note that, the label value does not indicate the order of execution of the statements! They follow the lexicographic (textual) order in the source code. Consequently, multiple occurrences of the same label are possible (see Figure 4.4).

```
xfor(i0 = 0,  i1 = 0;
     i0 < 10, i1 < 5;
     i0++,    i1++;
     1,       1; // Grains
     0,       0 // Offsets
) {
  0: foo(i0); // S0
  1: bar(i1); // S1
  0: boo(i0); // S2
}
```

(a) XFOR with two occurrences of label 0



(b) Corresponding iteration domain

Figure 4.4 – Example of multiple occurrences of the same label value in XFOR

Nested XFOR constructs are also possible. Each XFOR in the nest must count the same number of loops as the outermost XFOR. At each level, statements may access the iterator defined in the associated XFOR matching their label, as well as all the iterators from parent XFORs referring to the same label, in the same way like in case of nested for-loops. Statements of a given loop shall not reference iterators of the other loops. Moreover, values of iterators can not be modified within the loop bodies.

In order to illustrate how nested XFOR works, we remind the example from Section 4.1. The trivial implementation from Listing 4.1 features two separate for-loops executed consecutively. As both of these loops iterate over the same picture, in the optimized version, we fusioned the loops into one. Still, the function *filter2* should be applied only on a subpart of the picture. Therefore, the execution of the second loop is limited to the values of i from a to $b-1$ and the values of j from c to $d-1$. The XFOR implementation in Listing 4.3 allows us to express this code transformation thanks the offset parameter. Indeed, with XFOR we have, at first, merged both of the for-loops, then using the offset parameter we have shifted the second loop by a on i and by c on j. Like in previous examples, we represent the resulting iteration domain of the XFOR graphically in Figure 4.5.

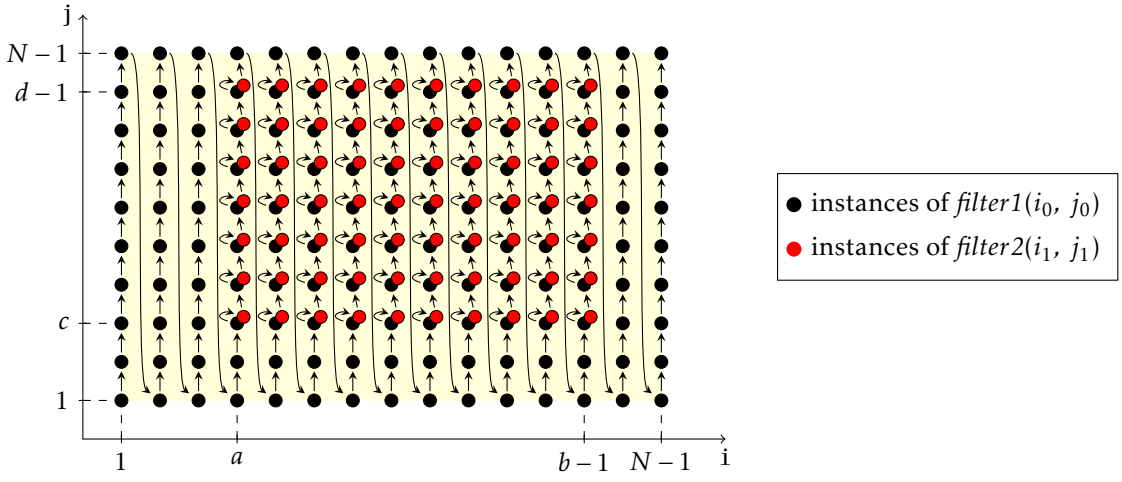


Figure 4.5 – Iteration domain of the XFOR in Listing 4.3

4.3 Building polyhedral representation

Being a custom made control structure, XFOR is not specified in the C language standard. Therefore, it is not recognized and can not be compiled using ordinary C production compilers, e. g. gcc. Although, using the polyhedral representation as an intermediate, XFOR can be expressed in function of standard for-loops and if-conditionnals, perfectly understandable by any C compiler.

4.3.1 Common iteration domain

Each statement within an XFOR body defines its own iteration domain delimited by the lower and the upper bounds of the associated for-loop handled by the XFOR. The disjoint union of these individual domains forms the common iteration domain covering the entire XFOR expressed in function of a single referential iterator. Within the union, the original iteration domains have to be expressed using the referential iterator, instead of their original indices. Formally, we use the formula in Figure 4.6 to recompute the original loop bounds relatively to the common iteration domain. Considering an XFOR handling k for-loops, let us denote $OldLower_i$ and $OldUpper_i$ the original lower and upper bounds of the i^{th} for-loop of the XFOR with $0 < i \leq k$. Accordingly, $NewLower_i$ and $NewUpper_i$ are the bounds projected into the common iteration domain. $Grain_i$ and $Offset_i$ denote the associated XFOR parameter values and $lcm(Grain_1, \ldots, Grain_k)$ matches the least common multiple of all the grains of the XFOR.

$$NewLower_i = Offset_i$$
$$NewUpper_i = (OldUpper_i - OldLower_i) \,/\, (lcm(Grain_1, \ldots, Grain_k) \,/\, Grain_i) + Offset_i$$

Figure 4.6 – Projection of the original loop bounds into the common iteration domain

Let us illustrate the computation of the common iteration domain on the examples from Section 4.2. First, we consider the XFOR in Listing 4.5. The original iteration domains of the two statements in the XFOR are defined in Figure 4.7a. According to the previously defined formula, the lower bounds of the projected iteration domains equal 0 so as both of the offset values. The upper bound of $\mathcal{D}_{S_0}$ equals $(9-0) \,/\, (lcm(1,\ 2) \,/\, 1) + 0 = 4$ and the upper bound of $\mathcal{D}_{S_1}$ equals $(4-0) \,/\, (lcm(1,\ 2) \,/\, 2) + 0 = 4$ which is the same as $\mathcal{D}_{S_0}$.

$$\mathcal{D}_{S_0} = \{\, i_0 \in \mathbb{Z}^1 \mid 0 \leq i_0 \leq 9 \,\}$$
$$\mathcal{D}_{S_1} = \{\, i_1 \in \mathbb{Z}^1 \mid 0 \leq i_1 \leq 4 \,\}$$

(a) Original iteration domains

$$\mathcal{D}_{S_0} = \{\, i \in \mathbb{Z}^1 \mid 0 \leq i \leq 4 \,\}$$
$$\mathcal{D}_{S_1} = \{\, i \in \mathbb{Z}^1 \mid 0 \leq i \leq 4 \,\}$$

(b) Projected iteration domains

Figure 4.7 – Projection of the iteration domains associated to the XFOR in Listing 4.5

In Figure 4.8a, we define the original iteration domains of the statements in the XFOR in Listing 4.6. The lower bound of the projected $\mathcal{D}_{S_0}$ is simply 0 so as the corresponding offset value. Its upper bound equals $(9-0) \,/\, (lcm(1,\ 1) \,/\, 1) + 0 = 9$. In case of $\mathcal{D}_{S_1}$, the lower bound corresponds to the associated offset value, which is 4. The upper bound equals $(4-0) \,/\, (lcm(1,\ 1) \,/\, 1) + 4 = 8$.

$$\mathcal{D}_{S_0} = \{\, i_0 \in \mathbb{Z}^1 \mid 0 \leq i_0 \leq 9 \,\}$$
$$\mathcal{D}_{S_1} = \{\, i_1 \in \mathbb{Z}^1 \mid 0 \leq i_1 \leq 4 \,\}$$

(a) Original iteration domains

$$\mathcal{D}_{S_0} = \{\, i \in \mathbb{Z}^1 \mid 0 \leq i \leq 9 \,\}$$
$$\mathcal{D}_{S_1} = \{\, i \in \mathbb{Z}^1 \mid 4 \leq i \leq 8 \,\}$$

(b) Projected iteration domains

Figure 4.8 – Projection of the iteration domains associated to the XFOR in Listing 4.6

```
for(i = 0; i <= 4; i++) {
  foo(2 * i); // Takes even values from 0 to 9
  bar(i); // Takes all values from 0 to 4
  foo(2 * i + 1); // Takes odd values from 0 to 9
}
```

Listing 4.7 – for-loop equivalent implementation of the XFOR in Listing 4.5

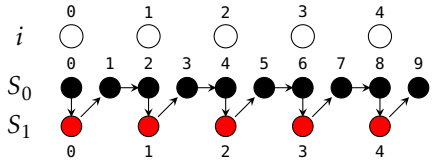
4.3.2 Global schedules

Eventually, we must associate a schedule function to each statement inside of an XFOR. Let us remind the one in Listing 4.5. According to the lexicographic order of the statements in this XFOR, we define their scheduling functions in Figure 4.9a. The statement S_0 counts two scheduling functions as its execution frequency was increased by 2 consequently to the reduction of the frequency of the loop associated with S_1 by the same factor. In other words, the iteration domain of S_0 was unrolled by 2.

Figure 4.9b represents the iteration domain and the execution order of the statements graphically. Listing 4.7 features the for-loop equivalent generated source code based on the common iteration domain and the scheduling functions we have determined.

$$\theta_{S_{0_0}}(i) = (0,\ i,\ 0)$$
$$\theta_{S_1}(i) = (0,\ i,\ 1)$$
$$\theta_{S_{0_1}}(i) = (0,\ i,\ 2)$$

(a) Scheduling functions of S_0 and S_1



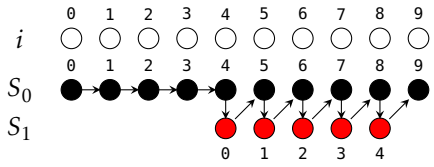
(b) Graphical scheme

Figure 4.9 – Polyhedral representation of the XFOR in Listing 4.5

The scheduling functions of the statements of the XFOR in Listing 4.6 are defined in Figure 4.10a. The shifting of the second loop in the XFOR relatively to the first one by 5 iterations is translated by simply adding the offset value of the second loop to the dimension of the scheduling vector linking the statement S_1 to the iterator. Figure 4.10b features the graphical polyhedral representation of the statements. Listing 4.8 shows the for-loop equivalent generated source code based on the common iteration domain and the scheduling functions we have determined.

$$\theta_{S_0}(i) = (0,\ i,\ 0)$$
$$\theta_{S_1}(i) = (0,\ i+4,\ 1)$$

(a) Scheduling functions of S_0 and S_1



(b) Graphical scheme

Figure 4.10 – Polyhedral representation of the XFOR in Listing 4.6

```
for(i = 0; i < 10; i++) {
  foo(i);
  if(i >= 4 && i < 9) // 'bar' has only 5 iterations.
    bar(i - 4); // Values expected by 'bar' are in range from 0 to 4!
}
```

Listing 4.8 – for-loop equivalent implementation of the XFOR in Listing 4.6

4.4 Target and Execution-Aware (TEA) programming

The XFOR structure is particularly useful for writing programs optimized in such a way as to make use of underlying computer architecture as much as possible. It allows the programmer to easily and intuitively perform cache memory usage optimizations as well as to expose the parallelism in his program by adjusting the way the latter is executed.

4.4.1 Data reuse distance minimization

In Section 3.8.2, we have seen why the minimization of the distance between a couple of memory accesses to the same memory location is important when improving the cache memory usage and the overall performance of a program. This manipulation is known as data reuse distance minimization.

The motivation example in Section 4.1 allowed us to demonstrate how the data reuse distance minimization can be easily performed thanks to the XFOR structure. Let us consider a more detailed example. We focus on the implementation of the one-dimensional Jacobi stencil `jacobi-1d` in Listing 4.9.

```
for (i = 1; i < _PB_N - 1; i++)
  B[i] = 0.33333 * (A[i - 1] + A[i] + A[i + 1]); // S0
for (i = 1; i < _PB_N - 1; i++)
  A[i] = B[i]; // S1
```

Listing 4.9 – Trivial implementation of `jacobi-1d`

Both for-loops can be fusioned in order to reduce the data reuse distances between the accesses to arrays A and B in statements S_0 and S_1. Although, the second loop must be shifted at least by one relatively to the first loop to respect the Write-After-Read dependency between the access to $A(i-1)$ in S_0 and the access to $A(i)$ in S_1. Otherwise, $A(i)$ would be written to by S_1 before it could be read as $A(i-1)$ in the following iteration of S_0. Listing 4.10 features the corresponding XFOR implementation.

```
xfor (i0 = 1, i1 = 1; i0 < _PB_N - 1, i1 < _PB_N - 1; i0++, i1++; 1, 1; 0, 1) {
  /* S0 */ 0: B[i0] = 0.33333 * (A[i0 - 1] + A[i0] + A[i0 + 1]);
  /* S1 */ 1: A[i1] = B[i1];
}
```

Listing 4.10 – XFOR implementation of `jacobi-1d` with minimized data reuse distances

Table 4.1 shows us that, the XFOR version with minimized data reuse distances is 49 % faster than the trivial implementation. Note that, we have conducted our experiments on an Intel Core i5 M520 processor having 4 logical cores each clocked at 2.40 GHz. We used the Intel C Compiler (icc) of version 19.0.4.243 launched with options `-O3` and `-march=native` with disabled automatic vectorization (`-no-vec`) running on Debian 10 (codename Buster) with Linux kernel of version 4.19.

	Trivial implementation	XFOR implementation	Overall gain
Execution time	1.03 s	0.69 s	**49%**

Table 4.1 – Performance comparison between the trivial and the XFOR implementation of `jacobi-1d`

4.4.2 Loop vectorization

Nevertheless, minimizing data reuse distances is not always the most optimal approach, especially when attempting to make use of instruction vectorization. Modern processing units feature several types of vectorization technologies. The SIMD *(Single Instruction Multiple Data)* vectorization units are capable of executing multiple instances of a single instruction on a vector of multiple equally spaced data in memory. Current general purpose processors have 128 or 256 bits-wide vector registers. For loops, this means that, multiple iterations can be executed in parallel.

The XFOR-optimized implementation of `jacobi-1d` in Listing 4.10 exhibits the minimal offset values respecting the data dependencies. Our processor has 128 bit-wide vector registers and arrays A and B

contain both double precision floating-point values having 64 bits each. So one vector register can hold up to 2 array elements.

The data dependency in our XFOR does not allow us to execute 2 iterations in parallel. Indeed, each couple of successive iterations is bound by the dependency. In this case, the reuse distances were minimized too much. To make automatic vectorization profitable, we need to set the offset of the for-loop on j_1 to a value greater than 2 which is the capacity of vector registers. During our tests, we achieved the best performance setting the offset value to 7 (see Listing 4.11).

```
xfor (i0 = 1, i1 = 1; i0 < _PB_N - 1, i1 < _PB_N - 1; i0++, i1++; 1, 1; 0, 7) {
  /* S0 */ 0: B[i0] = 0.33333 * (A[i0 - 1] + A[i0] + A[i0 + 1]);
  /* S1 */ 1: A[i1] = B[i1];
}
```

Listing 4.11 – XFOR implementation of `jacobi-1d` aware of automatic vectorization

This time we enabled automatic vectorization in icc launching it without the `-no-vec` option. We have also considered the initial trivial implementation of `jacobi-1d`, the XFOR implementation minimizing the data reuse distances from Listing 4.10 as well as the XFOR version aware of automatic vectorization from Listing 4.11. Table 4.2 shows the differences among these three implementations in terms of performance.

	Trivial implementation	XFOR version with minimal offset	XFOR version with optimal offset	Overall gain
Offset	_PB_N - 2	1	7	
Execution time	0.93 s	12.50 s	**0.68 s**	**37 %**

Table 4.2 – Performance analysis of the vectorized implementations of `jacobi-1d`

We can confirm that, minimizing the data reuse distance as far as possible may even cause the program to have considerably poorer performance compared to the initial implementation when automatic vectorization is enabled. This is why a proper choice of the offset value is important in order to take advantage of automatic vectorization when available.

In the trivial implementation, the statements are split into two independent for-loops which makes vectorization profitable for both of them since the dependency issue is not present. Although, as the for-loops are executed sequentially, automatic vectorization does not improve the program performance as much as in the case of our XFOR-optimized implementation which remains more efficient thanks to the loop fusion we have performed.

4.4.3 Loop parallelization

Although being very effective, vectorization has an important drawback. It is a fine-grained operation and only elementary instructions are vectorizable. Thus, in case of nested loops it is generally applied to the innermost loop. During the internship, we were studying the possibility to apply a similar concept on any loop level using threads instead of vector registers. In Chapter 6, we present and compare two different approaches of parallelization of the XFOR structure.

4.5 Software tools

4.5.1 IBB compiler

Source codes containing raw XFOR constructs can not be compiled directly using ordinary production compilers like gcc, icc, etc. Indeed, the structure is not a standard part of the C language [2]. Therefore, source codes containing XFOR constructs must be first translated into equivalent codes using only common C language structures such as for-loops and if-conditionnals.

IBB "Iterate, But Better!" is a source-to-source compiler allowing to perform such translation. It takes the source code of an XFOR program as input and produces an equivalent C source code as output expressed

using only standard programming constructs. The usage of the compiler is very straightforward and comparable to gcc:

```
IBB xfor_program.c -o for_equivalent.c
```

The compiler begins by parsing the input source code. For each XFOR construct it encounters, IBB builds the corresponding polyhedral representation as described in Section 4.3 using the Clan software (see Section 3.9). Based on the polyhedral representation and using ClooG (see Section 3.9), the compiler produces the C source code corresponding to the given XFOR. The rest of the input source code without XFOR is simply recopied to the output file.

4.5.2 XFOR-WIZARD

XFOR-WIZARD is an integrated development environment meant to make writing of XFOR programs easier. It has two main functions. In the first place, XFOR-WIZARD allows to conveniently edit XFOR source codes by providing syntax highlighting functionality as well as the integration of the IBB compiler and the polyhedral optimization framework Pluto [10].

However, the most important are the capabilities of XFOR-WIZARD in terms of assistance to the programmer in transforming existing for-loop nests into XFOR constructs in order to optimize them as far as possible. XFOR-WIZARD features the automatic conversion of for-loop nests into equivalent XFOR nests. It is then possible to adjust the parameters of the latter with the software capable of telling the programmer whether his modifications are correct or not in terms of respect of the semantics of the program and potential data dependencies.

Figure 4.11 illustrates this functionality. In this case, the programmer choose an incorrect offset value for the second outermost loop in the featured XFOR. By clicking on the 'Verify dependencies' button, the program is able to detect the dependency issue, warn the programmer and generate the dependecy violation graph.

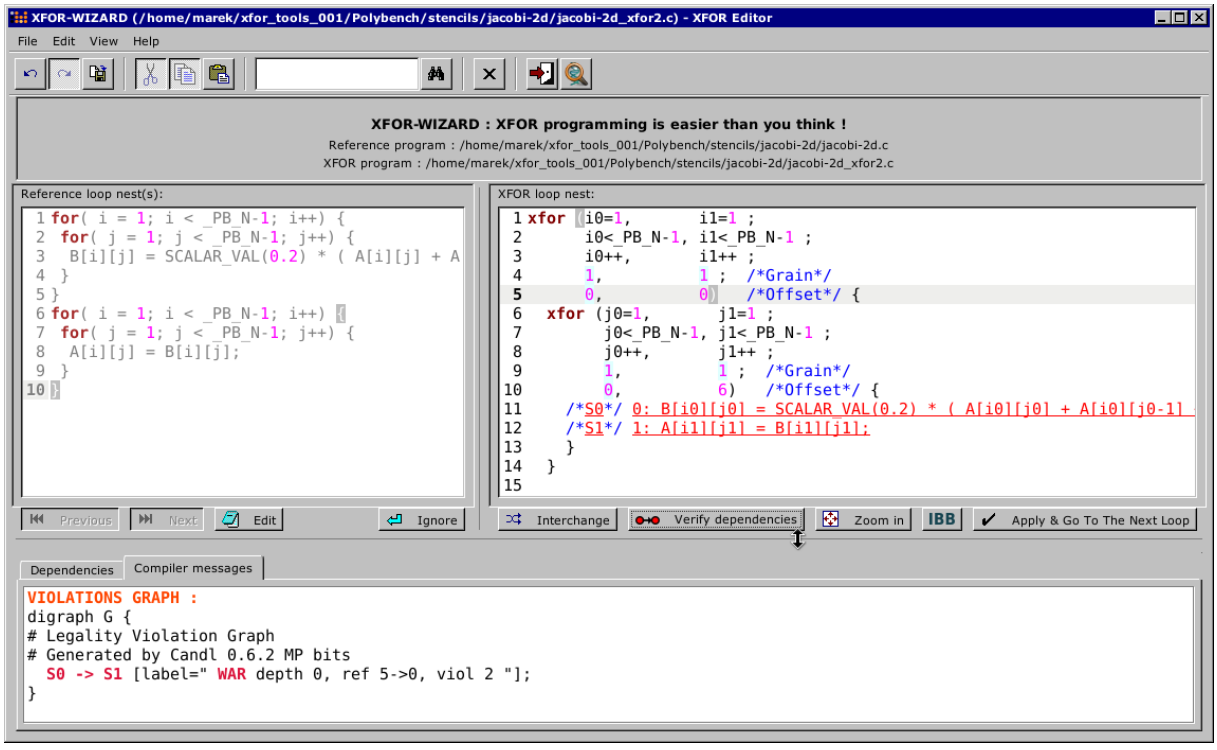


FIGURE 4.11 – Assisted modifcation of an XFOR program with XFOR-WIZARD

Chapter 5

Integration into the Clang/LLVM compiler

This internship made me work on two main objectives regarding the evolution of the XFOR structure. In Section 4.5.1, we have seen that, the only current option to compile an XFOR program is to transform all the XFOR constructs it contains into basic for-loops and if-conditionnals with the IBB compiler before compiling the program using a standard production C compiler such as gcc, icc or Clang/LLVM. Hence our motivation is to integrate XFOR into one of these in order to be able to compile XFOR programs directly and promote the structure within the C programmer community.

5.1 Compiler operation

Compilation includes multiple stages. At first, the lexical analysis transforms the input source code written in a high level programming language into a flow of keywords (`if`, `while`, `char`, `return`, ...), operators (+, -, >, ...), numbers, strings and so on, commonly referred to as tokens. This flow of tokens is then parsed during the semantic analysis which verifies the syntax of the program. In other words, the order of the tokens must match the grammar rules of the programming language. For example in C, `if` must be followed by a conditional expression enclosed in parenthesis and each statement must finish with a semicolon.

The product of semantic analysis is an Abstract Syntax Tree (AST). Being a type of program representation (see Chapter 3), the AST depicts the program in form of a tree, where the leaves are tokens that form nodes representing expressions, structures and functions. The root corresponds to the entire program.

This representation simplifies the manipulation of the program and its translation into the intermediate representation which can be seen as a type of low level language. It allows to express high level language constructs using only a limited set of elementary instructions such as conditional branches, jumps (similar to `goto`), system calls, memory manipulation instructions for loading, storing and moving data throughout memory locations, etc. Unlike assembly languages, it does not feature target CPU-specific instructions. It is language independent. Intermediate representation is designed to help in the analysis of the program for potential further processing such as code-improving transformations.

In a compiler, the above phases correspond to the front-end. The back-end covers the translation of the intermediate representation into the assembly and finally to the machine language of the target computer architecture. The general operations of a compiler are illustrated in Figure 5.1.

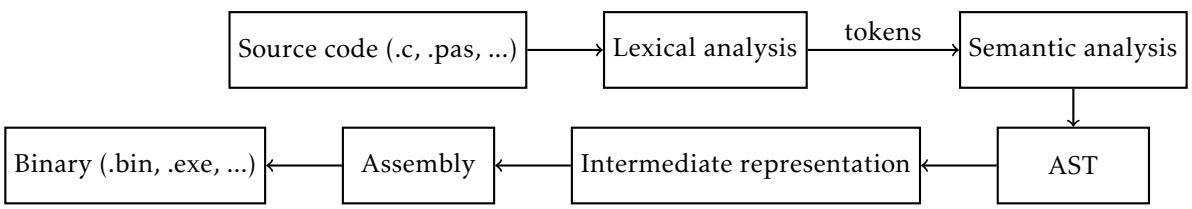


Figure 5.1 – General operation of a compiler

5.2 Clang/LLVM

Clang/LLVM is a widely used and open-source C and C++ compiler written in C++. It provides a dedicated programming interface allowing us to extend it by writing our own compilation passes.

Actually, Clang is a front-end for LLVM. It parses C or C++ source code on input and translates it into the intermediate representation (IR) used by LLVM. An example of LLVM IR is provided in Figure 5.2.

```c
int main(void) {
  int i;

  for(i = 0; i < 10; i++) {
    printf("%d\n", i);
  }

  return 0;
}
```

(a) High level C source code

```
; beginning of the main function
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4 ; allocate the return value
  %i = alloca i32, align 4 ; allocate the iterator variable 'i'
  store i32 0, i32* %retval, align 4 ; pick 0 as return value
  store i32 0, i32* %i, align 4 ; store 0 to 'i'
  br label %for.cond ; jump to the for-loop condition

for.cond: ; for-loop condition block
  %0 = load i32, i32* %i, align 4 ; load 'i' value into '%0'
  %cmp = icmp slt i32 %0, 10 ; check if '%0' is less than '10'

  ; if yes, go to the loop body, else jump to the loop end
  br i1 %cmp, label %for.body, label %for.end

for.body: ; for-loop body block
  %1 = load i32, i32* %i, align 4 ; load 'i' value into '%1'

  ; call the 'printf' function (simplified)
  %call = call i32 (i8*, ...) @printf(... i32 %1)
  br label %for.inc ; jump to the incrementation of 'i'

for.inc: ; 'i' increment block
  %2 = load i32, i32* %i, align 4 ; load 'i' value into '%2'
  %inc = add nsw i32 %2, 1 ; add '1' to '%2'
  store i32 %inc, i32* %i, align 4 ; store the result into 'i'
  br label %for.cond ; jump to the loop condition

for.end: ; end of the for-loop
  ret i32 0 ; return 0
}
```

(b) Annotated LLVM IR equivalent

Figure 5.2 – Example of the LLVM IR generated for a simple program written in C

LLVM is a compiler framework capable of applying series of custom compilation passes to the IR which is eventually translated into the machine binary form corresponding to the target computer architecture (see Figure 5.3). The passes often perform some sort of code analysis, optimization or transformation. A simple example would be a pass that inserts a call to a print function after every variable assignment found in the program in order to print the value being assigned.

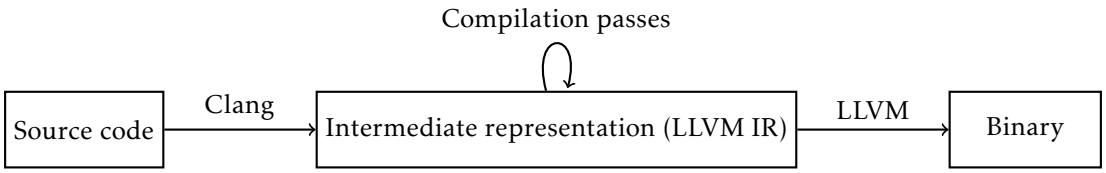


Figure 5.3 – Simplified operation chart of Clang and LLVM

Note that, Clang is not the only front-end for LLVM. There are numerous compilers for a number of different programming languages using LLVM as back-end. Indeed, thanks to the above mentionned programming interface of LLVM, one can use it to develop the front-end for a custom language and build a fully featured compiler.

During this mission I had to intervene in various phases of the compilation process. In the first stage, I had to extend the lexical and the semantic analyzers of Clang to make it understand and correctly parse source codes with XFOR constructs. The second stage consisted of translating the AST provided by the Clang parser into LLVM IR used to produce the final executable file.

5.3 Extending Clang

At the beginning of the internship, I was studying the source code of Clang and LLVM and learning how to proceed with extensions. Although being very limited, the official documentation [30] helped me to identify the parts of the source code to focus on. Figure 5.4 illustrates the most important portions of the structure of the source files of Clang.

Clang is composed of multiple modules dedicated to every compilation phase handled by the front-end, e. g. parsing, IR code generation or AST construction. They are easily recognizable in the Clang's sources structure too. I begun by focusing on the modules involved in lexical analysis.

- clang
  - include/clang
    - AST
    - Basic
    - CodeGen
    - Parse
    - Sema
    - ...
  - lib
    - AST
    - Basic
    - CodeGen
    - External
    - Parse
    - Sema
    - ...
  - test
  - xfortest
  - ...
  - CMakeLists.txt
  - ...

Figure 5.4 – Excerpt of the structure of the source files of Clang

5.3.1 Lexical analysis

Together with the XFOR structure, we have introduced a new keyword to the C programming language. Therefore, I have implemented the recognition of the `xfor` token into the lexical analyzer of Clang. The latter simply uses a specific file where known tokens are defined. I have simply added the line `KEYWORD(xfor, KEYALL)` to file `/include/clang/Basic/TokenKinds.def`.

5.3.2 Semantic analysis

A successful semantic analysis of XFOR loop nests requires a dedicated parsing method to be implemented to verify that the input flow of tokens from the lexical analyzer follows the syntax of the XFOR structure as defined in Section 4.2.

XFOR is a new kind of statement and in Clang, the statement parsing methods are implemented in file `/lib/Parse/ParseStmt.cpp`. Following the naming convention utilized in the source code, I have added the XFOR parsing method named `ParseXforStatement` to the file in question.

Unlike bison (yacc) parsers, Clang parser does not use a grammar file containing the set of syntax rules corresponding to the C programming language. In Clang's parsing methods, each token from the input flow is processed manually. Let us explain that on the XFOR parsing method implemented in our version of Clang.

Whenever the `xfor` keyword is encountered in the input source code, our XFOR parsing method is triggered. Then, it successively consumes every token following the `xfor` keyword until it reaches the end of the current XFOR statement represented by the last closing curly bracket.

The parsing method verifies whether the tokens forming the XFOR are coming in the correct order. If yes, the method discards the current and moves to the next token. Otherwise, the method throws a compilation error precising the nature as well as the location (concerned line and column number) of the problem in the source code. Clang has a special module for error reporting, that I provided with custom error types and messages to be used when an invalid XFOR loop nest is being parsed. The source

file involved is `/include/clang/Basic/DiagnosticParseKinds.td`. See an example of custom diagnostic message below:

```
def err_xfor_expected_colon_after_label :
  Error<"expected ':' after label in 'xfor' statement block">;
```

Often, a token is supposed to be followed by an expression. For example, after the `xfor` keyword there must be an opening parenthesis followed by a string corresponding to the name of the first index of the XFOR. Then, an equal sign (=) indicates that, the next token will be a beginning of the affine expression corresponding to the lower bound of the first for-loop handled by the XFOR (e. g. `i0= p + 1`). In this case, the parsing method consumes the equal sign and calls an existing method for parsing generic arithmetic expressions. Finally, I developed additional verification methods to verify whether parsed expressions corresponding to either loop bounds or offset parameter form valid affine expressions respecting the constraints of the polyhedral model (see Section 3.2).

Parsing of an XFOR loop nest is done in several steps. At first, we parse the XFOR specifier (e. g. `xfor(i0= 0, i1= 0; i0< 10, ...)`. The very first information obtained are the names and initial values of the loop indices (e. g. `i0= 0, i1= 0`). Then, the associated upper bounds, increment values, grain and offset expressions are parsed. Among other minor verifications, the parsing method checks also the count of the latter which must match the count of previously parsed indices.

XFOR body may contain either another nested XFOR or the statements associated to the for-loops defined in the specifier by the means of the label values preceeding the statement blocks. If a nested XFOR is detected, the statement label is set to −1 and the parsing method is called recursively.

Eventually, the XFOR parsing method gathers all the information about the XFOR loop nest necessary to build the corresponding AST node. To do this, I defined a new type of AST statement node for the XFOR structure in file `/include/clang/AST/Stmt.h` by implementing a new class `XforStmt` which inherits from the generic statement class `Stmt`. Moreover, I have added to the latter a new member in order to be able to label parsed statements.

Nevertheless, before actually building the XFOR AST node, it was necessary to implement a method performing the transfer of the data provided by the parsing method into the data structures of the XFOR AST node class. Once again, following the naming convention, I implemented this method under the name `ActOnXforStmt` in file `/lib/Sema/SemaStmt.cpp`. It is this method which actually produces the AST node corresponding to the currently parsed XFOR loop nest. Eventually, this node is integrated into the AST representing the entire input program. In a simplified manner, Figure 5.5 illustrates what kind of information needs to be stored in an XFOR AST node to be able to proceed with IR code generation used to produce the final binary executable file.

```
xfor(i0 = 0,  i1 = 0;
     i0 < 10, i1 <= 5;
     i0++,    i1++;
     1,       1;
     0,       PARAM/10
) {
  0: foo(i0);
  1: bar(i1);
}
```

(a) Illustrative XFOR loop

- XFOR statement
  - string ("i0") [1st index]
  - string ("i1") [2nd index]
  - affine expression (0) [lower bound of i_0]
  - affine expression (0) [lower bound of i_1]
  - relational operator (<) [corresponding to i_0]
  - relational operator (<=) [corresponding to i_1]
  - affine expression (10) [upper bound of i_0]
  - affine expression (5) [upper bound of i_1]
  - integer (1) [increment value of i_0]
  - integer (1) [increment value of i_1]
  - integer (1) [grain of i_0]
  - integer (1) [grain of i_1]
  - affine expression (0) [offset of i_0]
  - affine expression (PARAM / 10) [offset of i_1]
  - statement block (label: 0) [statement associated to the loop on i_0]
  - statement block (label: 1) [statement associated to the loop on i_1]

(b) Corresponding Clang AST node (simplified)

FIGURE 5.5 – Information kept by an XFOR AST node used in our version of Clang

5.4 Translation to LLVM IR

Once our version of Clang/LLVM became capable of building the AST representation of XFOR programs, I finally started to focus on the transformation of AST to the LLVM intermediate representation.

In Section 4.3, we have seen that, using the polyhedral representation, XFOR constructs can be expressed with solely elementary C language structures such as for-loops and if-conditionnals understandable by any compiler. In our version of Clang/LLVM, we also use the polyhedral model but, to transform the AST produced by Clang into the LLVM intermediate representation.

5.4.1 Dealing with mathematics

The first step is to build the polyhedral representation corresponding to every XFOR node in the AST generated by the Clang parser. To represent $\mathbb{Z}$-polyhedra and manipulate mathematical entities such as sets, maps and functions, I used the Integer Set Library (isl) [31].

Before being able to actually take advantage of the library, I had to include it into the Clang sources and the CMake system used to build the latter. I placed the sources of isl into directory `/lib/External` which is meant for external software and libraries. Eventually, I modified the configuration files of the CMake build system in order to get the library built and linked together with the sources of Clang. To the main `CMakeLists.txt` file, I added the location of the source files of isl. Then, I prepared an individual `CMakeLists.txt` file for the library itself. Also, the CMake configuration files of every module where the library is used had to be updated. At the end, I was able to use isl by simply including its header files using the C++ `#include` directive.

Unlike in IBB (see Section 4.5.1), we use only the isl library to deal with polyhedral representations. Including multiple external libraries would complexify the maintenance of our version of Clang/LLVM.

5.4.2 Computing the polyhedral representation

The polyhedral representation of an XFOR nest is computed in two stages. The iteration domains of the for-loops handled by the XFOR are built at first within the XFOR parsing method `ParseXforStmt` (see Section 5.3.2). I developed dedicated methods for transforming the affine expressions of loop bounds and offsets parsed in Clang format into data structures used by the isl library.

Furthermore, in Section 4.2.3, we have seen that, at each level in a nested XFOR loop, statements may access the iterator defined in the associated XFOR matching their label, as well as all the iterators from parent XFORs referring to the same label. This must be taken into account while computing the iteration domains of the for-loops handled within the nested XFOR, as references to the iterators from upper levels may appear in the definitions of loop bounds or offsets of inner XFOR loops in the nest. In other words, the inner XFOR loops in the nest must be aware of the indices defined in upper levels of the XFOR nest. For this, I created dedicated data structures to pass necessary information between the nest levels during the parsing.

In order to not overcharge the implementation of the parsing method, the corresponding scheduling functions are determined during the creation of the XFOR AST node in the method `ActOnXforStmt` (see Section 5.3.2).

Nevertheless, as we have seen in Section 3.4, the number of parameters of the scheduling functions of all the statements in an XFOR loop nest depends on the number of levels of the latter. So, the final count of XFOR nest levels should be known at the moment of scheduling functions computation. Although, this is not the case in our implementation. But, we add the missing parameters once the XFOR AST is complete, more exactly, at the moment of transforming the AST to the LLVM IR.

5.4.3 From polyhedra to intermediate representation

The LLVM IR code generation method `EmitXforStmt` for the XFOR structure is implemented in file `/lib/CodeGen/CGStmt.cpp`. The first step to do in this method, is to use the polyhedral representation of the parsed XFOR loop nest to build a partial AST using the isl library which expresses the XFOR nest in question with only elementary constructs such as for-loops and if-conditionnals. The final LLVM IR corresponding to the XFOR nest is eventually generated based on this new partial AST.

isl allows us to take the iteration domains and combine them with the associated scheduling functions in a common data structure. Indeed, the library features maps which are basically sets with separate input and output dimensions. Using maps, we can restrain the scheduling functions to specific iteration domains. Eventually, a simple call to a single function of isl is capable of generate the partial AST based on the union of the scheduling maps (see Section 4.3).

In the last stage, the partial AST provided by isl is walked over by the LLVM IR generating method that I developed. Depending on the type of node encountered (for-loop, if-conditionnal, affine expression, ...), a proper code generation method is called. The entire process is summarized in Figure 5.6.

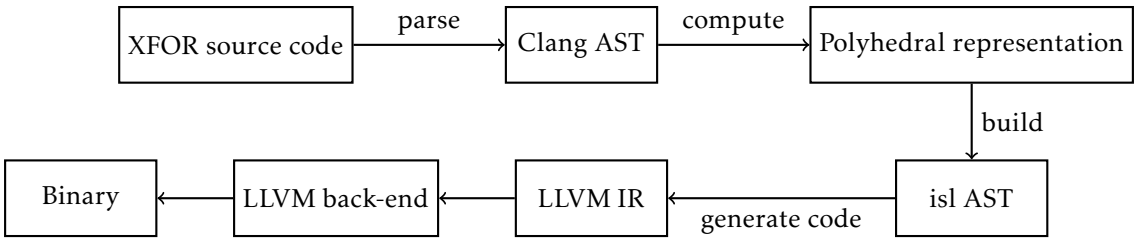


FIGURE 5.6 – Integration of the XFOR structure into the Clang/LLVM compiler

To illustrate our implementation, let us consider the XFOR loop in Listing 5.1. The isl AST generated from the polyhedral representation of this XFOR loop can be retranslated into a simple C source code as proposed in Listing 5.2. Finally, Listing 5.3 features the LLVM IR generated by our version of Clang based on that partial AST generated by isl.

```
xfor(i0 = 0,  i1 = 5;
     i0 < 20, i1 < 15;
     i0++,    i1++;
     1,       1;
     0,       2
) {
  0: foo(i0);
  1: bar(i1);
}
```

LISTING 5.1 – Illustrative XFOR loop

```
for(_mfr_ref0 = 0; _mfr_ref0 <= 19; _mfr_ref0++) {
  foo(_mfr_ref0);

  // 10 iterations of 'bar' after 2 iterations of 'foo'
  if(_mfr_ref0 >= 2 && _mfr_ref0 <= 12)
    // 'bar' expects values in range from 5 to 15!
    bar(_mfr_ref0 + 3);
}
```

LISTING 5.2 – Equivalent implementation based on the partial AST generated by isl based on the polyhedral representation of the XFOR loop

Being generated by isl which is an external library to Clang/LLVM, it was not possible to use existing methods for LLVM IR generation implemented in Clang to perform the transformation.

To resolve the issue, we were initially thinking about using the tool called Polly [20] which is part of LLVM framework and allows to perform loop and data reuse distance optimizations on polyhedral model-compatible loops (see Chapter 3). Most importantly, Polly also uses isl and we expected to take advantage of its isl AST to LLVM IR transformation module.

Unfortunately, I discovered that it will not be possible to reuse Polly's module directly. As Polly is part of the LLVM framework, it is composed of a series of compilation passes (see Section 5.2). As such, it expects LLVM IR on input and the module we were interested in is not entirely standalone, so it cannot be used separately by simply providing it only with an isl AST structure. Eventually, with Prof. Clauss we decided to perform the translation of isl AST into LLVM IR by our own means using the programming interface of LLVM combined with some of existing transformation functions in Clang.

When I was constrained to abandon the idea of taking advantage of Polly, I realized that the mission got more complex than expected. So, for future development, I decided to proceed in an incremental manner. I put together a list of XFOR features split in multiple levels of difficulty that I have been implementing one by one (see details in Figure 5.7). This technique also allowed me to perform continuous testing of my implementation throughout the entire compilation process.

```
define dso_local i32 @main() #0 { ; Main function
entry: ; Preamble
  %retval = alloca i32, align 4 ; Allocate the return value.
  %_mfr_ref0 = alloca i32, align 4 ; Allocate the common iterator value.
  store i32 0, i32* %retval, align 4 ; Set the return value to 0.
  store i32 0, i32* %_mfr_ref0, align 4 ; Initialize the iterator value to 0.
  br label %for.cond ; Jump to the beginning of the loop.

for.cond: ; Loop condition
  %0 = load i32, i32* %_mfr_ref0, align 4 ; Load the iterator value to '%0'.
  %cmp = icmp sle i32 %0, 19 ; Check if it is less or equal to 19.

  ; If yes, go to the loop body. Otherwise, go to the loop end.
  br i1 %cmp, label %for.body, label %for.end

for.body: ; Loop body
  %1 = load i32, i32* %_mfr_ref0, align 4 ; Load the iterator value to '%1'.
  call void @foo(i32 %1) ; Call 'foo' with '%1'.
  %2 = load i32, i32* %_mfr_ref0, align 4 ; Load the iterator value to '%2'.
  %cmp1 = icmp sge i32 %2, 2 ; Check if it is greater or equal to 2.

  ; If yes, jump to the other part of the condition after the 'AND' operator. Otherwise, go to
       the end of the conditionnal statement.
  br i1 %cmp1, label %land.lhs.true, label %if.end

; If the iterator is greater or equal to 2, check if it is also less or equal to 12.
land.lhs.true:
  %3 = load i32, i32* %_mfr_ref0, align 4
  %cmp2 = icmp sle i32 %3, 12

  ; If yes, execute 'bar'. Otherwise, go to the end of the conditionnal statement.
  br i1 %cmp2, label %if.then, label %if.end

if.then:
  %4 = load i32, i32* %_mfr_ref0, align 4 ; Load the iterator value to '%4'.
  %add = add nsw i32 %4, 3 ; Add 3 to it.
  call void @bar(i32 %add) ; Call 'bar' with '%4'.
  br label %if.end ; Jump to the end of the conditionnal statement.

if.end: ; End of the conditionnal statement
  br label %for.inc ; Jump to the loop incrementation.

for.inc: ; Loop incrementation
  %5 = load i32, i32* %_mfr_ref0, align 4 ; Load the iterator value to '%5'.
  %inc = add nsw i32 %5, 1 ; Increment it by 1.

  ; Store the new value to the iterator.
  store i32 %inc, i32* %_mfr_ref0, align 4
  br label %for.cond ; Return to the loop condition.

for.end: ; Loop end
  ret i32 0 ; Return 0.
}
```

Listing 5.3 – LLVM IR generated by our version of Clang corresponding to Listing 5.2 based on the partial AST provided by isl for the XFOR loop in Listing 5.1

Following the example of the official test suite provided with Clang/LLVM, I composed my own series of tests for every level of implementation. Each test runs Clang/LLVM on an illustrative source code. In addition, the annotations in the latter tells the test running script whether the compilation of the given source code should succeed or fail. In case of failure, the error type expected is also provided within the annotations and must match the one produced during the compilation. See an example in Listing 5.4 where the first line is the name of the test and the second line specifies the command line options of Clang to run the test file with.

```
// Failure on common syntax errors (part 1)
// RUN: %clang_cc1 -fsyntax-only -verify -pedantic %s
void foo(void) {}
int main(void) {
  int i, j;
  xfor(i = 0, j = 0; i < 10; // expected-error {{expected 2 condition expressions in 'xfor'
      statement specifier}}
    i++, j += 2; 1, 2; 0, 0) { // expected-error {{expected expression}}
    0: foo();
    1: { foo(); foo(); }
  }
  return 0;
}
```

Listing 5.4 – XFOR test source code with annotations for the test running script

Eventually, I have reached the fifth level of my incremental implementation plan. Therefore, our version of Clang is currently capable of compiling even nested XFOR constructs while taking into account the grain and the offset parameters. Also, the latter as well as the loops bounds of the handled for-loops may be expressed using valid affine expressions. Macro-defined value occurrences are also supported. Due to time constraints of the internship I could not implement the last level of my plan. Although, it represents rather rarely used XFOR features. So, it does not limit the direct compilation of the most common types of XFOR contructs.

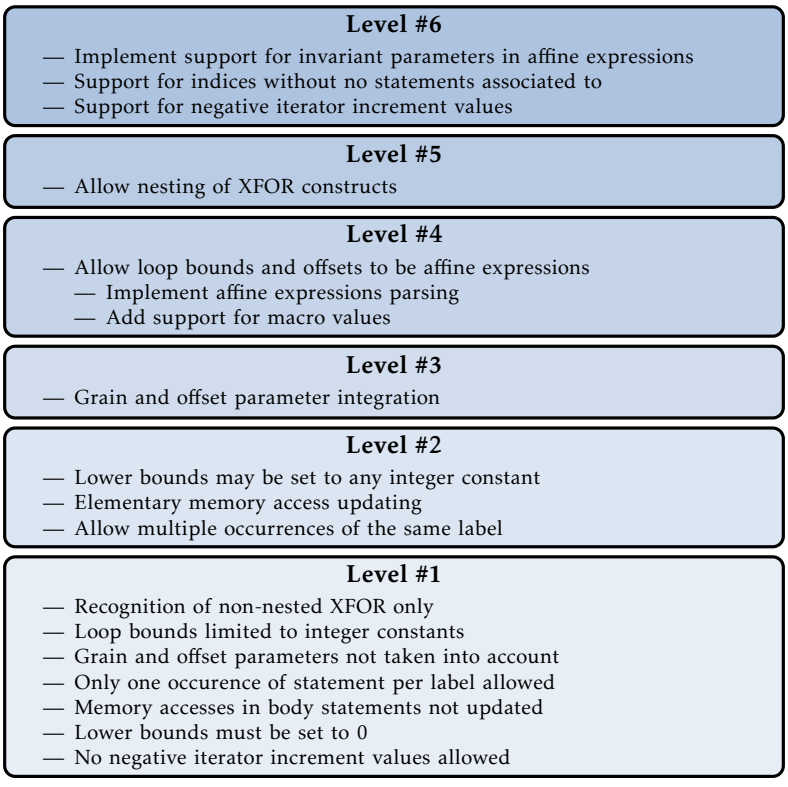


Figure 5.7 – Incremental implementation of XFOR features into the Clang/LLVM compiler

Chapter 6

Extension to parallel programming

Over a second phase of my internship, I was experimenting the possibilities of parallelization of the XFOR structure. In extension to the programming strategies presented in Section 4.4, we have studied two methods of parallelizing the loops handled by XFOR. Both are based on the OpenMP parallelization library but, present different approaches of using the latter and address different kinds of loop nests. In addition, we were focused on adapting existing XFOR software tools according to this new approach.

6.1 Classical approach

The OpenMP library features the `#pragma omp parallel for` directive for automatical parallelization of for-loops. When placed immediately before a for-loop, the latter is automatically parallelized during the compilation. Note that, compilers must be run with a specific option in order to make them take into account the OpenMP parallelization directives. In case of both gcc and icc, one should use the `-fopenmp` option.

`#pragma omp parallel for` splits the iterations of the target loop into a given number of groups. Multiple groups can then be run at the same time on multi-core processors. Usually, the loop iterations are equally split among the cores of the target processor.

Let us consider the simple for-loop in Listing 6.1. It counts 4 000 iterations in total. Assuming a processor with 4 logical cores, the OpenMp parallelization directive can split 4 000 iterations of the loop among 4 groups of 1 000 iterations. Each of them can be executed on one of the 4 cores of the processor which means that all of the groups can be run at the same time. Figure 6.1 illustrates this behavior.

The `private` and the `shared` attributes used in Listing 6.1 allow to specify whether each thread should have a local (private) copy of a variable or the same variable should be shared by all the threads.

```
#pragma omp parallel for private(i) shared(array)
for (i = 0; i < 4000; i++)
  array[i] = array[i] + 10;
```

Listing 6.1 – Simple for-loop parallelized with OpenMP

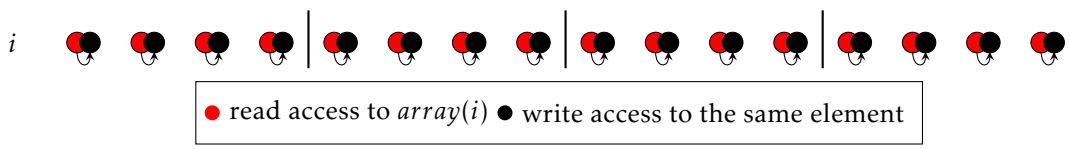


Figure 6.1 – Iteration domain of the for-loop in Listing 6.1 featuring a Write-After-Read data dependency

It is possible to apply this parallelization directive on XFOR loops too. The directive is then placed in front of every for-loop resulting from the polyhedral representation of the XFOR loop nest in question. Also the global iterators introduced during the transformation are conveniently set as private variables.

Let us consider the XFOR implementation of the matrix vector multiplication kernel in Listing 6.2. We have parallelized the outermost XFOR loop using the `#pragma omp parallel for` directive. The corresponding for-only implementation is proposed in Listing 6.3.

```
#pragma omp parallel for shared(x1, x2, A, y1, y2)
xfor(i0 = 0,     i1 = 0;
     i0 < _PB_N, i1 < _PB_N;
     i0++,       i1++;
     1,          1;
     0,          0) {
  xfor(j0 = 0,     j1 = 0;
       j0 < _PB_N, j1 < _PB_N;
       j0++,       j1++ ;
       1,          1;
       0,          0) {
    0: x1[i0] = x1[i0] + A[i0][j0] * y1[j0];
    1: x2[i1] = x2[i1] + A[j1][i1] * y2[j1];
  }
}
```

Listing 6.2 – Parallelized XFOR implementation of `mvt` kernel

```
#pragma omp parallel for private(i, j) shared(x1, x2, A, y1, y2)
for(i = 0; i <= _PB_N - 1; i++) {
  for(j = 0; j <= _PB_N - 1; j++) {
    x1[i] = x1[i] + A[i][j] * y1[j];
    x2[i] = x2[i] + A[j][i] * y2[j];
  }
}
```

Listing 6.3 – for-only implementation equivalent to Listing 6.2

The performance benchmarks in [14] show that this strategy can produce very satisfying results. Parallelized XFOR implementations can be up to 6 times faster compared to their original parallel versions.

Nevertheless, this approach is not directly applicable on loop nests carrying a data data dependency. By definition, such loop nests are not parallelizable because if the groups of iterations executed all in parallel are mutually dependent, the results produced by the program are very likely to be incorrect as the execution order of statements might not be respected. So, in its current implementation, the XFOR structure does not allow to fully express the parallelism of underlying loops.

6.2 Parallelizable chunks

Insprired by the principle of loop vectorization, to fill this gap, we have considered an alternative approach allowing to expose parallelism also in loops where the iterations are bound by a data dependency.

In Section 4.4.2, we have seen that, an appropriate adjustment of the offset value in XFOR can make automatic vectorization profitable even for loops for which it would normally not be possible due to data dependencies preventing parallel execution of multiple iterations.

Unfortunately, loop vectorization is a fine-grained form of parallelism. In general, it is solely applicable to the innermost loop in a given loop nest. To be able to parallelize this way loops of any level, we have to make use of multithreading.

Let us explain the approach we propose using an example. Listing 6.4 contains a trivial implementation of the two-dimensional Jacobi stencil. It features two independent and successive for-loops easily parallelizable using the `#pragma omp parallel for` OpenMP directive.

```
#pragma omp parallel for private(i,j) shared(A,B)
for (i = 1; i < _PB_N - 1; i++)
  for (j = 1; j < _PB_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]);
#pragma omp parallel for private(i,j) shared(A,B)
for (i = 1; i < _PB_N - 1; i++)
  for (j = 1; j < _PB_N - 1; j++)
    A[i][j] = B[i][j];
```

Listing 6.4 – Trivial implementation of `jacobi-2d`

Nevertheless, the for-loop nests may be fusioned into one. Using XFOR, it is also possible to favorize the vectorization of the innermost loop. See our XFOR implementation in Listing 6.5.

```
xfor(i0 = 1,         i1 = 1;
     i0 < _PB_N - 1, i1 < _PB_N - 1;
     i0++,           i1++;
     1,              1;
     0,              1) {
  xfor(j0 = 1,         j1 = 1;
       j0 < _PB_N - 1, j1 < _PB_N - 1;
       j0++,           j1++;
       1,              1;
       0,              7) {
    0: B[i0][j0] = 0.2 * (A[i0][j0] + A[i0][j0 - 1] + A[i0][1 + j0] + A[1 + i0][j0]
                         + A[i0 - 1][j0]);
    1: A[i1][j1] = B[i1][j1];
  }
}
```

Listing 6.5 – XFOR implementation of `jacobi-2d` (sequential)

Similarly to `jacobi-1d` in Section 4.4.2, we observe the best results by setting the offset value on the second innermost loop on $j1$ to 7. Although, a new issue appeared. The minimization of the data reuse distance between the outermost loops in our XFOR implementation makes simple parallelization of the outermost XFOR impossible using simply `#pragma omp parallel for`. Indeed, the Write-After-Read dependency between the accesses to A via $A(i0-1,\ j0)$ in S_0 and via $A(i1,\ j1)$ in S_1 prevents the outermost XFOR from being parallelized the classical way.

Our idea, is to shift the second outermost for-loop in the XFOR nest enough to create sufficiently large chunks contaning only independent instructions, as we did in the case of `jacobi-1d` in order to favorize the vectorization. The resulting chunks would be executed successively in a sequential manner, although we would apply the `#pragma omp parallel for` directive on each chunk. As such, the independent instructions it contains could be executed in parallel.

Unfortunately, such modification can not be currently done using nor IBB nor XFOR-WIZARD software tools. For experimentation purposes, we applied it manually on the for-only implementation corresponding to our XFOR version of `jacobi-2d`. Instead of 1, the offset of the second outermonst loop in the XFOR was set to 1 024. During our experimentations, we could obtain the best performance using this size of chunks. According to [29], the size of chunks should be greater than 256 for this strategy of parallelization to be profitable.

In addition to that, we inserted an enclosing for-loop on k to the implementation, in order to scan underlying loop iterations by chunks of 1 024 where each of them is parallelized using the `#pragma omp parallel for` directive. See the corresponding source code in Listing 6.6.

As Table 6.1 shows, using our parallelization approach, we could expose loop parallelism on a dependency-bound loop level and also outperform the trivial parallelized version by 57%.

	Trivial implementation	XFOR implementation	Overall gain
Execution time	43.11 s	27.41 s	**57**%

TABLE 6.1 – Performance comparison of the parallelized implementations of `jacobi-2d`

6.3 ADAPTING SOFTWARE TOOLS

Currently, our alternative parallelization approach can only by applied manually without any support from the XFOR software tools.

To be more easily applicable, we propose to introduce a new more expressive `#pragma` parallelization directive to ease the application of our parallelization strategy. This would also allow to generate the appropriate output source code by IBB automatically.

Moreover, XFOR-WIZARD should be extended as well in such a way as to be able to insert parallelization directives into the source code while editing XFOR loop nests. Also, the dependency analysis should be capable of verifying the respect of data dependencies in the case of parallel XFOR nests.

```
#pragma omp parallel for private(i, j) shared(A, B)
for(i = 0; i <= 1023; i++) // Parallel execution of the first chunk of 1024 iterations
  for(j = 0; j <= _PB_N - 3; j++)
    B[i + 1][j + 1] = 0.2 * (A[i + 1][j + 1] + A[i + 1][j] + A[i + 1][2 + j] + A[2 + i][j + 1]
         + A[i][j + 1]);

for(k = 1024; k <= _PB_N - 3; k += 1024) { // Scan the iterations by chunks of 1024
#pragma omp parallel for private(i, j) shared(A, B) firstprivate(k) // Parallelize each chunk
  for(i = k; i <= min(k + 1024, _PB_N - 3); i++) {
    for(j = 0; j <= 6; j++)
      B[i + 1][j + 1] = 0.2 * (A[i + 1][j + 1] + A[i + 1][j] + A[i + 1][2 + j]
                               + A[2 + i][j + 1] + A[i][j + 1]);

    for(j = 7; j <= _PB_N - 3; j++) {
      B[i + 1][j + 1] = 0.2 * (A[i + 1][j + 1] + A[i + 1][j] + A[i + 1][2 + j]
                               + A[2 + i][j + 1] + A[i][j + 1]);
      A[i - 1023][j - 6] = B[i - 1023][j - 6];
    }

    for(j = _PB_N - 2; j <= _PB_N + 4; j++)
      A[i - 1023][j - 6] = B[i - 1023][j - 6];
  }
}

#pragma omp parallel for private(i, j) shared(A, B)
for(i = _PB_N - 2; i <= _PB_N + 1021; i++) // Parallel execution of the last chunk of 1024
    iterations
  for(j = 7; j <= _PB_N + 4;j ++)
    A[i - 1023][j - 6] = B[i - 1023][j - 6];
```

LISTING 6.6 – Parallelized implementation of `jacobi-2d` previously optimized with XFOR

Chapter 7

Conclusion

In my work, I always try to put emphasis on autonomy and effectiveness. But, whenever I was facing a serious issue or had an important decision to make, I could rely on the expertise of Prof. Clauss.

Despite the fact that, my first mission got considerably more complex than expected, I have successfully integrated the XFOR structure into the Clang/LLVM compiler with its most important and most used functionalities. There are still several features of the structure to include and test. Although, thanks to an adapted working method, the greatest part of the work could have been done. The compiler currently handles source codes contaning either simple or nested XFOR loops with support for affine expressions in loop bound definitions, offset parameter values and memory references. There is also an implicit support for macro-defined values and some macro functions.

My study of parallelization strategies for the XFOR structure has revealed that, with our alternative approach there is a potential to fill a gap in parallel programming using XFOR. It allows to expose parallelism on loop levels on which it would normally not be possible due to data dependencies between loop iterations. It has also proven to have the capacity to outperform the classical way of parallelizing XFOR loops using the dedicated `#pragma omp parallel for` directive of the OpenMP library. Nevertheless, an important work of implementation has to be done in this direction in order to extend the software tools dedicated to the XFOR structure in such a way as to ease the parallel programming with XFOR as far as possible.

In addition to my main missions, I could participate on various seminars held by the members of the ICPS team. Plus, a presentation of the ICPS team and its activities was organized for Bachelor's degree students within the lecture of 'Introduction à la recherche'. At this occasion, I had a chance to talk about and demonstrate the XFOR structure as well as my work during a 20 minutes long session. I had another similar occasion by the end of the internship.

Globally, I could significantly extend my knowledge of the polyhedral model and other important concepts of loop optimization. I earned a valuable experience by working with Clang and LLVM which is widely used even in companies to build custom made compilers and optimization frameworks.

Our compilation courses were obviously essential to understand principles of the compilation process. Also, the courses of advanced compilation provided me with necessary knowledge about loop transformations, terms like data locality and data reuse distance, control flow graph etc. Thanks to practical exercises, I have learned necessary basics of LLVM. Our multiple courses of parallelism were particulary useful for me notably during my second mission. My previous internship with Prof. Clauss also helped me a lot while working on the latter.

Besides the scientific courses, I also made good use of English on daily basis whether it was related to work (documentation, reports, articles, thesis) or for communicating with international team members.

Eventually, I lived a rewarding professional experience that allowed me to reintegrate the scientific research environment and only confirmed my motivation for the domain, that my previous experience in ICube has already kindled in me.

Bibliography

Publications

[2] ISO/IEC JTC 1/SC 22. *ISO/IEC 9899:2018: Programming languages – C*. Geneva, Switzerland: International Organization for Standardization (ISO), June 2018. URL: https://www.iso.org/standard/74528.html. Price: CHF 198.

[3] Cédric Bastoul. "Code Generation in the Polyhedral ModelIs Easier Than You Think". In: *PACT'13 IEEE International Conference onParallel Architecture and Compilation Techniques*. Juan-les-Pins, Sept. 2004, pp. 7–16.

[4] Cédric Bastoul. *Extracting polyhedral representation from high level languages*. Tech. rep. LRI, Paris-Sud University, 2008.

[5] Cédric Bastoul. *Generating loops for scanning polyhedra*. Tech. rep. PRiSM, Versailles University, 2002.

[6] Cédric Bastoul. *OpenScop: A Specification and a Library for Data Exchange in Polyhedral Compilation Tools*. Tech. rep. Paris-Sud University, France, Sept. 2011. URL: http://icps.u-strasbg.fr/people/bastoul/public_html/development/openscop/docs/openscop.pdf.

[7] Cédric Bastoul and Louis-Noël Pouchet. *Candl: the chunky analyzer for dependences in loops*. Tech. rep. LRI, Paris-Sud University, 2012.

[8] Cédric Bastoul et al. *Putting Polyhedral Loop Transformations to Work*. Research Report RR-4902. INRIA, 2003. URL: https://hal.inria.fr/inria-00071681.

[9] Mohamed-Walid Benabderrahmane et al. "The Polyhedral Model Is More Widely Applicable Than You Think". In: *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*. Paphos, Cyprus: Springer-Verlag, Mar. 2010, pp. 283–303. URL: https://hal.inria.fr/inria-00551087.

[10] Uday Bondhugula et al. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer". In: *SIGPLAN Not.* 43.6 (June 2008), pp. 101–113. ISSN: 0362-1340. DOI: 10.1145/1379022.1375595. URL: http://doi.acm.org/10.1145/1379022.1375595.

[13] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (Jan. 1998), pp. 46–55. ISSN: 1070-9924. DOI: 10.1109/99.660313.

[14] Imen Fassi. "XFOR (Multifor): A New Programming Structure to Ease the Formulation of Efficient Loop Optimizations". Thesis. Université de Strasbourg, Nov. 2015. URL: https://hal.inria.fr/tel-01251721.

[16] Imen Fassi and Philippe Clauss. "XFOR: Filling the Gap between Automatic Loop Optimization and Peak Performance". In: *14th International Symposium on Parallel and Distributed Computing*. Ed. by IEEE. Limassol, Cyprus, June 2015. DOI: 10.1109/ISPDC.2015.19. URL: https://hal.inria.fr/hal-01155144.

[17] Paul Feautrier. "Dataflow analysis of array and scalar references". In: *International Journal of Parallel Programming* 20.1 (Feb. 1991), pp. 23–53. ISSN: 1573-7640. DOI: 10.1007/BF01407931. URL: https://doi.org/10.1007/BF01407931.

[18] Paul Feautrier. "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time". In: *International Journal of Parallel Programming* 21.6 (Dec. 1992), pp. 389–420. ISSN: 1573-7640. DOI: 10.1007/BF01379404. URL: https://doi.org/10.1007/BF01379404.

[19] Paul Feautrier and Christian Lengauer. "Polyhedron Model". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1581–1592. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_502. URL: https://doi.org/10.1007/978-0-387-09766-4_502.

[20] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. "Polly — Performing polyhedral optimizations on a low-level intermediate representation". In: *Parallel Processing Letters* 22.04 (2012), p. 1250010. DOI: 10.1142/S0129626412500107. URL: https://doi.org/10.1142/S0129626412500107.

[22] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[29] Benoit Pradelle. "Static and Dynamic Methods of Polyhedral Compilation for an Efficient Execution in Multicore Environments". Thesis. Université de Strasbourg, Dec. 2011. URL: https://tel.archives-ouvertes.fr/tel-00733856.

[31] Sven Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model". In: vol. 6327. Sept. 2010, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49.

Online sources

[1] *'L'université de Strasbourg et Inria donnent un nouvel élan à leur collaboration à l'ère de l'IA'*. Jan. 2019. URL: https://www.inria.fr/centre/nancy/actualites/l-universite-de-strasbourg-et-inria-donnent-un-nouvel-elan-a-leur-collaboration-a-l-ere-de-l-ia. (consulted on April 18, 2019).

[11] *Clang: a C language family frontend for LLVM*. URL: https://clang.llvm.org/. (consulted on July 10, 2019).

[12] *Cursus Master Ingénierie. Informatique, Systèmes et Réseaux*. URL: http://cmi-informatique.unistra.fr/reseaux/. (consulted on July 10, 2019).

[15] Imen Fassi and Philippe Clauss. *IBB : The Multifor Compiler*. URL: https://team.inria.fr/camus/ibb/. (consulted on July 10, 2019).

[21] Caroline Laplane. *Michel de Mathelin : 'ICube nous permet d'atteindre une taille suffisante pour assurer notre visibilité'*. 2013. URL: http://www.lactu.unistra.fr/index.php?id=14614#c62156. (consulted on April 18, 2019).

[23] *Official web site of the French Institute for Research in Computer science and Automation (INRIA). Presentation of the Camus team*. URL: https://team.inria.fr/camus/en/. (consulted on April 18, 2019).

[24] *Official web site of the French Institute for Research in Computer science and Automation (INRIA). List of software tools developed by the Camus team*. URL: https://team.inria.fr/camus/en/software/. (consulted on April 18, 2019).

[25] *Official web site of the French Institute for Research in Computer science and Automation (INRIA). APOLLO*. URL: https://team.inria.fr/camus/en/apollo/. (consulted on April 18, 2019).

[26] *Official web site of the ICube Laboratory*. URL: http://icube.unistra.fr/en/. (consulted on April 18, 2019).

[27] *Official web site of the ICube Laboratory. Research teams*. URL: http://icube.unistra.fr/en/equipes/. (consulted on April 18, 2019).

[28] *Official web site of the Scientific and Parallel Computing research team. Main page*. URL: http://icube-icps.unistra.fr/index.php/Main_Page. (consulted on April 18, 2019).

[30] The Clang team. *Clang CFE Internals Manual. How to add an expression or statement*. 2019. URL: https://clang.llvm.org/docs/InternalsManual.html#how-to-add-an-expression-or-statement.