Internship Report, jointly EPITA CSI 2006 and UPS Master 2
(March - September 2006)

Louis-Noël Pouchet
<louis-noel.pouchet@inria.fr>

WHEN ITERATIVE OPTIMIZATION MEETS THE POLYHEDRAL MODEL: ONE-DIMENSIONAL DATE

Under the direction of A. Cohen & C. Bastoul

INRIA Futurs
Parc Club Orsay Université, ZAC des vignes
4 rue Jacques Monod - Bâtiment G
F-91893 Orsay – France
Tel. +33/0 1 72 92 59 40 – Fax +33/0 1 60 19 69 63
`http://www-futurs.inria.fr`

Ecole pour l'Informatique et les Techniques Avancées
14-16, rue Voltaire – F-94270 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 44 08 01 01 – Fax +33 1 44 08 01 99
`http://www.epita.fr`

In collaboration with:

Abstract

Emerging micro-processors introduce unprecedented parallel computing capabilities and deeper memory hierarchies, increasing the importance of loop transformations in optimizing compilers. Because compiler heuristics rely on simplistic performance models, and because they are bound to a limited set of transformations sequences, they only uncover a fraction of the peak performance on typical benchmarks. Iterative optimization is a maturing framework addressing these limitations, but so far, it was not successfully applied complex loop transformation sequences because of the combinatorics of the optimization search space.

We focus on the class of loop transformation which can be expressed as one-dimensional affine schedules. *We define a systematic exploration method to enumerate the space of all legal, distinct transformations in this class* This method is based on an upstream characterization, as opposed to state-of-the-art downstream filtering approaches. Our results demonstrate orders of magnitude improvements in the size of the search space and in the convergence speed of a dedicated iterative optimization heuristic.

Feedback-directed and iterative optimizations have become essential defenses in the fight of optimizing compilers fight to stay competitive with hand-optimized code: they freshen the static information flow with dynamic properties, adapting to complex architecture behaviors, and compensating for the inaccurate single-shot of model-based heuristics. Whether a single application (for client-side iterative optimization) or a reference benchmark suite (for in-house compiler tuning) are considered, the two main trends are:

- tuning or specializing an individual heuristic, adapting the profitability or decision model of a given transformation;
- tuning or specializing the selection and parameterization of existing (black-box) compiler phases.

This study takes a more offensive position in this fight. To avoid diminishing returns in tuning individual phases or combinations of those, we collapse multiple optimization phases into a single, unconventional, iterative search algorithm. By construction, the search space we explore encompasses *all legal program transformations* in a particular class. Technically, we consider the whole class of loop nest transformations that can be modeled as *one-dimensional schedules*, a significant leap in model and search space complexity compared to state-of-the-art applications of iterative optimization. We make the following contributions:

- we statically construct the optimization space of all, arbitrarily complex, arbitrarily long sequences of loop transformations that can be expressed as one-dimensional affine schedules (using a polyhedral abstraction);
- this search space is built free of illegal and redundant transformation sequences, avoiding them altogether at the very source of the exploration;
- we demonstrate multiple orders of magnitude reduction in the size of the search space, compared to filtering-based approaches on loop transformation sequences or state-of-the-art affine schedule enumeration techniques;
- these smaller search spaces are amenable to fast-converging, mathematically founded operation research algorithms, allowing to compute the exact size of the space and to traverse it exhaustively;

- our approach is compatible with acceleration techniques for feedback-directed optimization, in particular on machine-learning techniques which focus the search to a narrow set of most promising transformations;
- our source-to-source transformation tool yields significant performance gains on top of a heavily tuned, aggressive optimizing compiler.

Eventually, we were stunned by the intricacy of the transformed code, which was far beyond our expectations; a confirmation that whatever the performance model and whatever the expertise of the programmer, designing a predictive model for loop transformation sequences seems out of reach.

Contents

List of Figures

List of Algorithms

List of Programs

Chapter 1

Technical Report

Abstract

Transformation of programs is an incontrovertible compilation step towards the full mining of a given architecture capabilities. It is a hard task for a compiler or a compilation expert to find or apply appropriate and effective transformations to a program, due to the complexity of modern architectures and to the rigidity of compilers. We propose to transpose the problem in the *polyhedral model*, where it is possible to transparently apply arbitrarily complex transformations. We focus on the class of transformation which can be expressed as affine monodimensional schedules. We combine the definition of this search space to an iterative compilation method, and propose a systematic exploration of this space in order to find the best transformation. We present experimental results demonstrating the feasibility of the proposed method and its high potential for program optimization.

Keywords

Optimization, iterative compilation, polyhedral model, legal transformation space, affine scheduling, Fourier-Motzkin algorithm.

1.1 Introduction

High level optimization consists in transforming a source program to another source program in order to take advantage of the architecture capabilities (parallelism, memory hierarchy, etc.). The privileged target for these optimizations are loop nests, where a high proportion of the execution time is spent. Typically we look for a sequence of transformations to apply to the program, in order to produce a more effective transformed version where outputs are identical.

Compilers represent programs as Abstract Syntax Trees. This rigid representation makes extremely difficult or even impossible the application of complex rescheduling transformations. *De facto* compilers do not dispose (or in a limited manner) of powerful high level optimization tools. We are going to transpose the problem in a formal model where the representation and application of these complex transformations is natural: the *polyhedral model*. Once the program is conveniently transformed, we go back to a syntactic representation thanks to a *code generation* phase.

If the polyhedral model makes easier the application of arbitrarily complex transformations, finding the best transformation is an extremely complex problem. We propose here a practical iterative compilation method to find the best transformation within a certain class of schedules.

The contribution is threefold: first, we propose an efficient and mathematically founded method to compute the set of legal affine monodimensional schedules for a program; second we propose a short study on the Fourier-Motzkin projection algorithm, to improve its efficiency; third we provide experimental results to underline the high potential of this iterative compilation method for program optimization.

The manuscript is organized as follows. In Section 1.2 we briefly introduce our motivations on transformation process and parallelization, and we recall some related work in Section 1.3. In Section 1.4 we introduce the polyhedral model on which relies our method and then in Section 1.5 and 1.6 we formally introduce our method to compute and bound the legal monodimensional affine scheduling space. In Section 1.7 we discuss a slight reformulation of the Fourier-Motzkin projection algorithm which propose for an approximately identical complexity to remove what we define as local redundancy. Various experimental results are discussed in Section 1.8. Finally, short and long term future works are discussed in Section 1.9 before concluding.

1.2 Motivation

If we consider the following trivial example Program 1.1, we can observe a program doing an m-relaxation of a vector of size n.

Program 1.1 A first example

```
     do i = 0, m
     |  do j = 0, n
S    |  |  s(j) = s(j) * 0.25
     |  end do
     end do
```

Let us consider a simple memory hierarchy where there is one cache level of size S_c (on a bi-processor embedded in some system), and Random Access Memory which has a 5 times lower access speed than the cache memory. The cycles needed to do the operation S is 1 if the data is in the cache, and 11 if not (we consider a naive caching algorithm which fills the cache to the maximum, and then removes elements one by one to replace them with new elements loaded from the RAM).

A run of our program would last $m \times n$ cycles if $n \leq Sc$. But if $n > S_c$, $m \times (n - S_c)$ *cache-misses* would occur. Plus, the statement S does not take any benefit from this bi-processor architecture, since no parallelism is expressed here.

Applying a single transformation on this program, by interchanging loops i and j would drastically reduce the cache-misses, by improving *data locality* (the "distance" between data used during the program). Also, a simple dependence analysis tells that the dependences between each execution of S let the program be parallelizable. The Program 1.2 illustrates this.

On this very simple example, we have manually improved the speed of our program by dividing its execution time by a factor 2 with parallelism, and removed $m \times (n - S_c)$ cache-misses with locality improvement.

Our goal in this study is to try to automatically discover the best transformation for a given machine. We will consider the whole set of possibilities to *legally reschedule* the program with

Program 1.2 A first transformation example

```
    doall j = 0, n
    |  do i = 0, m
S   |  |  s(j) = s(j) * 0.25
    |  end do
    end doall
```

affine monodimensional schedules (see Section 1.4.4), representing the composition of transformations in the polyhedral model.

We also claim that since processor architectures and optimizing compilers have attained such a level of complexity that it is no longer possible to model the optimization phase in a software, we need to consider them as black boxes. We propose an iterative compilation procedure to take the benefits from those optimizing compilers, and yet do ourselves a part of the work by detecting parallelism and enabling transformation compositions that those compilers cannot model at the moment.

We propose an approach to overcome the impossibility to model this processor plus compiler combination, by exhaustively testing an accurate subset of transformations on a given machine and with a given compiler.

The question is: does the expressiveness and algebraic structure of the polyhedral model contribute to accelerate the convergence of iterative methods and to discover significant opportunities for performance improvements ?

1.3 Related Work

The growing complexity of architectures became a challenge for compiler designers to achieve the peak performance for every program. In fact, the term *compiler optimization* is now biased since both compiler writers and users know that those program transformations can result in performance degradation in some scenarii that may be very difficult to understand [9, 11, 40].

Iterative compilation aims at selecting the best parametrization of the optimization chain, for a given program or for a given application domain. It typically affects optimization flags (switches), parameters (e.g., loop unrolling, tiling) and the phase ordering. [1, 3, 9, 11, 25, 31].

This paper studies a different search space: instead of relying on the compiler options to transform the program, we statically construct a set of candidate program versions, considering the distinct result of all legal transformations in a particular class. Our method is more tightly coupled with the compiler transformations and is thus complementary to other forms of iterative optimization. Furthermore, it is completely independent from the compiler back-end.

Because iterative compilation relies on multiple, costly "runs" (including compilation and execution), the current emphasis is on improving the profile cost of individual program versions [19, 25], or the total number of runs, using, e.g., genetic algorithms [24] or machine learning [1, 39]. Our meta-heuristic is tuned to the rich mathematical properties of the underlying *polyhedral* model of the search space, and exploits the regularity of this model to reduce the number of runs. Combining it with more generic machine learning techniques seems promising and is the subject of our ongoing work.

The polyhedral model is a well studied, powerful mathematical framework to represent loop nests and to remove the main limitations of classical, syntactic loop transformations. Many studies have tried to assess a predictive model characterizing the best transformation within this model, mostly to express parallelism [17, 26] or to improve locality [30, 36, 46]. We present experimental results showing that such models, although associated with optimal strategies, fail to scratch the complexity of the target architecture and the interactions with the back-end compiler, yielding far from optimal results even on simple kernels.

Iterative compilation associated to the polyhedral model is not a very common combination. To the best of our knowledge, only Long et al. tried to define a search space based on this model [27, 28], using the Unified Transformation Framework [23] and targeting Java applications. Long's search space includes a potentially large number of redundant and/or illegal transformations, that need to be discarded after a legality check, and the fraction of distinct and legal transformations decreases exponentially to zero with the size of program to optimize. On the contrary, we show how to build and to take advantage of a search space which, by construction, contains no redundant and no illegal transformation.

The polyhedral model makes easier the composition of transformations and the legality check, but it is also possible to characterize the whole set of legal transformations of a program. When Feautrier defined a method to express the set of legal affine positive monodimensional schedules, in order to minimize an objective function [16], he implicitly proposed a method powerful enough to compute and explore the set of legal transformations. The drawbacks were the limitation to positive schedules and the poor scalability of the method. The expression of the set of legal schedules we use was first formalized by Bastoul and Feautrier [6, 8] but was never used for exploration. Feautrier proposed several ways to improve scalability, including his recent paper about modular scheduling [18]; but we propose here a very simple and yet powerful method to compute this legal transformation set with a full scalability and no criterion on the dependence graph walk.

1.4 Polyhedral Representation of Programs

In the following section we describe the needed notions of the polyhedral model, its applications and its essential workaround.

1.4.1 Static Control Parts

The polyhedral model does not aim to model loop nests in general, but a widely used subclass called *Static Control Parts* (SCoP). A SCoP is a maximal set of instruction such that:

- the only allowed control structures are the `for` loops and the `if` conditionals,
- loop bounds and conditionals are affine functions of the surrounding parameters and the global parameters.

At first glance this definition may seem restrictive, but many programs which does not respect those conditions directly can thus be expressed as SCoPs. A preprocessing stage (typically inside a compiler architecture) can ease the automatic discovery of SCoPs. The Figure 1.1, grabbed from [6], brings out some automatic processing for SCoP discovery.

Another key concept is the idea of *static references*. A reference is said to be static when it refers to an array cell by using an affine subscript function that depends only on outer loop counters and parameters [6]. Pointers arithmetic is thus forbidden, and function calls have to be inlined (which, as said earlier, is a step that can be done inside a preprocessing stage).

It is worth noting that many scientific codes does respect the SCoP and static reference conditions, at least on hot spots of the code. A survey in [21] brings out the high proportion of SCoP in these codes. An empiric well-known constatation is that 80% of the processor time is spent on

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

original non-static loop target static loop

(a) Constant propagation example

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

original non-static loop target static loop

(b) While-loop to do-loop conversion

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

original static loop target static loop

(c) Loop normalization example

```
                            do i=1, m
do i=1, m                   | do j=1, n
| function(i,m)             | | S1
```

original non-static loop target static loop

(d) Inlining example

```
ind = 0                     do i=1, 100
do i=1, 100                 | do j=1, 100
| do j=1, 100               | | a(200*i+2*j-200)
| | ind = ind + 2           | | = a(200*i+2*j-200)
| | a(ind) = a(ind)         | |       + b(j)
| |          + b(j)         | c(i) = a(200*i)
| c(i) = a(ind)
```

original non-static loop target static loop

(e) Induction variable substitution example

Figure 1.1: Preprocessing for static control examples

less than 20% of the code, yielding the need to optimize these small code segments. These code segments are most of the time in loop nests, and we refer to them as *kernels*. The polyhedral model aims at modeling these kernels (under the SCoP and static reference conditions), and to be able to perform transformations (meaning changing the execution order but keep the output order) on these kernels.

1.4.2 Iteration Domain

In the polyhedral model, program information is represented as $\mathbb{Z}$-polyhedra. Let us consider the example Program 1.3 (*matvect*).

Program 1.3 `matvect`

```
    do i = 0, n
R   |  s(i) = 0
    |  do j = 0, n
S   |  |  s(i) = s(i) + a(i,j) * x(j)
    |  end do
    end do
```

The statement R is surrounded by one loop, with iterator i. Its *iteration vector* $\vec{x}_R$ is (i). The iterator i takes values between 0 and n, so the polyhedron containing all the values taken by i is $\mathcal{D}_R : \{i \mid 0 \leq i \leq n\}$. Intuitively, to each point of the polyhedron correspond an execution of the statement R (an *instance*), where the loop iterator i has the point coordinates in the polyhedron as value. With a similar reasoning we can express the iteration domain of statement S: $\vec{x}_S = \binom{i}{j}$. The polyhedron representing its iteration domain is $\mathcal{D}_S : \{i, j \mid 0 \leq i \leq n,\ 0 \leq j \leq n\}$.

In the remainder of this study, we use matrix form in homogeneous coordinates to express polyhedra. For instance, for the iteration domain of R, we get :

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

1.4.3 Dependence Expression

In order to describe a program, it is mandatory to know the sequential order of the execution of instructions. Such an order is constrained by the *dependence graph* of the program. If we consider the preceding example, data-flow analysis gives two dependences : statement S depends on R (R produces values used by S), and we note $D_1 : R\delta S$; similarly we have $D_2 : S\delta S$.

The dependence D_1 doesn't occur for each value of $\vec{x}_R$ and $\vec{x}_S$, it occurs only when $i_R = i_S$. We can then define a dependence polyhedron, being a subset of the Cartesian product of the iteration domains, containing all the values of i_R, i_S and j_S for which the dependence exists. We can write this polyhedron in matrix form (the first line represents the equality $i_R = i_S$) :

$$\mathcal{D}_{D_1} : \left[\begin{array}{ccccc} 1 & -1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \end{array}\right] . \begin{pmatrix} i_R \\ i_S \\ j_S \\ n \\ 1 \end{pmatrix} \begin{array}{c} = 0 \\ \hline \geq \vec{0} \end{array}$$

The exact matrix construction of the affine constraints of the dependence polyhedron we use was formalized by Bastoul and Feautrier [6, 8], and is recalled in the following.

Definition 1 *A statement R depends on a statement S (written $S\delta R$) if there exists an operation $S(x_S)$ and $R(x_R)$ and a memory location m such that:*

1. *$S(x_S)$ and $R(x_R)$ refer to the same memory location m, and at least one of them writes to that location,*
2. *x_S and x_R belongs to the iteration domain of R and S,*
3. *in the original sequential order, $S(x_S)$ is executed before $R(x_R)$.*

It is so possible to define a *dependence polyhedron* with affine (in)equalities, where each (integral) point of the polyhedron will represent an instance of the statement S and R where the dependence exists. This polyhedron will be a subset of the Cartesian product of S and R iteration domains. It can be constructed as the following:

1. *Same memory location*: assuming m is an array location, this constraint is the equality of the subscript functions of a pair of references to the same array: $F_S\vec{x}_S + a_S = F_R\vec{x}_R + a_R$.
2. *Iteration domains*: both S and R iteration domains can be described using affine inequalities, respectively $A_S\vec{x}_S + c_S \geq 0$ and $A_R\vec{x}_R + c_R \geq 0$.
3. *Precedence order*: this constraint can be separated into a disjunction of as many parts as there are common loops to both S and R. Each case corresponds to a common loop depth, and is called a *dependence level*. For each dependence level l, the precedence constraints are the equality of the loop index variables at depth lesser to l: $x_{R,i} = x_{S,i}$ for $i < l$ and $x_{R,l} > x_{S,l}$ if l is less than the common nesting loop level. Otherwise, there is no additional constraint and the dependence only exists if S is textually before R. Such constraints can be written using linear inequalities: $P_S\vec{x}_S - P_R\vec{x}_R + b \geq 0$.

Thus, the dependence polyhedron for $S\delta R$ at a given level l and for a given pair of references p can be described as follows:

$\mathcal{D}_{S\delta R,l,p}$:

$$D\begin{pmatrix}\vec{x}_S\\ \vec{x}_R\end{pmatrix} + d = \left[\begin{array}{cc} F_S & -F_R \\ \hline A_S & 0 \\ 0 & A_R \\ PS & -P_R \end{array}\right]\begin{pmatrix}\vec{x}_S\\ \vec{x}_R\end{pmatrix} + \begin{pmatrix}a_S - a_R\\ c_S\\ c_R\\ b\end{pmatrix} \begin{array}{c} = 0 \\ \hline \geq \vec{0} \end{array}$$

A simple algorithm for dependence analysis (which was implemented in the `Candl` tool) is to build the polyhedron $\mathcal{D}_{S\delta R,l,p}$ for each S, R, l and p of the program, and to check if there is a point in the polyhedron (with for example the PipLib tool [14, 15]). If so, an edge between R and S is added in the dependence graph, labeled by the $\mathcal{D}_{S\delta R,l,p}$ polyhedron.

1.4.4 Scheduling a Program

A schedule is a function which associates a timestamp to each execution of each statement. This function is constrained by the dependence graph. Indeed, when there is a dependence $D_1 : R\delta S$, the execution date of S has to be greater than the execution date of R, for each value of $\vec{x}_S$ and $\vec{x}_S$ where the dependence exists, enforcing S to be executed *after* R.

Two executions having the same date can be executed in parallel. This date can be either a scalar (we will talk of monodimensional schedules), or a vector (multidimensional schedules). A monodimensional schedule, if it exists, express the program as a single sequential loop, possibly surrounding one or more parallel loops. A multidimensional schedule express the program as one or more nested sequential loops, possibly surrounding one or more parallel loops.

In the polyhedral model, we use affine functions to schedule programs. A schedule of an instruction S will be an affine combination of the iteration vector $\vec{x}_R$ and the global parameters $\vec{n}$. It can be written (T is a constant matrix, and a constant row matrix in the monodimensional case):

$$\theta_S(\vec{x}_S) = T \begin{pmatrix} \vec{x}_S \\ \vec{n} \\ 1 \end{pmatrix}$$

Schedules are chosen affine in order to be able to generate the target code, with efficient code generation algorithms. Plus, this is the only case where we are able to exactly decide the legality of a transformation [6].

The schedule of a program will be the set of the schedule functions of each statement.

The "original" schedule of a program can be seen as the identity matrix. But we need to be able to express sequential orders inside loop nests. Let us consider the example Program 1.4.

Program 1.4 `multidim`

```
    do i = 1, n
       do j = 1, n
R         x(i) = x(i - 1)
S         x(i) = x(i) * 2
       end do
    end do
```

If we set for R $\theta_R(\vec{x}_R) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} . \binom{i}{j}$ and for S $\theta_S(\vec{x}_S) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} . \binom{i}{j}$ we apply the original lexicographic order on the iteration domains of R and S but we do not capture the sequentiality between R and S (R has to be executed before S). The two executions can have the same date, so they could be executed in parallel, which is false.

A possible and elegant method is to add an extra column in the instance vector (the vector exactly representing the timestamp of the instance of the instruction) to specify the static statement ordering inside the loop nest, as described in [21].

But the problem can be solved without this formulation. Instead, if we use $\theta_R(\vec{x}_R) = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} . \binom{i}{j}$ and $\theta_S(\vec{x}_S) = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} . \binom{i}{j} + \binom{0}{1}$ we ensure the two instructions will always have different execution dates and the condition $\theta_R(\vec{x}_R) \prec \theta_S(\vec{x}_S)$ will be respected. In fact, it is always possible to impose a sequential order for the statements inside a loop nest only with the T matrix.

The schedule of a program must be *legal*, that is, must not violate the dependence graph. The following definition holds:

Definition 2 (Legal schedule) *A legal schedule for a given dependence graph labeled by dependence polyhedra, and a given set of operations Ω is a function $\theta : \Omega \longrightarrow \mathbb{Z}^n$ such that:*

$$\forall R, S \in \Omega, \; \forall \; \vec{x}_s \times \vec{x}_R \in \mathcal{D}_{R\delta S}$$
$$R\delta S \Rightarrow \theta_S(\vec{x}_s) \preceq \theta_R(\vec{x}_R) + 1$$

Where n is the dimension of the schedule, $\preceq$ is the lexicographic order on vectors, and 1 is the null-vector of size n with 1 on its last column.

1.4.5 Transformations in the Polyhedral Model

Finding a legal transformation of a program consists in finding values for the matrix T in order to let the schedule of the program be legal with respect to the dependence graph.

One of the strength of the polyhedral model is to be able to express in a natural way an arbitrarily complex composition of transformations.

If we consider, for instance, loop permutation (*interchange*), the transformation (i becomes j and j becomes i) can be expressed $\theta\binom{i}{j} = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right].\binom{i}{j}$. If we consider *reversal* transformation (i becomes $-i$), it can be expressed likewise by $\theta\binom{i}{j} = \left[\begin{smallmatrix} -1 & 0 \\ 0 & 1 \end{smallmatrix}\right].\binom{i}{j}$.

Composing these two transformations consists in multiplying the two transformation matrices, yielding a unique and identically sized matrix representing the equivalent compound transformation. The Figure 1.3 illustrate this composition.

Expressing constraints on the coefficients of this matrix is equivalent to *express constraints on the result of a composition of transformations*, which can be an arbitrarily complex compound of transformations.

In this study, we focus on affine one-dimensional schedules: T is a constant row matrix. Such a representation is much more expressive than sequences of primitive transformations, since a single one-dimensional schedule may represent a potentially intricate and long sequence of any of the transformations shown in Figure 1.2.

Figure 1.2: Possible transformations embedded in a one-dimensional schedule

Transformation	Description
`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`

There exist robust and scalable algorithms and tools to reconstruct a loop nest program from a polyhedral representation (i.e., from a set of affine schedules) [5, 22, 35]. We will thus generate transformed versions of each SCoP by exploring its legal, distinct affine schedules, regenerating a loop nest program every time we need to profile its effective performance.

1.5 Affine Monodimensional Schedules

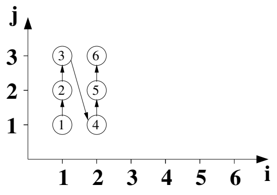
In the following section, we present a method to compute the set of legal monodimensional schedules of a program, in the polyhedral model.

1.5.1 Legal Affine Schedules Set

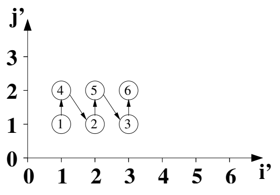
Let R and S be two statements. If there is a dependence $D_1 : R\delta S$, then each (integral) point of the dependence polyhedron $\mathcal{D}_{D_1}$ represents a value of the iteration vectors $\vec{x}_R$ and $\vec{x}_S$ where the dependence needs to be satisfied. It is possible to express the set of affine, non-negative functions over $\mathcal{D}_{D_1}$ thanks to the affine form of the Farkas lemma [37] (a geometric intuition of the Lemma is shown in Appendix A):

Interchange Transformation

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



$\Longrightarrow$



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

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

$$\begin{pmatrix}i'\\j'\end{pmatrix}=\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}\begin{pmatrix}i\\j\end{pmatrix}$$

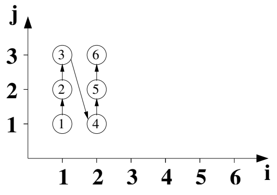
(b) transformation function $\vec{y}=T\vec{x}$

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

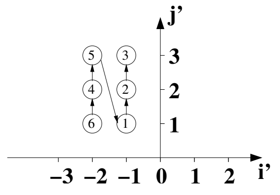
(c) target polyhedron $(AT^{-1})\vec{y}+\vec{a}\geq\vec{0}$

Reversal Transformation

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



$\Longrightarrow$



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

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

$$\begin{pmatrix}i'\\j'\end{pmatrix}=\begin{bmatrix}-1 & 0\\0 & 1\end{bmatrix}\begin{pmatrix}i\\j\end{pmatrix}$$

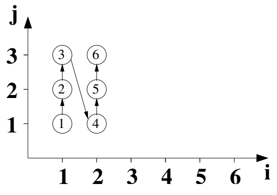
(b) transformation function $\vec{y}=T\vec{x}$

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

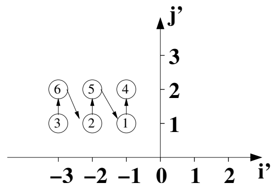
(c) target polyhedron $(AT^{-1})\vec{y}+\vec{a}\geq\vec{0}$

Coumpound Transformation

The transformation matrix is the composition of an interchange and reversal



$\Longrightarrow$



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

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

$$\begin{pmatrix}i'\\j'\end{pmatrix}=\begin{bmatrix}0 & -1\\1 & 0\end{bmatrix}\begin{pmatrix}i\\j\end{pmatrix}$$

(b) transformation function $\vec{y}=T\vec{x}$

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

(c) target polyhedron $(AT^{-1})\vec{y}+\vec{a}\geq\vec{0}$

Figure 1.3: A compound transformation example

Lemma 1 (Affine form of Farkas lemma) *Let $\mathcal{D}$ be a nonempty polyhedron defined by the inequalities $A\vec{x} + \vec{b} \geq \vec{0}$. Then any affine function $f(\vec{x})$ is non-negative everywhere in $\mathcal{D}$ iff it is a positive affine combination:*

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

λ_0 and $\vec{\lambda}^T$ are called the Farkas multipliers.

In order to satisfy the dependence (to force S to be executed after R), the schedules have to satisfy $\theta_R(\vec{x}_R) < \theta_S(\vec{x}_S)$, for each point of $\mathcal{D}_{D_1}$. So one can state that:

$$\Delta_{R,S} = \theta_S(\vec{x}_S) - \theta_R(\vec{x}_R) - 1$$

Must be non-negative everywhere in $\mathcal{D}_{D_1}$. Since we can express the set of affine non-negative functions over $\mathcal{D}_{D_1}$, the set of legal schedules satisfying the dependence D_1 is given by the relation:

$$\theta_S(\vec{x}_S) - \theta_R(\vec{x}_R) - 1 = \lambda_0 + \vec{\lambda}^T \left(\mathcal{D}_{D_1} \begin{pmatrix} \vec{x}_R \\ \vec{x}_S \end{pmatrix} + \vec{d}_{D_1} \right) \geq 0$$

Let us go back to our example (*matvect*). The two prototype affine schedules for R and S are:

$$\begin{array}{rcl} \theta_R(\vec{x}_R) & = & t_{1_R}.i_R + t_{2_R}.n + t_{3_R}.1 \\ \theta_S(\vec{x}_S) & = & t_{1_S}.i_S + t_{2_S}.j_S + t_{3_S}.n + t_{4_S}.1 \end{array}$$

Using the previously defined dependence representation, we can directly split the system into as many inequalities as there are independent variables, and equate the coefficients in both sides of the formula ($\lambda_{D_{1,x}}$ is the Farkas multiplier attached to the x^{th} constraint of $\mathbb{D}_{D_1}$):

$$\left\{ \begin{array}{rrcrcl} D_1 & i_R & : & -t_{1_R} & = & \lambda_{D_{1,1}} - \lambda_{D_{1,2}} + \lambda_{D_{1,7}} \\ & i_S & : & t_{1_S} & = & \lambda_{D_{1,3}} - \lambda_{D_{1,4}} - \lambda_{D_{1,7}} \\ & j_S & : & t_{2_S} & = & \lambda_{D_{1,5}} - \lambda_{D_{1,6}} \\ & n & : & t_{3_S} - t_{2_R} & = & \lambda_{D_{1,2}} + \lambda_{D_{1,4}} + \lambda_{D_{1,6}} \\ & 1 & : & t_{4_S} - t_{3_R} - 1 & = & \lambda_0 \end{array} \right.$$

In order to get the constraints on the schedule coefficients, we need to solve the system, with for example the Fourier-Motzkin projection algorithm [4, 16]. It is worth noting that, since all Farkas multipliers are positive, the system is sized enough to be computable. If there is no solution, then no affine monodimensional schedule is possible to solve this dependence.

If we build then solve with Fourier-Motzkin algorithm the system for the dependence D_1, we obtain a polyhedron $\mathcal{D}_t^1$, being the projection of the λ constraints on the t dimensions (of R and S). This polyhedron represents the set of legal values for the schedule coefficients, in order to satisfy the dependence. To build the set of legal schedule coefficients for the whole program, we have to build the intersection of each polyhedron obtained for each dependence.

One may note that we do not impose any specification for the dependence graph, neither any heuristic to walk it. We instead store in a global polyhedron $\mathcal{D}_t$, with as many dimensions as there are schedule coefficients for the SCoP, the intersection of the constraints obtained for each dependence. With this method, the transitivity of the dependence relation is preserved in the global solution but all systems are built and solved for one dependence at a time. In fact, the computation of the legal space can be done simultaneously with the dependence analysis. The intersection operation implicitly extends the dimensionality of polyhedra to the dimensionality of $\mathcal{D}_t$, and sets the missing dimensions to 0.

So we have for k dependences:

$$\mathcal{D}_t = \bigcap_k \mathcal{D}_t^k$$

The resulting polyhedron $\mathcal{D}_t$ represents the set of acceptable coefficients values for the program. Intuitively, to each (integral) point of $\mathcal{D}_t$ corresponds a different schedule for the program. If $\mathcal{D}_t$ is empty, then there is no possible affine monodimensional schedule for this program.

1.6 Legal Transformation Space Computation

1.6.1 The Algorithm

Bounding the Legal Transformation Space

The previous formulation gives a practical method to compute the (possibly infinite) set of legal monodimensional schedules for a program. We need to bound $\mathcal{D}_t$ into a polytope in order to make achievable an exhaustive search of the transformation space. The bounds are given by considering a side-effect of the used code generation algorithm [5]: *higher the transformation coefficients, the more likely the generated code to be ineffective*.

We use an heuristic to enable different bounds for the transformation coefficients, regarding their type (coefficients of iterators, parameters or constant). The iterator coefficients gives the "general form" of the schedule. We can, for instance, represent classical loop transformations like reversal, skewing, etc. only with these coefficients. Their values directly imply the complexity of the control bounds, and may generate modulo operations in the produced code; and bounds between -1 and 1 are accurate most of the time. On the other hand, parameter coefficients may have a hidden influence on locality. Let us consider the example of Program 1.5 (*locality*).

Program 1.5 `locality`

```
    do i = 0, n
    |  do j = 0, n
R   |  |  b(j) = a(j)
S   |  |  c(j) = a(j + m)
    |  end do
    end do
```

One can note that the parameter m, which is in no initial loop bound, influences on locality. If $m < n$, then a subset of the a array is read by both R and S statements. So the schedule of these statements should be the same for the corresponding subset of the iteration domains ($m \leq j_R \leq n$ and $0 \leq j_S \leq n - m$), to maximize the data reuse of a. It implies the m parameter has to be considered in the loop bounds of the transformation.

Had we used Feautrier's method to compute positive affine schedules by applying Farkas lemma also on schedule functions [16], we would have missed the bounds on m, since it is not involved in any loop bound (no Farkas multiplier is attached to it, its schedule coefficient would always have been 0).

The Algorithm

We can now propose an algorithm to compute a polytope of legal affine monodimensional schedules for a program.

The proposed algorithm does an intensive use of the Fourier-Motzkin projection algorithm. This algorithm is known to be super-exponential, and to generate redundant constraints [4, 37].

Algorithm 1.1 Bounded legal transformation space construction

```
1. Build the initial $\mathcal{D}_t$ polytope, by adding $sup$ and $inf$ bounds for each variable.
2. For each dependence $k$
    (a) Build the system:
        • equate the coefficients between the Farkas multipliers of the dependence ($\lambda$) and the schedule coefficients ($t$)
        • add the positivity constraints on the Farkas multipliers
    (b) Solve the system with the Fourier-Motzkin projection algorithm, and keep only $\mathcal{D}_t^k$ the polyhedron on the $t$ dimensions.
    (c) Intersect the obtained polyhedron $\mathcal{D}_t^k$ with $\mathcal{D}_t$.
```

We use a specific implementation of this algorithm, which reduces the number of redundant inequalities, for an approximately identical complexity (see Section 1.7). In addition to that, the polyhedral intersection operation, which is known to be exponential, was adapted: it is done only on sets of constraints generated by our implementation of the projection algorithm, yielding a quadratic operation for the intersection itself. We delayed to the exploration phase the emptiness test of the resulting polyhedron, since we deal with non-simplified polyhedra. Experiments have shown that this approach is suitable even for an exhaustive exploration of the polytope.

1.6.2 Discussions

Size of the Problems

It is worth noting that the size of the systems to solve can be easily expressed. For each statement S, there is exactly:

$$S_S = d_S + |n| + 1$$

Schedule coefficients, where d_S is the depth of the statement S and $|n|$ is the number of structure parameters. An empirical fact is that the domain of a statement S is defined by $2.d_S$ inequalities. Since the dependence polyhedron is a subset of the Cartesian product of statements domains, the number of Farkas multipliers for the dependence (one per row in the matrix, plus λ_0) is:

$$S_{D_{R,S}} = 2.d_R + 2.d_S + p + s + 1$$

Where p is size of the precedence constraints (at most $\min(d_R, d_S) - 1$) and s is the subscript equality.

So the dimension of the computed systems are at most:

$$\begin{array}{rl} R, S \in \Omega, & R\delta S \\ S_{syst} = & S_R + S_S + S_{D_{R,S}} \\ = & 3.d_R + 3.d_S + 2.|n| + min(d_R, d_S) + 4 \end{array}$$

Since d_R, d_S, and $|n|$ are in practice very small integers, it is noticeable that computed input systems are small and tractable. They contain $d_R + d_S + |n| + 1$ equalities and $S_{D_{R,S}}$ positivity constraints for the Farkas multipliers, yielding a system sized enough to be computed. The projection operation consists in projecting a space of size S_{syst} on a space of size $S_R + S_S$ (or a space of size $S_{syst} - d_S$ on a space of size S_S if considering a self-dependence $S\delta S$).

The dimension of $\mathcal{D}_t$ is exactly:

$$S = \sum_{S \in \Omega} (d_S + |n| + 1)$$

Accuracy of the Method

Had we applied Feautrier's method to compute the set of legal values for the Farkas multipliers of the schedules, and explore this (infinite) space, we would have faced the problem that the function giving the schedule coefficients from the schedule Farkas multipliers is not injective (many points of the obtained polyhedron can give the same schedule). This method, under contrary, is very well suited for an exploration of the different legal schedules since every (integral) point in $\mathcal{D}_t$ is a different schedule.

Some other methods (see Long et *al.*[27], for instance) define heuristics to explore the scheduling space of a program for a given combination of transformation (that is, a subset of the schedule space). These methods apply a legality test for the transformation, to ensure the respect of the dependence graph. It is merely impossible to exhaustively explore a search space of schedules, and then for each point test if the transformation is possible. Under contrary, exhaustive search of (an accurate subset of) the *legal* affine monodimensional schedule space is possible.

Figure 1.4 emphasizes the size of the computed search space, and the time to compute it. We compare, for a given example (with #Dependences and #Statements), the number of different Schedules to the number of Legal different schedules, with the same bounds for the schedule coefficients. These results and all the following are computed on a Pentium 4 Xeon, 3.2GHz.

Figure 1.4: Search space computation

Test	#Dep	#St	Bounds	#Sched	#Legal	Time
matvect	5	2	$-1, 1$	3^7	129	0.024
locality	2	2	$-1, 1$	3^{10}	6561	0.022
matmul	7	2	$-1, 1$	3^9	912	0.029
Gauss	18	2	$-1, 1$	3^{10}	506	0.047
CRout	26	4	$-3, 3$	7^{17}	798	0.046

1.7 Polyhedral Projections

There are many methods to solve a system of linear inequalities, some of them giving *one* solution, or simply deciding if a solution exists (see [37] for a comprehensive survey). In the polyhedral model we deal with parametrized $\mathbb{Z}$-polyhedra (also called lattice-polyhedra). But it is a well-known fact that most of the Parametrized Integer Programming algorithms are $\mathcal{NP}$-complete [37]. We chose to extend the computation of polyhedral projections to $\mathbb{Q}$-polyhedra, and then to consider only the integer part (the intersection between the polyhedron and the integer lattice of the same dimension) of the computed polyhedron. The following method holds for reals, and so for rationals.

Also, we need to express the solution set of a system, and not only *one* solution. Since we need to express this set (and implicitly do a polyhedral projection operation) we chose to use the Fourier-Motzkin algorithm as a basis of our work.

In this section we first recall in detail the Fourier-Motzkin elimination method, then we propose a reformulation of the algorithm to heavily reduce the number of redundant constraints. Finally, complexity issues are discussed.

1.7.1 Fourier-Motzkin Elimination

Since the elimination method works well to solve a system of linear equalities, it was natural to investigate a similar method to solve linear inequalities. Fourier first designed the method, which was rediscovered and studied several times, including by Motzkin in 1936 and Dantzig [13].

Practical Formulation

The principle is as follows, and is dragged from [4].

Given a system of linear inequalities:

$$\sum_{i=1}^{m} a_{ij}x_i \leq c_j \quad (1 \leq j \leq n) \tag{1}$$

Where a_{ij} and c_j are rational constants. To solve this system, we eliminate the variables one at a time in the order $x_m, x_{m-1}, ..., x_1$. The elimination consist in a projection of the polyhedron represented by (1) on the polyhedron being the union of the constraints of dimension $m-1$ and less, and the projection of the constraints of dimension m to dimension $m-1$. This polyhedron is represented by:

$$\sum_{i=1}^{m-1} t_{ij}x_i \leq q_j \quad (1 \leq j \leq n') \tag{2}$$

(a) Sort The first step is to rearrange the system (1) such that inequalities where a_{mj} is positive come first, then those where a_{mj} is negative, an those where a_{mj} is 0. We find integers n_1 and n_2 (which will later ease the computation) such that:

$$a_{mj} = \begin{cases} > 0 & \text{if } 1 \leq j \leq n_1, \\ < 0 & \text{if } n_1 + 1 \leq j \leq n_2, \\ = 0 & \text{if } n_2 + 1 \leq j \leq n, \end{cases} \tag{3}$$

(b) Normalize For $1 \leq j \leq n_2$, divide the j^{th} inequality by $|a_{mj}|$, to get:

$$\begin{aligned}
&\sum_{i=1}^{m-1} t_{ij}x_i + x_m \leq q_j \quad (1 \leq j \leq n1)\\
&\sum_{i=1}^{m-1} t_{ij}x_i + x_m \geq q_j \quad (n_1 + 1 \leq j \leq n2)
\end{aligned} \tag{4}$$

Where (for $1 \leq i \leq m-1, 1 \leq j \leq n2$)

$$\begin{cases} t_ij &= a_{ij}/|a_{mj}| \\ q_j &= c_j/|a_{mj}| \end{cases}$$

From (4) we derive $-\sum_{i=1}^{m-1} t_{ij}x_i + q_j$, an upper bound for x_m for $1 \leq j \leq n_1$, and a lower bound for x_m for $n_1 + 1 \leq j \leq n_2$. It is so possible to define respectively b_m, the "lower bound ball", and B_m the "upper bound ball" as:

$$\begin{aligned}
b_m(x_1, ..., x_{m-1}) &= \max_{n+1 \leq j \leq n2} \left(-\sum_{i=1}^{m-1} t_{ij}x_i + q_j \right)\\
B_m(x_1, ..., x_{m-1}) &= \min_{1 \leq j \leq n1} \left(-\sum_{i=1}^{m-1} t_{ij}x_i + q_j \right)
\end{aligned} \tag{5}$$

We define $b_m = -\infty$ if $n_1 = n_2$ (no lower bound) and $B_m = \infty$ if $n_1 = 0$ (no upper bound). So we can express the range of x_m:

$$b_m(x_1, ..., x_{m-1}) \leq x_m \leq B_m(x_1, ..., x_{m-1}) \tag{6}$$

The equation (6) is a description of the solution set for x_m.

(c) Create projection We now have the solution set for x_m, and we need to build the constraints for the x_{m-1} dimensions. In addition to the constraints where $a_{mj} = 0$, we simply linearly add each constraints where $a_{mj} > 0$ to each constraints where $a_{mj} < 0$, and add the obtained constraint to the system. One may note that we add $n_1(n_2 - n_1)$ inequalities, and the new system has $n' = n - n_2 + n_1(n_2 - n_1)$ inequalities for $m - 1$ variables. The original system (1) has a solution iff this new system has a solution, and so on from m to 1. If during this step, a contradiction occurs ($0 \leq q_j$ with $q_j < 0$) then the system has no solution.

Once the algorithm terminated (and did not yield an unsolvable system), it is possible to build the set of solutions by simply computing, for $k = 1$ to $k = m$, the values of b_k and B_k (yielding the bounds of acceptable values for x_k).

The Algorithm

From the previous formulation, we can derive the simple Algorithm 1.2.

1.7.2 Redundancy Reduction

The major drawback of the Fourier-Motzkin elimination algorithm is the strong possibility to generate redundant constraints during the step 4 of the algorithm. A constraint is redundant if it is implied by (an)other constraint(s) of the system. One may note that removing a redundant

Algorithm 1.2 Fourier-Motzkin elimination

Input: A system of n linear inequalities of the form

$$\sum_{i=1}^{m} a_{ij}x_i \leq c_j \quad (1 \leq j \leq n)$$

Output: The solution set of the input system

```
• for $k = m$ to 1 do
    1. **Sort** the system regarding the sign of $a_{kj},\ \forall j$. Compute $n_1$ the number of inequalities
       where $a_{kj} > 0$, $n_2$ the number of inequalities where $a_{kj} < 0$.
    2. **Normalize** the system, by dividing each inequalities where $a_{kj} \neq 0$ by $|a_{kj}|,\ \forall j$.
    3. Store $b_k$ and $B_k$ the lower and upper bound inequalities for $x_k$.
    4. **Create** the system for $x_{k-1}$, by adding each inequality where $a_{kj} < 0$ to each one
       where $a_{kj} > 0$ (use $n_1$ and $n_2$ to find bounds for $j$). If a contradiction occurs, **stop: no
       solution**. Add the inequalities where $a_{kj} = 0$. If the system is empty, **stop: finished**.
```

constraint can make the other redundant constraints irredundant, so it is not generally possible to remove them all at the same time.

Definition of Redundancy

We propose to distinguish two types of redundancy. We differentiate the **local redundancy**, where the redundancy can be observed between two constraints, and the **global redundancy** where a constraint has to be checked against the whole system.

Definition 3 (Local redundancy) *Given two constraints:*
$C : \sum_{i=1}^{n} c_i x_i \leq q$
$D : \sum_{i=1}^{n} d_i x_i \leq q'$
C is said to be locally redundant with D if one of the following holds:

(i) $\forall k,\ c_k = d_k$ *and* $q' \leq q$ *(parallel hyperplanes)*

(ii) $\exists k,\ c_k \neq 0,\ C/|c_k| = D/|d_k|$ *(coefficients equality)*

The three following examples bring to light local redundancy.

- $x + y \geq 2$ is redundant with $x + y \geq 1$
  (The two constraints define parallel hyperplanes.)

- $2x + 2y \geq 2$ is redundant with $x + y \geq 1$
  (The two constraints are equal when normalized.)

- $2x + 2y \geq 2$ is redundant with $x + y \geq 0$
  (The two constraints define parallel hyperplanes.)

Definition 4 (Global redundancy) *A constraint is said to be globally redundant if it is not locally redundant with any other constraint of the system, and the system defines the same polyhedron if we remove this constraint.*

An intuition of globally redundant constraints is given in Figure 1.5. A sure way to detect globally redundant constraints is to replace the constraint in the system by its opposite, and to check if there is a solution (with for example the Simplex algorithm). If so, then the constraint is irredundant.

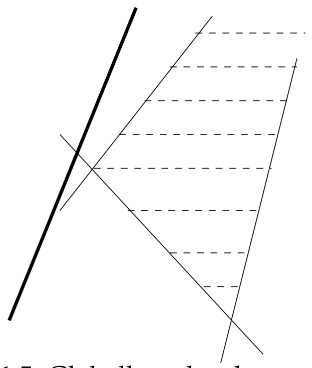


Figure 1.5: Globally redundant constraint

Reformulation of the Algorithm

The Fourier-Motzkin projection algorithm can be modified to guarantee the elimination of local redundancy. In addition to that, we can also reduce the complexity by using six smaller sets of constraints instead of one (their cumulative size is at most the same as the original set), as shown in Algorithm 1.3.

The benefits of this formulation is twofold: first, the **sort** step is removed, since it has been replaced by a simple test and three different sets; second, the STORE operation is done only on normalized constraints.

The STORE operation is the core of the local redundancy elimination. We assure by construction that we only store normalized constraints (let us recall that this normalization step was already mandatory to the projection algorithm). To detect if a constraint $C : \sum_{i=1}^{m} c_i x_i \leq q_c$ is locally redundant we only have to check, with each constraint D already present in the set, if $\forall i, c_i = d_i$. If so, we check if $q_d \leq q_c$. A very simple hash function for the constraints helps the redundancy check to be fast.

Related Work

The study of eliminating redundant constraints is a long-term issue, and was treated by various authors. Let us cite Chernikov's work [10, 29] who produced the *reduced convolution method*, which relies on the concept of linear-dependence of the vectors generating a cone. Let us also recall the work of Pugh [34] on the Omega Test, a practical method to solve integer linear programs; and Sehr et *al.* [38] who worked on redundancy reduction heuristics based on the Chernikov criterion. We may also note the work of Weispfenning [44] who produced a derived of the Fourier-Motzkin elimination with an exponential worst case upper bound complexity.

Algorithm 1.3 Modified Fourier-Motzkin elimination

```
Input: A system of $n$ linear inequalities of the form
$\sum_{i=1}^{m} a_{ij}x_i \leq c_j \quad (1 \leq j \leq n)$
Output: The solution set of the input system

• $\forall j$ where $a_{mj} \neq 0$, **Normalize** the constraint by $|a_{mj}|$. Apply the following rule:
    (i) if $a_{mj} > 0$ STORE the inequality in $S_+$
    (ii) if $a_{mj} < 0$ STORE the inequality in $S_-$
    (iii) if $a_{mj} = 0$ STORE the inequality in $S_0$
• $B_m = S_-,\ b_m = S_+$
• For k = m - 1 to 1 do
    1. $\forall s_+, s_- \in S_+ \times S_-, C = s_+ + s_-$. If a contradiction occurs, **stop: no solution**. $\forall s_0 \in S_0$
       where $s_{0k-1} \neq 0,\ C = s_0$. If $k > 1$, $\forall C$, **normalize** $C$ on the $k - 1^{th}$ variable.
        (i) if $c_{k-1} > 0$ STORE the inequality in $S'_+$
        (ii) if $c_{k-1} < 0$ STORE the inequality in $S'_-$
        (iii) if $c_{k-1} = 0$ STORE the inequality in $S'_0$
    2. If $S'_+ = \emptyset \wedge S'_- = \emptyset \wedge S'_0 = \emptyset \wedge S_0 = \emptyset$ **stop: finished**
       If $k > 1,\ B_k = S_-, b_k = S_+$
       If $k = 1,\ B_k = \max(S_+), b_k = \min(S_-)$
       $S_+ = S'_+,\ S_- = S'_-,\ S_0 = S'_0$
```

1.7.3 Complexity Issues

The Fourier-Motzkin projection algorithm is known to be super-exponential. We proposed a slight reformulation, which obviously solves the input system (it is possible to go back to the original, proved formulation of the Fourier-Motzkin algorithm from this formulation only by reordering the algorithm steps).

Complexity of the Original Fourier-Motzkin Algorithm

The total number of polynomials in the output of the Fourier-Motzkin algorithm is bounded in the worst case by ($|S|$ is the size of the original system) [44]:

$$\sum_{i=1}^{m} \frac{|S|^{2(i-1)}}{2^{2^i-1}} + \frac{|S|^{2^m}}{2^{(2^{m+1}-2)}}$$

And hence, more roughly if $|S| \geq 2$, by

$$(m+1)\left(\frac{|S|}{2}\right)^{2^m}$$

At each elimination step k of the algorithm (the elimination of a variable of dimension k), we can approximate the number of operations done, for n inequalities in the system:

- The *sort* step can be done in n operations.
- The *normalize* step can be done in k operations per line, so $k \times n$ operations.
- The *create projection* step can be done in, at worse, $\frac{n^2}{4} \times k$ operation (there is exactly $\frac{n}{2}$ positive constraints and $\frac{n}{2}$ negative ones to add to each other).

So the total count of operations can be approximately bounded in the worst case by:

$$\sum_{k=m}^{1} \left(n_k + k \left(n_k + \frac{n_k^2}{4} \right) \right)$$
$$\text{Where } n_k = \frac{n_{k-1}^2}{4}, n_m = n$$

Complexity of the Modified Fourier-Motzkin Algorithm

The modified Fourier-Motzkin algorithm obviously has the same worst bound complexity for polynomials output (the input system may at worse never produce locally redundant constraint, and so the algorithm behaves like the original formulation).

At each elimination step k of the modified algorithm, we can approximate the number of operations done, for n inequalities in the system:

- The *normalize* step can be done in k operations per line, so $k \times n$ operations.
- The *create projection* step can be done in, at worse, $\frac{n^2}{4} \times k$ operation (there is exactly $\frac{n}{2}$ positive constraints and $\frac{n}{2}$ negative ones). There will be $\frac{n^2}{4}$ *store* operations.

The complexity of the *store* operation can be bounded in the worst case by $St = k \times \frac{n^2}{4}$, but we provide an implementation empirically shown to have a $St = k \log \frac{n^2}{4}$ complexity.

So the total count of operations can be approximately bounded in the worst case by:

$$\sum_{k=m}^{1} \left(k \left(n_k + \frac{n_k^2}{4} . \left(1 + \log \left(\frac{n_k^2}{4} \right) \right) \right) \right)$$
$$\text{Where } n_k = \frac{n_{k-1}^2}{4}, n_m = n$$

Which is a small complexity overhead with regards to the original formulation. Let us recall that it is a worst case bound, and we do not take in account the huge advantage of local redundancy elimination. We experimentally noticed systems where the local redundancy elimination can reduce by a factor 2 the number of constraints at each variable elimination step, yielding a 2^m reduction of the polynomials output.

1.8 Experimental Results

We aim here at computing the performance distribution of a set of legal schedules for a given program. In the following section we present the experimental protocol we used, and discuss the obtained results.

1.8.1 Experimental Protocol

We implemented three separated tools for this protocol, all in `C`: `Candl`, a dependence analyzer in the polyhedral model; `FM`, a fast and redundancy-reduced implementation of the Fourier-Motzkin projection algorithm; and `LetSee`, the Legal Transformation SpacE Explorator which implements the algorithm of Section 1.1 for the legal space construction, and a polytope explorer.

The `LeTSeE` software aims at computing the legal transformation space of a program, given a matrix representation of the iteration domains of all statements, the dependence graph labeled by the dependence polyhedra, and the bounds of the transformation coefficients. Once the legal transformation polytope is computed, it is exhaustively explored, generating one `CLooG` [1] formatted file (containing the schedule and all the necessary information to generate the kernel code) per point in the polytope.

The whole test protocol respects the following:

- Use `LeTSeE` to generate a `CLooG` formatted file per transformation.
- For each generated file:
  - Generate the kernel code with `CLooG` (with by default options).
  - Post-treatment of the generated code, to add measure tools and kernel input initialization.
  - Compile and run, for various compilers and compiler options.
- Collect and format the results.

The measure tools count the number of cycles used by the program. In order to limit interferences with other programs, we set the scheduler for the tested program as FIFO.

In order to be consistent, the original code is included in the test protocol, as the identity transformation (it may be a multidimensional transformation in this case, to respect the exact original form of the program).

1.8.2 Legal Transformation Space Exploration

The `LeTSeE` software implements a naive recursive polytope explorer. In order to achieve a fast computation of the legal transformation polytope we built a non-simplified polytope, we only assure by construction that there are no locally redundant constraints. In fact, it is even possible to have contradictory constraints generating an empty set of legal monodimensional schedules. As said earlier, the emptiness test is delayed to the exploration phase. For the sake of simplicity, and since it is not of a critical matter, we implemented an exhaustive recursive polytope explorer.

It is thus possible to simplify $\mathcal{D}_t$ with for example the PolyLib, a state-of-the-art polyhedral computation library.

Transformations are named with the exploration point number. The exploration is done dimension by dimension, from the inner to the outer most dimension (the inner most dimension corresponds to the last statement coefficient – the 1 coefficient, while the outer most corresponds to the first statement's first iterator coefficient).

[1]Chunky Loop Generator, freely available at `http://www.cloog.org`

1.8.3 Performance Distribution

In the following we compare the performance distributions for some known examples, emphasizing the variation regarding the compiler, the compiler options or the parameters size. Let us recall that this study does not aim to compare compilers themselves.

1.8.4 The Compiler as an Element of the Target Platform

Our iterative optimization scheme is independent from the compiler and may be seen as a higher level to classical iterative compilation. In the same way as a given program transformation may better exploit a feature of a given processor, it also may enable more aggressive options of a given compiler. Because production compilers have to generate a target code in any case in a reasonable amount of time, their optimizations are very fragile, i.e. a slight difference in the source code may enable or forbid a given optimization phase.

To study this behavior and estimating how a higher level iterative optimization scheme may lead to better performances, we achieved a exhaustive scan of our search space for various programs and compilers with aggressive optimization options. We illustrate our results in Figure B.2, and with more details in Figure 1.6 for the matrix-multiply kernel shown in Figure 1.6, a very classic computational kernel. This kernel benchmark has been extensively studied, and is a typical target of aggressive optimizations of production compilers.

Program 1.6 `matmul`

```
    do i = 1, n
    |  do j = 1, n
S1  |  |  C(i,j) = 0
    |  |  do k = 1, n
S2  |  |  |  C(i,j) = A(i,k) * B(k,j)
    |  |  end do
    |  end do
    end do
```

We tested the whole set of legal schedules within the bounds $-1, 1$ for all coefficients (912 points), and checked the speedup for various compilers with aggressive optimizations enabled. Matrices are `double` arrays of size 250×250. We compared, for a given compiler, the number of cycles the original code took (Original) to the number of cycles the best transformation took (Best) (results are in millions of cycles).

Figure 1.6: Results for the `matmul` example

Compiler	Option	Original	Best	Schedule	Speedup
GCC 3.4.2	-O3	1076	686	$\theta_{S1}(\vec{x}_{S1}) = -i + j - n - 1$ $\theta_{S2}(\vec{x}_{S2}) = k + n + 1$	57%
GCC 4.1.1	-O3	958	643	$\theta_{S1}(\vec{x}_{S1}) = -i + n - 1$ $\theta_{S2}(\vec{x}_{S2}) = k + n + 1$	49%
ICC 9.0.1	-fast	465	72	$\theta_{S1}(\vec{x}_{S1}) = -i + n$ $\theta_{S2}(\vec{x}_{S2}) = k + 1$	645%
PathCC 2.5	-Ofast	228	79	$\theta_{S1}(\vec{x}_{S1}) = j - n - 1$ $\theta_{S2}(\vec{x}_{S2}) = k$	308%

Figure 1.6 shows significant speedups achieved by the best transformations for each back-end compiler. Such speedups are not uncommon when dealing with the matrix-multiplication kernel. The important point is that we do not perform any tiling (it requires multi-dimensional schedules), contrary to nearly all other works (see [2, 47] for useful references). It was possible to check using PathScale EKOPath that many optimization phases have been enabled or disabled, depending on the version generated from our exploration tool. Nevertheless it is technically hard to know precisely the contribution of the one-dimensional schedule (which has a high potential, by itself, as an optimizing transformation) with respect to the enabled compiler optimizations. But another striking result is the high variation of the best schedules depending on the compiler. For instance the lack of the j iterator in $\theta_{S1}(\vec{x}_{S1})$ for GCC or the lack of the n parameter $\theta_{S2}(\vec{x}_{S2})$ for ICC.

These results, which are consistent with the other tested programs, emphasize the need of a compiler-dedicated transformation to achieve the best possible performance. One possible explanation is the difference between optimization phases in the different back-end compilers. Compilers have attained such a level of complexity that it is no longer possible to model the effects of downstream phases on upstream ones. Yet it is mandatory to rely on the downstream phases of a back-end compiler to achieve a decent performance, especially those which cannot be embedded naturally in the polyhedral model.

Let us note that we do not perform any tiling or Index Set Splitting, and we consider only monodimensional schedules to the contrary of other well-known works (see [46] for instance).

We can also emphasize the transformation specificity for each different compiler, by comparing the speedup of the best obtained transformation for a given compiler when runned on another compiler (and compare it to the speedup attained by the best transformation found specifically for this compiler). The result is shown in Figure 1.7.

Figure 1.7: Compiler specifics

Comp.	GCC 3.4.2	GCC 4.0.1	ICC 9.0.1
GCC 3.4.2	–	$-1,8\%$	$-3,2\%$
GCC 4.0.1	$-0,3\%$	–	$-3,2\%$
ICC 9.0.1	$-1,5\%$	$-1,2\%$	–

Variation Between Compiler Options

Experiments have shown a dependence between the best transformation and the compiler options used. For instance, in the *matmul* case with the `ICC 9` compiler used with the aggressive `-fast` option, the best transformation obtained yields a 4.5% speeddown when this transformation is compiled with `-O2` and compared to the best one found for this compiler option. This behavior was observed on all the tested programs.

Importance of Parameters Sizes

Another observed behavior is the consistency of the best transformation among parameters sizes. Most of the performance obtained comes from locality improvements, that is, a better use of the different processor cache levels. It seems obvious that when the program uses a memory footprint larger than the cache size, increasing the memory footprint will not influence on the choice of the best transformation. But we also observed an approximate consistency of the ranking of transformations with small memory footprints. This observation comes from the fact (as shown in distributions of Figure 1.8 and 1.9) that an important rate of legal schedules are highly ineffective regarding the original code. And these schedules are of course also ineffective on small inputs.

A possible exploration heuristic could be to run the whole set of legal transformations on very small inputs (so the time per explored point would be reduced) in order to eliminate some ineffective schedules. Then, on the remaining set of possibly good schedules, the exploration is runned again with an accurate memory footprint regarding the memory hierarchy of the machine.

Notes on the Distribution

An exhaustive exploration of a legal transformation space let us be able to print and observe the performance distribution among this space. Figure 1.8 shows this distribution for the *matmul* and the *locality* example. We can observe that for the *matmul* example, many schedules have a similar performance with the best one while in the *locality* example it is opposite. If the distribution for *matmul* could seem chaotic (except the regularity for the good schedules), under contrary we can observe many regularities for *locality* (especially two clouds of ineffective points). The very small set of good schedules for *locality* ($\approx 30/6500$) let us claim inefficiency of a random exploration method, at least for this example.

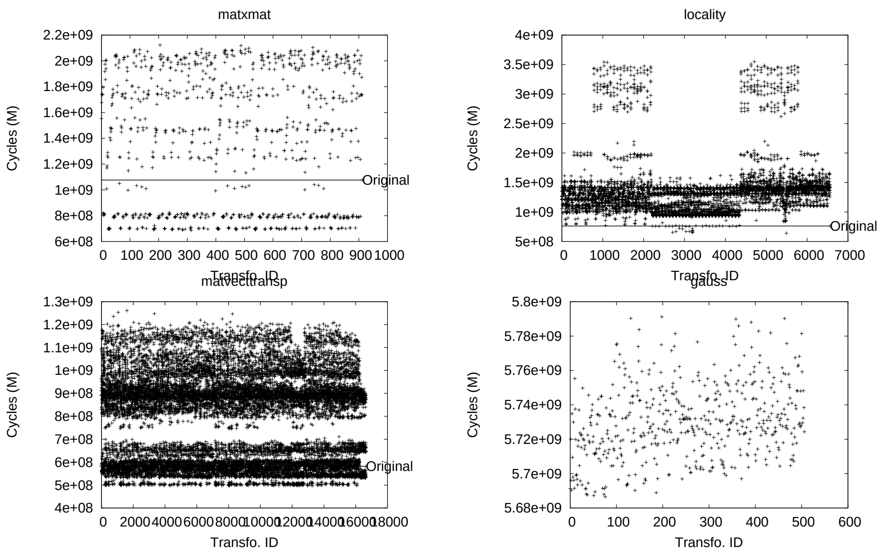


Figure 1.8: Performance distribution for *matmul*, *locality*, *MVT* and *Gauss* (`GCC 4 -O2`)

The shape of the distribution can be even more emphasized with the *crout* example (see Program 1.7), as shown in Figure 1.9.

The differences in the distribution regarding the compiler option is emphasized in Figure 1.10. We compare, for an identical original program, the shape of the distribution on `GCC 4`, with the `-O1`, `-O2`, `-O3` and `-fast` options on the *locality* example.

Program 1.7 `crout`

```
     do j = 1, n
       do i = 1, j
         do l = 1, i
S1         K(i,j) = K(i,j) - K(l,i) * K(l,j)
         end do
       end do
       do i = 1, j
S2       T(i) = K(i,j)
S3       K(i,j) = T(i) / K(i,i)
S4       K(j,j) = K(j,j) - T(i) * K(i,j)
       end do
     end do
```

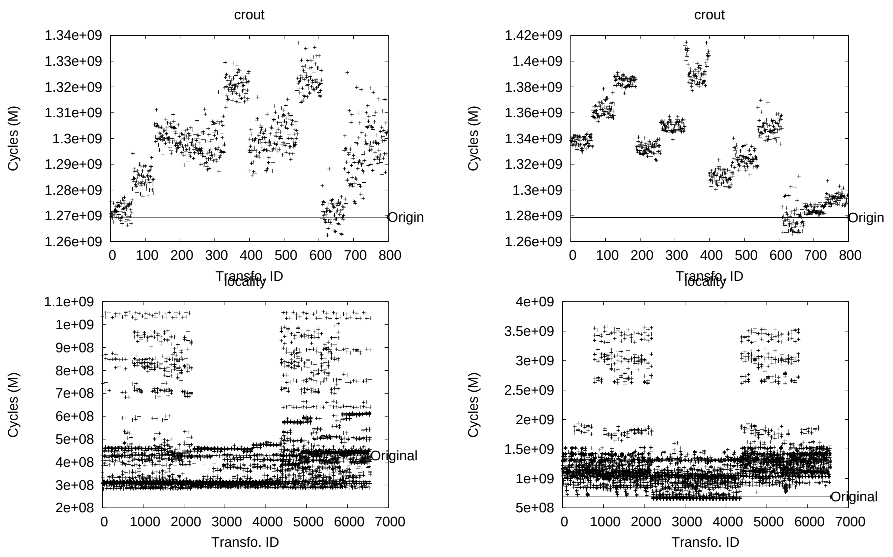


Figure 1.9: Performance distribution for *crout* and *locality* (`ICC 9 -fast` and `GCC 4 -O3`)

More Experimental Results

The table of Figure 1.13 summarize performance improvements obtained for some of the examples we tested. We give the speedup obtained with `ICC 9.0.1 -fast` (S. ICC) and with `GCC 4.0.1 -O3` (S. GCC). The *MVT* sample is the computation of a vector by a matrix, and another vector with the transpose of this matrix (see Program 1.8). All input data are `long double`. The reader is encouraged to report to Appendix B for an exhaustive survey of the obtained results.

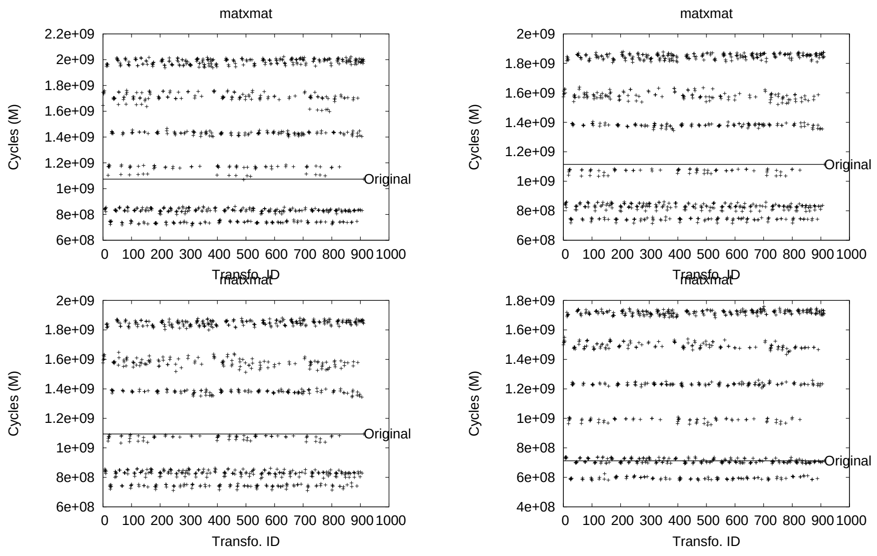


Figure 1.10: Performance distribution for *matmul* (`ICC 9`, options `-O1`, `-O2`, `-O3` and `-fast`)

Program 1.8 `MVT`

```
    do i = 1, n
S1  |  x1(i) = 0
S2  |  x2(i) = 0
    |  do j = 1, n
S3  |  |  x1(i) = x1(i) + A(i,j) * y1(j)
S4  |  |  x2(i) = x2(i) + A(j,i) * y2(j)
    |  end do
    end do
```

Figure 1.11: Experiments summary

Test	#Dep	#St	Bounds	#Sched	#Legal	S. ICC	S. GCC
locality	2	2	$-1, 1$	3^{10}	6561	51%	28%
matmul	7	2	$-1, 1$	3^{9}	912	23%	49%
Gauss	18	2	$-1, 1$	3^{10}	506	2%	1%
Crout	26	4	$-3, 3$	7^{17}	798	0.5%	1.3%
MVT	14	4	$-1, 1$	3^{14}	16641	26%	30%

1.8.5 Discussions

The polyhedral model aims at parallelism detection and modelisation. We chose to emphasize here the benefits of our method only on single processor machines, to show how program rescheduling can generate performance improvements. All the programs we tested have at least one degree of parallelism, not exploited in our experiments. It is worth noting that the observed speedup should increase had we taken benefits from the parallelism we implicitly expressed, especially on multi-core chips.

We may also note that many observed best transformations have a weird shape, and would seem inaccurate at first glance to a compiler specialist. This comes from the conjunction of the schedule, the code generator and the compiler. The code generator applies a first optimization phase, embedded in the code generation algorithm [5, 41]. Then, the compiler applies its own optimization phases, yielding a complex and multiple transformation process for the schedule. This led us to observe potentially ineffective code (that is, with too much complex controls) produced by the code generator be able to trigger optimizations in the compiler the original code would not.

1.8.6 Heuristic Traversal of the Optimization Space

Since it is unpractical to explore the whole search space on real-world benchmarks, we propose a heuristic to enumerate only a high-potential sub-space, using the properties of the polyhedral model to characterize the highest potential and narrowest one.

Decoupling Heuristic

We represent the schedule coefficients of a statement as a three component vector:

$$\theta_S(\vec{x}_S) = (\vec{\imath}\,\vec{p}\,c)\begin{pmatrix}\vec{x}_S\\ \vec{n}\\ 1\end{pmatrix}$$

Where $\vec{\imath}$ represents the iterators coefficients, $\vec{p}$ the parameters coefficients and c the constant coefficient.

In this search space representation, two neighbor points may represent a very different generated code, since a minor change in the $\vec{\imath}$ part can drastically modify the compound transformation (a program where *interchange* and *fusion* are applied can be the neighbor of a program with none of these transformations). The most significant impact on the generated code is caused by iterator coefficients, and we intuitively assume their impact on performance will be equally important. Conversely, modifying parameters or constant coefficients is less critical (especially when one-dimensional schedules are considered). Hence it is relevant to propose an exploration heuristic centered on the enumeration of the possible combinations for the $\vec{\imath}$ coefficients.

The proposed heuristic is window-search based. It decouples iterator coefficients from the others, enabling a systematic exploration of all the possible combinations for the $\vec{\imath}$ part. At first, we do not care about the values for the $\vec{p}$ and c part (they can be chosen arbitrarily in the search space, as soon as they are compatible with the $\vec{\imath}$ sequence). The resulting subset of program versions is then filtered with respect to effective performance, keeping the top points only. Then, we repeat the systematic exploration of the possible combination of values for the $\vec{p}$ and c coefficients to refine the program transformation sequence.

The heuristic can be sketched in 5 steps.

1. Build the set of all different possible combinations of coefficients for the $\vec{\imath}$ part of the schedule, inside the set of all legal schedules. Choose $\vec{p}$ and c at random in the space, according to the $\vec{\imath}$ part.

2. For each schedule in this set, generate and instrument the corresponding program version and run it.
3. Filter the set of schedules by removing those associated with a run time more than $x\%$ slower than the best one (combined with a bound on the limit of selected schedules).
4. For each schedule in the remaining set, explore the set of possible values for the $\vec{p}$ and c part (inside the set of all legal schedules) while the $\vec{\imath}$ part is left unchanged.
5. Select the best schedule and generated program in this set.

Discussion

Figure 1.12 details a run of our decoupling heuristic, and compares it with a plain random search for some of our kernel benchmarks. It shows the relative percentage of the best speedup achieved as a function of the number of iterative runs.

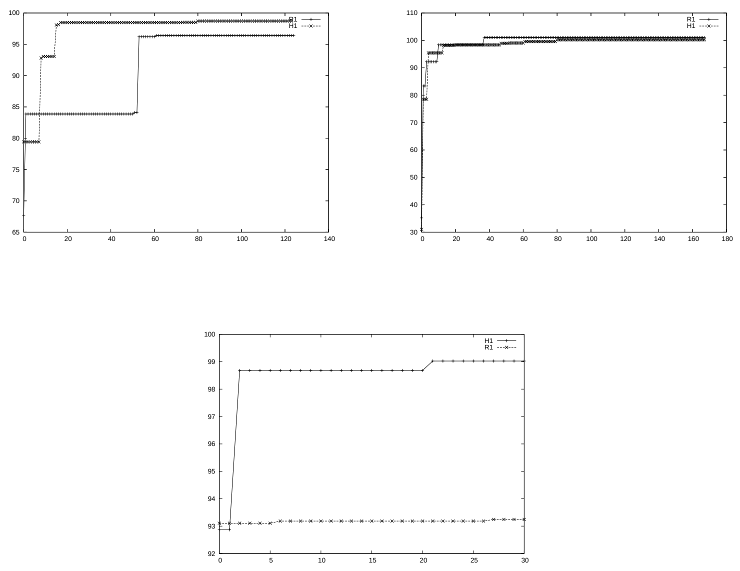


Figure 1.12: Comparison between random and decoupling heuristics (`locality`, `matmul`, `mvt` examples)

A fast convergence of the decoupling heuristic is attained on some examples like `crout` or `locality`, where regularities in the distribution can be observed. On these tested examples, more than 95% of the maximum speedup can be achieved with an order of magnitude reduction of the number of runs, compared to an exhaustive scan.

On the other hand, we observed a suboptimal behavior of the heuristic comparatively to a full random driven approach, as Figure 1.12 shows for the `matmul` kernel. Not surprisingly, as soon as the density of interesting transformations is large, a random space scan may converge faster than our enumeration-based method.

A more important problem is the scalability to larger SCoPs. To prevent the possibly large set of legal values for the $\vec{\imath}$ coefficients, it is possible to:

1. impose a static or dynamic limit to the number of runs, which should be coupled to an exploration strategy starting with coefficients as close as possible to 0 (remember 0 may not correspond to any legal schedule);
2. to replace an exhaustive enumeration of the $\vec{\imath}$ combinations by a limited set of random draws in the $\vec{\imath}$ space.

The choice between the exhaustive, limited or random exploration of the $\vec{\imath}$ space can be heuristically determined with regards to the size of the original SCoP (this size gives a good intuition of the order of magnitude of the size of the search space).

The table of Figure 1.13 summarizes the obtained results of the heuristic comparatively to a full random driven one. The filtering level is 5%.

Figure 1.13: Heuristic convergence

Example	#Schedules	Heuristic.	#Runs	%Speedup
locality	6561	Rand	125	96.1%
		DH	123	98.3%
matmul	912	Rand	170	99.9%
		DH	170	99.8%
mvt	16641	Rand	30	93.3%
		DH	31	99.0%

1.9 Future Work

In the following section we describe some short and long term objectives of our research.

1.9.1 Affine Multidimensional Schedules

It is always possible to find a multidimensional affine schedule to a SCoP, while a monodimensional schedule may not exists. But the generalization of our method to multidimensional schedules lead us to a combinatorial barrier: if there is exactly one way to choose the set of dependences to satisfy in the monodimensional case (they must all be satisfied in one dimension, i.e. in one set), there is a combinatorial way to choose the sets as soon as there is more than one dimension. Feautrier proposed a greedy algorithm to solve the maximal set of dependences at a given depth, and increment the depth if unresolved dependences remain [17]. This would give us the minimal sequential depth of the schedule [43], but the combinatorial remains if we want to explore all the possibilities to satisfy the dependences. We are currently investigating non-exhaustive space construction, including a method implying correction of monodimensional schedules to multidimensional schedules, thanks to the correction of violated dependences [42]. We also investigate the expression of equivalences between transformations.

1.9.2 Exploration of the Legal Transformation Space

There are many possible ways to explore the legal transformation space. We are currently investigating elimination of subsets of the space (we can anticipate inefficiency of the code with regards to the code generation phase), and different exploration heuristics regarding the type of the dimensions (iterators, parameters or constant). We also plan to evaluate efficiency of stochastic methods to estimate the space distribution.

1.9.3 Improving the Legal Space Computation

To the contrary of many improvements proposed by Feautrier, we do not simplify the systems computed for each dependence. More, we claim to compute a non simplified polyhedron of legal schedules. We are investigating the reduction of each computed system thanks to a Gaussian elimination to reduce the complexity. We are also looking for a local (which means to stop needing to store a global solution polyhedron) definition of the legal transformation space, by using the properties of the dependence graph. This would let us be able to reconstruct a global and yet simplified solution polyhedron.

1.9.4 Projection Algorithm

The work initiated on the Fourier-Motzkin projection algorithm lead us to a practical and yet general reformulation of the algorithm. We plan to use some geometric properties of the legal transformation space to make an ad-hoc version of the algorithm, which would get rid of global redundancy for a small complexity overhead.

1.9.5 Scalability and Integration

One of the drawback of the polyhedral model is to be known as a highly costly computational model, mainly due to the complexity of algorithms on parametrized integer linear programming. Many important works have been done to make the model be able to handle real-life programs in real-life compilers [7, 21], and the method we propose aims at being integrated in the Polyhedral Loop Optimization part of GCC (see [32, 33] for some recent work about it).

The iterative compilation domain is in constant progress thanks to the popularity of the model, and integrating these methods in production compilers (see [20] for instance) are a hot topic today. Our method, which at the moment is at the prototype status, has to take benefits from these researches before being integrated in a production compiler.

1.9.6 Parallelism

The polyhedral model is designed to express in a natural way parallelism inside loop nests. Our study was only applied to mono-processor machines, but it is a short term assignment to exploit this parallelism with a state-of-the-art parallel library like OpenMP [12]. We simply need to hardly modify the code generation phase in order to generate an OpenMP equipped C code.

1.10 Conclusion

Iterative and empirical search techniques are one of our last hope to harness the complexity of modern processors and compilers. Unfortunately, all approaches published so far rely on a separate legality checking to filter any illegal candidate program transformation.

The proposed method express the whole set of legal affine monodimensional schedules for a program, that is the expression of *all* the possible combination of transformations for this class of schedules. Our first experimental results bring to light the capability of the method to discover the best transformation for a program thanks to an exhaustive exploration of that space, for a given compiler, architecture and dataset. To our knowledge, this is the first time such a space is explored.

It is expectable that the systematic exploration process will not be doable on large programs, or with multidimensional schedules. But our study let us do observations on the performance distributions in the transformation space. We plan to extract this knowledge in order to generalize our method to multidimensional schedules and then to be able to scale to real-life programs.

Bibliography

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[2] J. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[3] L. Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 231–239, New York, 2004.

[4] Utpal Banerjee. *Loop Transformations for Restructuring Compilers, the Foundations*. Kluwer Academic Publishers, 1993.

[5] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, september 2004.

[6] C. Bastoul. *Improving Data Locality in Static Control Programs*. PhD thesis, University Paris 6, Pierre et Marie Curie, december 2004.

[7] C. Bastoul, A. Cohen, A. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, pages 209–225, College Station, october 2003.

[8] Cédric Bastoul and Paul Feautrier. Adjusting a program transformation for legality. *Parallel processing letters*, 15(1):3–17, March 2005.

[9] F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback Directed Compilation*, Paris, October 1998.

[10] S.N. Chernikov. The convolution of finite systems of linear inequalities. *Zh. vychisl. Mat. mat. Fiz.*, 5:3 – 20, 1969.

[11] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *2nd Workshop on Feedback-Directed Optimization*, Israel, November 1999.

[12] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998. See http://www.openmp.org.

[13] George B. Dantzig and B. Curtis Eaves. Fourier-motzkin elimination and its dual. *J. Comb. Theory, Ser. A*, 14(3):288–297, 1973.

[14] P. Feautrier, J. Collard, and C. Bastoul. Solving systems of affine (in)equalities. Technical report, PRiSM, Versailles University, 2002.

[15] Paul Feautrier. Parametric integer programming. *RAIRO Recherche opérationnelle*, 22(3):243–268, 1988.

[16] Paul Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *Int. J. Parallel Program.*, 21(5):313–348, 1992.

[17] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Program.*, 21(5):389–420, 1992.

[18] Paul Feautrier. Scalable and modular scheduling. In Andy D. Pimentel and Stamatis Vassiliadis, editors, *SAMOS*, volume 3133 of *Lecture Notes in Computer Science*, pages 433–442. Springer, 2004.

[19] Grigori Fursin, Albert Cohen, M. O'Boyle, and Olivier Temam. A practical method for quickly evaluating program optimizations. In *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'05)*, number 3793 in LNCS, pages 29–46, Barcelona, November 2005. Springer-Verlag.

[20] Grigori Fursin, Albert Cohen, Michael F. P. O'Boyle, and Olivier Temam. Quick and practical run-time evaluation of multiple program optimizations. *Trans. on High Performance Embedded Architectures and Compilers*, 1(1):13–31, 2006.

[21] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 2006. Accepted for publication.

[22] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symposium on the frontiers of massively parallel computation*, McLean, 1995.

[23] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical report, College Park, MD, USA, 1993.

[24] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 12–23, San Diego, California, USA, 2003. ACM Press.

[25] Prasad A. Kulkarni, Stephen R. Hines, David B. Whalley, Jason D. Hiser, Jack W. Davidson, and Douglas L. Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Trans. Archit. Code Optim.*, 2(2):165–198, 2005.

[26] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 201–214, New York, NY, USA, 1997. ACM Press.

[27] Shun Long and Grigori Fursin. A heuristic search algorithm based on unified transformation framework. In *ICPPW '05: Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, pages 137–144, Washington, DC, USA, 2005. IEEE Computer Society.

[28] Shun Long and Grigori Fursin. Systematic search within an optimisation space based on unified transformation framework, 2006.

[29] A.V. Lotov, V.A. Bushenkov, and G.K. Kamenev. *Feasible Goals Method – Search for Smart Decisions*. Computing Centre RAS, Moscow, 2001.

[30] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, 1996.

[31] Antoine Monsifrot, Francois Bodin, and Rene Quiniou. A machine learning approach to automatic production of compiler heuristics. In *AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50, London, UK, 2002. Springer-Verlag.

[32] Sébastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developper's Summit (to appear)*, Ottawa, Canada, June 2006.

[33] Sébastian Pop, Albert Cohen, P. Jouvelot, and G.-A. Silber. The new framework for loop nest optimization in GCC: from prototyping to evaluation. In *Proc. of the 12th Workshop Compilers for Parallel Computers (CPC'06)*, A Coruña, Spain, January 2006.

[34] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, New York, NY, USA, 1991. ACM Press.

[35] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, october 2000.

[36] Robert Schreiber and Gilles Villard. Lattice-based memory allocation. *IEEE Trans. Comput.*, 54(10):1242–1257, 2005. Member-Alain Darte.

[37] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[38] G. Stehr, H. Graeb, and K. Antreich. Analog performance space exploration by fourier-motzkin elimination with application to hierarchical sizing. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 847–854, Washington, DC, USA, 2004. IEEE Computer Society.

[39] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Not.*, 38(5):77–90, 2003.

[40] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.

[41] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS, pages 185–201, Vienna, Austria, March 2006. Springer-Verlag.

[42] Nicolas Vasilache, Cédric Bastoul, Sylvain Girbal, and Albert Cohen. Violated dependence analysis. In *Proceedings of the ACM International Conference on Supercomputing (ICS'06)*, Cairns, Australia, june 2006. ACM.

[43] Frédéric Vivien. On the optimality of Feautrier's scheduling algorithm. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 299–308, London, UK, 2002. Springer-Verlag.

[44] V. Weispfenning. Parametric linear and quadratic optimization by elimination. Number MIP-9404. 1994.

[45] T. Wiegand, G. Sullivan, and A. Luthra. Itu-t rec. h.264 – iso/iec 14496-10 avc - final draft. Technical report, Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, May 2003.

[46] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, New York, NY, USA, 1991. ACM Press.

[47] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, 1995.

Appendix A

Affine Form of Farkas Lemma

(This appendix is grabbed verbatim from C. Bastoul PhD Thesis)

Farkas Lemma is fundamental to the theory of polyhedra; it was proved by the Hungarian physicist and mathematician Gyula Farkas in 1901. There exists many forms of this Lemma. The affine form is of a particular interest for affine schedule purpose since it allows to solve the dependence constraints (see section 1.6 or [16]). More generally it may be used to *linearize* non-affine constraints.

Lemma 2 *(Affine form of Farkas Lemma [37]) Let $\mathcal{D}$ be a nonempty polyhedron defined by the inequalities $A\vec{x} + \vec{b} \geq \vec{0}$. Then any affine function $f(\vec{x})$ is nonnegative everywhere in $\mathcal{D}$ iff it is a positive affine combination:*

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

where λ_0 and $\vec{\lambda}^T$ are called Farkas multipliers.

As an illustration, let us consider an affine function $f(x) = ax + b$, supposed to be non-negative everywhere in the domain $[1, 3]$. To find the constraints that a and b have to satisfy is geometrically simple: if $a > 0$ then every function $f(x) = ax + b$ with $b \geq -a$ is non-negative in $[1, 3]$, and if $a < 0$ then we must have $b \geq -3a$ (see Figure A.1). We can use Farkas Lemma to find these constraints algebraically. The domain $\mathcal{D}$ is defined by the following inequalities

$$\mathcal{D} : \left\{ \begin{array}{rcrcc} x & - & 1 & \geq & 0 \\ -x & + & 3 & \geq & 0 \end{array} \right.$$

Thus according to the Lemma, $f(x)$ is nonnegative everywhere in this domain if and only if $f(x) = \lambda_0 + \lambda_1(x - 1) + \lambda_2(3 - x)$ where $\lambda_0 \geq 0$, $\lambda_1 \geq 0$ and $\lambda_2 \geq 0$. The we can equate the coefficients of the components of $f(x)$ and build the constraint system:

$$\left\{ \begin{array}{ccccccc} & & \lambda_1 & - & \lambda_2 & = & a \\ \lambda_0 & - & \lambda_1 & + & 3\lambda_2 & = & b \\ \lambda_0 & & & & & \geq & 0 \\ & & \lambda_1 & & & \geq & 0 \\ & & & & \lambda_2 & \geq & 0 \end{array} \right.$$

Then we can use the Fourier-Motzkin elimination method to remove the λ_i from the constraint system as described in Section 1.7. First, we eliminate λ_0:

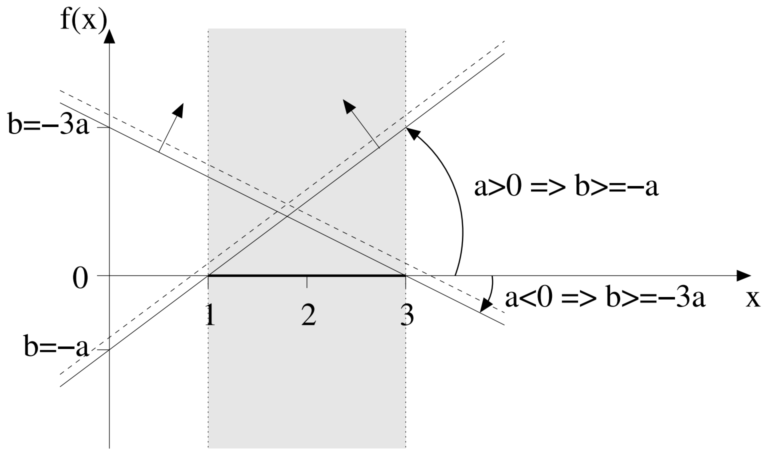


Figure A.1: Geometric example of Farkas Lemma

$$\left\{\begin{array}{rcrcr} \lambda_1 & - & \lambda_2 & = & a \\ \lambda_1 & - & 3\lambda_2 & \geq & -b \\ \lambda_1 & & & \geq & 0 \\ & & \lambda_2 & \geq & 0 \end{array}\right.$$

then we eliminate λ_1:

$$\left\{\begin{array}{rcl} \lambda_2 & \geq & -a \\ -2\lambda_2 & \geq & -a-b \\ \lambda_2 & \geq & 0 \end{array}\right.$$

hence the range of the last variable is

$$a+b \quad \geq \quad 2\lambda_2 \quad \geq \quad max(0,-2a)$$

we deduce that the constraint on a and b is $a+b \geq max(0,-2a)$ which is clearly equivalent to the result of our geometric reasoning.

Appendix B

Complete Experimental Results

We made several tests to compare our approach, taking into account only the legal schedules, to considering every schedules and filter legal ones thanks to a legality check, as Long et al. suggests [27]. We used different compute-intensive kernel benchmarks coming from various origins and listed in Figure B.1. `h264` is a fractional sample interpolation of the H.264 standard [45]. `fir` and `fft` are DSP kernels extracted from UTDSP benchmark suite [45]. `lu`, `gauss`, `crout` and `matmul` are well known mathematical kernels corresponding to LU factorization, Gaussian elimination, Crout matrix decomposition and matrix-matrix multiply. `MVT` is a kernel including two matrix-vector multiplies, one matrix being the transposition of the other. `locality` is an hand-written memory access intensive kernel.

These kernels are typically small, from 2 to 17 statements. They are quite well adapted to the present study since first, they should not challenge present production compiler optimization schemes, and second, they will make it possible to achieve an exhaustive visit of our search space which is necessary to evaluate the potential of the method and to design heuristic techniques. Dealing with larger benchmarks presents some technical difficulties. First, every SCoP do not have a one-dimensional schedule and some preprocessing (such as using a single assignment form) may be necessary. We still not have tools to apply such preprocessing or even to extract useful program informations automatically (iteration domains, subscript functions, etc.) from the source code. The GRAPHITE framework inside GCC should soon provide such a facility and allow us to enlarge our benchmark set [32].

The results of our study on the search spaces are summarized in Figure B.1. The first column presents the various kernel benchmarks; the second one labeled `#Dep.` precises the number of dependence sets for the corresponding kernel; $\vec{\imath}$`-Bounds` gives the iterator coefficient bounds used for search space bounding; $\vec{p}$`-Bounds` gives the parameter coefficient bounds; c`-Bounds` gives the constant coefficient bounds; `#Schedules` gives the total number of schedules, including illegal ones; `#Legal` gives the number of actual schedules in our space, i.e. the number of legal schedules; lastly `Time` precises the search space computation time on a Pentium 4 Xeon, 3.2GHz.

Results shows the very high benefit to work directly on a space including only legal transformations since it lowers the number of considered transformations by one to many orders of magnitude for a quite acceptable computation time. On the contrary, these results shows that without such a politic, achieving an exhaustive search is not possible even for small kernels. While these results shows profitability, it is not a demonstration of scalability, in the following we will propose to actually visit the search space exhaustively or using an heuristic way.

Our iterative optimization scheme is independent from the compiler and may be seen as a

Kernel Benchmark	#Dep.	$\vec{\imath}$-Bounds	$\vec{p}$-Bounds	c-Bounds	#Schedules	#Legal	Time
`h264`	15	$-1, 1$	$-1, 1$	$0, 4$	3165	360	0.011
`fir`	12	$-1, 1$	$-1, 1$	$-1, 1$	4.78×10^6	432	0.004
`fft`	36	$-2, 2$	$-2, 2$	$0, 6$	5.8×10^{25}	804	0.079
`lu`	14	$0, 1$	$0, 1$	$0, 1$	3.2×10^4	1280	0.005
`gauss`	18	$-1, 1$	$-1, 1$	$-1, 1$	5.9×10^4	506	0.021
`crout`	26	$-3, 3$	$-3, 3$	$-3, 3$	2.3×10^{14}	798	0.027
`matmult`	7	$-1, 1$	$-1, 1$	$-1, 1$	19683	912	0.003
`MVT`	10	$-1, 1$	$-1, 1$	$-1, 1$	4.7×10^6	16641	0.001
`locality`	2	$-1, 1$	$-1, 1$	$-1, 1$	59049	6561	0.001

Figure B.1: Search Space Computation

higher level to classical iterative compilation. In the same way as a given program transformation may better exploit a feature of a given processor, it also may enable more aggressive options of a given compiler. Because production compilers have to generate a target code in any case in a reasonable amount of time, their optimizations are very fragile, i.e. a slight difference in the source code may enable or forbid a given optimization phase.

To study this behavior and estimating how a higher level iterative optimization scheme may lead to better performances, we achieved a exhaustive scan of our search space for various programs and compilers with aggressive optimization options. We illustrate our results in Figure B.2.

We implemented tools for dependence analysis, legal transformation space construction and scanning. We used for that purpose external publicly available tools as `PipLib`, a linear algebra tool [15] and `CLooG`, a code generator in the polyhedral model [5]. We designed our tools to be able to use them as a plugin in the future GRAPHITE GCC's polyhedral framework [32].

We ran our experiments on an Intel workstation based on Xeon 3.2GHz, 8KB L1, 512KB L2 caches. We used four different compilers: GCC 3.4.2, GCC 4.1.1, Intel ICC 9.0.1 and PathScale EKOPath 2.5. We used hardware counters to measure the number of cycles used by various programs. In order to avoid interferences with other programs and the system, we set the system scheduler policy to FIFO for every test.

Because our search space is only based on legal solutions, the acceptable number of solutions for our various kernel benchmarks makes it possible to achieve an exhaustive search in a reasonable amount of time. Figure B.2 summarizes our results. The `Benchmark` column states the input original program; the `Compiler` column shows the compiler used to build each program version of the search space (GCC version was 4.1.1); the `Options` column precises the compiler options; the `Parameters` column gives the values of the global parameters (for instance the array sizes); the `Improved` column shows the number of transformations that achieves a better performance than the original program (the total number of versions is shown in Figure B.1); the `ID best` gives the "number" of the best solution; lastly, the `Speedup` column gives the speedup achieved by the best solution with respect to the original program performance.

Benchmark	Compiler	Options	Parameters	#Improved	ID best	Speedup
`h264`	PathCC	-Ofast	none	11	352	36.1%
`h264`	GCC	-O2	none	19	234	13.3%
`h264`	GCC	-O3	none	26	250	25.0%
`h264`	ICC	-O2	none	27	290	12.9%
`h264`	ICC	-fast	none	0	N/A	0%
`fir`	PathCC	-Ofast	N=150000	240	72	6.0%
`fir`	GCC	-O2	N=150000	259	192	15.2%
`fir`	GCC	-O3	N=150000	119	289	13.2%
`fir`	ICC	-O2	N=150000	420	242	18.4%
`fir`	ICC	-fast	N=150000	315	392	3.4%
`fft`	PathCC	-O2	N=256 M=256 O=8	21	267	7.2%
`fft`	GCC	-O2	N=256 M=256 O=8	10	285	0.9%
`fft`	GCC	-O3	N=256 M=256 O=8	11	289	1.8%
`fft`	ICC	-O2	N=256 M=256 O=8	17	260	6.9%
`fft`	ICC	-fast	N=256 M=256 O=8	20	112	6.4%
`lu`	PathCC	-Ofast	N=1000	100	224	6.5%
`lu`	GCC	-O2	N=1000	321	339	1.6%
`lu`	GCC	-O3	N=1000	330	337	3.9%
`lu`	ICC	-O2	N=1000	281	770	9.0%
`lu`	ICC	-fast	N=1000	262	869	8.7%
`gauss`	PathCC	-Ofast	N=150	212	4	3.1%
`gauss`	GCC	-O2	N=150	204	2	1.7%
`gauss`	GCC	-O3	N=150	52	2	0.01%
`gauss`	ICC	-O2	N=150	63	288	0.05%
`gauss`	ICC	-fast	N=150	15	39	0.03%
`crout`	PathCC	-Ofast	N=150	0	N/A	0%
`crout`	GCC	-O2	N=150	132	638	3.6%
`crout`	GCC	-O3	N=150	56	628	1.7%
`crout`	ICC	-O2	N=150	37	625	0.5%
`crout`	ICC	-fast	N=150	63	628	2.9%
`matmul`	PathCC	-Ofast	N=250	402	283	308.1%
`matmul`	GCC	-O2	N=250	318	284	38.6%
`matmul`	GCC	-O3	N=250	345	270	49.0%
`matmul`	ICC	-O2	N=250	390	311	56.6%
`matmul`	ICC	-fast	N=250	318	641	645.4%
`MVT`	PathCC	-Ofast	N=2000	5652	4934	27.4%
`MVT`	GCC	-O2	N=2000	3526	13301	18.0%
`MVT`	GCC	-O3	N=2000	3601	13320	21.2%
`MVT`	ICC	-O2	N=2000	5826	14093	24.0%
`MVT`	ICC	-fast	N=2000	5966	4879	29.1%
`locality`	PathCC	-Ofast	N=10000, M=2000	6069	5430	47.7%
`locality`	GCC	-O2	N=10000, M=2000	30	5494	19.0%
`locality`	GCC	-O3	N=10000, M=2000	589	4332	6.0%
`locality`	ICC	-O2	N=10000, M=2000	3269	2956	38.4%
`locality`	ICC	-fast	N=10000, M=2000	4614	3039	54.3%

Figure B.2: Search Space Statistics

Glossary

C

Candl	Chunky Analyzer for Dependences in Loops library.

F

FIFO	Firt In First Out (scheduler policy).
FM	Fourier-Motzkin library.

I

integer lattice	The ordered set of all integer vectors of dimension n.

L

LetSee	LEgal Transformation SpacE Explorator library.

P

polyhedron	A set of affine inequalities representing a convex (possibly finite) set of dimension n of points (points may be in $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{Q}$, $\ldots$). A polyhedron can be parameterized.
projection	Polyhedral operation which consists in geometrically projecting a space of dimension n to a space of dimension m where $m \leq n$.

S

statement	Atomic element handled in the polyhedral model. It is described by a number of executions and which (array) elements are read and written. There is no particular informations on *which* operations are actually executed.

Index