Polyhedral Model in ROSE

Tristan Vanderbruggen

Université de Rennes 1

MRI Internship

Polyhedral Model in ROSE

Author:
Tristan Vanderbruggen

Supervisor:
Daniel Quinlan

August 18, 2010

Abstract

The Polyhedral Model has been an academic topic since the early eighties. It has primarily been used for systolic architecture generation and loop tranformations. During the last ten years, interest in this model has increased for many reasons: computing power has increased (Polyhedral Model is compute intensive), classical heuristic loop transformation methods have reached their limits, architectural behaviour is becoming unpredictable because of its complexity and new hardware (like graphic accelerators) are opening new horizons.
This report presents the implementation in ROSE, a research compiler, of Farkas' Algorithm. A new way to apply this algorithm has been developed; this variation of the original algorithm enables elimination of all hidden variables created by Farkas' Algorithm. This algorithm can also be applied in parallel to use multicore processors, alleviating the cost of the modification.

Résumé

Le Modèle Polyédrique est un sujet de recherche depuis le début des années quatre-vingt, il a été utilisé pour la génération d'architectures systoliques ainsi que pour les transformations de boucles. Au cours des dix dernières années, l'intérêt pour ce modèle s'est développé, de nombreuses raisons existe : l'augmentation de la puissance des ordinateurs (le Modèle Polyédrique coûte cher en calcul), les méthodes classiques de transformations de boucles par heuristique ont atteintes leurs limites, la complexité des architectures rend leurs comportements imprévisibles, les nouveaux systèmes (comme les accélérateurs graphiques) ouvrent de nouveaux horizons...
Ce rapport présente l'implémentation dans ROSE, un compilateur dédié à la recherche, de l'algorithme de Farkas. Une nouvelle façon d'appliquer cet algorithme a été développée, cette variation de l'algorithme original permet l'élimination de l'ensemble des variables cachées générées par la méthode de base. De plus, cet algorithme pouvant être appliqué en parallèle, le surcoût de lélimination des variables cachées est en grande partie masqué.

Acknowledgements

I want to thank:

- Daniel Quinlan, who gave me the opportunity to work with him.
- Christophe Wolinski, who put me in contact with Daniel and pushed me to begin working on Polyhedral Model.
- ROSE team, for their support and cheerfulness.
- A special mention for Justin Frye, who reviewed this report.

Contents

Introduction

This report discusses the implementation of the Polyhedral Model in the research compiler ROSE developed at the Lawrence Livermore National Laboratory. The Polyhedral Model is a convenient mathematical abstraction for a sub-class of programs, called Static Control Programs (SCoP).

The Polyhedral Model has been an academic topic since the early eighties. It has primarily been used for systolic architecture generation and loop tranformations. During the last ten years, interest in this model has increased for many reasons: computing power has increased (Polyhedral Model is compute intensive), classical heuristic loop transformation methods have reached their limits, architectural behaviour is becoming unpredictable because of its complexity and new hardware (like graphic accelerators) are opening new horizons.

After a fast overview of the ROSE project, we examine the current state of the art research on the Polyhedral Model and Schedule Generation. I used previous works from Paul Feautrier, Albert Cohen, Cédric Bastoul and Louis-noel Pouchet [Fea93a, PBCV07]. Their works are based on Farkas' Algorithm, which enables the generation of all one-dimensional causal affine schedules associated with an SCoP.
Next, I describe the improvement that I have found for Farkas' Algorithm. This improvement enables the elimination of every hidden variable that is introduced by Farkas' Algorithm.
Lastly, I present my future work on this topic: reducing the number of valid schedules by adding architectural constraints.

Duality of my internship

During my internship, I also had some engineering tasks to complete. I made the decision to not discuss this part of my internship here for two reasons: I have a separate report to do for my Engineering Grade and I have enough

to discuss with the Polyhedral Model implementation (which is my research topic).

Chapter 1

ROSE Project

1.1 Lawrence Livermore National Laboratory

The Lawrence Livermore National Laboratory (LLNL) is a laboratory of the U.S. Departement Of Energy (DOE). I had my internship in the Institute for Computer Science Research (ISCR). My advisor, Dan Quinlan, is a member of the Computer Science Group in the Center for Applied Scientific Computing (CASC). He manages the ROSE project.

1.2 ROSE

ROSE is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale Fortran 77/95/2003, C, C++, OpenMP, and UPC applications. The intended users of ROSE could be either experienced compiler researchers or library and tool developers who may have minimal compiler experience. ROSE is particularly well suited for building custom tools for static analysis, program optimization, arbitrary program transformation, domain-specific optimizations, complex loop optimizations, performance analysis, and cyber-security.

1.3 ROSE's Development Environment

One really interesting point, which I will briefly cover in this document, is the development environment used to maintain and improve this software: continous integration. Indeed, Rose uses Hudson[1], a build and test automa-

[1] http://hudson-ci.org/

tion server and GIT[2], a distributed version control system.
With this system, all commits to a "release candidate" remote branch will be intensively tested before being merged with the master branch. Furthermore, as everyone is asked to create tests for their features, the system ensures that a modification will not have a side-effect on a different part of the project.

This usage makes possible the development of a compiler with a small team of permanent developers and many interns, without constantly breaking the master branch.
This continous development environment has a cost (computers and maintenance) but increases the productivity of the team and the reliability of the software.

[2]http://git-scm.com/

Chapter 2

Previous Work on Polyhedral Model

2.1 Modelling of Static Control Programs

2.1.1 Static Control Programs

Static Control Programs (SCoP) are a sub-class of programs (see [Fea93a, PBCV07] and for Affine Control Loop (sub-class of SCoP) [Raj02]). This kind progam contains only **for loops** and **if conditionals** as control structures. Loops bounds and conditions are affine functions of iterators and global variables.

```
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
3         if (i<=n-j+2)
4                 s[i] = ...;
```

Figure 2.1: A SCoP example

In SCoP, Memory Accesses are also affine functions of iterators and global variables.

2.1.2 Modelling

From SCoP's control structures, we can extract for each statement an iteration domain. Because loop bounds and conditions are affine functions of iterators and global variables, these domains are integer polyhedrons.

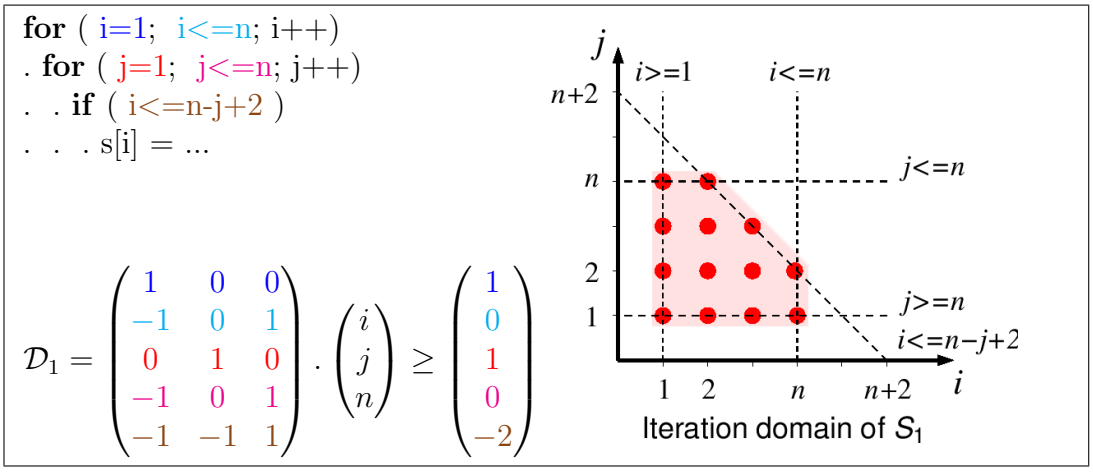

Figure 2.2: Polyhedral Domain associated with one statement in a SCoP.

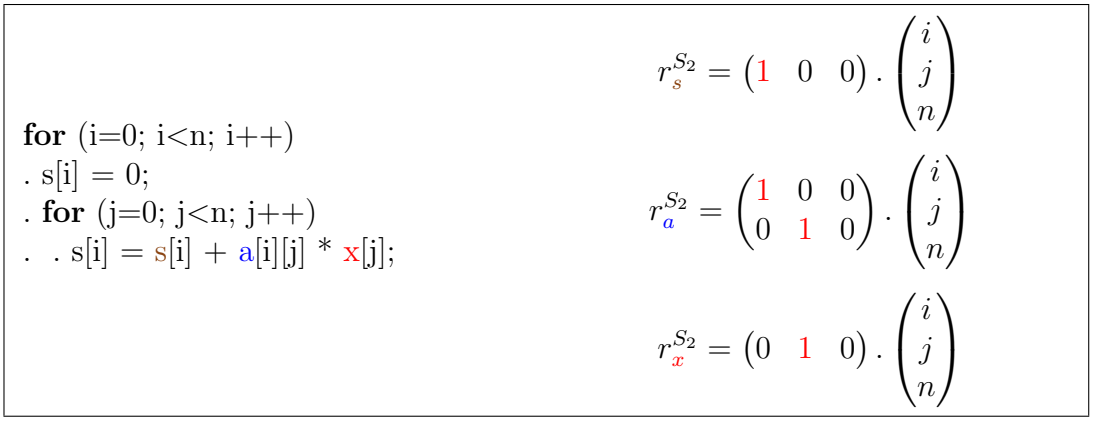

Figure 2.3: Affine Functions associated to memory read/write in a SCoP.

2.2 Data Dependencies and Generalized Dependency Graph

In [Fea93a], Paul Feautrier introduced the Generalized Dependency Graph (GDG). In these graphs, nodes are statements of one SCoP and edges, that represent dependencies, are qualified by a polyhedron.
The polyhedron defining one dependency is the result of an exact dependency analysis. This analysis can be provided by the Fuzzy Array Dependency Analysis (FADA, [BCF97]).
The resulting polyhedron is a subset of the cartesian product of statements' iteration domains. We compute this subset with two affine relations: a **condition** on iterators from the statement on which it depends and a **relation**

between statements' iterators.

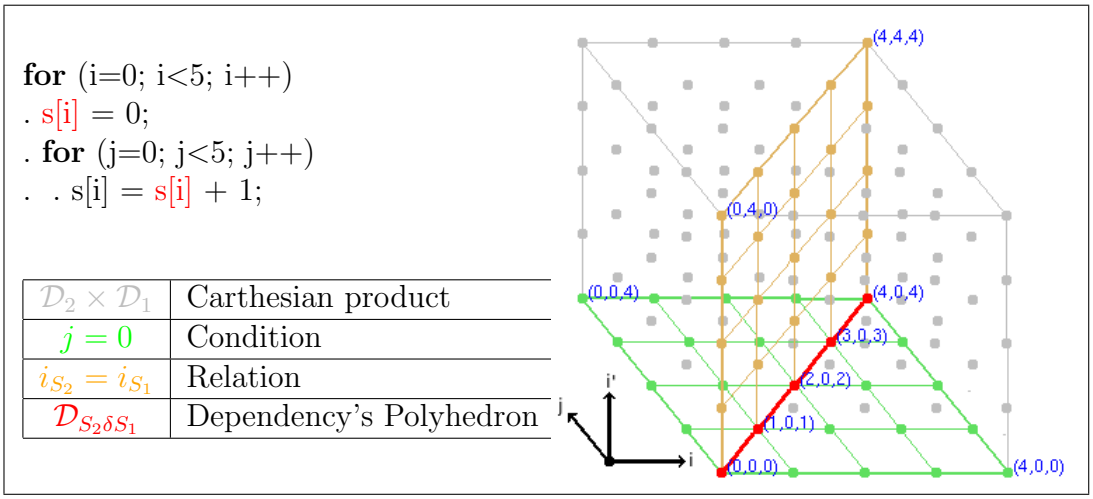

Figure 2.4: Polyhedron associated with the "read after write" dependency from statement 2 to statement 1.

The **Condition** $j = 0$ means that only the first j-iteration of statement 2 has a dependency. The **Relation** $i_{S_2} = i_{S_1}$ means that the dependency exists only for the same i-iteration, when both statements access the same element of the array s.
A point of $\mathcal{D}_{S_2\delta S_1}$ can be read as:

$$\begin{pmatrix} i_{S_2} \\ j_{S_2} \\ i_{S_1} \end{pmatrix} \in \mathcal{D}_{S_2\delta S_1} \iff S_2(i_{S_2}, j_{S_2}) \text{ depends of } S_1(i_{S_1})$$

2.3 One-Dimensional Causal Affine Schedules

2.3.1 One-Dimensional Affine Schedules

A Schedule is a function that associates an execution time to each statement in a program. We focus here on **One-Dimensional Affine Schedules**.

Given a statement S, $\mathcal{D}_S$ iteration domain of S, and $\mathcal{G}$ global variables' domain[1], Θ is a One-Dimensional Affine Schedule of S iff:

$$\forall(\bar{z}, \bar{g}) \in \mathcal{D}_S \times \mathcal{G}, \quad \Theta_S(\bar{z}, \bar{g}) = \overline{\alpha_S}.\bar{z} + \overline{\beta_S}.\bar{g} + \kappa_S$$

Where:

[1]for example: $\{(n, m) \,|\, n \geq 0 \wedge m \geq n\}$

- $\overline{\alpha_S} \in \mathbb{Z}^{card(\mathcal{D}_S)}$
- $\overline{\beta_S} \in \mathbb{Z}^{card(\mathcal{G})}$
- $\kappa_S \in \mathbb{Z}$

2.3.2 Causality

One-Dimensional Causal Affine Schedules are One-Dimensional Affine Schedules that respect a causality relation.
This relation comes from data dependencies: if iteration $\overline{z_2}$ of statement S_2 depends on iteration $\overline{z_1}$ of statement S_1 then:

$$\Theta_{S_2}(\overline{z_2}, \overline{g}) \geq \Theta_{S_1}(\overline{z_1}, \overline{g}) + \Delta_1$$

Where Δ_1 is the latency of statement 1.

So, for each edge $S_2 \rightarrow S_1$ in the Generalized Dependency Graph:

$$\forall \begin{pmatrix} \overline{z_2} \\ \overline{z_1} \\ \overline{g} \end{pmatrix} \in \mathcal{D}_{S_2 \delta S_1},\ \Theta_{S_1}(\overline{z_1}, \overline{g}) + \Delta_{S_1} \leq \Theta_{S_2}(\overline{z_2}, \overline{g}) \tag{2.1}$$

Where $\mathcal{D}_{S_2 \delta S_1}$ is the polyhedron associated with the edge $S_2 \rightarrow S_1$.

2.3.3 Code Transformation Embedded in One-Dimensional Affine Schedules

The Polyhedral Model and Schedules enable us to achieve many loop transformations, but the difference with common compilers is that a One-Dimensional Affine Schedule can embed a composition of many loop transformations (see 2.1), whereas common compilers only apply some transformations sequentially.

Table 2.1: Transformations that can be embbeded in a One-Dimensional Affine Schedule [PBCV07].

reversal	Changes the direction in which a loop traverses its iteration range
skewing	Makes the bounds of a given loop depend on an outer loop counter
interchange	Exchanges two loops in a perfectly nested loop, a.k.a. permutation
peeling	Extracts one iteration of a given loop
index-set splitting	Partitions the iteration space between different loops
shifting	Allows to reorder loops
fusion	Fuses two loops, a.k.a. jamming
distribution	Splits a single loop nest into many, a.k.a. fission or splitting

2.4 Farkas' Algorithm

2.4.1 Affine Form of Farkas' Lemma

Let $\mathcal{P}$ be a nonempty polyhedron defined by p affine inequalities:

$$\overline{z} \in \mathcal{P} \iff \forall k \in [\![1, p]\!], a_k.\overline{z} + b_k \geq 0$$

Then an affine form ψ is non negative everywhere in $\mathcal{P}$ iff it is a positive affine combination:

$$\forall \overline{z} \in \mathcal{P}, \psi(\overline{z}) \geq 0$$

$$\iff$$

$$\forall k \in [\![0, p]\!],\ \exists \lambda_k \geq 0 \text{ such as } \forall \overline{z} \in \mathcal{P},\ \psi(\overline{z}) = \lambda_0 + \sum_{k=1}^{p} \lambda_k.(a_k.\overline{z} + b_k)$$

2.4.2 Farkas' Algorithm

Farkas' Algorithm [Fea93a] is the application of the Affine Form of Farkas' Lemma to all causality relations 2.1 implied by the GDG.

Indeed, $\mathcal{D}_{S_2 \delta S_1}$ is a polyhedron and

$$\Theta_{S_1}(\overline{z_1}, \overline{g}) + \Delta_{S_1} \leq \Theta_{S_2}(\overline{z_2}, \overline{g}) \iff \Theta_{S_2}(\overline{z_2}, \overline{g}) - \Theta_{S_1}(\overline{z_1}, \overline{g}) - \Delta_{S_1} \geq 0$$

Resulting from the application of Farkas' Lemma on the edge $S_2 \rightarrow S_1$, we have a system of equations and inequations that involve $\overline{\alpha_{S_1}}$, $\overline{\alpha_{S_2}}$, $\overline{\beta_{S_1}}$, $\overline{\beta_{S_2}}$, κ_{S_1}, κ_{S_2} and many λ introduced by the algorithm. These λ are called Farkas' Multiplicators and in previous work a few of them are eliminated

using Fourrier-Motzkin Elimination.

The application of Farkas' Lemma on the causality relation for each dependency enables the construction of a new polyhedron. This polyhedron has for components every schedules' coefficiants and Farkas' Multiplicators (more details: 3.2.1).

Finally, the polyhedron generated by Farkas' Algorithm contains **all** One-Dimensional Causal Affine Schedules.

Chapter 3

Improvement to Farkas' Algorithm

3.1 Farkas' Multiplicators Issues

Farkas' Multiplicators are numerous and each one adds a dimension to the final search space (polyhedron containing all one-dimensional causal affine schedules). Previous work [Fea93a, PBCV07] uses Fourrier-Motzkin Elimination to decrease their numbers and bound those remaining.

Some issues come from this method: (1) Fourrier-Motzkin Elimination can't remove more than a few Farkas' Multiplicators, (2) even bounded Farkas' Multiplicators are over dimensions in the polyhedron, (3) by bounding Farkas' Multiplicators, we can loose some schedules.

3.2 Using Cylindrification Operator

Cylindrification Operator is a mathematic operation [Mon00], that unconstrains a variable from a polyhedron, but conserves the rest of the polyhedron:

$$\forall k \in [\![0, dim(\mathcal{P})-1]\!] cyl_k(\mathcal{P}) = \{\overline{w} \in \mathbb{Z}^{dim(\mathcal{P})} \,|\, \exists \overline{v} \in \mathcal{P},\ \forall i \in [\![0, dim(\mathcal{P})-1]\!] - \{k\}\ v_i = w_i\} \tag{3.1}$$

I used this operator to construct cyl_n^* operation:

$$\forall n \in [\![0, dim(\mathcal{P}) - 1]\!]\ cyl_n^*(\mathcal{P}) = \{\overline{w} \in \mathbb{Z}^n \,|\, \exists \overline{v} \in \mathcal{P},\ \forall i \in [\![0, n-1]\!]\ v_i = w_i\} \tag{3.2}$$

This operation encloses: the cylindrification of the $dim(\mathcal{P}) - n$ last dimensions and the projection on the n first dimensions.

The projection on the n first dimensions of any element in $\mathcal{P}$ is in $cyl_n^*(\mathcal{P})$ (proof 3.1). Any element of $cyl_n^*(\mathcal{P})$ is the projection of an element of $\mathcal{P}$ (trivial).

Proof

$$\forall k \in \mathbb{N},\ \forall \overline{u} \in \mathbb{K}^n,\ proj_k(\overline{u}) = \begin{pmatrix} u_0 \\ \vdots \\ u_{k-1} \end{pmatrix} \tag{3.3}$$

$$\forall (\overline{u}, \overline{v}) \in \mathbb{K}^n \times \mathbb{K}^m,\ \overline{u} \square \overline{v} = \begin{pmatrix} u_0 \\ \vdots \\ u_{n-1} \\ v_0 \\ \vdots \\ v_{m-1} \end{pmatrix} \tag{3.4}$$

Proof by contrapositive:

$$\begin{aligned}
& \overline{x_0} = proj_n(\overline{z_0}) \notin cyl_n^*(\mathcal{P}) && (3.5)\\
\iff & proj_n(\overline{z_0}) \notin \{\overline{x} \in \mathbb{Z}^n \mid \exists \overline{z} \in \mathcal{P},\ \forall i \in [\![0, n-1]\!]\ x_i = z_i\} && (3.6)\\
\iff & proj_n(\overline{z_0}) \notin \{\overline{w} \in \mathbb{Z}^n \mid \exists \overline{v} \in \mathcal{P},\ \overline{x} = proj_n(\overline{z})\} && (3.7)\\
\iff & \nexists\, \overline{z} \in \mathcal{P},\ proj_n(\overline{z_0}) = proj_n(\overline{z}) && (3.8)\\
\implies & \nexists\, \overline{y} \in \mathbb{Z}^m,\ proj_n(\overline{z_0}) \square \overline{y} \in \mathcal{P} && (3.9)\\
\implies & \overline{z_0} \notin \mathcal{P} && (3.10)
\end{aligned}$$

Figure 3.1: $\forall \overline{z} \in \mathbb{Z}^{n+m},\ \overline{z} \in \mathcal{P} \implies \overline{x} = proj_n(\overline{z}) \in cyl_n^*(\mathcal{P})$

We can use this operation to eliminate all Farkas' Multiplicators. But the cost of this operation is over-linear so, for large SCoP, elimination of Farkas' Multiplicators from the final search space is not possible.
However, we can construct a search space for each dependency in the GDG, eliminate in every resulting search space all Farkas' Multiplicators and finally compute the intersection of all these search spaces.

3.2.1 Application of Cylindrification operation to Farkas' Algorithm

With $dim(\mathcal{A}) = n_1$ and $dim(\mathcal{B}) = n_2$.

$$\mathcal{A} \overset{k}{\bigcap} \mathcal{B} = \left\{ \overline{z} \in \mathbb{Z}^{n_1+n_2-k} \,\middle|\, \exists (\overline{x}, \overline{y_1}, \overline{y_2}) \in \mathbb{Z}^k \times \mathbb{Z}^{n_1-k} \times \mathbb{Z}^{n_2-k},\ \overline{z} = \overline{x} \square \overline{y_1} \square \overline{y_2} \ \wedge\ \overline{x} \square \overline{y_1} \in \mathcal{A} \ \wedge\ \overline{x} \square \overline{y_2} \in \mathcal{B} \right\} \tag{3.11}$$

The operation defined in 3.11 represents a step of the classical Farkas' Algorithm: each dependency produces a new set of Farkas' Multiplicators ($\overline{\lambda}$).

In both following figure (3.2, 3.3), notation: $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$ represent a polyhedron with 3 dimensions associated with variables called a, b and c. A vector, $\overline{a}$ represent a set of variables $\{a_0, ..., a_n\}$.

$$\begin{pmatrix} \overline{\alpha_{S_1}} \\ \vdots \\ \kappa_{S_n} \end{pmatrix} \overset{\delta_1}{\rightarrow \bigcap^k} \begin{pmatrix} \overline{\alpha_{S_1}} \\ \vdots \\ \kappa_{S_n} \\ \overline{\lambda_0} \end{pmatrix} \overset{\delta_2}{\rightarrow \bigcap^k} \begin{pmatrix} \overline{\alpha_{S_1}} \\ \vdots \\ \kappa_{S_n} \\ \overline{\lambda_0} \\ \overline{\lambda_1} \end{pmatrix} \overset{\delta_i}{\rightarrow \bigcap^k} \cdots \overset{\delta_m}{\rightarrow \bigcap^k} \begin{pmatrix} \overline{\alpha_{S_1}} \\ \vdots \\ \kappa_{S_n} \\ \overline{\lambda_0} \\ \vdots \\ \overline{\lambda_m} \end{pmatrix}$$

Figure 3.2: Classic Farkas' Algorithm Implementation: For each dependency, we add some dimensions (for Farkas' Multiplicators) then applied equation/inequation generated by the algorithm. These operations can be represented by 3.11 equation.

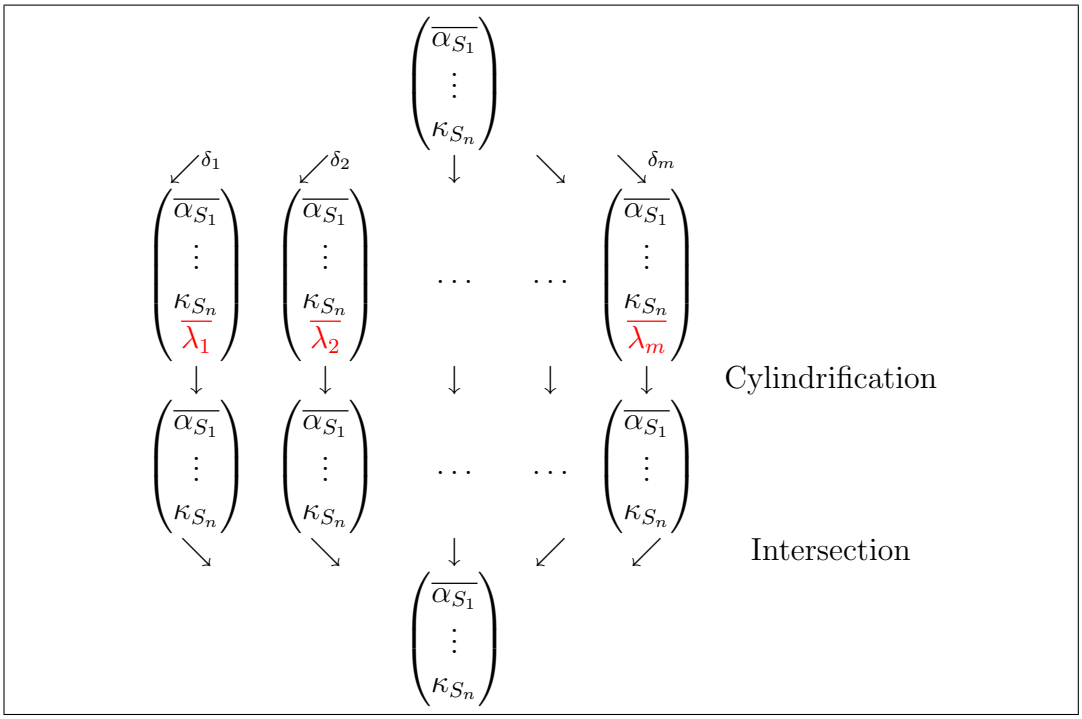

Figure 3.3: Proposed application of Farkas' Algorithm using the Cylindrification operation.

It is interesting to notice that this algorithm can be easily parallelized. Indeed, adding a dependency and removing generated Farkas' Multiplicators can be done independentely for each dependency.

Proof of equivalence

I showed before (3.1) that $cyl_n^*(\mathcal{P})$ (with n number of schedule coefficiants) conserves all causal schedules and contains only causal schedules.

$$cyl_k^*(\mathcal{A} \bigcap^k \mathcal{B}) = \left\{\overline{x} \in \mathbb{Z}^k \,\middle|\, \exists(\overline{y_1}, \overline{y_2}) \in \mathbb{Z}^{n_1-k} \times \mathbb{Z}^{n_2-k},\ \overline{x} \,\square\, \overline{y_1} \in \mathcal{A} \ \wedge\ \overline{x} \,\square\, \overline{y_2} \in \mathcal{B}\right\} \tag{3.12}$$

$$= \left\{\overline{x} \in \mathbb{Z}^k \,\middle|\, \exists \overline{y} \in \mathbb{Z}^{n_1-k},\ \overline{x} \,\square\, \overline{y} \in \mathcal{A}\right\} \bigcap \left\{\overline{x} \in \mathbb{Z}^k \,\middle|\, \exists \overline{y} \in \mathbb{Z}^{n_2-k},\ \overline{x} \,\square\, \overline{y} \in \mathcal{B}\right\} \tag{3.13}$$

$$= cyl_k^*(\mathcal{A}) \bigcap cyl_k^*(\mathcal{B}) \tag{3.14}$$

Figure 3.4: $cyl_k^*(\mathcal{A} \bigcap^k \mathcal{B}) = cyl_k^*(\mathcal{A}) \bigcap cyl_k^*(\mathcal{B})$

I show in 3.4 that application of Farkas' Algorithm separately for each dependency followed by the elimination of corresponding Farkas' Multiplicators, before computing the intersection of all generated polyhedrons is equivalent to computing Farkas' Algorithm for all dependencies before eliminating all Farkas' Multiplicators at one time.

3.2.2 Conclusion

This variation of Farkas' Algorithm enables the generation of a search space that is not polluted by Farkas' Multiplicators and the highcost of cylindrification can be partially covered by the ability to parallelize this process.

Chapter 4

Future Work

The Polyhedral Model implementation in ROSE still needs a code generator. After an initial attempt to implement the proposed algorithm in [Bas04], it seems the best method is to use CLooG [Bas02], a library implementing the previous algorithm.
Another remaining task is to implement multi-dimensional schedule generation [Fea93b]; with this kind of schedule it will be possible to achieve more complex code transformations on more complex programs.

Regarding future research work: I want to investigate the possibility of constraining the search space with architectural considerations. By extracting some constraints from statements, it can be possible to generate a search space where schedules that imply common architectural bottlenecks will be eliminated.
General Purpose Graphic Processing Units (GPGPU) seem to be really suitable for this kind of study; indeed, GPGPU have many "linear" constraints (for example: memory access from one multiprocessor need to be contigous to be execute at the same time).

Chapter 5

Implementation details

I have taken care during the implementation to simplify migration from one library to another and maximize the portability my work to other projects. For this, I have developed 4 modules with minimal inter-dependencies (only for data transfer purposes). These modules are:

- **Exact Dependency Analysis**:
  This module uses FADAlib to obtain an exact dependency analysis. It transforms the ROSE internal representation (SageIII: an Abstract Syntax Tree) of an SCoP to FADA internal representation (another Abstract Syntax Tree) before performing the analysis.
  It can be used for other purposes, like the auto-parallelisation project inside ROSE (This project does automatic annotation of code with OpenMP directives).
  A lower level of internal representation of FADAlib can be used to perform full dependency analysis on object-oriented programs.

- **Data Dependencies Polyhedron**:
  Here, a polyhedron (like in 2.4) is generated for each dependency in the input program.

- **Causal Affine Schedules**:
  This module generates the polyhedron containing all One-Dimensional Causal Affine Schedules.

- **Code Generation**:
  This module needs to generate for a given schedule the SageIII representation of the transformed program.
  This can be done using CLooG [Bas02].

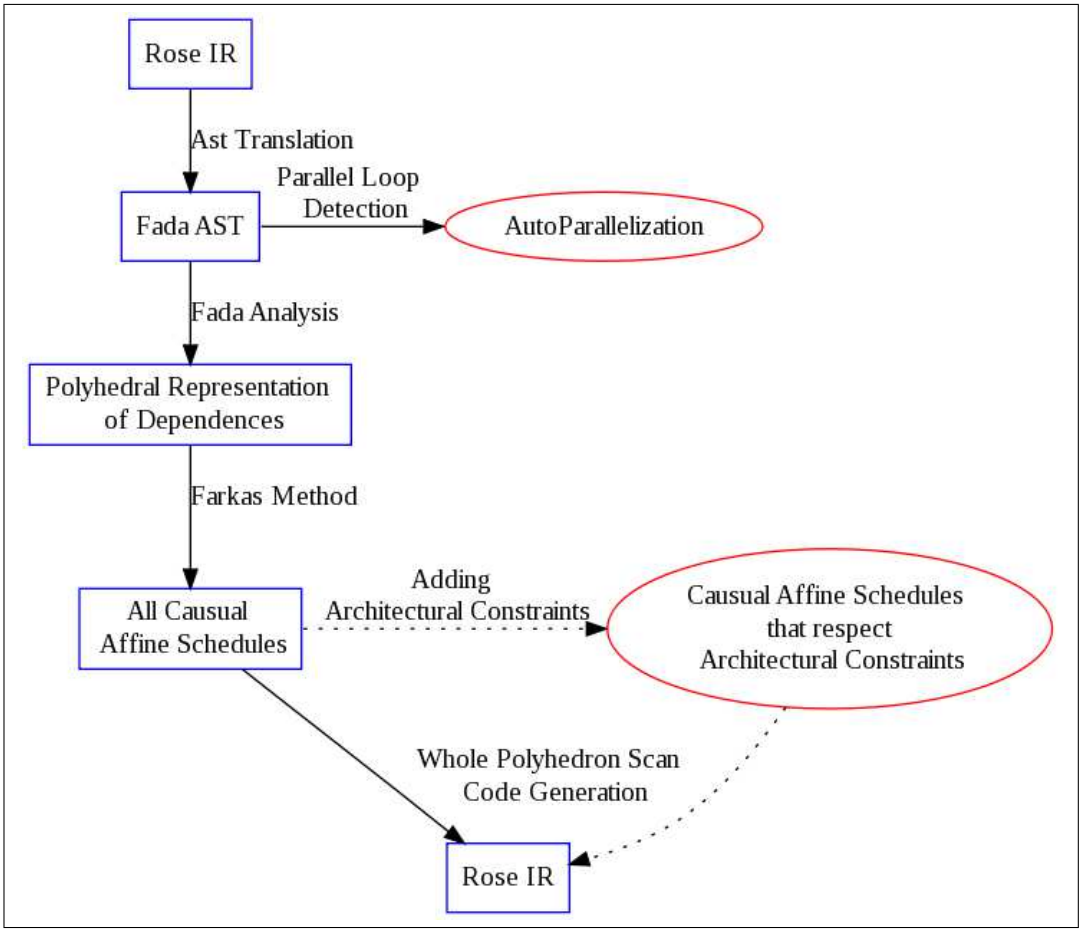

Figure 5.1: This graph describes the data transformation in the project. In blue, the existing chain where edges are the four modules. In red, some possible developments.

Chapter 6

Conclusion

My first plan for this internship was to work on schedule generation under architectural constraints (applied to graphic accelerator). Unfortunately, it is not possible to implement the Polyhedral Model in six months. So, to finish it and start the second part, my advisor, Daniel Quinlan, offers me to come back. I hope to have the opportunity to return and finish this project, which is really exciting.

I learned a lot during this summer. Continuous Integration is one of the thing I learned about and I'm convinced that it's something to promote. I also practiced full-time research during five months; it thought me how to conduct my work over a longer period than ever before and gave me the occasion to have results that can be used in other research.

This internship has been an occasion to work in a leading computer science laboratory and to encounter really interesting poeple. I also discovered an interesting topic which mixes compilation, architecture and linear algebra.

Bibliography

[Bas02] Cdric Bastoul. Generating loops for scanning polyhedra: Cloog users guide. Technical report, 2002.

[Bas04] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[BCF97] Denis Barthou, Jean-François Collard, and Paul Feautrier. Fuzzy array dataflow analysis. *Journal Parallel Distributed Computing*, 40(2), 1997.

[Fea93a] Paul Feautrier. Some efficient solutions to the affine scheduling problem – part i one-dimensional time. *International Journal of Parallel Programming*, 1993.

[Fea93b] Paul Feautrier. Some efficient solutions to the affine scheduling problem – part ii one-dimensional time. *International Journal of Parallel Programming*, 1993.

[Mon00] JD Monk. An introduction to cylindric set algebras. *Logic Jnl IGPL*, 8(4):451–496, 2000.

[PBCV07] Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. *Code Generation and Optimization, 2007. CGO '07. International Symposium*, 2007.

[Raj02] Sanjay V. Rajopadhye. Dependence analysis and parallelizing transformations. In *The Compiler Design Handbook*, pages 329–372. 2002.