THÈSE DE L'UNIVERSITÉ PARIS-SUD

Spécialité : Informatique

présentée par

Cédric BASTOUL

pour obtenir

L'HABILITATION À DIRIGER DES RECHERCHES DE L'UNIVERSITÉ PARIS-SUD

Sujet de la thèse :

Contributions à l'optimisation des programmes de haut niveau

Contributions to High-Level Program Optimization

Soutenue le *12 décembre 2012*

Devant le jury composé de :

Pr. Philippe	CLAUSS	Président
Pr. François	BODIN	Rapporteur
Pr. François	IRIGOIN	Rapporteur
Pr. Sanjay	RAJOPADHYE	Rapporteur
Pr. Pierre	BOULET	Examinateur
Pr. Albert	COHEN	Examinateur
Pr. Daniel	ETIEMBLE	Examinateur

Thèse préparée à l'Université Paris-Sud
au sein du Laboratoire de Recherche en Informatique (LRI), UMR 8623 CNRS et de l'équipe
Grand Large, INRIA Saclay Île-de-France

“Helping programmers to develop efficient applications without sacrificing productivity.” This is, in one sentence, the rationale behind my involvement in research since the defense of my PhD in December 2004. This document presents an overview of my related activities during the past eight years in various research groups, as postdoc at University of Auvergne, as assistant professor at Paris-Sud University, as visiting professor at Reservoir Labs Inc. and more recently as visiting researcher at CNRS.

Contents

Chapter 1

Introduction

1.1 The Programming Wall

Frequency scaling used to bring performance improvements without any effort from application developers. This time is now over. The reason is, silicon transistor technology reached a physical and financial limit. Chip power dissipation is the consequence of the switching power, which is proportional to the frequency, and of various leakage currents, which become higher as the feature size decreases. As a consequence, high-frequency processors require a significant amount of energy and generate overheating while suitable cooling systems may be too expensive. To continue to benefit from the increasing number of transistors per surface unit, the processor industry shifted to designs including several *cores*. Cores are processors that share some resources such as a cache level or a memory bus. We refer to *multicore* when a chip includes from two to few dozens of cores (such as current general purpose processors) and to *manycore* when this number grows to hundreds (such as current hardware accelerators like GPGPUs). This solution allows the use more transistors on the same chip without increasing the clock frequency and still increasing the theoretical peak performance. However, the price to pay to benefit from such architectures is often to rewrite applications. And this price is high.

Multicore or manycore architectures are not bringing new fundamental problems with respect to program parallelization. However, their availability on desktop and embedded systems is dramatically changing the scope of these problems. While parallelization was expert's business, any programmer writing a new application should now take into account this new dimension. While compute-intensive applications (usually very regular) were the typical targets for parallelization, all softwares are now potentially affected. While a parallel computer was usually dedicated to running a single application until its termination, dozens of parallel programs should now coexist on the same system.

Despite these facts, several years after multicore architectures became ubiquitous, the massive shift to parallel programming has not happened yet. Systems are more responsive because the illusion of parallelism provided by multitasking (the central processing unit switches quickly from one program to another) is becoming a reality. But despite few exceptions, notably modern web browsers and antiviruses, most programs are still sequential. The main reason is the complexity of parallel programming which is still out of reach for non-experts. Despite many attempts, no proposal for a new language or programming model has been successful at establishing a new standard combining productivity and performance. Hence, at the time this habilitation thesis is written, the shift to multicore architectures corresponds to a brutal raise of the development costs to reach the same portion of peak performance than on monocore architectures, a “programming wall”. If no satisfactory solution is provided, programs will not benefit from the increasing number of cores and performance may soon stall.

Computer architects are working on solutions to avoid such a situation. The most likely scenario is the introduction of heterogeneity within a single chip. Rather than providing hundreds of identical cores, it suggests to use a part of the silicon surface to specialized hardware accelerators (such as GPUs or neural networks). Introducing new challenges on programming models, compilation and optimization, this solution would bring another brick to the programming wall.

1.2 Limits of Program Optimization Tools

Apart from profiling tools that aim at determining program critical parts and at observing their behavior, the tool set of developers looking at parallelizing and optimizing their programs consists in three main families: parallel languages, libraries and compilers. They correspond to complementary approaches with their own strengths and limitations,

Parallel languages can provide high-level abstractions (mathematical objects, trees, graphs, etc.) or parallel programming idioms (parallel loops, reduction, tasks etc.) or a fine control over the program execution (data distribution, alignment etc.). Many proposals have been presented (for instance Cilk, Chapel, C^n, CUDA, Fortress, HPF, UPC, X10 etc.). Nevertheless, none of them made its way (except on proprietary hardware) against the three de-facto parallel programming approaches, namely threads, OpenMP and MPI (note that none of them is a language), despite their weaknesses. Reasons are twofold. On one hand, programmers have legitimate concerns about developing applications using a language or a library while its long-term viability is unknown; and obviously, programming habits die hard. On the other hand, those languages are still leaving much of the work to the programmers, such as extracting parallelism or detailing data layout and distribution.

Optimized libraries (such as LAPACK, MKL, IPP etc.) provide to the programmers very efficient, ready to use functions (BLAS routines, FFT, etc.). Some advanced libraries have auto-tuning capabilities on some variants of a given target architecture (ATLAS, FFTW, SPIRAL...). While they may be very efficient solutions to some dedicated problems, function and target architecture coverage is, by essence, limited. Moreover, it is not possible to benefit from global optimization. For instance if two matrix multiplications reuse a given matrix, two calls will be necessary while a global optimization is likely to provide better results.

The last major family of tools is compilers (such as GCC, IBM XL or Intel ICC). They can provide language extensions (through pragmas, such as OpenMP or OpenACC) or offer automatic parallelization and optimization capabilities (vectorization, loop transformations...). Optimizing compilers may provide the best productivity, since they do not require to learn a new programming language, along with the best portability since they achieve the optimization work according to the target architecture. Compilers have a good record at taking advantage of the available resources within a single core, e.g., extracting instruction level parallelism or short-vector parallelism. Unfortunately those results are fragile and depend on the input source code and compilation options in a difficult way to understand and to predict. Moreover, they still have a poor support at extracting coarse grain parallelism.

1.3 High-Level Optimizations for High-Level Programs

Our work on helping programmers to develop efficient applications without sacrificing productivity belongs to the *high-level* compilation approach. Low-level compilers like GCC are multi-layer programs, translating the original source code into successive intermediate representations until the target object code is generated. During those translations, high-level constructions inherited from programmer's work such as arrays or iterative loops may be lost (or altered so aggressively that it is difficult to extract them back) before the optimization process. For instance GCC first translates the input code to GENERIC, a language-independent representation, then to GIMPLE, a three-address representation, then to RTL, a low-level, machine dependent representation. Optimizations are done at GIMPLE and RTL levels, far away from the input source code.

Contrary to low-level compilation, high-level compilation builds on the high-level structures available from the input code to design high-level optimizations altering those structures. Source-to-source compilers use this strategy to transform an input source code to another, optimized source code with a similar (or possibly higher) abstraction level. Obviously, high-level approaches are possible within a low-level compiler by reconstructing high-level structures, when this is possible.

High-level optimizations are essential to optimize and parallelize codes on modern architectures with several cores and deep memory hierarchies. Low-level compilation techniques are good at optimizing and parallelizing on a fine scale, e.g., achieving an efficient use of registers or extracting fine grain parallelism such as instruction level parallelism. The reason is that they can analyze precisely close enough instructions. Coarse grain parallelism or optimizing for large memories is quite different because it requires to analyze instructions that may be thousands of instructions away with respect to the dynamic sequential execution. Unrolling factors of thousands are not acceptable, hence, it is necessary to rely on high-level abstractions that can represent large sets of dynamic executions of program instructions and be compact at the same time.

The *polyhedral model* provides such a compact yet powerful abstraction for a subset of programs corresponding to regular computation codes (such that the control flow is not data-dependent and loop bounds, conditions and array accesses are affine forms). In this model, each dynamic execution of an instruction is encoded as an integer point in a geometrical space. It is restrictive enough to compute exact data dependences and data flow on programs that perfectly fits the model and it is possible to rely on conservative analyses on more general programs (dependences are over-approximated). This model has been the cornerstone of major advances in high-level optimization and parallelization over the last two decades [Fea92b, BDSV98, Lim01, Gri04, BHRS08, CB-PBCC08a, CB-PBB$^+$11]. Our work contributes to this approach.

1.4 Challenges Addressed in this Document

Back to the days of my PhD thesis, the polyhedral model was known to suffer from strong scalability problems (in particular for data dependence analysis and the code generation phase which translates the polyhedral abstraction back to a code). Moreover, while optimal parallelism extraction algorithms had been designed, exploiting this parallelism to reduce the execution time was essentially an open question. Our work during the past eight years focused on pushing the limits of the polyhedral model and to contribute at building a convincing high-level compiler approach to cross the programming wall. We addressed four main challenges: designing practical ways to interact with the user, building efficient optimizations with respect to performance, improving the scalability of the data dependence analysis and the code generation step, and enlarging the scope of polyhedral compilation techniques.

User Accessibility Compilation techniques developed for automatic optimization and parallelization provide powerful analyses and complex code restructuring capabilities [Fea92a, CB-GVB$^+$06]. However, the fully-automatic nature of compiler techniques is missing several opportunities for better optimizations. (1) The programmer should help the compiler to extract the high-level abstraction of the critical parts of his programs. (2) The compiler should analyze the reasons why an optimization cannot be applied, rather than the usual go/no go answer from data dependence analysis. (3) The programmer should be able to drive the compiler through high-level interfaces. Hence, the first challenge we address is how to build an efficient partnership between the programmer and the compiler to design the best optimizations.

We present how a programmer can actively collaborate with a high-level optimization engine. We advocate for the use of a simple programming model, powerful enough to write most compute intensive codes, and simple enough to be analyzed in depth by a compiler. We introduce the concept of *semantic feedback*, that a compiler can use to help programmers (or optimizing algorithms) to design semantics-preserving optimizations. We show how to build a set of high-level transformation primitives by revisiting classical loop transformations and how a compiler can correct a transformation script to respect data dependences, if possible and if necessary.

Parts of these results notably involved the supervision of Nicolas Vasilache and collaborations with Albert Cohen, Sylvain Girbal and Sebastian Pop [CB-VBGC06, CB-PCB$^+$06].

Optimization Quality Optimizing compilers should actually improve performance. However, because of the complex interplay between the many hardware resources and because of the complexity of low-level compilers, if is difficult to predict the behavior of a given code, along with a given compiler option set. The second challenge we address is to design an automatic optimization technique able to find a near-optimal optimization, as independent as possible from imprecise cost and performance models.

We present a feedback-directed, high-level iterative optimization approach to answer this challenge. We show how we can build a legal transformation space where each point represents a unique, semantically equivalent version of the input code. We achieve an in-depth analysis on small problems to design efficient heuristic traversal techniques to find the best performing versions for a given target platform.

These results notably involved the supervision of Louis-Noël Pouchet and Nicolas Vasilache, and collaborations with Uday Bondhugula, John Cavazos, Albert Cohen, J. Ramanujam and P. Sadayappan [CB-PBCV07, CB-PBCC08b, CB-PBCC08a, CB-PBB$^+$10, CB-PBB$^+$11]

Scalability High-level compilation techniques based on a polyhedral representation of programs are relying on high complexity techniques at nearly every stages. Those techniques behave well in practice because simple transformations are applied on simple problems. The third challenge we address is to design scalable techniques able to face complex real world examples and to demonstrate precise analyses can be issued in a reasonable amount of time and efficient solutions can be found to complex problems.

We study this problem on two aspects of high-level compilation frameworks, namely data dependence analysis and code generation. We present a fast dependence checking algorithm and a demonstration of scalability of instance-wise data dependence analysis on large benchmarks. We present scalable code generation methods that make possible the application of increasingly complex transformations on increasingly large programs. By studying the transformations themselves, we show how it is possible to benefit from their properties to dramatically improve both code generation quality and space/time complexity, with respect to the best state-of-the-art code generation tool. In addition, we build on these improvements to ensure an efficient generated code even for complex scenarios.

These results notably involved the supervision of Nicolas Vasilache and collaborations with Albert Cohen and Reservoir Labs Inc. [CB-VBGC06, CB-VBC06, CB-BVL$^+$09].

Applicability The main limitation of the polyhedral model is known to be its restriction to statically predictable, loop-based program parts. The fourth challenge we address is to relax this limitation and to study how it can be used to operate on general data-dependent control-flow.

To answer this challenge, we embed control and exit predicates as first-class citizens of the algebraic representation, from program analysis to code generation. Complementing previous (partial) attempts in this direction, our work concentrates on extending the code generation step and does not compromise the expressiveness of the model. We present experimental evidence that our extension is relevant for program optimization and parallelization, showing performance improvements on benchmarks that were thought to be out of reach of the polyhedral model.

These results notably involved the supervision of Mohamed-Walid Benabderrahmane and Louis-Noël Pouchet, and a collaboration with Albert Cohen [CB-BPCB10].

1.5 Habilitation Thesis Overview

This document is organised as follows. Chapter 2 describes the polyhedral representation of programs. It revisits the program transformation process through the “relation” abstraction used in state-of-the-art frameworks. Chapter 3 presents our efforts on designing efficient ways for a high-level compiler to interact with its users. It discusses different strategies such as syntactic and semantic feedback and details a semi-automatic optimization approach based on the relation abstraction and augmented with automatic correction capabilities. Chapter 4 describes the very first high-level iterative compilation approach based on the legal transformation space. It presents various heuristics to find optimizing transformations efficiently and a coupling between iterative and model-driven approaches. Chapter 5 depicts our efforts on providing scalable techniques for data dependence analysis and code generation. It presents an empirical study of exact analyses on large programs and of complex code generation problems. Chapter 6 shows our methodology to support irregular codes within a polyhedral framework. It demonstrates that general codes can benefit from existing high-level compilation techniques thanks to our extensions. Finally, Chapter 7 sums up the contributions of this work and details some future research directions.

Chapter 2

The Polyhedral Path

Since the very first compilers, the internal representation of programs has been in direct correspondence with their operational semantics. In such abstract syntaxes, each statement appears only once even if it is executed many times. This representation has severe limitations. First of all, it may limit the accuracy of program analysis. For instance, if a statement in a loop has some data dependence relation with another statement, we would consider both of them as single entities while the dependence relation may involve only very few of the dynamic iterations of these statements. This is particularly common in loop-based programs accessing arrays. Next, it may limit program transformation applicability. For instance, loop transformations operate on *individual statement iterations*. Lastly, it limits the expressiveness of program transformations: the most impactful loop nest transformations cannot be expressed as structural, incremental updates of the loop tree structure [CB-GVB$^+$06].

The polyhedral model is a semantical, algebraic representation which combines analysis power, transformation expressiveness and flexibility to design sophisticated optimization heuristics. The polyhedral model is closer to the program execution than operational/syntactic representations because it operates on individual statement iterations, or *statement instances*. For each instance, the optimizing algorithm will compute a *mapping* which will determine at which time (time mapping, or *scheduling*) and/or on which processor (space mapping, or *placement*) this instance has to be executed.

The origin of this model goes back to the late Sixties with the work of Karp, Miller and Winograd for scheduling systems of uniform recurrence equations [KMW67]. This seminal work has been transposed on one side to systolic array design [Qui84, RPF86, RK88, QD89] and on the other side to automatic parallelization of programs, with Lamport's hyperplane method [Lam74]. Until the early Nineties, most subsequent optimization and parallelization techniques relied on syntactic loop transformations rather than affine scheduling [Wol82, AK87, Wol87, WL91, ST92, Ban93]. But they also relied on an ever increasing data dependence abstraction precision, always defined as some class of convex affine sets (see Yang et al. for a survey of those abstractions and their relations with transformation techniques [YAI95]). The convergence of program representation, dependence abstraction and mapping, all expressed using affine sets, ultimately led to the polyhedral model (also referred in the literature as the *polytope model*) [Pug91b, Fea92a, Len93]. This model has been the basis for major advances in automatic optimization and parallelization of programs [Fea92b, BDSV98, Lim01, Gri04, BHRS08, CB-PBCC08a, CB-PBB$^+$11]. After decades of research, production compilers are getting closer to making effective use of the polyhedral model to compile for multicore architectures, including GCC [CB-PCB$^+$06, TCE$^+$10], LLVM [GZA$^+$11], IBM XL [BGDR10] and Reservoir Labs Inc. high level compiler R-Stream [MVW$^+$11].

Roughly, program restructuring in the polyhedral model is a three step process. First, a program (a source code or its abstract syntax tree) which can fit the model is translated to the polyhedral representation, then an optimizing algorithm computes a sequence of optimizing transformations *in the model*, finally the model is translated back to a program. The expectations are (1) to facilitate the extraction of the properties of the program (e.g., data reuse, parallelism) through the mathematical representation, (2) to compute a sequence of optimizing and parallelizing transformations in the model to exploit those properties using classical linear programming techniques and (3) to apply those optimizations as a single and straightforward model transformation step.

This chapter presents the polyhedral framework using the *union of relations* abstraction, which corresponds to the current state of the art. Section 2.1 presents this abstraction and how it is used to represent static control programs. Section 2.2 shows how to apply a transformation in this model and the constraints a transformation must satisfy to preserve the original program semantics. Section 2.3 presents different techniques to generate a program from the model abstraction and focuses on the state-of-the art technique. Finally, Section 2.4 concludes and summarizes our contributions to the polyhedral path.

2.1 Polyhedral Representation of Programs

The polyhedral model is a mathematical representation of programs which eases both analysis and restructuring. It allows the description of the parallelization and optimization problem in a compact and expressive way. This representation is also the key to solving this problem efficiently thanks to powerful and scalable polyhedral or mathematical libraries such as PIP, PolyLib, CLooG, PPL, isl or Omega (see Appendix A for some details about those tools). This model, or parts of it, has been used in most successful work on automatic parallelization and optimization. Program parts for which this model can encode the exact semantics are called *static control parts* [Fea91], or *SCoPs*, for short.

The polyhedral model is about representing and restructuring programs through affine sets. Early abstractions had various restrictions, on either the program representation (e.g., ability to process only perfectly nested loops [Len93]), or the mapping (e.g., only unimodular transformations [Ban90]) or the code generation process (e.g., only one polyhedron at a time [LVW94]). The evolution of the model followed the evolution of fundamental algorithms on polyhedral manipulation. The parametric extension of Chernikova's algorithm to convert a system of inequalities to vertices is the current basis for efficient polyhedral operations in several libraries [LW97, Le 92]. The parametric extension to the Simplex/Gomory algorithm [Fea88a] and the integer extension of the Fourier-Motzkin elimination method [Pug91a] are widely used to check the existence of an integer point inside a polyhedron, for linear programming or code generation. More recent works show how to manipulate $\mathbb{Z}$-polyhedra within an optimization framework [GR07, Ver10, SLM12].

Latest research and development allows unions of general affine *relations* to be used in every steps of a polyhedral optimization framework. This abstraction is presented in Section 2.1.1. Three mathematical objects based on unions of relations need to be defined to manipulate codes within the polyhedral model. First, *iteration domains* provide the relevant information about the various executions of the same statement, they are detailed in Section 2.1.2. Second, *space-time mapping functions* provide the order of the various instances with respect to each other and their placement amongst different processors. Lastly access functions model reading and writing on memory cells. They are described in Section 2.1.3.

2.1.1 Polyhedral Relations

A *relation* is a mapping from a set of input coordinates in an input space to a set of output coordinates in an output space. A *polyhedral relation* $\mathcal{R}(\vec{p})$ is a finite union of basic relations $\mathcal{R}(\vec{p}) = \bigcup_i \mathcal{R}_i(\vec{p})$, each basic relation being a function associating to $\vec{p}$ a relation that can be represented using m affine constraints in the following way:

$$\mathcal{R}_i(\vec{p}) = \left\{ \vec{x}_{in} \to \vec{x}_{out} \in \mathbb{Z}^{dim(\vec{x}_{in})} \times \mathbb{Z}^{dim(\vec{x}_{out})} \;\middle|\; \exists \vec{l}_i \in \mathbb{Z}^{dim(\vec{l}_i)} : \begin{bmatrix} A_{out,i} & A_{in,i} & L_i & P_i & \vec{c}_i \end{bmatrix} \begin{pmatrix} \vec{x}_{out} \\ \vec{x}_{in} \\ \vec{l}_i \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\},$$

where:

- $\vec{x}_{in} \in \mathbb{Z}^{dim(\vec{x}_{in})}$ is an input coordinate,
- $\vec{x}_{out} \in \mathbb{Z}^{dim(\vec{x}_{out})}$ is an output coordinate,
- $\vec{l}_i \in \mathbb{Z}^{dim(\vec{l}_i)}$ is the vector of *local variables* (also referred as *set variables* in the literature),
- $\vec{p} \in \mathbb{Z}^{dim(\vec{p})}$ is the vector of *parameters* (unknown but fixed integer values, a.k.a. *free variables*),
- $\vec{c}_i \in \mathbb{Z}^{d_m}$ is a constant vector,
- $A_{out,i} \in \mathbb{Z}^{m \times dim(\vec{x}_{out})}$, $A_{in,i} \in \mathbb{Z}^{m \times dim(\vec{x}_{in})}$, $L_i \in \mathbb{Z}^{m \times dim(\vec{l}_i)}$ and $P_i \in \mathbb{Z}^{m \times dim(\vec{p})}$ are integer matrices.

Polyhedral relations have been originally suggested by Kelly and Pugh as a convenient object to represent programs and to compute data dependences [KP93, KP95]. However, early polyhedral relations had to respect various constraints and could not be used directly to represent all aspects of a polyhedral framework (e.g., mapping had to be invertible). Modern implementations of polyhedral relations such as `isl` [Ver10] allow the use of the full expressiveness of relations.

Without loss of generality and except stated otherwise, we will only consider basic relations without local variables, to simplify notations in this document.

2.1.2 Representing Statement Instances: Iteration Domains

The key aspect of the representation of programs in the polyhedral model is to consider *statement instances*. A statement instance is *one* particular execution of a statement. Each instance of a statement that is enclosed inside a loop can be associated with the value of the outer loop counters (also called *iterators*). For instance, let us consider the polynomial multiply code in Figure 2.1: the instance of statement `S1` for $i = 2$ is `z[2] = 0`.

In the polyhedral model, statements are considered as functions of the outer loop counters that may produce statement instances: instead of simply "`S1`", the notation `S1(i)` is preferred. For instance, statement `S1` for $i = 2$ is written `S1(2)` and statement `S2` for $i = 4$ and $j = 2$ is written `S1`$\binom{4}{2}$. The vector of the iterator values is called the *iteration vector*.

Obviously, dealing with statement instances does not mean that unrolling all loops is necessary. First because there would probably be too many instances to deal with, and second because the number of

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

     for (i = 0; i < N; i++)
       for (j = 0; j < N; j++)
S2:      z[i+j] += x[i] * y[j];
```

Figure 2.1: Polynomial Multiply Kernel

instances may not be known. For instance, when the loops are bounded with constants that are unknown at compile time (called "parameters"), e.g., `N` in the example code in Figure 2.1. A compact way to represent all the instances of a given statement is to consider the set of all possible values of its iteration vector. This set is called the statement's *iteration domain*. It can be conveniently described by all the constraints on the various iterators that the statement depends on. When those constraints are affine and depend only on the outer loop counters and some parameters, the set of constraints defines a *polyhedron* (more precisely this is a *$\mathbb{Z}$-polyhedron*, but *polyhedron* is used for short). We can use a special case of relation to represent the iteration domain, since no input (or more exactly a constant input) is necessary:

$$\mathcal{D}_S(\vec{p}) = \left\{ () \to \vec{\imath}_S \in \mathbb{Z}^{dim(\vec{\imath}_S)} \;\middle|\; [D_S] \begin{pmatrix} \vec{\imath}_S \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\},$$

where $\vec{\imath}_S$ is the $dim(\vec{\imath}_S)$-dimensional iteration vector and $D_S \in \mathbb{Z}^{m_{\mathcal{D}_S} \times (dim(\vec{\imath}_S)+dim(\vec{p})+1)}$ is an integer matrix where $m_{\mathcal{D}_S}$ is the number of constraints. For instance, here are the iteration domains for the polynomial multiply example in Figure 2.1:

- $\mathcal{D}_{S1}(N) = \left\{ () \to \begin{pmatrix} i \end{pmatrix} \in \mathbb{Z} \;\middle|\; \left[\begin{array}{r:c:r} 1 & 0 & 0 \\ -1 & 2 & -2 \end{array} \right] \begin{pmatrix} i \\ \hdashline N \\ \hdashline 1 \end{pmatrix} \geq \vec{0} \right\},$

- $\mathcal{D}_{S2}(N) = \left\{ () \to \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \;\middle\|\; \left[\begin{array}{rr:c:r} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & -1 \end{array} \right] \begin{pmatrix} i \\ j \\ \hdashline N \\ \hdashline 1 \end{pmatrix} \geq \vec{0} \right\}.$

Iteration domains can be unions of basic relations, e.g., when a condition in the input code splits the iteration domain into a union of disjoint polyhedra. For instance, statements in Figure 2.2 are guarded by a condition which separates the iteration domain.

The iteration domain of $S3$ would be the following (note that constraints above the dashed line in the matrix are equalities while constraints under this line are inequalities):

```
     for (i = 0; i < N; i++) {
       for (j = 0; j <= i; j++) {
         if (i == j || j == 0)
S3:        A[i][j] = 1;
         else
S4:        A[i][j] = A[i-1][j] + A[i-1][j-1];
       }
     }
```

Figure 2.2: Pascal Triangle Kernel

$$\begin{aligned}\mathcal{D}_{S3}(N) = \quad & \left\{ () \to \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \;\middle|\; \left[\begin{array}{cc:c:c} -1 & 1 & 0 & 0 \\ \hdashline 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \end{array}\right] \left(\begin{array}{c} i \\ j \\ \hdashline N \\ \hdashline 1 \end{array}\right) \begin{array}{c} = \\ \hdashline \geq \end{array} \vec{0} \right\} \cup \\ & \left\{ () \to \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \;\middle|\; \left[\begin{array}{cc:c:c} 0 & 1 & 0 & 0 \\ \hdashline 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \end{array}\right] \left(\begin{array}{c} i \\ j \\ \hdashline N \\ \hdashline 1 \end{array}\right) \begin{array}{c} = \\ \hdashline \geq \end{array} \vec{0} \right\}.\end{aligned}$$

Local variables can be introduced in the iteration domain when existential quantifiers are useful. They can be used for instance when non-unit loop steps (a.k.a. *strides*) or modulo conditions are present in the input code. For instance, the loop in Figure 2.3 shows a loop with a stride 2.

```
     for (i = 0; i < 2*N; i += 2) {
S5:    c[i]   = (a[i]   * b[i]) - (a[i+1] * b[i+1]);
S6:    c[i+1] = (a[i+1] * b[i]) + (a[i]   * b[i+1]);
     }
```

Figure 2.3: Complex Vector Multiply Kernel

The iteration domain of $S5$ can be represented using a local variable l to express the fact that i is even: $\exists l \in \mathbb{Z}$ s.t. $i = 2 * l$ (note that again, constraints above the dashed line in the matrix are equalities while constraints under this line are inequalities):

$$\mathcal{D}_{S5}(N) = \left\{ () \to \begin{pmatrix} i \end{pmatrix} \in \mathbb{Z} \;\middle|\; \exists l \in \mathbb{Z} : \left[\begin{array}{c:c:c:c} -1 & 2 & 0 & 0 \\ \hdashline 1 & 0 & 0 & 0 \\ -1 & 0 & 2 & -1 \end{array}\right] \left(\begin{array}{c} i \\ \hdashline l \\ \hdashline N \\ \hdashline 1 \end{array}\right) \begin{array}{c} = \\ \hdashline \geq \end{array} \vec{0} \right\}.$$

2.1.3 Representing Order and Placement: Mapping Relations

Iteration domains do not provide any information about the relative execution order between statement instances, nor do they inform about the processor that has to execute them. Such information is provided by other mathematical objects called *space-time mappings*. They describe a relation between a statement instance and the logical date *when* it has to be executed and the processor coordinate *where* it has to be executed. In the literature, the part of those relations dedicated to time is called *scheduling* while the part dedicated to space is called *placement* (or *allocation*, or *distribution*).

In the case of *scheduling*, the logical dates express at which time a statement instance has to be executed, with respect to the other statement instances (from the same statement and other statements as well). It is typically denoted θ_S for a given statement S. For instance, let us consider the three statements in Figure 2.4(a) and their scheduling functions in Figure 2.4(b). The first and third statements have to be executed both at logical date 1. This means they can be executed in parallel at date 1 but they have to be executed before the second statement since its logical date is 2. The target code implementing this scheduling using OpenMP pragmas is shown in Figure 2.4(c), where a fictitious variable t stands for the time. It can be seen that at time $t = 1$, both $S1$ and $S3$ are run in parallel, while $S2$ is executed afterward at time $t = 2$.

```
S1: x = a + b;
S2: y = x + d;
S3: z = c * e;
```

(a) Original Code

$$\theta_{S1} = \{() \to (1)\}$$
$$\theta_{S2} = \{() \to (2)\}$$
$$\theta_{S3} = \{() \to (1)\}$$

(b) Scheduling

```
     t = 1;
     #pragma omp parallel sections
     {
       #pragma omp section
       {
S1:      x = a + b;
       }
       #pragma omp section
       {
S3:      z = c * e;
       }
     }
     t = 2;
S3:  y = x + d;
```

(c) Target Code

Figure 2.4: One-Dimensional Scheduling Example

Logical dates may be multidimensional, like clocks: the first dimension could correspond to days (most significant), next one to hours (less significant), the third one to minutes and so on. The order of multidimensional dates with a decreasing significance for each dimension is called the *lexicographic* order. Again, it is not possible to assign one logical date to each statement instance for two reasons: this would probably lead to an intractable number of logical dates and the number of instances may not be known at compile time. Hence, a more compact representation called the *scheduling relation* is used. A scheduling relation is a mapping from statement instances to logical dates. They have the following form for a statement S:

$$\theta_S(\vec{p}) = \left\{ \vec{\imath}_S \to \vec{t}_S \in \mathbb{Z}^{dim(\vec{\imath}_S)} \times \mathbb{Z}^{dim(\vec{t}_S)} \;\middle|\; [T_S] \begin{pmatrix} \vec{t}_S \\ \vec{\imath}_S \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\},$$

where $\vec{t}_S$ is the $dim(\vec{t}_S)$-dimensional iteration vector and $T_S \in \mathbb{Z}^{m_{\theta_S} \times (dim(\vec{t}_S)+dim(\vec{\imath}_S)+dim(\vec{p})+1)}$ is an integer matrix where m_{θ_S} is the number of constraints. Scheduling relations can easily encode a wide range of usual transformations such as skewing, interchange, reversal, shifting tiling etc. Many program transformation frameworks have been proposed on top of such relations (or some particular cases of such relations like scheduling *functions*), such as UTF [KP93], URUK [Gir05] or CHiLL [CCH08].

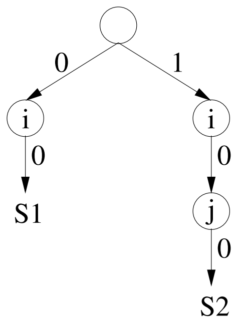


Figure 2.5: Polynomial Multiply Abstract Tree

A very useful example of multi-dimensional mapping relation is the **scheduling of the original program**. A method to compute it has been presented by Feautrier as a demonstration of the existence of a legal scheduling [Fea92b]. The technique is to build an abstract syntax tree of the program such that, (1) the nodes correspond to the loops and they are labelled with the corresponding loop counters. A fictitious loop running once and enclosing the whole program is used as the root of the tree. (2) Statements are leaves. (3) Arcs connect loops with their internal loops or statements. They are labelled with the corresponding textual order of the loop or the statement in the source code. The original scheduling for each statement is simply the list of labels from the root to the according leaf, the i^{th} label corresponding to the i^{th} dimension of the scheduling. For instance, let us consider the polynomial multiply code in Figure 2.1. Its abstract tree is shown in Figure 2.5. According to this tree, the original scheduling relation is:

- $$\theta_{S1}(N) = \left\{ \left(\begin{array}{c} i \end{array} \right) \to \left(\begin{array}{c} t^1_{S1} \\ t^2_{S1} \\ t^3_{S1} \end{array} \right) \in \mathbb{Z} \times \mathbb{Z}^3 \;\middle|\; \left[\begin{array}{ccc|c|c|c} -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \end{array} \right] \left(\begin{array}{c} t^1_{S1} \\ t^2_{S1} \\ t^3_{S1} \\ \hline i \\ \hline N \\ \hline 1 \end{array} \right) = \vec{0} \right\},$$

  which simply corresponds to the function $\theta_{S1}(N)\left(\begin{array}{c} i \end{array} \right) = \left(\begin{array}{c} 0 \\ i \\ 0 \end{array} \right)$

- $$\theta_{S2}(N) = \left\{ \left(\begin{array}{c} i \\ j \end{array} \right) \to \left(\begin{array}{c} t^1_{S2} \\ t^2_{S2} \\ t^3_{S2} \\ t^4_{S2} \\ t^5_{S2} \end{array} \right) \in \mathbb{Z}^2 \times \mathbb{Z}^5 \;\middle|\; \left[\begin{array}{ccccc|cc|c|c} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \end{array} \right] \left(\begin{array}{c} t^1_{S2} \\ t^2_{S2} \\ t^3_{S2} \\ t^4_{S2} \\ t^5_{S2} \\ \hline i \\ j \\ \hline N \\ \hline 1 \end{array} \right) = \vec{0} \right\},$$

  which simply corresponds to the function $\theta_{S2}(N)\left(\begin{array}{c} i \\ j \end{array} \right) = \left(\begin{array}{c} 1 \\ i \\ 0 \\ j \\ 0 \end{array} \right)$.

Placement is similar to scheduling, only the semantics is different: instead of logical dates, a placement relation π_S maps instances of statement S to processor coordinates corresponding to the processor that has to execute the instance. A space-time mapping relation σ_S is a multidimensional relation embedding both space and time information for statement S: some dimensions are devoted to scheduling while some others are dedicated to placement. In the polyhedral representation, the semantics of each dimension is not relevant: after code generation, each dimension will be translated to some loops that can be post-processed to become parallel or sequential according to their semantics (obviously, semantics information can be used to generate a better code, but this is out of the scope of this introduction).

Mapping unions can be used to apply a different mapping on different parts of the input space (e.g., to perform an index-set splitting) and/or to duplicate some parts of the input space (e.g., to apply unrolling or to achieve versioning). A mapping union does not need to be disjoint, but it will be translated into a disjoint union before applying it (i.e., there is no duplication of instances with the same mapping).

2.1.4 Representing Memory Accesses: Access Relations

After iteration domain and space-time mapping, the third main mathematical object used to represent a program in the polyhedral model is the *access relation*. The access relation models the memory reference behavior for a given access. It maps statement instances to memory cells for a given statement and a given array reference. When those relations are affine and depend only on the outer loop counters and some parameters, they can be written in the following general form:

$$\mathcal{A}_{S,r}(\vec{p}) = \left\{ \vec{\imath}_S \to \vec{a}_{S,r} \in \mathbb{Z}^{dim(\vec{\imath}_S)} \times \mathbb{Z}^{dim(\vec{a}_{S,r})} \;\middle|\; [A_{S,r}] \begin{pmatrix} \vec{a}_{S,r} \\ \vec{\imath}_S \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\},$$

where r denotes the reference number in the statement, $\vec{\imath}_S$ is the $dim(\vec{\imath}_S)$-dimensional iteration vector, $\vec{a}_{S,r}$ is the $dim(\vec{a}_{S,r})$-dimensional access vector where the i^{th} element corresponds to the index of the i^{th} array dimension and $A_{S,r} \in \mathbb{Z}^{m_{\mathcal{A}_{S,r}} \times (dim(\vec{\imath}_S)+dim(\vec{a}_{S,r})+dim(\vec{p})+1)}$ is an integer matrix where $m_{\mathcal{A}_{S,r}}$ is the number of constraints. For instance, let us consider the polynomial multiply code in Figure 2.1, page 18. The corresponding access relations would be the following:

- $$\mathcal{A}_{S1,1}(N) = \left\{ \begin{pmatrix} i \end{pmatrix} \to \begin{pmatrix} a_{S1,1} \end{pmatrix} \in \mathbb{Z} \times \mathbb{Z} \;\middle|\; \left[\begin{array}{c:c:c:c} -1 & 1 & 0 & 0 \end{array} \right] \begin{pmatrix} a_{S1,1} \\ \hdashline i \\ \hdashline N \\ \hdashline 1 \end{pmatrix} = \vec{0} \right\},$$

- $$\mathcal{A}_{S2,1}(N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} a_{S2,1} \end{pmatrix} \in \mathbb{Z}^2 \times \mathbb{Z} \;\middle|\; \left[\begin{array}{c:cc:c:c} -1 & 1 & 1 & 0 & 0 \end{array} \right] \begin{pmatrix} a_{S2,1} \\ \hdashline i \\ j \\ \hdashline N \\ \hdashline 1 \end{pmatrix} = \vec{0} \right\},$$

- $$\mathcal{A}_{S2,2}(N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} a_{S2,2} \end{pmatrix} \in \mathbb{Z}^2 \times \mathbb{Z} \;\middle|\; \left[\begin{array}{c:cc:c:c} -1 & 1 & 0 & 0 & 0 \end{array}\right] \begin{pmatrix} a_{S2,2} \\ \hdashline i \\ j \\ \hdashline N \\ \hdashline 1 \end{pmatrix} = \vec{0} \right\},$$
- $$\mathcal{A}_{S2,3}(N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} a_{S2,3} \end{pmatrix} \in \mathbb{Z}^2 \times \mathbb{Z} \;\middle|\; \left[\begin{array}{c:cc:c:c} -1 & 0 & 1 & 0 & 0 \end{array}\right] \begin{pmatrix} a_{S2,3} \\ \hdashline i \\ j \\ \hdashline N \\ \hdashline 1 \end{pmatrix} = \vec{0} \right\}.$$

2.2 Applying a Mapping in the Polyhedral Model

2.2.1 Transformation in the Model

Iteration domains can be extracted directly from the input code. They represent for each statement the set of their instances. In particular they do not encode any ordering information: iteration domains are nothing but "bags" of unordered statement instances. On the opposite, mapping relations, typically computed by an optimizing or parallelizing algorithm, provide the ordering information for statement instances. It is necessary to collect all this information into a polyhedral representation before the final code generation. There exist two ways to achieve this task:

Inverse Transformation Let us consider an iteration domain defined by the system of affine constraints $D\vec{\imath} + \vec{d} \geq \vec{0}$ where $\vec{d}$ is a constant vector, possibly parametric, and the transformation function leading to a target index $\vec{t} = T\vec{\imath}$. By noticing that $\vec{\imath} = T^{-1}\vec{t}$ it follows that the transformed polyhedron in the new coordinate system can be defined by:

$$\mathcal{T}(\vec{p}) = \left\{ () \to \vec{t} \in \mathbb{Z}^{dim(\vec{t})} \mid [DT^{-1}]\vec{t} + \vec{d} \geq \vec{0} \right\}.$$

Generalized Change of Basis Alternatively, new dimensions corresponding to the ordering in leading positions can be introduced while using the relation notation we used for the iteration domain in Section 2.1.2 and the mapping relation in Section 2.1.3:

$$\mathcal{T}(\vec{p}) = \left\{ () \to \begin{pmatrix} \vec{t} \\ \vec{\imath} \end{pmatrix} \in \mathbb{Z}^{dim(\vec{t})+dim(\vec{\imath})} \;\middle|\; \left[\begin{array}{c:c} \multicolumn{2}{c}{T} \\ \hdashline 0 & D \end{array}\right] \begin{pmatrix} \vec{t} \\ \hdashline \vec{\imath} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\}.$$

The inverse transformation solution has been introduced since the seminal work on parallel code generation by Ancourt and Irigoin [AI91]. It is simple and compact but has several issues: the transformation matrix must be invertible, and even when it is invertible, the target polyhedra may embed some integer points that have no corresponding elements in the iteration domain (this happens when the transformation matrix is not unimodular, i.e., whose determinant is neither $+1$ or -1). This necessitates specific code generation processing, briefly discussed in Section 2.3.1. The second formula is attributed to Le Verge, who named it the *Generalized Change of Basis* [Le 95]. It does not require any property on the transformation matrix. Nevertheless, the additional dimensions may increase the complexity of the code generation process. It has been rediscovered independently from Le Verge's unpublished work and used in production code generators only recently [CB-Bas04a]. Both formulas are used, and possibly mixed, in current code generation tools, depending on the desired transformation properties.

In the case of a mapping union, the mapping is first translated into a union of disjoint mappings to avoid the duplication of instances with the same mapping. Then, for a mapping union with n parts, each iteration domain is duplicated into n iteration domains, and the i^{th} of them is mapped with the i^{th} union part.

As an example, a compiler may suggest the following space-time mapping for the polynomial multiply code shown in Figure 2.1, page 18:

- $\sigma_{S1}(N) = \left\{ \left(i \right) \to \begin{pmatrix} p \\ t \end{pmatrix} \in \mathbb{Z} \times \mathbb{Z}^2 \,\middle|\, \left[\begin{array}{rr|r|r|r} -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \end{array} \right] \left(\begin{array}{c} p \\ t \\ \hline i \\ \hline N \\ \hline 1 \end{array} \right) = \vec{0} \right\},$

- $\sigma_{S2}(N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} p \\ t \end{pmatrix} \in \mathbb{Z}^2 \times \mathbb{Z}^2 \,\middle|\, \left[\begin{array}{rr|rr|r|r} -1 & 0 & 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 \end{array} \right] \left(\begin{array}{c} p \\ t \\ \hline i \\ j \\ \hline N \\ \hline 1 \end{array} \right) = \vec{0} \right\}.$

Its first mapping dimension p is a placement that corresponds to a wavefront parallelism for $S2$ and improves locality by executing the initialization of an array element by $S1$ on the same processor where it is used by $S2$. The second mapping dimension t is a very simple constant scheduling that ensures the initialization of the array element is done before it is used (it is usual to add the identity schedule at the last dimensions, however this will not be necessary for the continuation of this example). To apply the space-time mapping of the polynomial multiply proposed in Section 2.1.3 page 18, it is convenient to use the Generalized Change of Basis because the transformation matrices are not invertible (note that in the following formula, constraints "above" the line are equalities while constraints "under" the line are inequalities):

- $\mathcal{T}_{S1}(N) = \left\{ () \to \left(\begin{array}{c} p \\ t \\ \hline i \end{array} \right) \in \mathbb{Z}^3 \,\middle|\, \left[\begin{array}{rr|r|r|r} -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & -1 & 1 & 0 \end{array} \right] \left(\begin{array}{c} p \\ t \\ \hline i \\ \hline N \\ \hline 1 \end{array} \right) \begin{array}{c} = \\ \hline \geq \end{array} \vec{0} \right\},$

- $\mathcal{T}_{S2}(N) = \left\{ () \to \left(\begin{array}{c} p \\ t \\ \hline i \\ j \end{array} \right) \in \mathbb{Z}^4 \,\middle|\, \left[\begin{array}{rr|rr|r|r} -1 & 0 & 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 1 & 0 \end{array} \right] \left(\begin{array}{c} p \\ t \\ \hline i \\ j \\ \hline N \\ \hline 1 \end{array} \right) \begin{array}{c} = \\ \hline \geq \end{array} \vec{0} \right\}.$

In the target polyhedra, whatever the chosen formula, the order of the dimensions is meaningful: the ordering is encoded as the lexicographic order of the integer points. Thus, the parallel code generation problem is reduced to generating a code that enumerates the integer points of several polyhedra, with respect to the lexicographic ordering of the mapping dimensions.

2.2.2 Expressing Data Dependences

Not all mappings do preserve the original program semantics. Hence, it is critical to formalize the necessary information to preserve it. This is done through *data dependence* abstractions. Those abstractions characterize the fact that some ordered statement instances access the same memory location [Ber66]. Many abstractions exist, depending on the precision of the analysis and on the requirements of the user. The simplest and least precise one is called *dependence levels*, it specifies for a given loop nest which loop carry the dependence. It has been introduced in the Allen and Kennedy parallelization algorithm [AK87]. The *direction vectors* is a more precise abstraction where the i^{th} element approximates the value of all the i^{th} elements of the *distance vectors* (which shows the difference of the loop counters of two dependent instances). It has been introduced by Lamport [Lam74] then formalized by Wolfe [Wol95] and is clearly the most widely used representation. The most precise abstraction is the *dependences between iterations* [IT87, Fea92a] which is able to determine exactly the set of statement instances in dependence relation. The choice of a given dependence abstraction is crucial for further study. For some subsets of input codes and desired transformations, simple abstractions may encode the exact necessary information [YAI95, Iri11]. But in our context of general domain, access and mapping relations, choosing an imprecise dependence abstraction can result in blacking out interesting optimizations.

In this document, we use dependence between iterations, however we will use the term *dependence relations* for consistency and to highlight the fact that unlike most existing work, we use access relations instead of access functions to describe them. A dependence relation $\delta_{S,r_S \xrightarrow{d} T,r_T}(\vec{p})$ is a mapping from instances and accessed memory locations of a *source* statement S to instances and accessed memory locations of a *target* statement T, at a given *dependence depth* d (defined below), for a given pair of memory references r_S and r_T. The dependence relation is not empty for two instances iff they access the same memory locations and the source instance accesses the memory locations first. The dependence relation is built using three sub-relations:

1. **Existence condition** of the instances: the instances must belong to the iteration domains of their respective statements. The constraints involved are those of the iteration domains, see Section 2.1.2.
2. **Conflict condition** of the memory locations: the memory locations must belong to the access relation of their respective accesses and they must refer to the same locations. The constraints involved are those of the access relations, see Section 2.1.4, plus the equality of the access dimensions.
3. **Causality condition** of the instances: the instances of the source statement in the dependence relation are executed before the corresponding instances of the target statement. The constraints involved in this condition depend on the situation. They can be separated into a disjunction with as many parts as common loops to both S and T. Each part corresponds to a common loop depth and is called a *dependence depth*. For a given dependence depth $d > 0$, the causality condition is made of two parts:
   - the equality of the iteration vector elements at depth less than d: $i_S^x = i_T^x$ for $x < d$,
   - $i_T^d \geq i_S^d$ if S is textually before T, $i_T^d > i_S^d$ otherwise.

   If no loop is shared by S and T, there is no causality constraint and the dependence may only exist if S is textually before T.

Hence, the general form of a dependence relation with causality constraints is the following. We use the notation $M^{\vec{v}}$ for the submatrix of M made of the columns of M to be multiplied with the vector elements of $\vec{v}$:

$$\delta_{S,r_S \xrightarrow{d} T,r_T}(\vec{p}) = \left\{ \begin{pmatrix} \vec{\imath}_S \\ \vec{a}_{S,r_S} \end{pmatrix} \rightarrow \begin{pmatrix} \vec{\imath}_T \\ \vec{a}_{T,r_T} \end{pmatrix} \in \mathbb{Z}^{dim(\vec{\imath}_S)+dim(\vec{a}_{S,r_S})} \times \mathbb{Z}^{dim(\vec{\imath}_R)+dim(\vec{a}_{T,r_T})} \;\middle|\; \Delta_{S,r_S \xrightarrow{d} T,r_T} \right\},$$

where $\Delta_{S,r_S \xrightarrow{d} T,r_T}$ is the following dependence constraint system:

$$\Delta_{S,r_S \xrightarrow{d} T,r_T} : \left[\begin{array}{c|c|c|c|c|c} D_S^{\vec{\imath}_S} & 0 & 0 & 0 & D_S^{\vec{p}} & D_S^c \\ \hline 0 & 0 & D_T^{\vec{\imath}_T} & 0 & D_T^{\vec{p}} & D_T^c \\ \hline A_{S,r_S}^{\vec{\imath}_S} & A_{S,r_S}^{\vec{a}_{S,r_S}} & 0 & 0 & A_{S,r_S}^{\vec{p}} & A_{S,r_S}^c \\ \hline 0 & 0 & A_{T,r_T}^{\vec{\imath}_T} & A_{T,r_T}^{\vec{a}_{T,r_T}} & A_{T,r_T}^{\vec{p}} & A_{T,r_T}^c \\ \hline 0 & I & 0 & -I & 0 & 0 \\ \hline I^{1..d-1,\bullet} & 0 & -I^{1..d-1,\bullet} & 0 & 0 & 0 \\ \hline I^{d,\bullet} & 0 & -I^{d,\bullet} & 0 & 0 & 0 \text{ or } -1 \end{array} \right] \begin{pmatrix} \vec{\imath}_S \\ \vec{a}_{S,r_S} \\ \vec{\imath}_T \\ \vec{a}_{T,r_T} \\ \vec{p} \\ 1 \end{pmatrix} \begin{array}{c} \geq \\ \geq \\ \geq \\ \geq \\ = \\ = \\ \geq \end{array} \vec{0}$$

The complete information about data dependences of an input program is stored in the *data dependence graph*. In this directed graph, each program statement is represented using a unique vertex, and the existing dependence relations between statement instances are represented using edges. Each vertex is labelled with the iteration domain of the corresponding statement and the edges are labelled with the dependence relation between the source and destination statements. As an example, the data dependence graph of the polynomial multiply kernel of Figure 2.1 page 18 is shown in Figure 2.6. The dependence constraint systems are the following:

- $$\Delta_{S1,1 \xrightarrow{0} S2,1} : \left[\begin{array}{r|r||rr|r||r|r} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 2 & -2 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & -1 \\ \hline\hline 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 & -1 & 0 & 0 \\ \hline 0 & -1 & 0 & 0 & 1 & 0 & 0 \end{array} \right] \begin{pmatrix} i \\ a_{S1,1} \\ i' \\ j \\ a_{S2,1} \\ N \\ 1 \end{pmatrix} \begin{array}{c} \geq \\ \geq \\ = \\ = \\ = \end{array} \vec{0},$$

- $$\Delta_{S2,1 \xrightarrow{1} S2,1} : \left[\begin{array}{rr|r||rr|r||r|r} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 & -1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & -1 \\ \hline\hline 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 \\ \hline 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ \hline\hline -1 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{array} \right] \begin{pmatrix} i \\ j \\ a_{S1,1} \\ i' \\ j' \\ a_{S2,1} \\ N \\ 1 \end{pmatrix} \begin{array}{c} \geq \\ \geq \\ = \\ = \\ = \\ \geq \end{array} \vec{0}.$$

Finally, to avoid enforcing unnecessary constraints in reductions or scans [AK02], it is also possible to consider fundamental properties such as commutativity and associativity, to further refine the data dependence graph. We present in chapters 3 and 4 how to use the data dependence information either to check the preservation of the semantics (i.e. the legality of the transformation), or to build a semantic-preserving transformation, or to transform an illegal transformation into a legal one.

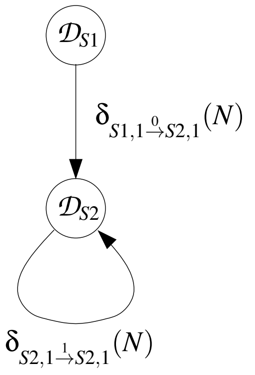

$\mathcal{D}_{S1}$
$\delta_{S1,1 \xrightarrow{0} S2,1}(N)$
$\mathcal{D}_{S2}$
$\delta_{S2,1 \xrightarrow{1} S2,1}(N)$

Figure 2.6: Dependence Graph of the Polynomial Multiply Kernel

2.3 Scanning Polyhedra

Once the target code information has been encoded into some polyhedra that embed the iteration spaces as well as the scheduling and placement constraints, the code generation problem translates to a *polyhedra scanning* problem. The problem here is to find a code (preferably efficient) visiting each integral point of each polyhedra, once and only once, with respect to the lexicographic order of some dimensions. Three main methods have been successful in doing this. Fourier-Motzkin elimination-based techniques have been the very first, introduced by the seminal work of Ancourt and Irigoin [AI91]. They are discussed briefly in Section 2.3.1. While Fourier-Motzkin-based techniques aim at generating loop nests, an alternative method based on Parametric Integer Programming has been suggested by Boulet and Feautrier to generate lower-level codes [BF98]. This method is discussed briefly in Section 2.3.2. Lastly, Quilleré, Rajopadhye and Wilde showed how to take advantage of high-level polyhedral operations to generate efficient codes directly [QRW00]. As this later technique is now widely adopted in production environments, it is discussed in some depth in Section 2.3.3.

2.3.1 Fourier-Motzkin Elimination-Based Scanning Method

Ancourt and Irigoin [AI91] proposed in 1991 the first solution to the polyhedron scanning problem. Their work is based on the Fourier-Motzkin pair-wise elimination technique [Sch86]. The scope of their method was quite restrictive since it could be applied to only one polyhedron, with unimodular transformation matrices. The basic idea was, for each dimension from the first one (outermost) to the last one (innermost), to project the polyhedron onto the axis and to deduce the corresponding loop bounds. For a given dimension i_k, the Fourier-Motzkin algorithm can establish that $L(i_1,...,i_{k-1})+\vec{l} \le c_k i_k$ and $c_k i_k \le U(i_1,...,i_{k-1})+\vec{u}$, where L and U are constant matrices, $\vec{l}$ and $\vec{u}$ are constant vectors of size m_l and m_u respectively, and c_k is a constant. Thus, the corresponding scanning code for the dimension i_k can be derived:

```
for  ($i_k = \max_{j=1}^{m_l} \lceil (L_j(i_1,...,i_{k-1}) + l_j)/c_k \rceil$ ;
      $i_k \le \min_{j=1}^{m_u} \lfloor (U_j(i_1,...,i_{k-1}) + u_j)/c_k \rfloor$ ;
      $i_k$++)  {
  Body($i_k$);
}
```

The main drawback of this method is the large amount of redundant control since eliminating a variable with the Fourier-Motzkin algorithm may generate up to $n^2/4$ constraints for the loop bounds where n is the initial number of constraints. Many of those constraints are redundant and it is necessary to remove them for efficiency.

Most further works tried to extend this first technique in order to reduce the redundant control and to deal with more general transformations. Le Fur presented a new redundant constraint elimination policy by using the simplex method [Le 96]. Several works proposed to relax the unimodularity constraint of the transformation to an invertibility constraint by using the Hermite Normal Form [Sch86] to avoid scanning “holes” in the polyhedron [LP94, Xue94, DR94, Ram95]. Griebl, Lengauer and Wetzel [GLW98] relaxed the constraints of code generation further to transformation matrices with non-full rank, and also presented preliminary techniques for scanning several polyhedra using a single loop nest. Kelly, Pugh and Rosser showed how to scan several polyhedra in the same code by generating a naive perfectly nested code and then (partly) eliminating redundant conditionals [KPR95]. Their implementation relies on an extension of the Fourier-Motzkin technique called the Omega test. The implementation of their algorithm within the Omega Calculator is one of the most popular parallel code generators [KMP$^+$96]. This technique has been refined by Chen to minimize control overhead further [Che12].

2.3.2 Parametric Integer Programming-Based Scanning Method

Boulet and Feautrier proposed in 1998 a parallel code generation technique which relies on Parametric Integer Programming (PIP for short) to build a code for scanning polyhedra [BF98]. The PIP algorithm computes the lexicographic minimal integer point of a polyhedron. Because the minimum point may not be the same depending on the parameter values, it is returned as a tree of conditions on the parameters where each leaf is either the solution for the corresponding parameter constraints or $\perp$ (called *bottom*), i.e., no solution for those parameter constraints.

The basic idea of the Boulet and Feautrier algorithm (in the simplified case of scanning one polyhedron) is to find the first integer point of the polyhedron, called *first*, then to build a function *next* which for a given integer point returns the next integer point in the polyhedron according to the lexicographic ordering. Both *first* and *next* computations can be expressed as a problem of finding the lexicographic minimum in a polyhedron. Finally, the code can be built according to the following canvas, where x is an integer point of the polyhedron that represents the iteration domain:

```
      $x$ = $first$;
L1:   if ($x$ == $\perp$)
        goto L2;
      Body();
      $x$ = $next$;
      goto L1;
L2:   ...
```

Generalizing this method to many polyhedra implies combining the different trees of conditions and subsequent additional control cost and code duplication. While this technique has no widely used implementation, it is quite different than the others since it does not aim at generating high-level loop statements. This property may be relevant for specific targets, e.g., when the generated code is not the input of a compiler but of a high-level synthesis tool.

2.3.3 QRW-Based Scanning Method

Quilleré, Rajopadhye and Wilde proposed in 2000 the first code generation algorithm that builds a target code without redundant control *directly* [QRW00]. While previous schemes started from a generated code with some redundant control and then tried to improve it, their technique (referred as the QRW algorithm) never fails at removing control, and the processing is easier. Eventually it generates a better code more efficiently.

The QRW algorithm is a generalization to several polyhedra of the work of Le Verge, Van Dongen and Wilde on loop nest synthesis using polyhedral operations [LVW94]. It relies on high-level polyhedral operations (like polyhedral intersection, union, projection etc.) which are available in various existing polyhedral libraries. The basic mechanism is, starting from (1) the list of polyhedra to scan and (2) a polyhedron encoding the constraints on the parameters called the *context* C, to recursively generate each level of the abstract syntax tree of the scanning code (AST).

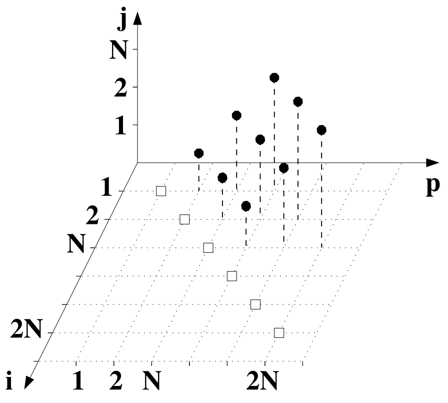
QRW: build a polyhedra scanning code AST without redundant control.

Input: a polyhedron list, a context C, the current dimension d.
Output: the AST of the code scanning the input polyhedra.

1. Intersect each polyhedron in the list with the context C;
2. Project the polyhedra onto the outermost d dimensions;
3. Separate these projections into disjoint polyhedra (this generates loops for dimension d and new lists for dimension $d+1$);
4. Sort the loops to respect the lexicographic order;
5. Recursively generate loop nests that scan each new list with dimension $d+1$, under the context of the dimension d;
6. Return the AST for dimension d.

Figure 2.7: Sketch of the QRW Code Generation Algorithm

The algorithm is sketched in Figure 2.7 and a simplified example is shown in Figure 2.8. It corresponds to the generation of the code implementing the polynomial multiply space-time mapping introduced in Section 2.2. Its input is the list of polyhedra to scan, the context and the first dimension to scan. This corresponds to Figure 2.8(a) in our example, with the first dimension to scan being p. The first step of the algorithm intersects the polyhedra with the context to ensure no instance outside the context will be executed. Then it projects them onto the first dimension and separates the projections into disjoint polyhedra. For instance, for two polyhedra, this could correspond to one domain where the first poly-

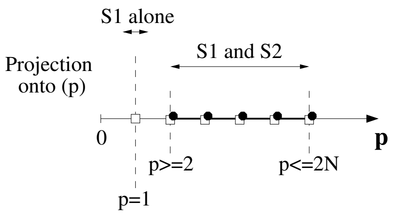


Context: $n \geq 3$

$$\mathcal{T}_{S_1}(N) : \begin{cases} p = i \\ t = 0 \\ 1 \leq i \leq 2N \end{cases}$$

$$\mathcal{T}_{S_2}(N) : \begin{cases} p = i + j \\ t = 1 \\ 1 \leq i \leq N \\ 1 \leq j \leq N \end{cases}$$

(a) Polyhedra to Scan and Context Information (degenerated dimension t is not drawn)



```
#pragma omp parallel for
for (p = 1; p <= 1; p++)
```

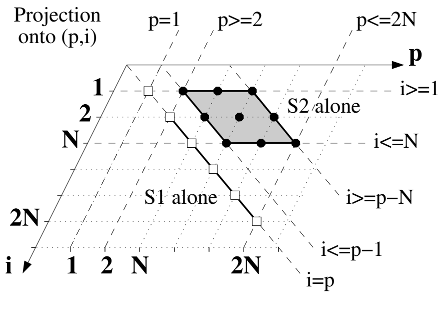
$$\mathcal{T}'_{S_1}(N) : \begin{cases} t = 0 \\ i = 1 \end{cases}$$

```
#pragma omp parallel for
for (p = 2; p <= 2*N; p++)
```

$$\mathcal{T}''_{S_1}(N) : \begin{cases} t = 0 \\ i = p \end{cases}$$

$$\mathcal{T}'_{S_2}(N) : \begin{cases} t = 1 \\ 1 \leq i \leq N \\ 1 \leq j \leq N \\ i + j = p \end{cases}$$

(b) Intersection with the Context, Projection and Separation onto the First Dimension



```
    #pragma omp parallel for
    for (p = 1; p <= 1; p++)
      for (t = 0; t <= 0; t++)
        for (i = 1; i <= 1; i++)
S1:       z[i] = 0;

    #pragma omp parallel for
    for (p = 2; p <= 2*N; p++) {
      for (t = 0; t <= 0; t++)
        for (i = p; i <= p; i++)
S1:       z[i] = 0;
      for (t = 1; t <= 1; t++)
        for (i  = max(1, p-N);
             i <= min(N, p-1); i++)
          for (j = p-i; j <= p-i; j++)
S2:         z[i+j] += x[i] * y[j];
    }
```

(c) Recursion on the Next Dimensions

Figure 2.8: QRW Code Generation Example

hedron is “alone”, one domain where the second polyhedron is “alone” and one domain where the two polyhedra coexist. This is depicted in the Figure 2.8(b) for our example: it depicts the projection onto the p axis and the separation. Two disjoint polyhedra are created: one where S_1 is alone on p (it has only one integer point but a loop is generated to scan it, for consistency) and one where S_1 and S_2 are together on p (it can be seen here that the domain where S_2 is “alone” is empty). The constraints on dimension p for the resulting polyhedra give directly the loop bounds. As the semantics of the placement dimension is to distribute instances across different processors, this loop is parallel. Then the algorithm recursively generates the next dimension loops for each disjoint polyhedron separately as shown in Figure 2.8(c) for our example. First, the projection/separation on (p,t) is done. It is trivial because t is a constant in every polyhedron: it only enforces disjonction and ordering of the polyhedra inside the second `doall` loop. Next the same processing is applied for (p,t,i): the loop bounds of the remaining dimensions can be deduced from the graphical representation (the trivial dimension t is not shown).

The QRW algorithm is simple and efficient in practice, despite the high theoretical complexity of most polyhedral operations. However, in its basic form, it tends to generate codes with costly modulo operations, and the separation process is likely to result in very long codes. Several extensions to this algorithm have been proposed to overcome those issues [CB-Bas04a, CB-VBC06]. CLooG, a popular implementation of the extended QRW technique demonstrated effectiveness and robustness of the extended algorithm [CB-Bas04a]. It is now used in production environments such as in GCC, LLVM or IBM XL.

2.4 Conclusion

The polyhedral model encodes the complete information about dynamic iterations, data dependences and memory accesses of a restricted class of programs through affine sets. In this representation, it is possible to rely on mathematical tools to analyse programs and to build optimizing and parallelizing instance-wise mappings. This chapter presented the *relation* abstraction to represent static control programs, and how it can be used to manipulate codes from source to source, provided the availability of libraries for manipulating $\mathbb{Z}$-polyhedra. This abstraction is already at work, e.g., in recent versions of the isl library [Ver10] or in the Clay framework (see Section 3.3), but this is probably the first document that revisits the polyhedral path with this abstraction.

The relation concept goes back to the work of Kelly and Pugh in the early Nineties [KP93, KP95]. However, it sufferred from various limitations because of the lack of fundamental algorithms and scalable tools to manipulate the unrestricted polyhedral representation from the original source code analysis to the code generation.

The “polyhedral path” presented in this chapter is the result of a slow maturation with recent additions. I made some contributions to its construction. First I showed how a general mapping could be applied, hence allowing mapping relations while the most general alternative at the moment was invertible functions[1] (i.e., a particular case where the constraints are only equalities and the constraint matrix has full-row rank). Second, I contributed to popularize and I extended the QRW algorithm through the code generation tool and library CLooG [CB-Bas04a]. By that time the model was still lacking a complete support for $\mathbb{Z}$-polyhedra (however it was possible to mimic them in CLooG using additional dimensions), and mapping unions. $\mathbb{Z}$-polyhedra has been the subject of recent works by Gautam and Rajopadhye [GR07], Seghir et al. [SLM12] and Verdoolaege [Ver10] which implemented them inside the

[1]I recall here that this transformation was first discovered independently by Le Verge, see Section 2.2.1 for details.

isl library, and added the support for that library to CLooG. Another recent addition is mapping unions proposed by Tobias Großer[2].

State-of-the art polyhedral library isl [Ver10], extraction tools like Clan [CB-Bas08] or Pet [Ver12], data dependence analyzers like Candl [CB-BP12] or isl [Ver10] and the code generation tool CLooG [CB-Bas04a] support the program abstraction discussed in this chapter. Relations allow the expression of, e.g., tiling from the mapping, and union of relations offers, e.g., index-set splitting. Still, efficient automatic mapping construction algorithms benefiting from the full expressiveness of this abstraction are yet to be designed.

[2]Informal discussion around CLooG improvements in 2009.

Chapter 3

User Accessibility

This chapter is about tools, abstractions, techniques and lessons on how to help users to take advantage of high-level optimization techniques. It is necessary to consider two main user categories for the optimization frameworks we are developing. The *final user* aims at optimizing a program. He may or may not be the author of this program. He can be an optimization expert, i.e., knowledgeable on both the architectural details of the target hardware and the usual program optimization techniques. He can be an expert of the input program as well, i.e., knowledgeable about the internals of the application, from the data structures to the algorithms. Automatic or semi-automatic optimization tools may be useful for any final user if he lacks some part of the expertise or if he looks at accelerating and improving the reliability of the optimization process. The way the tools will support the final user depends on his expertise and his needs. Hence, it is necessary to provide suitable solutions to the various final user profiles.

Contrary to the final user, the *intermediate user* goal is not to optimize programs directly, but to develop tools for final users. Those tools, such as compilers, may be very complex. We are also developing ways for the intermediate users to integrate the technologies we are designing, with the best trade-off between technical ease and processing power. Providing the intermediate user with a pertinent access path to optimization techniques, we contribute to their dissemination and to their use by final users.

In this chapter, we first build on our experience at designing high-level optimization frameworks to advocate considering static control as a programming model and to provide *syntactic feedback* to the programmer. Then we show in Section 3.2 how to provide a *semantic feedback* to the programmer, with respect to the optimization sequence he is willing to apply. Section 3.3 offers a high-level entry point to the programmer, by revisiting the classical loop transformations in the light of the relation abstraction, and by building on the semantic feedback to help the user to define legal transformation scripts. Finally, Section 3.4 discusses the OpenScop effort for standardizing the polyhedral abstraction to facilitate the dissemination of polyhedral frameworks and to ease their construction.

3.1 Syntactic Feedback: Static Control as a Programming Model

A largely underestimated challenge to a successful use of the polyhedral model is the *raising* problem, i.e., the action of translating a source code or an equivalent internal representation into a polyhedral abstraction. We can distinguish two schools of raising techniques which correspond to two very different approaches of high-level compilation techniques:

- *IR raising* extracts the polyhedral abstraction from the internal representation of a compiler and is not guaranteed to be the first pass of that compiler.

- *Direct raising* extracts the polyhedral abstraction directly from a high-level representation of the code, i.e., from the source code itself or from an intermediate representation such that it is possible to generate back the unmodified original source code.

IR raising has to automatically detect the parts of the intermediate representation amenable to a polyhedral abstraction because various processing of the internal representation may have considerably modified its properties, for better or for worse. It is typically related to implementations of polyhedral frameworks inside low-level compilers, such as WRAP-IT for ORC [CB-BCG$^+$03a, Gir05, CB-GVB$^+$06], GRAPHITE for GCC [CB-PCB$^+$06, TCE$^+$10] or Polly for LLVM [GZA$^+$11].

Independently from the paramount iteration domain and access function construction challenges (e.g., see Trifunovic et al. [TCE$^+$10, Tri11] and Pop [Pop06] for these problems with respect to GCC), integrating polyhedral compilation techniques in production low level compilers raised several issues. Some of them were expected, some of them were more surprising. First, high-complexity techniques are not welcome in production compilers. Polyhedral analyses have a very high worst-case complexity. This complexity is mostly hidden when dealing with human-written codes, since the code parts relevant to a polyhedral analysis are typically simple and small. However problems are likely to surface because irregular extensions are now possible (see Chapter 6), because larger and more complex codes are now automatically generated, and because of aggressive inter-procedural analyses. Second, a compiler is made of many passes. While some are good for polyhedral analyses (e.g., constant propagation), some are wasting the efforts of a user to write a code which should benefit from the polyhedral framework (e.g., induction variable substitution or array linearization may introduce non-affine array accesses). This is a general statement for any optimization, but it is extremely critical for a technology which may change the performance by an order of magnitude. Third, the impact of program transformations on iterators (whose original values may be reused or, more maliciously, which may overflow) has been an underestimated problem which needs to be investigated. Hence, bringing all the benefits of a polyhedral framework to production compilers requires further research and development on, e.g., lower complexity techniques or internal compiler construction we are currently investigating.

Direct rising typically relies on the user to mark the code part to be processed by the polyhedral framework and/or to counterbalance analysis weaknesses (to, e.g., ensure the absence of pointer aliasing or function side-effects). It is typically related to high-level, source-to-source frameworks like PoCC [CB-PBB$^+$10, CB-PBB$^+$11], or R-Stream [MVW$^+$11, CB-MLV$^+$09, CB-BVL$^+$09, CB-LVM$^+$10] or Pluto [BHRS08].

Contrary to their low-level compiler counterparts, high-level polyhedral frameworks require some efforts from their users: the code parts to be processed must be written in such a way that the direct raising is possible. In addition, if the raising system does not have the analysis power of a complete compiler, to, e.g., perform alias analysis, the user is required to provide the missing information. This task is usually done through language extensions such as pragmas. As an example, Figure 3.2 shows the rules to write a code that the raising tool Clan [CB-Bas08] is able to process. Modern direct raising tools like Clan or Pet [Ver12] also integrate diagnostic mechanisms providing syntactic feedback to users to help them to comply to the static control paradigm, as illustrated in Figure 3.1. Another significant difference with low-level compilers is the compilation time which is a weak constraint since users accepted to rely on high complexity techniques and have the guarantee that their codes can be processed.

```
$ cat foo.c

#pragma scop
for (i = 0; i < N*j; ++i)
  for (j = 0; j < N; ++j)
    S1();
#pragma endscop

$ clan foo.c

[Clan] Error: non-affine expression at line 2, column 19.
for (i = 0; i < N*j; ++i)
                ~~~^
[Clan] Error: a loop iterator was previously used as a parameter at line 3, column 8.
  for (j = 0; j < N; ++j)
    ~~~^
```

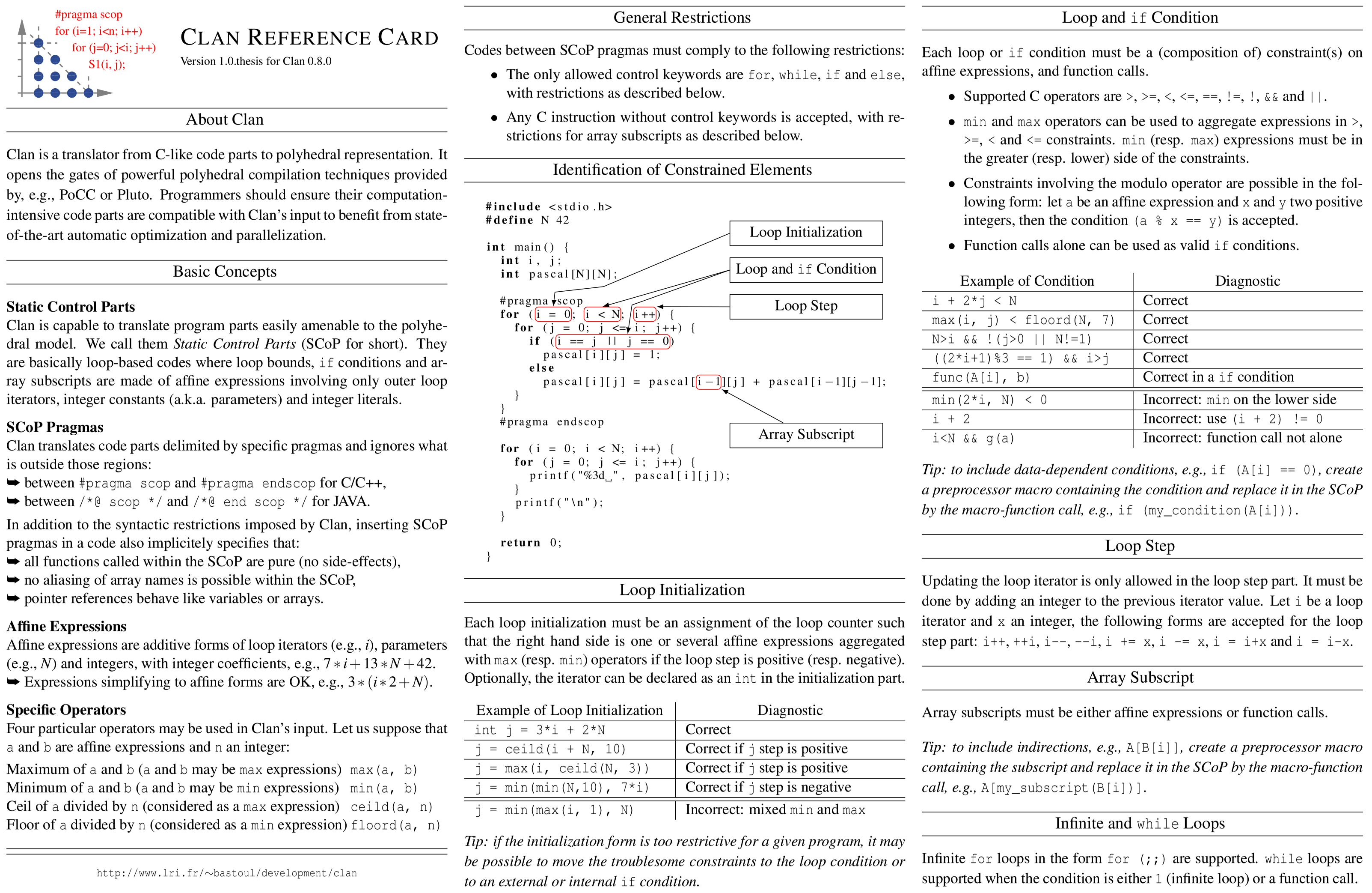
Figure 3.1: Syntactic feedback Example Using Clan

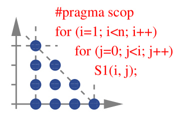
To achieve a reasonable amount of the peak performance on a given architecture, a programmer may choose (1) to learn the details and specific API and language layers of the target architecture like CUDA [nVi12], or (2) to use high level abstractions and tools like SPIRAL [PSX$^+$04] to automatically generate a target code, or (3) to make extensive calls to optimized libraries like BLAS [HKL73], or (4) to rely on an optimizing compiler. The drawbacks of the first approach are well known: high learning curve, tedious task of parallelism extraction, slow and error-prone programming, lack of portability of the languages and the optimizations. The second solution requires to learn a new abstraction model and is typically domain-specific (as SPIRAL for DSP algorithms). The third solution necessitates the availability of an adequate library for the target architecture and may miss optimizations across library calls. A common property of the first three approaches is to necessitate significant efforts from the programmers. The last approach is the less demanding for the application writers, but it raises the problem of the availability of a good enough optimizing compiler, usually several years after a given architecture is available.

High-level optimization techniques based on the polyhedral model demonstrated high retargetability as shown by Reservoir Labs R-Stream compiler (including targets to OpenMP, Tilera, Cell, ClearSpeed and CUDA with a general polyhedral mapper) or by the research compiler Pluto (including targets to OpenMP, CUDA and MPI). Hence it is a choice framework for optimizing compilers to rapidly adapt to new targets. As a result, we advocate here for the use of the static control paradigm *as a programming model*. The only effort required from the users is, if it proves to be possible, to rewrite their computational kernels in a subset of an existing language complying to, e.g., the constraints detailed in Figure 3.2 to benefit from high-level polyhedral frameworks, and to maximize the probability to enable the internal polyhedral frameworks of low-level compilers.

3.2 Semantic Feedback: the Violation Analysis Approach

Once a program has been raised to a polyhedral abstraction, we have to build an optimization which respects the original program semantics. In general, automatic or semi-automatic loop optimizers check that a given transformation does not alter the semantics of a program before actually applying it. When legality is enforced by construction, illegal transformations are not considered at all. When legality





Clan Reference Card

Version 1.0.thesis for Clan 0.8.0

About Clan

Clan is a translator from C-like code parts to polyhedral representation. It opens the gates of powerful polyhedral compilation techniques provided by, e.g., PoCC or Pluto. Programmers should ensure their computation-intensive code parts are compatible with Clan's input to benefit from state-of-the-art automatic optimization and parallelization.

Basic Concepts

Static Control Parts
Clan is capable to translate program parts easily amenable to the polyhedral model. We call them *Static Control Parts* (SCoP for short). They are basically loop-based codes where loop bounds, `if` conditions and array subscripts are made of affine expressions involving only outer loop iterators, integer constants (a.k.a. parameters) and integer literals.

SCoP Pragmas
Clan translates code parts delimited by specific pragmas and ignores what is outside those regions:
➥ between `#pragma scop` and `#pragma endscop` for C/C++,
➥ between `/*@ scop */` and `/*@ end scop */` for JAVA.

In addition to the syntactic restrictions imposed by Clan, inserting SCoP pragmas in a code also implicitely specifies that:
➥ all functions called within the SCoP are pure (no side-effects),
➥ no aliasing of array names is possible within the SCoP,
➥ pointer references behave like variables or arrays.

Affine Expressions
Affine expressions are additive forms of loop iterators (e.g., i), parameters (e.g., N) and integers, with integer coefficients, e.g., $7*i+13*N+42$.
➥ Expressions simplifying to affine forms are OK, e.g., $3*(i*2+N)$.

Specific Operators
Four particular operators may be used in Clan's input. Let us suppose that `a` and `b` are affine expressions and `n` an integer:

Maximum of `a` and `b` (`a` and `b` may be `max` expressions)	`max(a, b)`
Minimum of `a` and `b` (`a` and `b` may be `min` expressions)	`min(a, b)`
Ceil of `a` divided by `n` (considered as a `max` expression)	`ceild(a, n)`
Floor of `a` divided by `n` (considered as a `min` expression)	`floord(a, n)`

http://www.lri.fr/~bastoul/development/clan

General Restrictions

Codes between SCoP pragmas must comply to the following restrictions:

- The only allowed control keywords are `for`, `while`, `if` and `else`, with restrictions as described below.
- Any C instruction without control keywords is accepted, with restrictions for array subscripts as described below.

Identification of Constrained Elements

```
#include <stdio.h>
#define N 42

int main() {
  int i, j;
  int pascal[N][N];

  #pragma scop
  for (i = 0; i < N; i++) {
    for (j = 0; j <= i; j++) {
      if (i == j || j == 0)
        pascal[i][j] = 1;
      else
        pascal[i][j] = pascal[i-1][j] + pascal[i-1][j-1];
    }
  }
  #pragma endscop

  for (i = 0; i < N; i++) {
    for (j = 0; j <= i; j++) {
      printf("%3d␣", pascal[i][j]);
    }
    printf("\n");
  }

  return 0;
}
```

Loop Initialization
Loop and `if` Condition
Loop Step
Array Subscript

Loop Initialization

Each loop initialization must be an assignment of the loop counter such that the right hand side is one or several affine expressions aggregated with `max` (resp. `min`) operators if the loop step is positive (resp. negative). Optionally, the iterator can be declared as an `int` in the initialization part.

Example of Loop Initialization	Diagnostic
`int j = 3*i + 2*N`	Correct
`j = ceild(i + N, 10)`	Correct if `j` step is positive
`j = max(i, ceild(N, 3))`	Correct if `j` step is positive
`j = min(min(N,10), 7*i)`	Correct if `j` step is negative
`j = min(max(i, 1), N)`	Incorrect: mixed `min` and `max`

Tip: if the initialization form is too restrictive for a given program, it may be possible to move the troublesome constraints to the loop condition or to an external or internal `if` *condition.*

Loop and `if` Condition

Each loop or `if` condition must be a (composition of) constraint(s) on affine expressions, and function calls.

- Supported C operators are `>`, `>=`, `<`, `<=`, `==`, `!=`, `!`, `&&` and `||`.
- `min` and `max` operators can be used to aggregate expressions in `>`, `>=`, `<` and `<=` constraints. `min` (resp. `max`) expressions must be in the greater (resp. lower) side of the constraints.
- Constraints involving the modulo operator are possible in the following form: let `a` be an affine expression and `x` and `y` two positive integers, then the condition `(a % x == y)` is accepted.
- Function calls alone can be used as valid `if` conditions.

Example of Condition	Diagnostic
`i + 2*j < N`	Correct
`max(i, j) < floord(N, 7)`	Correct
`N>i && !(j>0 || N!=1)`	Correct
`((2*i+1)%3 == 1) && i>j`	Correct
`func(A[i], b)`	Correct in a `if` condition
`min(2*i, N) < 0`	Incorrect: `min` on the lower side
`i + 2`	Incorrect: use `(i + 2) != 0`
`i<N && g(a)`	Incorrect: function call not alone

Tip: to include data-dependent conditions, e.g., `if (A[i] == 0)`*, create a preprocessor macro containing the condition and replace it in the SCoP by the macro-function call, e.g.,* `if (my_condition(A[i]))`.

Loop Step

Updating the loop iterator is only allowed in the loop step part. It must be done by adding an integer to the previous iterator value. Let `i` be a loop iterator and `x` an integer, the following forms are accepted for the loop step part: `i++`, `++i`, `i--`, `--i`, `i += x`, `i -= x`, `i = i+x` and `i = i-x`.

Array Subscript

Array subscripts must be either affine expressions or function calls.

Tip: to include indirections, e.g., `A[B[i]]`*, create a preprocessor macro containing the subscript and replace it in the SCoP by the macro-function call, e.g.,* `A[my_subscript(B[i])]`.

Infinite and `while` Loops

Infinite `for` loops in the form `for (;;)` are supported. `while` loops are supported when the condition is either `1` (infinite loop) or a function call.

Figure 3.2: Clan Reference Card

is checked, illegal transformations are dropped. We show in this section why and how an *optimization-centric* approach is more helpful than a *legality-centric* approach to help the user (a compiler or a human) in the optimization process.

The classic approach of program optimization considers individual optimization directives (like *tile* or *skew* or *fuse* etc. [Wol95]), each of them being associated with specific legality conditions, and sometimes specific static analyses [AK02]. For instance, the unimodular transformations on one side, and the loop fusion/fission on the other side, require distinct legality checking code. This traditional approach has several drawbacks:

- it is almost impossible to define complex loop transformations with a global impact on the loop nest, since their legality conditions would be difficult to formally define [Kel96];
- hence complex transformations must be decomposed into sequences of primitive ones, a fragile and combinatorial task in general [CGP$^+$05];
- each individual transformation must be checked, leading to compile-time overhead and additional fragility, since a single conservative approximation for one of these checks may invalidate the whole sequence [CB-GVB$^+$06];
- in terms of compiler engineering, more effort is needed to scatter and specialize legality checking code in the loop transformation infrastructure [AK02].

Modern compiler optimization techniques ensure legality by construction: data dependences and causality conditions (see Section 2.2.2) are intimate parts of the transformation problem [Fea92a, BHRS08, CB-PBB$^+$11]. As a result, pre-processing has to be applied to the input program to remove as many data dependences as possible before the optimization step, e.g., privatization, total memory expansion, static single assignment form [LF98], index-set splitting [GFL00] etc. There is in general no easy way back from those pre-processing techniques while they may have a serious impact in memory use and/or control overhead. Hence, the ability to check transformations after they have been applied enables new ways to drive an optimization process: if the compiler can reason about *violated dependences*, some fundamental decision flaws of syntactic compilers disappear *by converting early decisions into delayed corrections of illegal transformations*. For example, a typical ill-formed optimization problem like “is there a loop peeling step that would enable fusion of two given loops ?” would simply be converted into the extraction of the minimal set of iterations that violate the fusion, followed by the natural peeling transformation to correct this violation.

In the context of these fundamental and compiler engineering motivations, this section explains how instance-wise dependence information can be used to delay legality checks *after* the application of complex transformations or long transformation sequences. Second we show that beyond dependence analysis, an illegal transformation is not necessarily a dead-end. We show how to exactly determine the violated dependences that need to be corrected. Identifying these violations can in turn enable automatic correction schemes to fix an illegal transformation sequence with minimal changes.

3.2.1 On the Need For Instance-wise Data Dependence Analysis

The power of an automatic optimizer or parallelizer greatly depends on its capacity to decide whether two portions of the program execution may be interchanged or run in parallel. Such knowledge is related to the difficult task of *dependence analysis* which aims at precisely disambiguating memory references.

Many data dependence tests and abstractions have been suggested, with various motivations such as computational cost, precision or application domain (see Section 5.2.4 for details and useful references). Several empirical studies have been conducted to compare those tests [PP91, GKT91, PP96, PK04]. The generally accepted conclusion is: "it is more interesting to use simple tests (like the *Banerjee-test* [WB87] or *I-test* [KKP90]), and simple abstractions (like *Direction Vectors* [Wol95]), because they capture most data dependence information at a low computational cost". Our view significantly contrasts with such generally accepted ideas and with the traditional use of data dependence analysis.

Using an exact instance-wise analysis, a dependence between two statements does not necessarily hamper the application of an optimizing/parallelizing transformation. Indeed, the comparison of data dependence tests in above-mentioned studies is quite biased: it only evaluates the ability to prove or disprove dependences between statements, and not to precisely tell which iterations of those statements are in dependence. The essence of data dependence analysis is to build or to check useful transformations. To prove or disprove dependences between statements is quite rough since optimization/parallelization may often be possible even if there exists data dependences (examples will be provided along with the transformation correction technique, see sections 3.2.3 and 3.3.3). On the contrary, instance-wise analyses and abstractions give the right precision level to decide whether or not to apply a given mapping. Empirical studies of dependence tests/analyses which ignore the impact on transformations are not powerful enough for advanced compiler design.

Few methods provide an exact solution to the dependence problem for general static control kernels, such as the Omega test [Pug91a] or the PIP test [Fea91]. They can enable finer program transformations, like affine mapping [Fea92a, Fea92b, LL97, Gri04, BHRS08, CB-PBB$^+$11], which corresponds to the granularity of our work, at the price of an higher complexity than their conservative counterparts. Reasonning about the scalability of an exact dependence test is postponed until Section 5.1.

3.2.2 Characterization of Violated Dependences

Once an optimizing transformation relation (which may correspond to a complex sequence of classic loop transformations) has been constructed, and applied in the model (see Section 2.2.1), the question arises whether the resulting program still executes correct code. Our approach consists in building the dependence graphs (see Section 2.2.2), before applying any transformation, then to apply a given transformation sequence, and eventually to run a legality check at the very end of the transformation sequence.

In the relation formalism, the legality of a transformation is guaranteed if, for any iteration vectors $\vec{\imath}_S$ and $\vec{\imath}_T$ involved in a dependence relation $\delta_{S,r_S \xrightarrow{d} T,r_T}(\vec{p})$,

$$\theta_S(\vec{\imath}_S) \prec \theta_T(\vec{\imath}_T),$$

where $\prec$ denotes the *lexicographic ordering*. Intuitively, this means that the source instance has to be executed before the target instance after the mapping has been applied.

The violated dependences analyzer computes the iterations that were in a dependence relation in the original program and *whose order has been reversed by the transformation*. These iterations, should they exist, do not preserve the causality of the original program. A violation relation $\upsilon_{S,r_S \xrightarrow{d,v} T,r_T}(\vec{p})$ is a mapping from mapped instances and accessed memory locations of a source statement S to mapped instances and accessed memory locations of a target statement T, at a given dependence depth d and a violation depth v (defined below), for a given pair of memory references r_S and r_T. The violation relation is not empty for two mapped instances iff they are in dependence relation and the source instance is mapped after or at the same time as the target instance. Hence, the violation relation is built using three subrelations:

1. **Dependence condition** of the instances, see Section 2.2.2.

2. **Mapping condition** of the instances, see Section 2.1.3.

3. **Causality violation condition** of the mapped instances: the mapped instances of the source statement in the violation relation are executed after or at the same time as the corresponding mapped instances of the target statement. The constraints involved in this condition depend on the situation. They can be separated into a disjunction with as many components as mapping dimensions in both S and T (i.e. the minimum mapping depth between S and T). Each component corresponds to a given mapping depth and is called a *violation depth*. For a given violation depth $v > 0$, the causality violation condition is made of two parts:

   - the equality of the mapping elements at depth less than v: $t_S^x = t_T^x$ for $x < v$,
   - $t_T^v < t_S^v$ if v is less than the minimum number of mapping dimensions between S and T, $t_T^v \leq t_S^v$ otherwise.

   If the minimum number of mapping dimension is 0, there is no causality violation constraint and the violation may only exist if a dependence exists.

Hence, the general form of a dependence relation with causality violation constraints is the following:

$$\upsilon_{S,r_S \overset{d,v}{\to} T,r_T}(\vec{p}) = \left\{ \left(\begin{array}{c} \vec{\imath}_S \\ \vec{a}_{S,r_S} \\ \vec{t}_S \end{array} \right) \to \left(\begin{array}{c} \vec{\imath}_T \\ \vec{a}_{T,r_T} \\ \vec{t}_T \end{array} \right) \in \begin{array}{c} \mathbb{Z}^{dim(\vec{\imath}_S)+dim(\vec{a}_{S,r_S})+dim(\vec{t}_S)} \\ \times \\ \mathbb{Z}^{dim(\vec{\imath}_R)+dim(\vec{a}_{T,r_T})+dim(\vec{t}_T)} \end{array} \;\middle|\; \Upsilon_{S,r_S \overset{d,v}{\to} T,r_T} \right\},$$

where $\Upsilon_{S,r_S \overset{d,v}{\to} T,r_T}$ is the following violation constraint system:

$$\Upsilon_{S,r_S \overset{d,v}{\to} T,r_T} : \left[\begin{array}{cc:c:cc:c:c:c} \multicolumn{8}{c}{\Delta^+_{S,r_S \overset{d}{\to} T,r_T}} \\ \hdashline T_S^{\vec{\imath}_S} & 0 & T_S^{\vec{t}_S} & 0 & 0 & 0 & T_S^{\vec{p}} & T_S^c \\ \hdashline 0 & 0 & 0 & T_T^{\vec{\imath}_T} & 0 & T_T^{\vec{t}_T} & T_T^{\vec{p}} & T_T^c \\ \hdashline 0 & 0 & I^{1..v-1,\bullet} & 0 & 0 & -I^{1..v-1,\bullet} & 0 & 0 \\ \hdashline 0 & 0 & I^{v,\bullet} & 0 & 0 & -I^{v,\bullet} & 0 & 0 \text{ or } -1 \end{array} \right] \left(\begin{array}{c} \vec{\imath}_S \\ \hdashline \vec{a}_{S,r_S} \\ \hdashline \vec{t}_S \\ \hdashline \vec{\imath}_T \\ \hdashline \vec{a}_{T,r_T} \\ \hdashline \vec{t}_T \\ \hdashline \vec{p} \\ \hdashline 1 \end{array} \right) \begin{array}{c} = or \geq \\ \hdashline \geq \\ \hdashline \geq \\ \hdashline = \\ \hdashline \geq \end{array} \vec{0},$$

where $\Delta^+_{S,r_S \overset{d}{\to} T,r_T}$ is the dependence constraint matrix (see Section 2.2.2) where additional columns set to zero have been added for the corresponding $\vec{t}_S$ and $\vec{t}_T$ dimensions. For reference, the complete system is the following:

$$\left[\begin{array}{c:c:c::c:c:c::c:c}
D_S^{\vec{\imath}_S} & 0 & 0 & 0 & 0 & 0 & D_S^{\vec{p}} & D_S^c \\ \hdashline
0 & 0 & 0 & D_T^{\vec{\imath}_T} & 0 & 0 & D_T^{\vec{p}} & D_T^c \\ \hdashline
A_{S,r_S}^{\vec{\imath}_S} & A_{S,r_S}^{\vec{a}_{S,r_S}} & 0 & 0 & 0 & 0 & A_{S,r_S}^{\vec{p}} & A_{S,r_S}^c \\ \hdashline
0 & 0 & 0 & A_{T,r_T}^{\vec{\imath}_T} & A_{T,r_T}^{\vec{a}_{T,r_T}} & 0 & A_{T,r_T}^{\vec{p}} & A_{T,r_T}^c \\ \hdashline
0 & I & 0 & 0 & -I & 0 & 0 & 0 \\ \hdashline
I^{1..d-1,\bullet} & 0 & 0 & -I^{1..d-1,\bullet} & 0 & 0 & 0 & 0 \\ \hdashline
I^{d,\bullet} & 0 & 0 & -I^{d,\bullet} & 0 & 0 & 0 & 0 \text{ or } -1 \\ \hdashline \hdashline
T_S^{\vec{\imath}_S} & 0 & T_S^{\vec{\imath}_S} & 0 & 0 & 0 & T_S^{\vec{p}} & T_S^c \\ \hdashline
0 & 0 & 0 & T_T^{\vec{\imath}_T} & 0 & T_T^{\vec{\imath}_T} & T_T^{\vec{p}} & T_T^c \\ \hdashline \hdashline
0 & 0 & I^{1..v-1,\bullet} & 0 & 0 & -I^{1..v-1,\bullet} & 0 & 0 \\ \hdashline
0 & 0 & I^{v,\bullet} & 0 & 0 & -I^{v,\bullet} & 0 & 0 \text{ or } -1
\end{array}\right]
\left(\begin{array}{c}
\vec{\imath}_S \\ \hdashline \vec{a}_{S,r_S} \\ \hdashline \vec{t}_S \\ \hdashline \hdashline \vec{\imath}_T \\ \hdashline \vec{a}_{T,r_T} \\ \hdashline \vec{t}_T \\ \hdashline \hdashline \vec{p} \\ \hdashline 1
\end{array}\right)
\begin{array}{c}
\geq \\ \hdashline \geq \\ \hdashline \geq \\ \hdashline \geq \\ \hdashline = \\ \hdashline = \\ \hdashline \geq \\ \hdashline \hdashline \geq \\ \hdashline \geq \\ \hdashline \hdashline = \\ \hdashline \geq
\end{array}
\vec{0}$$

Note that we consider here *potential* violations as violations. For instance, when two instances in a relation dependence have the same mapping, the violation exists since it is not possible to know which one will be executed first from the mapping. However, the code generation step may still generate (by chance) a correct code. To compute only actual violations, simply consider that there is no violation if the minimum number of mapping dimension is 0 and that the second part of the causality violation condition is always $t_{T,v} < t_{S,v}$. Finally, if the mapping relation is a union, the violation analysis has to be performed on each component of the union.

For a given set of mapping relations and a data dependence graph, the non empty violation relations are gathered into a *violated dependence graph*. Beyond characterizing illegal transformation sequences, it is possible to reason about these graph of violated dependences and effectively derive more flexible optimization algorithms.

3.2.3 Removing Data Dependence Violations

When a violated dependence graph has been computed for a given mapping, it is possible to reason about it to modify either the input program or the mapping to remove violations. Some transformations like, e.g., privatization, or array expansion [LF98, Fea88b], do not generate new violations, but can remove some of them at the price of a higher memory footprint. Applying such a transformation after violated dependence analysis ensures the minimal expansion is made for the desired mapping to be applied.

Another way is to modify the mapping itself. Starting from an incorrect transformation sequence, the goal is to reestablish the legality of the final program while disrupting the mapping as little as possible. The root of this idea comes from Kelly and Pugh's work where they try to apply *schedule alignment* in the context of the UTF framework [KP93]. They use a depth by depth, conservative approach and let their alignment technique completely set the constant part of the schedules. We already designed a powerful automatic correction scheme based on instance-wise dependence information based on the space of legal transformations [CB-BF05]. Our technique could complete partially defined schedules, not only the constant part, and apply complex corrections in the specific context of data locality optimizing schedules. However, the complexity of the technique can make it impractical for large multidimensional problems. Vasilache et al. proposed a correction technique, which, provided a fully specified, but illegal transformation, and ought to compute a minimally intrusive adjustment to the schedule matrices [VCP07].

Assuming the given transformation is an upper bound to the peak performance achievable for this application, the adjustment to make it correct becomes an optimization problem in itself. This problem is NP-complete in general (e.g., when considering loop fusion/fission as a means to correct the schedule [Dar00]). We are thus working on correction strategies for which a minimal adjustment can be derived effectively, and on sub-optimal heuristics for other correction strategies.

We propose here a new mapping correction algorithm with the same goal as Vasilache et al.'s algorithm [VCP07], i.e., starting from an illegal mapping, find a constant (possibly parametric) shifting with minimal deviation. The two techniques are quite different. Contrary to Vasilache et al. the algorithm presented here is not iterative on depth (however it can be modified in this way to reduce complexity, see the depth-by-depth variant page 43). Moreover, rather than reasoning on the space where a violation exists (to find the extremal amount of time units between a source instance and a target instance), we focus on the space where no violation exists.

Our correction algorithm, named after its implementation "Candl", is depicted in Figure 3.3. The intuitive idea is the following:

- First, we introduce new *correction parameters* and we use them to *shift* the mapping dimensions to be corrected. A correction parameter is personal to a given dimension of a given mapping relation. We create one correction parameter per dimension we want to correct in each statement involved in a violation (even transitively). Then we shift each dimension we want to correct with its own correction parameter (i.e., we substitute t with $t-C$ in the relation if t is the dimension to correct and C is its correction parameter).

- Next, we find the constraints on the correction parameters such that no violation exists. We rely on Parametric Integer Programming (PIP) for such a task [Fea88a, CB-FcCB02]. For this we express the violation relations using the shifted mapping and we ask PIP for a solution. The solution is provided as a *quasi-affine selection tree*, or *quast*, i.e., a selection tree based on constraints on parameters. Each path to $\perp$ (no solution) in the quast corresponds to a part of the parameter space where the violation does not exist. We gather all those parts in a union of polyhedra defining the "safe" part of the space with respect to the parameters for a given violation relation. Lastly we intersect all the unions for all the violation relations.

- Finally, we select a solution in the "safe" part of the space, if it exists, such that it has a minimum deviation from the original mapping. This selection highly depend on the structure of the mapping (are there *beta* dimensions [CB-GVB$^+$06] so that some dimensions are more important than others?). Our general strategy is to minimize the absolute value of the correction values with decreasing priority, from first to last mapping dimensions. It is again an optimization problem where we use PIP. It is easy to adapt the strategy to the mapping structure if we know it.

A complete example of correction using the algorithm is presented in Figure 3.4. A user wishes to transform the original program in 3.4(a) using the mapping relation in 3.4(b), much probably to improve locality for the array A. Using this mapping, a code generator could generate the code in 3.4(c), it corresponds to a loop fusion. Unfortunately it is easy to see that in the target code, $S2$ is now consuming data before $S1$ produces them. A violated dependence analysis would show that the violation relation $\Upsilon_{S1,1 \xrightarrow{0,2} S2,2}$ is not empty. We can try to correct this mapping.

Because there are only two statements, it is not necessary to shift both. We choose to apply a parametric correction shifting to the mapping of $S1$. This shifting impacts the violated dependence relation

CANDL MAPPING CORRECTION ALGORITHM

```
• Input:
  $M$: set of original mapping relations
  $DDG$: data dependence graph
  $VDG$: violated dependence graph

• Output:
  $M_{shift}$: set of corrected mapping relations

1. choose the set of mapping relations $M_{shift}$ to correct by shifting (at least one mapping from
   each violated dependence in $VDG$)

2. for each mapping relation $\mathcal{M}$ in $M_{shift}$ do

   (a) extend $\mathcal{M}$ with $p$ correction parameters, where $p$ is the maximal violation depth of all
       the violations where $\mathcal{M}$ in involved
   (b) shift the $i^{th}$ mapping dimension of $\mathcal{M}$ with the $i^{th}$ correction parameter, $0 \leq i \leq p$ (i.e.,
       set the column corresponding to the $i^{th}$ correction parameter to the opposed column of
       the $i^{th}$ mapping dimension).

3. build the list $V$ of violated dependence relations which have to be recomputed with respect
   to the modified mappings

4. $\mathcal{D}_{shift} \leftarrow$ universe

5. for each violated dependence relation $\mathcal{V}$ in $V$ do

   (a) replace the original mapping(s) with the shifted mapping(s) in $\mathcal{V}$ (i.e., introduce the
       correction parameters as explained in step 2)
   (b) compute solutions to $\mathcal{V}$ as a quast on correction and global parameters using, e.g., PIP
   (c) build a union of polyhedra $\overline{\mathcal{V}}$ such that each union component corresponds to the
       parametric conditions leading to no solution (bottom or $\perp$ in PIP) in the quast; in those
       polyhedra, correction parameters are promoted to variables while global parameters
       remain parameters
   (d) $\mathcal{D}_{shift} \leftarrow \mathcal{D}_{shift} \cap \overline{\mathcal{V}}$

6. if $\mathcal{D}_{shift}$ is empty then

   • return $\emptyset$

   else

   (a) choose a low deviation solution to $\mathcal{D}_{shift}$ (computed using, e.g., PIP), i.e., a correcting
       value for each correction parameter (possibly parametric itself)
   (b) replace the correction parameters in $M_{shift}$ with their corresponding correcting value
       and remove the correction parameters (i.e., remove the corresponding columns)
   (c) return $M_{shift}$
```

Figure 3.3: Correction by shifting algorithm

$\Upsilon_{S1,1 \xrightarrow{0,2} S2,2}$, but also $\Upsilon_{S1,1 \xrightarrow{0,1} S2,2}$. Hence it is necessary to consider both, as shown in Figure 3.4(e) and 3.4(f) where we applied the parametric correction shifting in the relation. For both relations, we compute the "safe" space where, depending on the parameters, no violation exists. This is done using a call to PIP [Fea88a, CB-FcCB02] with the violation relation constraints. The quasts and their conversion to "safe" space are shown in 3.4(g) and 3.4(h). The intersection of the violation-relative "safe" spaces gives the global "safe" space shown in 3.4(i).

Each part of the "safe" space union contains possible solutions. Moreover, constraints on global parameters forces versioning: different corrections may be applied depending on the global parameter values. In our example, we can see from 3.4(i), that any correction is correct for $N < 1$ (because the dependence does not exist), but not for $N >= 1$. We are free to chose any solution correct for $N >= 1$. However they are not equivalent with respect to deviation from the original mapping. For instance two possible corrections are shown in 3.4(j) and 3.4(k). One of them totally cancel the loop fusion, while the other slightly modifies it. Our general approach is to look for the smallest possible shifting (in absolute value) for the first dimension, the order of priority being the order of dimensions, hence to achieve the correction shown in 3.4(k).

From the proposed algorithm, it is easy to derive useful variants:

Full-Depth variant For practical reasons, we restricted the number of shifted dimensions for a given mapping to the maximum violation depth where it is involved (see Step 2 of Figure3.3). However, shifting at depth d can generate violations at depth $d' > d$ (e.g., when shifting results in a loop fusion). While we can try to solve it at depth $< d$, we can allow to correct it at a further depth. To enable this, simply add correcting parameters to all dimensions.

Depth-by-Depth variant For complex cases, the computation of the "safe" part of the space can be difficult (PIP computation may not converge or the number of elements in the union may explode). A depth-by-depth approach can be derived if scalability problems rise. To achieve this, an additional outer loop on mapping depth has to be added to the algorithm. At the n^{th} iteration, only the correcting parameter for the n^{th} dimension is added to all mapping involved in a dependence relation, and the correcting values of the $n-1$ first dimensions are integrated for violation relation construction.

If the correction algorithm fails, then the mapping is considered as illegal and is simply discarded.

3.3 Semi-Automatic Mapping Construction

The polyhedral mapping abstraction is too complex to be used directly by non-expert human end-users or optimizing systems based on classical optimizing directives like *tile* or *fuse* or *skew*. However, they could strongly benefit from its unique properties of exact data dependence analysis to check or to correct their transformations, and from automatic code generation to handle the complex and error-prone optimization implementation in a transparent way.

Several frameworks have been proposed to expose a high-level interface on top of a polyhedral engine, UTF (Unified Transformation Framework) [KP93] and its implementation in the Petit tool [Kel96] being arguably the very first of them. The URUK (Unified Representation Universal Kernel) framework [Gir05, Vas07, CB-GVB$^+$06, CB-BCG$^+$03a] enables the composition of any complex sequence of classical loop transformations (including tiling) decoupled from any syntactic form of the program.

```
    for (i = 0; i <= N; i++)
S1:   A[i] = B[i];

    for (i = 0; i <= N; i++)
S2:   C[i] = A[i+1];
```

(a) Original Program

$$\theta_{S1} = \left\{ (\ i\ ) \rightarrow \begin{pmatrix} 0 \\ i \\ 0 \end{pmatrix} \right\}$$

$$\theta_{S2} = \left\{ (\ i\ ) \rightarrow \begin{pmatrix} 0 \\ i \\ 1 \end{pmatrix} \right\}$$

(b) Illegal Mapping

```
    for (i = 0; i <= N; i++) {
S1:   A[i] = B[i];
S2:   C[i] = A[i+1];
    }
```

(c) Illegal Target Code

$$\theta_{S1} = \left\{ (\ i\ ) \rightarrow \begin{pmatrix} 0+C1 \\ i+C2 \\ 0 \end{pmatrix} \right\}$$

(d) Parametric Shifting of θ_{S1}

$$\Upsilon_{S1,1\overset{0,2}{\rightarrow}S2,2} :$$

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1
\end{bmatrix}
\begin{pmatrix} i \\ a_{S1,1} \\ t_{S1,1} \\ t_{S1,2} \\ t_{S1,3} \\ i' \\ a_{S2,2} \\ t_{S2,1} \\ t_{S2,2} \\ t_{S2,3} \\ N \\ C1 \\ C2 \\ 1 \end{pmatrix}
\begin{matrix} \geq \\ \geq \\ = \\ = \\ = \\ = \\ = \\ = \\ \geq \end{matrix} \vec{0}$$

(e) Violated Dependences With Shifted θ_{S1}

$$\Upsilon_{S1,1\overset{0,1}{\rightarrow}S2,2} :$$

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1
\end{bmatrix}
\begin{pmatrix} i \\ a_{S1,1} \\ t_{S1,1} \\ t_{S1,2} \\ t_{S1,3} \\ i' \\ a_{S2,2} \\ t_{S2,1} \\ t_{S2,2} \\ t_{S2,3} \\ N \\ C1 \\ C2 \\ 1 \end{pmatrix}
\begin{matrix} \geq \\ \geq \\ = \\ = \\ = \\ = \\ = \\ = \\ \geq \end{matrix} \vec{0}$$

(f) Potential Violations With Shifted θ_{S1}

```
if (N >= 1)
  if (C1 >= 0)
    if (-C1 >= 0)
      if (C2 >= 0)
        solution
      else
        ⊥
    else
      ⊥
  else
    ⊥
else
  ⊥
```

$$\left\{ \begin{array}{l} N \geq 1 \\ C1 = 0 \\ C2 < 0 \end{array} \right. \cup \quad \left\{ \begin{array}{l} N \geq 1 \\ C1 > 0 \end{array} \right. \cup \quad \left\{ \begin{array}{l} N \geq 1 \\ C1 < 0 \end{array} \right. \cup \quad \left\{ \ N < 1 \right.$$

(g) Quast and safe space for $\Upsilon_{S1,1\overset{0,2}{\rightarrow}S2,2}$

```
if (N >= 1)
  if (C1 >= 1)
    solution
  else
    ⊥
else
  ⊥
```

$$\left\{ \begin{array}{l} N \geq 1 \\ C1 < 1 \end{array} \right. \cup \quad \left\{ \ N < 1 \right.$$

(h) Quast and safe space for $\Upsilon_{S1,1\overset{0,1}{\rightarrow}S2,2}$

$$\left\{ \ N < 1 \right. \cup \left\{ \begin{array}{l} N \geq 1 \\ C1 = 0 \\ C2 < 0 \end{array} \right. \cup \left\{ \begin{array}{l} N \geq 1 \\ C1 < 0 \end{array} \right.$$

(i) Result of the intersection of safe spaces

$$\theta_{S1} = \left\{ (\ i\ ) \rightarrow \begin{pmatrix} 0-1 \\ i+0 \\ 0 \end{pmatrix} \right\}$$

```
    for (i = 0; i <= N; i++)
S1:   A[i] = B[i];

    for (i = 0; i <= N; i++)
S2:   C[i] = A[i+1];
```

(j) High Deviation Depth 1 Correction

$$\theta_{S1} = \left\{ (\ i\ ) \rightarrow \begin{pmatrix} 0+0 \\ i-1 \\ 0 \end{pmatrix} \right\}$$

```
S1: A[0] = B[0];
    for (i = 0; i < N; i++) {
S1:   A[i+1] = B[i+1];
S2:   C[i] = A[i+1];
    }
S2: C[N] = A[N+1];
```

(k) Low Deviation Depth 2 Correction

Figure 3.4: Example of Mapping Correction

Transformations are done on both linear mapping *functions* with specific structure and iteration domains. Because each individual transformation in URUK preserves compositionality invariants, ensuring a target code corresponding to the sequence of transformations is guaranteed and it is possible to check for the legality of the complete sequence by maintaining the data dependence graph when domain transformations are performed. The CHiLL framework [CCH08] shares a lot with URUK with the exception of an *alignment* pre-processing to iteration spaces to ensure they have the same dimensionality and the necessity for intermediate dependence checks.

We present yet another revisiting of classical loop transformations in the polyhedral model, but based on the mapping relation abstraction discussed in Section 2.1.3. The new framework, named after its implementation *Clay*, is designed as a new generation URUK framework where domain transformations are not needed anymore (this is necessary in both URUK and CHiLL to support, e.g., index-set splitting, peeling, strip-mining or tiling). The complete mapping information is embedded in the mapping relation which makes this new framework more elegant and simpler (e.g., there is no need for intermediate data dependence graph updates or checks). The specific mapping structure used in Clay as well as the various notations and operators are defined in Section 3.3.1. The translation of the classical loop transformation using relations and our notations and operators are shown in Section 3.3.2.

3.3.1 Clay Mapping Structure, Notations and Operators

We use a particular mapping relation structure with special dimensions which must be preserved during the process of combining transformations. Simply, in a similar way to UTF, URUK and CHiLL, odd mapping dimensions (the first dimension being 1) are devoted to express the lexicographic order of loops and statements. They are called β in URUK and *auxiliary loops* in CHiLL. We use here the name β for consistency with URUK. β are restricted to be equal to non-parametric constant values. The vector formed from β values for a given mapping union component is called a β-*vector*, or $\vec{\beta}$. Each β-vector must be unique and cannot be a prefix of other β-vectors. Even mapping dimensions are called α[1]. Hence, the general form of the mapping dimension vector is $(\beta_1, \alpha_1, \beta_2, \alpha_2 \ldots, \beta_n, \alpha_n, \beta_{n+1})^T$. As an example, the scheduling of the original program, presented in Section 2.1.3 does respect this structure. It is commonly referred as a "$2d+1$" structure since in UTF, URUK and CHiLL, the mapping is $(2d+1)$-dimensional where d is the number of dimensions of the corresponding iteration domain. In our formalism, the number of mapping dimensions is odd, but there is no link with the dimensionality of the iteration domain.

The mapping structure ensures that (1) a target code corresponding to the desired sequence of transformations can be generated and (2) that the whole sequence can be checked for data dependence after it has been constructed. It is supposed that the input mapping representation respects this structure before being transformed, then the transformations will ensure it is preserved. Comparatively to Girbal's work [Gir05] where many invariants were necessary, the mapping structure corresponds to the *sequentiality of* β only.

To express classical loop transformations using the relation formalism and our mapping structure, we use some notations and new operators shown in Figure 3.5. Their purpose is to represent specific relation sets, β and α vectors and their elements. The notion of β-prefix is paramount. It will be necessary to select specific subsets of relations implied in a given transformation. From a syntactic point of view, a β-prefix can represent a specific statement or a specific loop (or, equivalently, the set of statements

[1]Those dimensions correspond to ϕ in URUK, but because our formalism does not require further decomposition, we prefer to use α here.

$\mathcal{M}$	a mapping union component
$\vec{\rho}$	a β-prefix, if empty it corresponds to the root level
$\vec{\beta}_{\mathcal{M}}$	β-vector of $\mathcal{M}$
$\vec{\alpha}_{\mathcal{M}}$	α-vector of $\mathcal{M}$
α^i	symbolic i^{th} element of any α-vector; when involved in a formula applying to $\forall\mathcal{M}$, it translates to $\alpha^i_{\mathcal{M}}$
$\mathcal{M}_*$	set of all mapping relations
$\mathcal{M}_{\vec{\rho}}$	subset of $\mathcal{M}_*$ restricted to union components such that $\vec{\rho}$ is a β-prefix
$\mathcal{M}_{\vec{\rho},i}$	i^{th} subset of $\mathcal{M}_{\vec{\rho}}$ when those subsets are created and sorted according to the value $\beta^{\dim(\vec{\rho})+1}_{\mathcal{M}}$
$\mathcal{M}_{\vec{\rho},next}$	subset of $\mathcal{M}_*$ restricted to union components such that $\vec{\rho}^{1..\dim(\vec{\rho})-1}$ is the β-prefix and the $\dim(\vec{\rho})^{\text{th}}$ beta dimension is the next value strictly greater than $\rho^{\dim(\vec{\rho})}$
$\mathcal{M}_{\vec{\rho},>}$	subset of $\mathcal{M}_*$ restricted to union components such that $\vec{\rho}^{1..\dim(\vec{\rho})-1}$ is the β-prefix and the $\dim(\vec{\rho})^{\text{th}}$ beta dimension is strictly greater than $\rho^{\dim(\vec{\rho})}$
$\uparrow_d$	unary relation extend operator: $\uparrow_d(\mathcal{M})$ inserts one new α dimension and one new β dimension before the d^{th} β dimension of $\mathcal{M}$. The new β dimension is set to 0 while the new α dimension is left free
$copy$	unary relation copy operator: $copy(\mathcal{M})$ creates a new relation as a carbon-copy of $\mathcal{M}$

Figure 3.5: Notations and Operators used in the Clay Formalism

embedded in that loop). The empty vector is a particular β-prefix value used to select all mappings, or, from a syntactic point of view, the root of the program.

3.3.2 Revisiting Classical Transformations in Clay

In this section we revisit classical loop transformations [Wol95] using the relation abstraction presented in Section 2.1.1 and notations and operators presented in Section 3.3.1. We suppose, before starting to apply any transformation, that each program has been translated to relations as described in Chapter 2 and more specifically, that mapping relations correspond to the *scheduling of the original program* described in Section 2.1.3. Each transformation is presented as a primitive whose parameters can be integers, integer vectors, affine expressions or affine constraints.

Most of those primitives are more general than classical loop transformations because they actually target sets of statement instances and not directly loops. For instance it is possible to achieve a loop interchange for a subset of statements inside the loops using only one directive, while it would require a composition of loop splitting, loop interchange and loop fusion using classical loop transformations [Wol95]. We do not try here to provide a one-to-one translation from classical transformations to relation formalism. Instead, we provide our own view where classical transformations are a particular case for each primitive. The most radical change is probably for *shifting* which can be constant or parametric as usual, but which is also generalized to *variable shifting*. It is clear from our formalism that, e.g., *skewing* or *reversal* are simply particular cases of variable shifting.

Loop Reordering

Syntax	reorder($\vec{\rho}$, $\vec{v}$)
Effect	$\forall i \;\; 0 \le i \le \dim(\vec{v}),\;\; \forall \mathcal{M} \in \mathcal{M}_{\vec{\rho},i},\;\; \beta_{\mathcal{M}}^{\dim(\vec{\rho})+1} \leftarrow v^i$
Condition	$\mathrm{card}(\mathcal{M}_{\vec{\rho},*}) = \dim(\vec{v})$ and $\forall i,j \;\; i \neq j,\;\; v^i \neq v^j$
Definition	Reorganize statements and loops included in the loop corresponding to $\vec{\rho}$ (or at the root level) according to the vector $\vec{v}$. The i^{th} element of the vector $\vec{v}$ corresponds to the new order of the statement or the loop which was originally at order i.
Note	Update the β value after $\vec{\rho}$ of the selected mappings according to the reordering vector.

Loop Interchange

Syntax	interchange($\vec{\rho}$, $d1$, $d2$)
Effect	$\forall \mathcal{M} \in \mathcal{M}_{\vec{\rho}},\;\; \alpha_{\mathcal{M}}^{d1} \leftrightarrow \alpha_{\mathcal{M}}^{d2}$
Condition	$\forall \mathcal{M} \in \mathcal{M}_{\vec{\rho}},\;\; 1 \le d1, d2 \le \dim(\vec{\alpha}_{\mathcal{M}})$
Definition	Interchange the loop at loop-depth $d1$ with the loop at loop-depth $d2$ for all statement instances with β-prefix $\vec{\rho}$.
Note	In the matrix representation, swap the columns corresponding to α^{d1} and α^{d2}.

Shifting

Syntax	shift($\vec{\rho}$, d, *amount*)
Effect	$\forall \mathcal{M} \in \mathcal{M}_{\vec{\rho}},\;\; \alpha_{\mathcal{M}}^{d} \leftarrow \alpha_{\mathcal{M}}^{d} + amount$
Condition	$1 \le d \le \dim(\vec{\rho})$ and *amount* must be an affine expression not involving mapping input variables.

Definition	Move all statement instances with β-prefix $\vec{\rho}$ from their iteration coordinates to those coordinates plus *amount*, *amount* being the shifting distance, possibly parametric and/or variable.
Note	This is a substitution of an α value: in the matrix representation of the mapping $\mathcal{M}$, add n times *amount* to each row, where n is the coefficient of $\alpha^d_{\mathcal{M}}$.

Loop Fusion

Syntax	fuse($\vec{\rho}$)
Effect	$\mathit{offset} \leftarrow \max_{m\in\mathcal{M}_{\vec{\rho}}}\left(\beta_m^{\dim(\vec{\rho})+1}\right);$ $\forall \mathcal{M} \in \mathcal{M}_{\vec{\rho},next},\ \begin{cases} \beta_{\mathcal{M}}^{\dim(\vec{\rho})} & \leftarrow \quad \rho^{\dim(\vec{\rho})} \\ \beta_{\mathcal{M}}^{\dim(\vec{\rho})+1} & \leftarrow \quad \beta_{\mathcal{M}}^{\dim(\vec{\rho})+1} + \mathit{offset} + 1 \end{cases}$
Condition	$\vec{\rho}$ and the existing β-prefix lexicographically directly after $\vec{\rho}$ correspond to loops.
Definition	Fuse the loop corresponding to $\vec{\rho}$ with its direct successor. The order of the statements and loops inside the fused loop is the same as before fusion.

Loop Splitting

Syntax	split($\vec{\rho}$, d)
Effect	$\forall \mathcal{M} \in \mathcal{M}_{\vec{\rho},>},\ \beta^d_{\mathcal{M}} \leftarrow \beta^d_{\mathcal{M}} + 1$
Condition	$1 \leq d \leq \dim(\vec{\rho})$
Definition	Split the loop at depth d for all statement instances with β-prefix $\vec{\rho}^{1..d}$ into two loops, the second one directly following the first one, such that the statement instances lexicographically after $\vec{\rho}$ at depth $\dim(\vec{\rho})$ in that loop belong to the second loop while the others belong to the first loop.

Index-Set Splitting

Syntax	iss($\vec{\rho}$, *constraint*)
Effect	$\mathit{offset} \leftarrow \max_{m\in\mathcal{M}_{\vec{\rho}}}\left(\beta_m^{\dim(\vec{\rho})+1}\right);$ $\forall \mathcal{M} \in \mathcal{M}_{\vec{\rho}}, \mathcal{M} \leftarrow \begin{array}{l} \left(\mathrm{copy}(\mathcal{M}) \cap \mathit{constraint}\right) \ \bigcup \\ \left(\mathrm{copy}(\mathcal{M}) \cap \overline{\mathit{constraint}};\ \beta_{\mathcal{M}}^{\dim(\vec{\rho})+1} \leftarrow \beta_{\mathcal{M}}^{\dim(\vec{\rho})+1} + \mathit{offset} + 1\right) \end{array}$
Condition	*constraint* is an affine constraint on the mapping variables and parameters, and $\vec{\rho}$ corresponds to a loop.

Definition	Transform every mapping of statement instances with β-prefix $\vec{\rho}$ to a union of two mappings, one including the new constraint *constraint* and the other one including its negation. The uniqueness of the β in all mapping union components is preserved to ensure further transformations can target either union.
Note	If $\vec{\rho}$ is a β-vector (hence if it corresponds to a statement instead of a loop), replace $\mathcal{M}_{\vec{\rho}}$ with $\mathcal{M}_{1..\dim(\vec{\rho})-1}$ in the offset computation, and $\dim(\vec{\rho})+1$ with $\dim(\vec{\rho})$ everywhere. We did not do the case distinction to maintain the notation simple.

Strip-Mining

Syntax	stripmine($\vec{\rho}$, d, l)
Effect	$\forall \mathcal{M} \in \mathcal{M}_{\vec{\rho}},\ \mathcal{M} \leftarrow \big((\uparrow_d(\mathcal{M})) \cap (l * \alpha^d_{\mathcal{M}} \leq \alpha^{d+1}_{\mathcal{M}} \leq l * \alpha^d_{\mathcal{M}} + l - 1)\big)$
Condition	$1 \leq d \leq \dim(\vec{\rho})$
Definition	Decompose the loop at depth d, for all statement instances with β-prefix $\vec{\rho}$, into two consecutive loops such that, for each iteration of the first loop, the second loop iterates over a chunk of at most l consecutive iterations of the original loop.
Note	In the matrix representation, insert two column before the one corresponding to the previous α^d and three rows: two for the inequalities making the relation between the new α^d and α^{d+1} (the previous one) and one for the equality stating that the new β dimension is 0.

Loop Reversal (Particular Case of Shifting)

Syntax	reverse($\vec{\rho}$, d)
Effect	shift($\vec{\rho}$, d, $-2 * \alpha^d$)
Condition	See shifting.
Definition	Reverse the iteration order of the loop at depth d for all statement instances with β-prefix $\vec{\rho}$.
Note	In the matrix representation, it translates to opposing the α^d column.

Loop Skewing (Particular Case of Shifting)

Syntax	skew($\vec{\rho}$, d, *factor*)
Effect	shift($\vec{\rho}$, $\dim(\vec{\rho})$, $factor * \alpha^d$)

Condition See shifting.

Definition Reshape the iteration space such that the loop iterator at depth $\dim(\vec{\rho})$ linearly depends on the loop iterator at depth d with a coefficient *factor* (*skew factor*).

Peeling (Composition of Index-Set Splitting and Loop Splitting)

Syntax peel($\vec{\rho}$, *bound*)

Effect $cut \leftarrow \max_{m \in \mathcal{M}_{\vec{\rho}}} \left(\beta_m^{\dim(\vec{\rho})+1}\right);$
iss($\vec{\rho}$, $\alpha^{\dim(\vec{\rho})} \leq bound$);
$\text{split}\Big(\binom{\vec{\rho}}{cut}, \dim(\vec{\rho})\Big);$

Condition $\vec{\rho}$ corresponds to a loop and *bound* is an affine expression on the parameters and the constant.

Definition Extract from the loop at depth $\dim(\vec{\rho})$ all iterations such that the iterator value is lower or equal to the parametric value *bound*. The extracted iterations are put inside a loop in their original order just before the loop where they have been extracted.

Many other transformations may be designed or implemented as a composition of other transformations. **tiling**, for instance, is the classic composition of strip-mining and interchange.

3.3.3 Correcting Transformation Scripts

An arbitrary long sequence of Clay primitives can be built by a tool or an expert guided by performance analyzers to form an optimization script. Such a script will, under the hood, be transposed to a set of mapping relations, one per statement of the original program. Those mappings can be checked for legality by the violation analysis but they can also benefit from the automatic correction as stated by Lemma 3.1.

Lemma 3.1 *The correction algorithm of Figure 3.3 preserves Clay's mapping structure, except for the last* β *dimension which can be updated to respect it.*

Proof. The correction algorithm may change the mappings only by adding a parametric constants to each dimension. Such shifting cannot generate new iterations. Moreover, in the case of the β dimensions of the Clay formalism, the correction constant cannot be parametric because those dimensions are not linked directly or indirectly to the parameters. Hence, PIP cannot issue parametric conditions involving the β dimensions, and they remain constant after correction.

β-vector uniqueness is not guaranteed to be preserved by the correction algorithm. However, two iterations cannot share the same mapping because it is considered as a violation in the violated dependence analysis. Thus, the last β dimension of statements with the same β-vector (or one β-vector and some β-prefix) is not meaningful for ordering. It follows they can be updated in such a way that each β vector is unique and is not the prefix of other β-vectors, provided other β values are updated to respect the relative orders, if necessary. ∎

```
$ cat ring_roberts.c

#pragma scop
/* Clay
   fuse(L1);
 */

/* Ring Blur Filter */
for (i = 1; i < length - 1; i++)
  for (j = 1; i < width - 1; j++)
    Ring[i][j] = (Img[i-1][j-1] + Img[i-1][j] + Img[i-1][j+1]+
                  Img[i][j-1]   +               Img[i][j+1]   +
                  Img[i+1][j-1] + Img[i+1][j] + Img[i+1][j+1])/8;

/* Roberts Edge Detection Filter */
for (i = 1; i < length - 2; i++)
  for (j = 2; i < width - 1; j++)
    Img[i][j] = abs(Ring[i][j]   - Ring[i+1][j-1])+
                abs(Ring[i+1][j] - Ring[i][j-1]);
#pragma endscop

$ clay --check ring_roberts.c

[Clay] Illegal mapping:
digraph G {
# Legality Violation Graph
# Generated by Candl 0.7.0 MP bits
  S1 -> S2 [label=" RAW depth 0, ref 1->3, viol 2 "];
  S1 -> S2 [label=" RAW depth 0, ref 1->4, viol 2 "];
  S1 -> S2 [label=" WAR depth 0, ref 2->1, viol 2 "];
  S1 -> S2 [label=" WAR depth 0, ref 3->1, viol 2 "];
  S1 -> S2 [label=" WAR depth 0, ref 4->1, viol 2 "];
# Number of edges = 5
}

[Clay] Suggested correction:
fuse(L1);
shift(S2, 1, 1);
```

Figure 3.6: Correction of Loop Fusion on a `Ring-Roberts` Edge Detection Filter for Noisy Images

Hence, corrections can translate to a sequence of shifting (for α dimensions) and relaxed[2] reordering (for β dimensions) as defined in Section 3.3.2. Note that while we are looking for a minimum deviation from the original transformation, the correction may undo it entirely if no better solution is found by the correction algorithm.

An example of the use of such property is shown in Figure 3.6 where two successive filters are applied on an image. An expert can notice the data reuse of the array `Img` and ask for a loop fusion to improve locality using Clay. The script, here a single primitive, is set thanks to a special comment. The implementation of Clay accepts β-prefix and also more user-friendly inputs, e.g., Ln targets all iterations inside the nth loop (nth occurrence of the `for` keyword in the source code text) or Sn targets the iterations of the nth statement (nth statement in the source code text). In this case the user asks to fuse L1 with the

[2]Our primitive asks for a list of unique logical dates, here they are not forced to be unique.

next loop. Unfortunately this transformation is not legal as shown by the list of violated dependences, but a correction by shifting is suggested automatically by the tool: as we can see it was necessary to let a complete column of pixels to be processed before fusion, a slight deviation from the optimizing transformation.

3.4 Easing Polyhedral Framework Integration: the OpenScop Initiative

In the previous sections, we presented means to interact with the final users that are looking to actually optimize their codes. In this section, we consider intermediate users which are designing optimizing tools. We present here shortly the OpenScop initiative to ease both polyhedral framework construction and their integration within optimizing tools like compilers.

OpenScop is an open specification that defines a file format and a set of data structures to represent a *static control part* (SCoP for short), i.e., a program part that can be represented in the polyhedral model. The goal of OpenScop is to provide a common interface to various polyhedral compilation tools in order to simplify their interaction.

Designing a single format for tools that have different purposes (e.g., as different as code generation and data dependence analysis) may sound strange at first. However we could observe that most available polyhedral compilation tools during the last decade were manipulating more or less the same kind of data (polyhedra, affine functions...) and were actually sharing a part of their input (e.g., iteration domains and context concepts are nearly everywhere). We could also observe that those tools may rely on different internal representations, mostly based on one of the major polyhedral libraries (e.g., Omega, Polylib, PPL or isl), and this representation may change over time (e.g., when switching to a more convenient polyhedral library). The OpenScop aim is to provide a stable, unified format that offers a durable guarantee that a tool can use an output or provide an input to another tool without breaking a tool chain because of some internal changes in one element. The other promise of OpenScop is the ability to assemble or replace the basic blocks of a polyhedral compilation framework at no, or at least low, engineering cost.

The policy that drives OpenScop can be summarized by these three rules:

- Embed the *minimum* information to build a complete polyhedral compilation framework in the so-called *core part* (to avoid as much as possible empty or useless information for each tool).
- Provide a *very stable* core part (so users have some guarantee that they will not need to update their tool because of frequent specification evolution),
- Provide a *very flexible* extension part (so it can also be used to test wild new ideas).

Another, more technical, rule may be added:

- Avoid any need for external library or tool to support it (i.e., it's not XML or YAML or anything like that).

The success of OpenScop in meeting its goals totally depends on its acceptance by the tool developers (that have to support it in their tools). To help them, we provide an example implementation: the OpenScop Library. This library (and in particular its API) is not part of the OpenScop specification (which includes only the file format and the set of data structures). It is licensed under the 3-clause BSD license so developers may feel free to use it in their code (either by linking it or copy-pasting its code). Most tools related inside this work (namely, Clan, Candl, Clay and CLooG) already support OpenScop. For more details about the OpenScop specification and library, the reader is welcome to read at its foundation document [CB-Bas11b].

3.5 Conclusion

Optimizing and parallelizing on modern architectures can change the performance by an order of magnitude. As a result, programmers may be reluctant to rely on fragile black-boxes like optimizing compilers to perform this task: it is not acceptable that a slight change in the code or a new compiler version can totally waste the performance. This chapter presented a more open strategy for polyhedral frameworks.

First, we related our efforts and conclusions about the construction of a number of optimizing frameworks, some based on IR raising like WRAP-IT for ORC [CB-BCG$^+$03a, CB-GVB$^+$06], or GRAPHITE for GCC [CB-PCB$^+$06], some based on direct raising like Reservoir Labs Inc. R-Stream [CB-BVL$^+$09, CB-MLV$^+$09, CB-LVM$^+$10, CB-HVB$^+$10] or PoCC [CB-PBB$^+$10, CB-PBB$^+$11]. A conclusion is to promote static control as a programming model, providing the users with clear directions and syntactic feedback to guarantee that his code can be processed. Our main technical contribution to this approach is the high-level raising tool Clan [CB-Bas08] which is currently used in both PoCC and Pluto high-level compilers.

Next, we presented how a user can interact with a polyhedral framework to design the best optimization sequence. The data dependence graph could already point out, e.g., which dependences prevent the parallelization of a given loop. We presented the violated dependence analysis which can be more specific by providing feedback to the user (a programmer or a compiler) about what prevents the application of a given transformation. This information can be used to drive a compiler to perform only the necessary pre-processing to enable that transformation (e.g., array expansion), or a user to design the right sequence of optimizations while preserving the original semantics of the program. On this respect, we built on the violated dependence analysis to design a mapping correction algorithm that corrects a transformation for legality with limited changes. The interaction between a user and the compiler can be done directly from high-level, using the set of directives we introduced. While the directive approach is not new, we presented the first formalism based on the relation abstraction and how it can benefit from the correction algorithm to reach another level of flexibility.

Finally, to help polyhedral framework dissemination and to ease their construction, we presented the first attempt to standardize polyhedral abstractions through OpenScop [CB-Bas11b]. Compilation is a very applicative matter and better tools raise new questions and new research results. Hence, we believe such an effort is essential.

Chapter 4

Optimization Quality: An Iterative Approach

High-level loop optimizations are necessary to achieve good performance over a wide variety of processors. Their performance impact can be significant because they involve in-depth program transformations that aim to sustain a balanced workload over the computational, storage, and communication resources of the target architecture. High-level optimization frameworks have to take into account both the low-level compiler and the complete behavior of the architecture. Unfortunately, low-level compilers and architectures are so complex, obscure or confidential that no model can be precise enough to select an optimal transformation:

- Low-level compilers are multi-layer softwares which transform the input code to various internal representations (e.g., abstract syntax trees, control flow graphs, data dependence graphs, static single assignment form...) and apply multiple passes on each of these representations, until they generate the target object code. As a result a slight change in the input code, or in the compiler options, or in the compiler itself (e.g., when using a new version of a given compiler) may result in a very different output and performance. It is currently infeasible to guarantee *at high-level compile time* which (version of a) compiler and which options will enable the best sequence of low-level optimizations for a given sequence of high-level optimizations.

- The efficient execution of a computation kernel on a modern multi-core architecture requires the synergistic operation of many hardware resources, via the exploitation of thread-level parallelism; the memory hierarchy, including prefetch units, different cache levels, TLBs, memory buses and interconnect; and all available computational units, including SIMD units. Because of the very complex interplay between all these components, it is currently infeasible to guarantee *at compile-time* which set of transformations is leading to maximal performance.

Because of this complexity, optimizing compilers use simplistic performance models that abstract away many of the architecture intricacies and that simply ignore low-level compilers. In addition, they usually rely on inaccurate dependence analysis (even when it is computable) and lack frameworks to express complex interactions of transformation sequences. As a result, they typically uncover only a fraction of the peak performance available on many applications. We propose an iterative framework in the polyhedral model to address these issues.

Feedback-directed and iterative optimizations have become essential technique to keep optimizing compilers competitive with hand-optimized code [BKK$^+$98, KKO00, CST02, TVA05, KHW$^+$05]. Building on operation research, statistical analysis and artificial intelligence, iterative optimization generalizes profile-directed approach to integrate precise feedback from the runtime behavior of the program into optimization algorithms. Whether a single application (for client-side iterative optimization) or a reference benchmark suite (for in-house compiler tuning) are considered, the two main trends are: (1) tuning or specializing an individual heuristic, adapting the profitability or decision model of a given transformation [SAMO03], and (2) tuning or specializing the selection and parameterization of existing (black-box) compiler phases [TVVA03, ABC$^+$06].

Through the many encouraging results that have been published in this area, it has become apparent that achieving better performance with iterative techniques depends on two major challenges:

Search space expressiveness To achieve good performance with iterative techniques that are portable over a variety of architectures, it is essential that transformation search space be expressive enough to let the optimizations target all important architecture components and address all dominant performance anomalies.

Search space traversal It is also important to construct search algorithms (analytical, statistical, empirical) and acceleration heuristics (performance models, machine learning) that efficiently traverse the search space by exploiting its static and dynamic characteristics.

We rely on the polyhedral model to construct and traverse large and expressive search spaces of affine transformations. Those spaces encompass only legal, distinct versions resulting from the restructuring of any static control loop nest. Girbal et al. show that complex sequences of loop transformations are needed to generate efficient code for full-size loop nests on modern architectures [CB-GVB$^+$06]. They also show that such transformation sequences are out of reach of classical loop optimization frameworks, whereas affine scheduling can successfully model them as one single optimization step [Fea92b, DRV00] (see Sections 2.1.3 and 3.3.2), and scales to large loop nests with hundreds of array references. Within this space of complex sequences of loop transformations, our work simultaneously address the two aforementioned challenges.

4.1 Legal Transformation Spaces

Program restructuring is usually broken into sequences of primitive transformations. In the case of loops, typical primitives are the loop *fusion*, loop *tiling*, or loop *interchange* [AK02]. This approach has severe drawbacks. First, it is difficult to decide the completeness of a set of directives and to understand their interactions. Many different sequences lead to the same target code and it is typically impossible to build an exhaustive set of candidate transformed programs in this way. Next, each basic transformation comes with its own application criteria such as legality check or pattern-matching rules. For instance it is unlikely that loop fusion would be applied by a compiler if the bounds of the original loops do not match (while this may be the result of a former transformation in the sequence). Finally, long sequences of transformations contribute to code size explosion, polluting instruction cache and potentially forbidding further compiler optimizations.

Instead of reasoning on transformation sequences, we look for a representation where composition laws have a simple structure, with at least the same expressiveness as classical transformations, but without conversions to or from transformation descriptions based on sequences of primitives. To achieve this goal, we rely on the polyhedral model presented in Chapter 2 where a given mapping may correspond to a large sequence of complex transformations (e.g., those presented in Section 3.3.2). We build *transformation spaces* where each dimension correspond to a mapping component and each integer *point* corresponds to a unique, legal mapping.

4.1.1 One-Dimensional Schedules

In this section, we focus on a sub-class of mappings that can be modeled through *one-dimensional scheduling functions* (a subset of scheduling relations). A one-dimensional schedule, if it exists, expresses the program as a single sequential loop, possibly enclosing one or more parallel loops. A multi-dimensional schedule expresses the program as one or more nested sequential loops, possibly enclosing one or more parallel loops. Affine schedules have been extensively used to design systolic arrays [QD89] and in automatic parallelization programs [Fea92a, DRV00, GFL04]. An affine one-dimensional schedules for a statement S is an affine form on outer loop iterators $\vec{\imath}_S$ and global parameters $\vec{p}$:

$$\theta_S(\vec{p}) = \left\{ \vec{\imath}_S \to t_S^1 \in \mathbb{Z}^{dim(\vec{\imath}_S)} \;\middle|\; \left[\begin{array}{c:c} -1 & T_S \end{array}\right] \begin{pmatrix} t_S^1 \\ \hdashline \vec{\imath}_S \\ \vec{p} \\ 1 \end{pmatrix} = 0 \right\}.$$

Or, using the more classical and compact function notation, since we are working with functions here:

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

where T_S is a constant row matrix called the mapping matrix or the transformation 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 4.1. All these transformations can be represented as a partial order in the space of all instances for all statements, and such orderings may be expressed with one-dimensional scheduling functions [Wol92].

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`

Figure 4.1: Possible Transformations Embedded in a One-Dimensional Schedule

In general, applying a mapping changes the semantics of a program. When a mapping preserves the original program semantics, we say it is *legal*. Previous works on iterative optimization using a polyhedral representation ensure this property by checking, after computing a transformation, whether it is legal or not [LF05, LF06] (non-iterative optimization algorithms use either a similar approach [KAP97], either consider programs simple enough that nearly every transformation is possible [Wol95]). This results in considering huge search spaces, since every illegal or redundant solutions have to be checked, and to a significant computation overhead corresponding to each legality check. Nisbet [Nis98], then Long and Fursin [LF06] experimentally observed that choosing a schedule at random is very likely to lead to an illegal program version. Moreover, the probability of finding a legal mapping decreases exponentially with program size [CB-PBCV07]. Hence such an approach cannot scale and data dependence information must be integrated into the construction of the search space.

Two statement instances are in dependence if they access the same memory location and at least one of these accesses is a write. Maintaining the relative order of such instances is a sufficient condition to preserve the original program semantics [Ber66]. The data dependence analysis as described in Section 2.2.2 gives the exact information on which statement instance pairs have to preserve their relative original execution order. Let S and T be two statements, r_S and r_T be two memory references of those statements respectively, and d be a given loop depth. Each (integral) point of the dependence relation $\delta_{S,r_S \xrightarrow{d} T,r_T}$ represents a value of the iteration vectors $\vec{\imath}_R$ and $\vec{\imath}_T$ where the dependence needs to be satisfied, i.e., where the precedence condition $\theta_S(\vec{\imath}_S) < \theta_T(\vec{\imath}_T)$ must hold. Hence, a mapping is legal if, for any integer point in any dependence relation $\delta_{S,r_S \xrightarrow{d} T,r_T}$,

$$\gamma_{S,T} = \theta_T(\vec{\imath}_T) - \theta_S(\vec{\imath}_S) - 1 \text{ is non-negative everywhere in } \delta_{S,r_S \xrightarrow{d} T,r_T}.$$

Unfortunately, this is not a set of affine constraints since both the components of θ_T (resp. θ_S) and $\vec{\imath}_T$ (resp. $\vec{\imath}_S$) are unknown. Feautrier found the way to translate it to affine constraints using the Farkas lemma [Fea92a]:

Lemma 4.1 *(Affine form of Farkas Lemma [Sch86]) 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:*

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

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

Applying Farkas Lemma to the dependence problem, a mapping is legal if, for any dependence relation $\delta_{S,r_S \xrightarrow{d} T,r_T}$, there exists a set of Farkas multipliers such that:

$$\Gamma_{S,T} = \lambda_0 + \vec{\lambda}^T \left(\Delta^-_{S,r_S \xrightarrow{d} T,r_T} \begin{pmatrix} \vec{\imath}_S \\ \vec{\imath}_T \\ \vec{p} \\ 1 \end{pmatrix} \right) \tag{1}$$

Where $\Delta^-_{S,r_S \xrightarrow{d} T,r_T}$ is the dependence constraint matrix (see Section 2.2.2) of the projection of $\delta_{S,r_S \xrightarrow{d} T,r_T}$ such that $\vec{a}_{S,r_S}$ and $\vec{a}_{T,r_T}$ dimensions have been removed[1]. The Farkas Lemma has to be applied for each individual union part of the dependence relation. The contribution to the constraint system describing the legal space of a given union part of a given dependence relation is built by equating the coefficients in both sides of the equation (1). The intersection of all the constraints gives a global polyhedron, with as

[1]Removing access dimensions is not necessary: it can be done later while removing Farkas multipliers and may not be possible at all if access relations are not functions, we do it here to simplify notations.

many dimensions as there are schedule coefficients for the SCoP (plus Farkas multipliers, to be removed by projection using, e.g., the Fourier-Motzkin projection algorithm [Fea92a]). This polyhedron is the space of legal one-dimensional scheduling functions.

The formulation was first proposed by Feautrier [Fea92a], the only notable difference is we accept coefficients in $\mathbb{Z}$ instead of $\mathbb{N}$. This generalization is simpler and has the property that two points of the space correspond to two different schedules which is an important property for our purpose.

As an example, let us consider the polynomial multiply kernel in Figure 2.1 page 18 and its two dependence relations shown in Section 2.2.2 page 26. The projection of the dependence relations to remove reference dimensions can lead to the following dependence constraint systems ans constraint matrices (note that equalities have been translated to two inequalities for an easier identification of Farkas multipliers):

- $$\Delta^{-}_{S1,1 \xrightarrow{0} S2,1} : \left[\begin{array}{r|rr|r|r} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 2 & -2 \\ \hline 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & -1 \\ \hline\hline 1 & -1 & -1 & 0 & 0 \\ -1 & 1 & 1 & 0 & 0 \end{array}\right] \left(\begin{array}{c} i \\ \hline i' \\ j \\ \hline N \\ \hline 1 \end{array}\right) \geq \vec{0},$$

- $$\Delta^{-}_{S2,1 \xrightarrow{1} S2,1} : \left[\begin{array}{rr|rr|r|r} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -1 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & -1 \\ \hline\hline 1 & 1 & -1 & -1 & 0 & 0 \\ -1 & -1 & 1 & 1 & 0 & 0 \\ \hline\hline -1 & 0 & 1 & 0 & 0 & -1 \end{array}\right] \left(\begin{array}{c} i \\ j \\ \hline i' \\ j' \\ \hline N \\ \hline 1 \end{array}\right) \geq \vec{0}.$$

The two affine schedule prototypes for $S1$ and $S2$ are:

$$\begin{array}{rclcccccc} \theta_{S1}(\vec{\imath}_{S1}) & = & T^{i}_{S1}\, i & + & & & T^{N}_{S1}\, N & + & T^{c}_{S1} \\ \theta_{S2}(\vec{\imath}_{S2}) & = & T^{i}_{S2}\, i & + & T^{j}_{S2}\, j & + & T^{N}_{S2}\, N & + & T^{c}_{S2} \end{array}$$

Using the dependence matrices, we can split the system into as many inequalities as there are independent variables, and equate the coefficients in both sides of the equation (1). The contribution of each dependence is the following:

- $$\delta_{S1,1 \xrightarrow{0} S2,1}(N) : \left\{ \begin{array}{rcrcrcrcrcrcr} i & : & -T^{i}_{S1} & = & & \lambda_{1,1} & - & \lambda_{1,2} & + & \lambda_{1,7} & - & \lambda_{1,8} \\ i' & : & T^{i}_{S2} & = & & \lambda_{1,3} & - & \lambda_{1,4} & - & \lambda_{1,7} & + & \lambda_{1,8} \\ j & : & T^{j}_{S2} & = & & \lambda_{1,5} & - & \lambda_{1,6} & - & \lambda_{1,7} & + & \lambda_{1,8} \\ N & : & T^{N}_{S2} - T^{N}_{S1} & = & & 2\lambda_{1,2} & + & \lambda_{1,4} & + & \lambda_{1,6} & & \\ 1 & : & T^{c}_{S2} - T^{c}_{S1} - 1 & = & - & 2\lambda_{1,2} & - & \lambda_{1,4} & - & \lambda_{1,5} & & \end{array}\right.$$

- $$\delta_{S2,1\xrightarrow{1}S2,1}(N):\left\{\begin{array}{rcrcrcrcrcrcrcr} i & : & -T^{i}_{S2} & = & & \lambda_{2,1} & - & \lambda_{2,2} & + & \lambda_{2,9} & - & \lambda_{2,10} & - & \lambda_{2,11} \\ j & : & -T^{j}_{S2} & = & & \lambda_{2,3} & - & \lambda_{2,4} & + & \lambda_{2,9} & - & \lambda_{2,10} & & \\ i' & : & T^{i}_{S2} & = & & \lambda_{2,5} & - & \lambda_{2,6} & - & \lambda_{2,9} & + & \lambda_{2,10} & + & \lambda_{2,11} \\ j' & : & T^{j}_{S2} & = & & \lambda_{2,7} & - & \lambda_{2,8} & - & \lambda_{2,9} & + & \lambda_{2,10} & & \\ N & : & 0 & = & & \lambda_{2,2} & + & \lambda_{2,4} & + & \lambda_{2,6} & + & \lambda_{2,8} & & \\ 1 & : & -1 & = & - & \lambda_{2,2} & - & \lambda_{2,4} & - & \lambda_{2,6} & - & \lambda_{2,8} & - & \lambda_{2,11} \end{array}\right.$$

where $\lambda_{x,y}$ is the Farkas multiplier attached to the x^{th} contribution and the y^{th} row of the corresponding constraint matrix. The space of legal one-dimensional scheduling functions for the polynomial multiply kernel is the intersection of the two contributions on the mapping dimensions (plus the fact that all Farkas multipliers must be greater or equal to zero). To ensure each integer point of this space have different mapping coordinates, i.e., corresponds to a unique sequence of transformations, we have to eliminate as much unknowns as possible, starting with Farkas multipliers. This can be done through Gaussian elimination, Fourier-Motzkin elimination or polyhedral projection using any polyhedral library. If this space is empty, then no affine one-dimensional schedule is possible for this program.

The legal one-dimensional schedule space is possibly infinite. For instance it is easy to see that if there is no data dependence at all, every value of the schedule coefficients is possible. It is necessary to bound this space in such a way that an exhaustive scan becomes possible. We bound the values of the scheduling coefficients, it is detailed in Section 4.2.1.

Intuitively, to each (integral) point of $\mathcal{T}$ corresponds a different schedule for the original program, i.e., a different program version (or also a valid, distinct transformation sequence). Enumerating points in this polyhedron can be done by polyhedral code generation algorithms, but even though our problem lies into the (simpler) convex case, they may not scale over thirty to forty dimensions [AI91, CB-VBC06] because of the intrinsic combinatorics of characterizing the polyhedron's integral hull. Fortunately, our problem happens to be much simpler than the "static" loop nest generation one: we only need to "dynamically" enumerate every integral point which respects the set of constraints provided by $\mathcal{T}$. We may thus incrementally pick a dimension then *pick an integer* in the polyhedron's projection onto this dimension. This incremental method combines low-complexity projections with the Fourier-Motzkin algorithm and simple enumerations of dense polyhedra.

4.1.2 Multidimensional Schedules

The set of programs which accept a one-dimensional scheduling is quite restricted since their dependences must be simple enough to require at most one external sequential loop. Considering multidimensional schedules with enough dimensions, it is possible to tackle any static-control loop nest (see Section 2.1.3.). The construction of the space of legal multimensional scheduling functions uses a mechanism similar to the mono-dimensional case seen in Section 4.1.1 with the generalization of the precedence constraint to:

$$\theta_S(\vec{\imath}_S) \prec \theta_T(\vec{\imath}_T),$$

where $\prec$ denotes the *lexicographic ordering*.[2]

Using one-dimensional schedules, all dependences have to be satisfied within a single time dimension and the transformation matrix is a row matrix. For multidimensional schedules, the precedence

[2] $(a_1,\ldots,a_n) \prec (b_1,\ldots,b_m)$ iff there exists an integer $1 \le i \le min(n,m)$ s.t. $(a_1,\ldots,a_{i-1}) = (b_1,\ldots,b_{i-1})$ and $a_i < b_i$.

constraints can be entirely satisfied at any dimension d. Dimensions $< d$ can solve the precedence constraint for a subset of the statement instances in dependence relation (possibly empty). We say that the dependence is *weakly solved* for those dimensions: $\theta_T^{1..d-1}(\vec{\imath}_T) - \theta_S^{1..d-1}(\vec{\imath}_S) \succeq 0$. Then at dimension d, the precedence constraint holds for all remaining statement instances in dependence relation. We say that the dependence is *strongly solved* at that dimension: $\theta_S^d(\vec{\imath}_S) - \theta_R^d(\vec{\imath}_R) > 0$. Once a dependence has been strongly solved, no additional constraint is required for legality at dimensions $> d$. Formally, a mapping is legal if (using $\delta_{S\to T}$ to express a dependence, to lighten notations):

$$\begin{aligned} & \forall \delta_{S\to T},\ \exists d \in \{1,\ldots,\min(\dim(\theta_S),\dim(\theta_T))\},\ \gamma^d_{\delta_{S\to T}} = 1 \\ \wedge\quad & \forall j < d,\ \gamma^j_{\delta_{S\to T}} = 0 \\ \wedge\quad & \forall j \le d, \forall \langle \vec{\imath}_S, \vec{\imath}_T\rangle \in \delta_{S\to T},\ \theta_T^j(\vec{\imath}_T) - \theta_S^j(\vec{\imath}_S) \ge \gamma^j_{\delta_{S\to T}} \end{aligned}$$

where the vector $\vec{\gamma}_{\delta_{S\to T}}$ encodes the dependence satisfaction: its entries are set to 0 for the dimensions that weakly satisfy the dependence $\delta_{S\to T}$ and the dimension which strongly satisfies $\delta_{S\to T}$ is set to 1. The proof directly derives from the lexicopositivity of dependence satisfaction [Fea92b].

Selecting the time dimension to strongly solve a dependence leads to a combinatorial problem: each choice can be considered and corresponds to a potentially different legal sub-space. It is also possible to arbitrarily increase the number of time dimensions of the schedule, resulting to an infinite set of scenarios. Hence, it is necessary to set a maximum number of dimensions for each scheduling function. Choosing $d_S + 1$ where d_S is the number of surrounding loops for the statement S is enough to be able to find a scheduling corresponding to the original execution order. Hence, a possible expression of a space of legal multidimensional scheduling functions is a union of polyhedra of legal scheduling functions, one union component for each possible decision.

When the mapping coefficients are bounded and (without any loss of generality) when we assume loop bounds are non-negative, it is possible to avoid the combinatorial selection of the dimensions strongly satisfying dependences and to build a convex legal space. To build such a space, it is necessary to avoid any constraint on scheduling dimensions if the dependence has been strongly satisfied at a previous dimension. We can notice that, as shown by Vasilache [Vas07], if the scheduling coefficients are bounded, it is always possible to find a value K such that for any dimension k:

$$\theta_S^k(\vec{\imath}_T) - \theta_S^k(\vec{\imath}_S) \ge -K\vec{p} - K$$

the computation of K comes from the evaluation of the maximum value of $\theta_S^k(\vec{\imath}_T) - \theta_S^k(\vec{\imath}_S)$, knowing the greatest absolute values of the mapping coefficients and the extremal values of the iterators (loop bounds). We can exploit this property to derive a convex form of semantics-preserving multi-dimensional affine schedules: a mapping is legal if the three following conditions hold:

$$\begin{aligned} &(i) && \forall \delta_{S\to T},\ \forall d \in \{1,\ldots,m_{ST}\},\ \gamma^d_{\delta_{S\to T}} \in \{0,1\} \\ &(ii) && \forall \delta_{S\to T},\ \sum_{d=1}^{m_{ST}} \gamma^d_{\delta_{S\to T}} = 1 \\ &(iii) && \forall \delta_{S\to T},\ \forall d \in \{1,\ldots,m_{ST}\},\ \forall \langle \vec{\imath}_S, \vec{\imath}_T\rangle \in \delta_{S\to T}, \\ & && \theta_T^d(\vec{\imath}_T) - \theta_S^d(\vec{\imath}_S) \ge -\sum_{k=1}^{d-1} \gamma^k_{\delta_{S\to T}}.(K.\vec{p}+K) + \gamma^d_{\delta_{S\to T}} \end{aligned}$$

with $m_{ST} = \min(\dim(\theta_S),\dim(\theta_T))$. Parts (i) and (ii) ensure the $\vec{\gamma}_{\delta_{S\to T}}$ is only made of binary decision variables and that only one (the one corresponding to the dimension where the dependence is fully satisfied). Part (iii) ensures that either the dependence has not been strongly satisfied in a previous dimension, and we have $\theta_T^d(\vec{\imath}_T) - \theta_S^d(\vec{\imath}_S) \geq \gamma_\delta^d$ or it has been strongly satisfied and instead we have $\theta_T^d(\vec{\imath}_T) - \theta_S^d(\vec{\imath}_S) \geq -K\vec{p} - K$. Using this form, we can build the space of all (bounded) legal affine multidimensional schedules with fixed dimensions. In the same way as one-dimensional scheduling, Farkas Lemma is used to linearize part (iii), all the sets of constraints generated for each dependence are intersected together then Farkas multipliers can be eliminated using, e.g. Fourier-Motzkin projection.

Note that mappings corresponding to loop tiling or index-set splitting are not included in the search space. The use of Farkas Lemma requires scheduling *functions*. However, tiling (or strip-mining) introduce dimensions with inequality constraints only, see 3.3.2. In the same way index-set splitting adds inequalities to the relations and new relation unions, which is not supported either. Because of this, our search space does not currently encompass loop tiling or index-set splitting. Recent results by Renganarayanan et al. and Bondhugula et al. are promising directions towards fully integrating loop tiling with affine scheduling algorithms [RKRS07, BBK$^+$08b, BHRS08]. We show an example of combining their techniques with our iterative approach in Section 4.6.

4.2 Practical Search Space

While Section 4.1 presents theoretical search spaces, we make some design choices for those spaces to be (1) bounded in such a way that we prune less interesting parts as shown in Section 4.2.1 and (2) tractable by avoiding combinatorial explosion as detailed in Section 4.2.2.

4.2.1 Mapping Coefficient Bounding

The legal mapping space for a given SCoP as described in section 4.1 is possibly infinite. We bound this space in a systematic way by defining a bounded interval for each mapping coefficient. In this way an exhaustive scan of the bounded search space becomes possible if it is not too large, and efficient traversal techniques can be designed. Bounding the space will remove some possible program transformations. We have to ensure we remove only the least interesting solutions for performance.

We can distinguish two families of coefficients in the schedule expressions, (1) input coefficients (i.e. iterator coefficients), (2) parameter and constant coefficients. Each family provides a specific contribution to the global program transformation [CB-BCG$^+$03a]. The iterator coefficients impact loop structure and bounds (*skewing*-like transformations for instance) while parameters and constant impact loop ordering and statement ordering within a loop (*shifting*-like transformations for instance). While the coefficient values of parameters and constant do not impact the execution time, large iterator coefficients result in excessive control overhead due to complex loop bounds and modulo operations [CB-Bas04a]. This overhead may wipe out the speedup expected from the transformation/new schedule. Hence we should bound the values of the iterator coefficients with small values.

We checked empirically that the bounding interval $[-1, 1]$ is wide enough in general [CB-PBCV07]. Although it eliminates some schedules from the space (e.g., non-unit skewing), these bounds are compatible with the expression of arbitrary compositions of loop fusion, distribution, interchange, code motion; in the worst case, it translates into additional time dimensions for multidimensional schedules. Overall, This solution is an good trade-off between the expressiveness of the mapping, the scalability of the compiler and the quality of the generated code.

The coefficients of the parameters and the constant (in other words, the coefficient of 1 of the affine expressions) have also to be bounded to avoid an infinite search space. The difference between the two extremal values of the interval should be greater than the number of statements to ensure that at least every ordering of the statements within or outside loops is possible. Greater intervals would offer more possibilities, for instance to achieve more peeling transformations, but a larger flexibility is rarely useful in practice.

4.2.2 Search Space Construction

Once the mapping coefficients have been bounded, the search space is ready to be scanned exhaustively or traversed using heuristics in the one-dimensional case. In the multidimensional case, we face a combinatorial problems because too many polytopes have to be considered[3]. For instance, the Ring-Roberts filter shown in Figure 3.6 has 12 dependence polyhedra, hence a huge number of possible strongly/weakly solved dependence scenarios.

Feautrier found a systematic solution to the explosion of the number of polyhedra: he considers a space of legal schedules leading to maximum fine-grain parallelism [Fea92b, Viv02]. To achieve this, a greedy algorithm maximizes the number of dependences solved for a given dimension. While this solution is interesting because it reduces the number of dimensions and may exhibit inner parallelism, it is not practical enough for several reasons. First, it requires the resolution of a system of linear inequalities involving every schedule coefficient *plus* a decision variable per dependence [Fea92b]. This makes the problem intractable for kernels with a large set of dependences. Moreover, minimizing the number of dimensions often translates into large schedule coefficients; these generally leads to algorithmic complexity and both significant loop bounds and control flow overhead after generation of the target imperative code [Kel96].

We suggest a simple variation to overcome these two issues. The algorithm in Figure 4.2 sketches our search space construction for a given static control part. This heuristic outputs for each schedule dimension d a space $\mathcal{L}_d$ of legal solutions.

The algorithm terminates if the intervals for bounding the mapping coefficients are large enough to guarantee that a solution exists. The scheduling of the original program presented by Feautrier (see Section 2.1.3) demonstrates that the interval $[0,1]$ is enough for all coefficients except the constant coefficient which must be n if the number of statements is n. In this case, the termination proof uses the same argument as Feautrier's multidimensional scheduling algorithm [Viv02]: at least one dependence can be strongly solved per time dimension d.

This construction algorithm differs from Feautrier's algorithm as it does not guarantee a maximal number of dependences solved per dimension. Therefore, it may not minimize the number of dimensions of the schedule[4]. This is not an issue since our purpose is not to expose the maximum parallelism. However, this algorithm is efficient and only needs one polyhedron emptiness test per dependence (Over $\mathcal{L}_d$ which contains exactly one variable per schedule coefficient), and the elimination of Farkas multipliers used to enforce the precedence constraint on schedule coefficients is performed dependence per dependence (i.e., on very small systems).

[3]We do not use the convex space expressed in Section 4.1.2 because we will build on the dimension-per-dimension structure to design traversal heuristics and we will benefit from extracted fine-grain parallelism à la Feautrier [Fea92b] to expose vectorizable inner loops to the low-level compiler.

[4]It is equivalent to Feautrier's algorithm when the mapping coefficients are not bounded however, because the maximal set of dependences which can be strongly solved for a given dimension is unique [Fea92b].

LEGAL MAPPING SPACE CONSTRUCTION ALGORITHM

```
Input: a SCoP.
Output: a space $\mathcal{L}_d$ of legal solutions for each mapping dimension $d$.

1. Compute the set $G$ of dependence relations for the SCoP, see Section 2.2.2
2. $d \leftarrow 1$
3. while $G \neq \emptyset$ do
   (a) Initialize $\mathcal{L}_d$ (the space of legal schedules for time dimension $d$) to the
       space of all mapping coefficients such that they belong to their respective
       bounding interval
   (b) for each dependence $\delta_{S\to T} \in G$
       • Compute $\mathcal{W}_{\delta_{S\to T}}$, the space of legal schedules weakly satisfying
         $\delta_{S\to T}$ at dimension $d$, i.e., such that:
         $\forall \langle \vec{\imath}_S, \vec{\imath}_T \rangle \in \delta_{S\to T}, \theta_T^d(\vec{\imath}_T) - \theta_S^d(\vec{\imath}_S) \geq 0$
       • $\mathcal{L}_d \leftarrow \mathcal{L}_d \cap \mathcal{W}_{\delta_{S\to T}}$
   (c) for each dependence $\delta_{S\to T} \in G$
       • Compute $\mathcal{S}_{\delta_{S\to T}}$, the space of legal schedules strongly satisfying $\delta_{S\to T}$
         at dimension $d$, i.e., such that:
         $\forall \langle \vec{\imath}_S, \vec{\imath}_T \rangle \in \delta_{S\to T}, \theta_T^d(\vec{\imath}_T) - \theta_S^d(\vec{\imath}_S) \geq 1$
       • if $\mathcal{L}_d \cap \mathcal{S}_{\delta_{S\to T}} \neq \emptyset$ then
         – $\mathcal{L}_d \leftarrow \mathcal{L}_d \cap \mathcal{S}_{\delta_{S\to T}}$
         – $G \leftarrow G - \delta_{S\to T}$
   (d) $d \leftarrow d + 1$
```

Figure 4.2: Practical Multidimensional Legal Space Construction Algorithm

So far, we have not defined the order in which dependences are considered when checking against strong satisfaction. This order can have a significant impact on the constructed space. A long term approach would be to consider this order as part of the search space, but this is not currently practical because of the combinatorial explosion. Instead, we use two analytical criteria to order the dependences.

First of all, each dependence is assigned a priority, depending on the memory traffic generated by the pair of statements in dependence. We use a simplified version of the model by Bastoul and Feautrier [CB-Bas03]: for each array A and dimension d, we approximate the traffic as $m_d^{r_A}$, where m_d is the size of the d^{th} dimension of the array, and r_A is the rank of the concatenation of the variable parts of the subscript matrices of all references to dimension d of array A in the statement. Thus, the generated traffic evaluation for a given statement is a multivariate polynomial in the parametric sizes of all arrays. We use profiling to instantiate these size parameters. Intuitively, maximizing the depth where a dependence is strongly solved maximizes reuse in inner loops and minimizes the memory traffic in outer loops. Therefore, we start with dependences involved in the statements with minimum traffic. Our second criterion is based on *dependence interference*; it is used in case of non-discriminating priorities resulting from the first criterion. Two dependences interfere if it is impossible to build a one-dimensional schedule strongly satisfying these two dependences. We first try to solve dependences interfering with the lowest number of other dependences, maximizing our chance to strongly solve more dependences within the current time dimension.

4.3 Legal Space Traversal

The algorithm presented in Section 4.2.2 builds one polytope per dimension of the mapping. In the one-dimensional case, picking a point inside the unique polytope fully describes one legal mapping. In the multidimensional case, we need to pick one point in every polytope to describe one multidimensional mapping. Each solution corresponds to one program version: the generated imperative codes will be distinct if the mapping matrices are distinct. To find the most interesting points in small enough search spaces, an exhaustive search (with respect to the mapping coefficient intervals) is possible, a scalable technique is presented in Section 4.3.1. For large spaces, a heuristic search is necessary as detailed in Section 4.3.2.

4.3.1 Exhaustive Search

Achieving an exhaustive search in the legal space is possible by using polyhedral code generation algorithms [AI91, KPR95, QRW00, CB-Bas04a] (see Section 2.3): we could generate a code to scan the integer points of the legal space and execute it to perform the exhaustive scan. However, even though our problem lies into the (simpler) convex case, they may not scale over thirty to forty dimensions [AI91, CB-VBC06] because of the intrinsic combinatorics of characterizing the polyhedron's integral hull. Fortunately, the exhaustive scan problem is much simpler than the "static" loop nest generation one: we only need to "dynamically" enumerate every integral point which respects the set of constraints defining the legal space. We may thus incrementally pick a dimension then *pick an integer* in the polyhedron's projection onto this dimension. This incremental method combines low-complexity projections with the Fourier-Motzkin algorithm and simple enumerations of dense polyhedra.

To build the d^{th} row of the mapping matrix T, we scan the legal polytope $\mathcal{L}_d$, by successively instantiating values for each coefficient in a predefined order[5]. Fourier-Motzkin elimination (a.k.a. projection) [Sch86] provides a representation of the affine constraints of a polytope suitable for its dynamic traversal. Computing the projection of all variables of a polytope $\mathcal{L}_d$ results in a set of constraints defining the same polytope, but where it is guaranteed that for a point $v \in \mathcal{L}_d$, the value of the k^{th} coordinate v_k *only depends* on $v_1, \dots, v_{k-1}$, that is the affine inequalities involve only $v_1, \dots, v_k$. Thus, the sequential order to build coefficients is simply the reverse order of the Fourier-Motzkin elimination steps. This scheme guarantees that provided a value in the projection of $v_1, \dots, v_{k-1}$, a value exists for v_k, for all k.[6]. To achieve scalability, Pouchet shows that a modified and redundancy-aware version of the Fourier-Motzkin algorithm is necessary [Pou10].

4.3.2 Heuristic Traversal

For SCoPs with more than 3 or 4 statements, the space construction algorithm leads to very large search spaces, challenging an exhaustive traversal procedure. It is possible to focus the search on some co-

[5]The order has no impact on the completeness of the traversal.

[6]The case of holes in $\mathbb{Z}$-polyhedra is handled through a schedule completion algorithm described in Section 4.3.2.

efficients of the schedule with maximal impact on performance, postponing the instantiation of a full schedule in a second heuristic step. We show that such a two-step procedure can be designed without breaking the fundamental legality property of the search space. This approach will be used extensively to simplify the optimization problem.

Schedule Completion Algorithm

Relying on the projection mechanism used for the exhaustive scan in Section 4.3.1, it is always possible to complete or to correct any coordinate set, by slightly modifying them, so it corresponds to a point in the legal space $\mathcal{L}_d$. Our completion algorithm is sketched below. Given a point v in a n-dimensional space with some undefined coordinates:

1. set all undefined coordinates to 0;
2. for each $k \in [1,n]$:
   (a) compute the lower bound lb and the upper bound ub of v_k in $\mathcal{L}_d$, provided the coordinate values for $v_1 \dots v_{k-1}$,
   (b) if $v_k \notin [lb, ub]$, then $v_k = lb$ if $v_k < lb$ or $v_k = ub$ if $v_k > ub$.[7]

Therefore it is possible to partially build a schedule prefix, e.g., values for the input (iterator) coefficients, leaving the other coefficients unspecified. Then, applying this completion algorithm will result in finding the minimal amount of complementary transformations to make the transformation lie in the computed legal space. Three fundamental properties are embedded in this completion algorithm:

1. if $v_1, \dots, v_k$ is a prefix of a legal point v in the legal space, a completion is always found;
2. this completion will only update $v_{k+1}, \dots, v_{d_{\max}}$, if needed;
3. when $v_1, \dots, v_k$ are the $\vec{\imath}$ coefficients, the heuristic looks for the smallest absolute value for the $\vec{p}$ and constant coefficients.

Picking coefficients as close as possible to 0 has several advantages in general: smaller coefficients tend to simplify code generation, improve locality, reduce latency, and increase the size of basic blocks in inner loops.

Decoupling Heuristic

Our approach to find the best points inside large legal space is called the *decoupling heuristic*. It leverages the completion algorithm and derives from the observation of the performance distribution, where density patterns hinted that not all schedule coefficients have a significant impact on performance [CB-PBCV07, CB-PBCC08b]. The most performance impacting transformations (interchange, skewing, reversal) are embedded in the input coefficients of the mapping (the original iterator coefficients, $\vec{\imath}$); followed by coefficients usually involved in fusion and distribution the $\vec{p}$ coefficients; and finally the less impacting constant coefficients, representing loop shifting and peeling [CB-PBCC08b]. The completion algorithm

[7]$\mathbb{Z}$-holes are detected by checking if $lb > ub$.

finds complementary transformations in order of least to most impacting, as it will not alter any vector prefix if a legal vector suffix exists in the space.

The principle of the decoupling heuristic for one-dimensional schedules is (1) to enumerate different values for the $\vec{\imath}$ coefficients, (2) to instantiate full schedules with the completion algorithm, and (3) to select the best completed schedules and further enumerate the different coefficients for the $\vec{p}$ part. A direct extension to the multidimensional case exhibits two major drawbacks. First, the relative performance impact of the different schedule dimensions must be quantified. Second, an exhaustive enumeration of $\vec{\imath}$ coefficients for all dimensions is out of reach, as the number of points exponentially increases with the number of dimensions [CB-PBCC08a].

To extend the decoupling approach to multidimensional schedules, we need to integrate interactions between dimensions. For instance, to distribute the outer loop of a nest (which can improve locality and vectorization [AK02]), one can operate on the $\vec{p}$ and constant parts of the schedule for the first dimension (a parametric shift). On the other hand, altering the $\vec{\imath}$ parts will lead to the most significant changes in the loop controls. Conversely, it is likely that the best performing transformations will share similar $\vec{\imath}$ coefficients in their schedules.

Furthermore, the first dimension is highly constrained in general, since all dependences need to be (weakly or strongly) considered. Conversely, the last dimension is the less constrained and often carries only very few dependences.[8]

We conducted an extensive experiment showing that the first time dimension of the schedule, θ^1 is a major discriminant of the overall performance distribution [CB-PBCC08b]. Therefore, the heuristic starts with an exploration of the different legal values for the coefficients of θ^1, and the completion algorithm is called to compute the remaining rows of θ. Furthermore, this exploration is limited to the subspace associated with the $\vec{\imath}$ coefficients of θ^1 (and the remaining coefficients of θ^1 are also computed with the completion algorithm), except if this subspace is smaller than a given constant L_1 ($L_1 = 50$ in our experiments). L_1 drives the exhaustiveness of the procedure: the larger the degree of freedom, the slower the convergence. By limiting the search to the $\vec{\imath}$ class we target only the most performance impacting subspaces.

To enumerate points in the polytopes, we incrementally pick a dimension then pick an integer in the polyhedron's projection onto this dimension. Note that the full projection is computed once and for all by the Fourier-Motzkin algorithm before traversal. Technically, to enumerate integer points of the subspace composed of the first m columns of $\mathcal{L}_d$, we define the following recursive procedure to build a point v:

EXPLORE $(v, k, \mathcal{L}_d)$:

1. compute the lower bound lb and the upper bound ub of v_k in $\mathcal{L}_d$, given the coordinate values for $v_1 \ldots v_{k-1}$;
2. for each $x \in [lb, ub]$:
   (a) set $v_k = x$,
   (b) if $k < m$ call EXPLORE $(v, k+1, \mathcal{L}_d)$ else output v.

The enumeration is initialized with a call to EXPLORE $(v, 1, \mathcal{L}_d)$. The completion algorithm is then called on each point v generated, to compute a legal suffix for v (corresponding to the columns $[m+1, n]$ of $\mathcal{L}_d$), finally instantiating a legal point of full dimensionality.

[8]This is typically the case when the final dimension is required to order the statements within an innermost loop.

Then, the heuristic selects the $x\%$ best values for θ^1 (x was set to 5% in our experiments). The heuristic then proceeds with the exploration of values for coefficients of θ^2 with the selected values of θ^1, and recursively until the last but one dimension of the schedule. The last dimension, corresponding to the innermost nesting depth in the generated code, is not traversed but completed with a single value: exploring it would yield a huge number of iterations, with limited impact on the generated code, and negligible impact on performance. Eventually, the number of schedules visited is bounded with a static limit (1000 evaluations in our experiments).

4.3.3 Evolutionary Traversal

Genetic algorithms [Gol89] is a well known approach to program optimization. Our contribution is to reconcile fine-grain control of a transformation heuristics (as opposed to optimization flag or pass selection [TVA05, ABC$^+$06]) with the guaranteed legality of the transformed program (as opposed to filtering approaches [Nis98, LO04, LF06] or always-correct transformations [SAMO03, KHW$^+$05].

The challenge of relying on genetic algorithms with instance-wise mappings is to preserve legality during mutation. We will build on the properties of our legal space and our previous algorithms to design legality-preserving mutation and crossover operators. The operators of our evolutionary approach are the following:

Initialization	We build an initial individual applying our completion algorithm (see Section 4.3.2) onto an undefined mapping, and a initial population of 30 to 100 individuals from aggressive mutations of the initial individual.
Mutation	First, we compute the distribution of probability to alter every coefficient, with respect to their performance impact (see Section 4.5.5). Then according to this distribution, we randomly pick a coefficient to be altered, we replace its value with a randomly selected value within the legal bounds for that coefficient and we use the completion algorithm to correct the following coefficients for legality.
Crossover	Starting with two individuals, we may apply a *row crossover* by randomly picking mapping rows of either individuals for each mapping dimensions to build a new individual. This preserves legality since there is no dependence between rows. Or we may apply a *column crossover*: if we noticed that there exist some sets of coefficients not connected together in any constraint defining the legal space, then it is safe to randomly pick the corresponding columns of either individuals to build a new individual, because they correspond to independent sets of coefficients.
Selection	We keep the best half of the current population and use mutation and crossover operators to generate the next generation.

4.4 Overview of the Experimental Platform

The implementation of the LEgal Transformation SpacE Exploration tool is called LeTSeE and its workflow is sketched in Figure 4.3. It is composed of several software components, the reader can find some information about each of them in Appendix A. LeTSeE is a source-to-source framework, where the input code is first translated to a polyhedral representation decorated with the data dependence graph (using Clan and Candl), next the legal mapping search space is built and sent to the traversal algorithm, e.g., using the techniques presented in Section 4.2.2 and 4.3. For every "point" chosen by the traversal algorithm, a target code is generated by CLooG, then compiled using the target low-level compiler and run onto the target architecture. Execution information (performance, cache misses etc.) reported using the target architecture hardware counters are communicated to the traversal algorithm to select the next "point" using either the decoupling heuristic shown in Section 4.3.2 or the evolutionary traversal discussed in Section 4.3.3. When the traversal algorithm converged, the best performing source code[9] is sent as output.

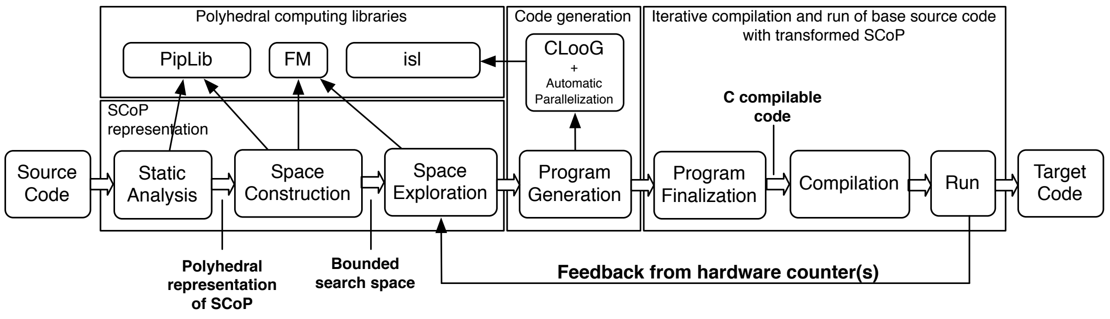


Figure 4.3: LeTSeE's Workflow

4.5 Lessons Learned from Experiments

The experimental studies the iterative compilation approach presented in this chapter and its variations and developments have been extensively published [CB-PBCV07, CB-PBCC08b, CB-PBCC08a, CB-PBB$^+$10, CB-PBB$^+$11] and discussed in Pouchet's PhD. dissertation [Pou10]. Instead of reproducing them here, we will recall in this section the main lessons learned from (or confirmed by) those experiments. Some lessons will be illustrated using benchmark studies. Those benchmarks are detailed by Pouchet [Pou10] and most of them are freely available in the PolyBench Benchmark suite (See Section A.10 for details).

4.5.1 Search Space Construction Cannot Avoid Legality

The first attempt to build a high-level iterative framework by Long et al. postponed the mapping legality question after its construction, by filtering out any illegal mapping using a legality check [LF05]. Using the legal space, we can evaluate the proportion of legal points in the complete space of bounded mapping coefficients. The results are detailed in Figure 4.4 on benchmarks accepting a one-dimensional mapping, and being simple enough to compute the number of legal points and to perform and exhaustive search. The figure presents the number of dependences for each kernel which is a metric for the complexity of the program, the interval used for the various mapping coefficients, the search space computation time on a Pentium 4 Xeon 3.2GHz, the number of possible mappings and the number of legal mappings.

This study shows that, even for simple programs with simple enough dependences to enable a legal one-dimensional mapping, the number of possible mappings is several orders of magnitude larger than

[9]With respect to the assumed input. While SCoPs are not sensitive to input values, they may be sensitive to the input size. In this work, we rely on profiling to evaluate parameter sizes. Versionning is possible as well but is not investigated here.

the number of legal mappings. Furthermore, the proportion of legal mappings decreases as the number of dependences grows. Hence, encoding the legality in the search space is mandatory for a practical implementation of a high-level iteration framework.

Benchmark	#Dependences	$\vec{\imath}$-Bounds	$\vec{p}$-Bounds	c-Bounds	Time	#Mappings	#Legal
`h264`	15	$-1,1$	$-1,1$	$0,4$	0.011	7.5×10^5	360
`fir`	12	$-1,1$	$-1,1$	$-1,1$	0.004	4.7×10^6	432
`fft`	36	$-2,2$	$-2,2$	$0,6$	0.079	5.8×10^{25}	804
`lu`	14	$0,1$	$0,1$	$0,1$	0.005	3.2×10^4	1280
`gauss`	18	$-1,1$	$-1,1$	$-1,1$	0.021	5.9×10^4	506
`crout`	26	$-3,3$	$-3,3$	$-3,3$	0.027	2.3×10^{14}	798
`matmult`	7	$-1,1$	$-1,1$	$-1,1$	0.003	1.9×10^4	912
`MVT`	10	$-1,1$	$-1,1$	$-1,1$	0.001	4.7×10^6	16641
`locality`	2	$-1,1$	$-1,1$	$-1,1$	0.001	5.9×10^4	6561

Figure 4.4: Comparison of Complete and Legal Spaces

4.5.2 Exhaustive Search Is Out of Reach In General

Working in the bounded legal space ensures to generate only possible mappings. However, an exhaustive enumeration of the mapping coefficients is out of reach in general, even when considering the input dimensions only because the number of points exponentially increases with the number of dimensions. Figure 4.5 illustrates this assertion by summarizing the size of the legal polytopes for different benchmarks, for all schedule dimensions. We consider 10 SCoPs requiring multi-dimensional mappings extracted from classical benchmarks. This table provides for each kernel the number of dependences, the number of dimensions of the mapping, the number of legal points for each mapping dimensions, and the total number of legal mappings. A value $> 10^n$ is a conservative lower bound when it was not possible to compute the exact space size in a reasonable amount of time.

While those kernels have small to medium sizes (from 3 statements for `edge` to 17 for `radar`, the number of legal mappings is not tractable for any of them. Hence, approximated techniques are necessary to find the most interesting mappings in a reasonable amount of time.

Benchmark	#Dep.	#Dim.	θ^1	θ^2	θ^3	θ^4	Total
`compress-dct`	56	3	20	136	10857025	n/a	2.9×10^{10}
`edge`	30	4	27	54	90534	43046721	5.6×10^{15}
`iir`	66	3	18	6984	$>10^{15}$	n/a	$>10^{19}$
`fir2`	36	2	18	52953	n/a	n/a	9.5×10^7
`lmsfir`	112	2	27	10534223	n/a	n/a	2.8×10^8
`latnrm`	75	3	9	1896502	$>10^{15}$	n/a	$>10^{22}$
`lpc`	85	2	63594	$>10^{20}$	n/a	n/a	$>10^{25}$
`ludcmp`	187	3	36	$>10^{20}$	$>10^{25}$	n/a	$>10^{46}$
`radar`	153	3	400	$>10^{20}$	$>10^{25}$	n/a	$>10^{48}$

Figure 4.5: Search space statistics

4.5.3 Model-Based Performance Models Are Not Enough

Exhaustive search on simple problems made it possible to find the best mapping for a given program and a given architecture and a given low-level compiler and its set of options. The form of the best transformed programs typically appears to be quite complex. Most of the time, it was not possible to easily understand which part of the transformation sequence was responsible for the speedup. We also noticed that optimization algorithms based on formal representations were sometimes far away from the optimal solution. A very simple but striking example is shown in Figure 4.6(A).

The simple, supposedly optimal locality transformation in our class suggests a schedule of (i) for $S1$ and $(j+N)$ for $S2$ [CB-Bas03], which results in maximizing the reuse of the array a (see Figure 4.6(B)). The very best schedules were in fact $(i-j)$ and $(i+j-N+1)$ (the code generated by our framework is given in Figure 4.6(C)). While the supposed optimal schedules generate a speedup of 147% with $N = 100$ and $M = 500000$ using GCC 3.4 on an Intel Xeon 3.2GHz, the very best schedules generate a speedup of 398% (with a similar number of L1 and L2 cache-misses but a heavily reduced data TLB misses). The ever growing complexity of architectures and compilers is not likely to improve the situation and high-level iterative, feedback-directed techniques will probably become more and more necessary for optimizing compilers.

```
/* (A) Original code: */
for (i = 0; i <= M; i++) {
  a[i] = i;
  for (j = 0; j <= N; j++) {
    b[j] = (b[j] - a[i]) / 2;
  }
}

/* (B) Chunked code: */
for (t = 0; t <= M; t++) {
  a[t] = t;
}
for (t = M; t <= M+N; t++) {
  for (i = 0; i <= M; i++) {
    b[t-M] = (b[t-M] - a[i]) / 2;
  }
}

/* (C) Best transformation: */
a[0] = 0;
for (t = -M+1; t <= 0; t++) {
  for (i = max(0, t+M-N-1); i <= t+M-1; i++) {
    b[t-i+M-1] = (b[t-i+M-1] - a[i]) / 2;
  }
  a[t+M] = t+M;
}
for (t = 1; t <= N+1; t++) {
  for (i = max(t+M-N-1, 0); i <= M; i++) {
    b[t-i+M-1] = (b[t-i+M-1] - a[i]) / 2;
  }
}
```

Figure 4.6: Intricacy of the transformed code

4.5.4 The Compiler Is a Part of the Experimental Platform

Our iterative optimization scheme is independent from the compiler and may be seen as a higher level of classical iterative compilation. A given program transformation may better exploit a feature of a given processor but it may also 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 optimizing heuristics are very fragile, i.e. a slight difference in the source code may enable or forbid a given optimization phase. Using PathScale EKOPath, we observed that many optimization phases are enabled or disabled depending on the version generated from our exploration tool [CB-PBCV07].

To illustrate this, we performed an exhaustive scan of the legal space to find the best mapping for the matrix multiplication kernel, which is a typical target of aggressive optimizations of production compilers, for various compilers with their most aggressive set of optimization options. 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). The results are shown in Figure 4.7. Besides the fact that, contrary to most other works, good speedups are achieved without tiling, we can observe that the best mappings are quite different, which emphasises the need for a compiler-dedicated transformation to achieve the best possible performance. This is also true depending on the compiler options [CB-PBCV07], advocating for a coupling of low-level iterative compilation techniques to find the best set of compiler options and our high-level techniques.

Compiler	Option	Original	Best	Schedule	Speedup
GCC 3.4.2	-O3	519	163	$\theta_{S1}(\vec{\imath}_{S1}) = -1$ $\theta_{S2}(\vec{\imath}_{S2}) = k+1$	318.4%
GCC 4.1.1	-O3	515	207	$\theta_{S1}(\vec{\imath}_{S1}) = -i-j+n-1$ $\theta_{S2}(\vec{\imath}_{S2}) = k+n$	248.7%
ICC 9.0.1	-fast	465	72	$\theta_{S1}(\vec{\imath}_{S1}) = -i+n$ $\theta_{S2}(\vec{\imath}_{S2}) = k+1$	645.4%
PathCC 2.5	-Ofast	228	79	$\theta_{S1}(\vec{\imath}_{S1}) = j-n-1$ $\theta_{S2}(\vec{\imath}_{S2}) = k$	308.1%

Figure 4.7: Best Mappings for the `matmult` Kernel (matrices are 250×250 doubles)

4.5.5 All Mapping Coefficients Are Not Equal With Respect to Performance

We conducted variance studies to capture the relative impact of schedule coefficients. We observe that the impact on performance distribution of the $\vec{p}$ coefficients is lower than the $\vec{\imath}$ ones, while the impact of the c coefficients is almost negligible. Overall, all the conducted experiments confirm the hypothesis of the Decoupling Heuristic presented in Section 4.3.2. As a result the Decoupling Heuristic performs very well at finding a near-optimal mapping while scanning only a very restricted set of mappings as shown in Figure 4.8. It shows the relative percentage of the best speedup achieved as a function of the number of iterative runs. The decoupling heuristic (the *DH* plot) yields fast convergence, bringing to light the correlation between the speedup and the $\vec{\imath}$-coefficients. On these tested examples, one may achieve over 98% of the maximum speedup within less than 20 iterations.

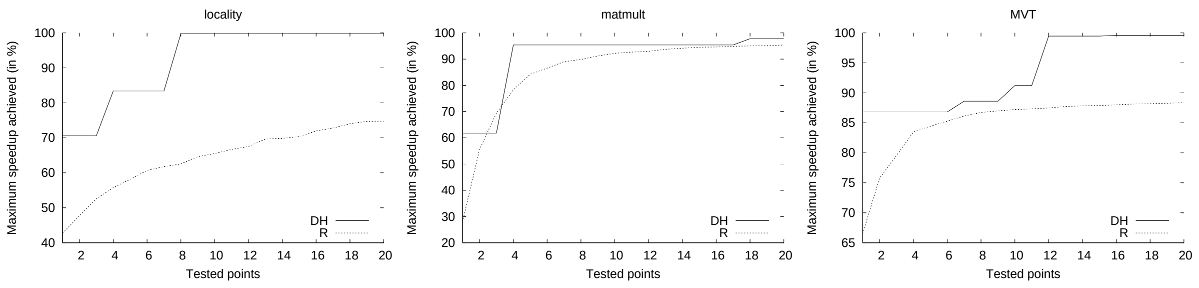


Figure 4.8: Comparison between the random (R) and the decoupling heuristics (DH)

4.5.6 Performance Distribution Is Not Random

When an exhaustive scan of the legal mappings is possible, we can observe the complete performance distribution. Figure 4.9 shows this distribution for `matmult` and `locality` which are compiled with `GCC 4.1.1 -O2` and for `crout` compiled with `ICC -fast` (left hand side) and `GCC 4.1.1 -O3` (right hand side). Each graph represents the computation time of every point in the search space as a function of its number in the scanning order. The horizontal line shows the performance of the original program: every point below this line corresponds to a more efficient program version.

Although the scanning order seems an irrational criteria for such representation, it shows that the performance distribution is not random. It is not an absurd ordering though: the scanning procedure could be seen as a very deep loop nest were the outer loop iterates on values of the first iterator coefficient of the first statement and the inner loop iterates on values of the constant coefficient of the last statement.

From these observations, we conclude that:

- in most cases, contiguous regions of similar performance can be identified;
- several transformations may be close to the best performance, but the probability of finding them at random can be very low (e.g., on `locality`);
- the performance distribution depends on the compiler;
- for some benchmarks (e.g., on `matmult`), strong correlations do exist but are not easily observable without reordering the index space of the transformations (the X axis on the performance distribution figures).

Understanding performance regularities may help to find "hot" regions in the search space, thus avoiding useless runs in low-interest regions and diminishing-return searches among nearly optimal solutions.

4.5.7 Bad Solutions Are Close to Good Ones

An interesting property of the search space is, if a given mapping coefficient can significantly degrade the performance when iterating over it, then this mapping coefficient is more likely to significantly improve the performance as well. We illustrate this property with the performance distribution of the

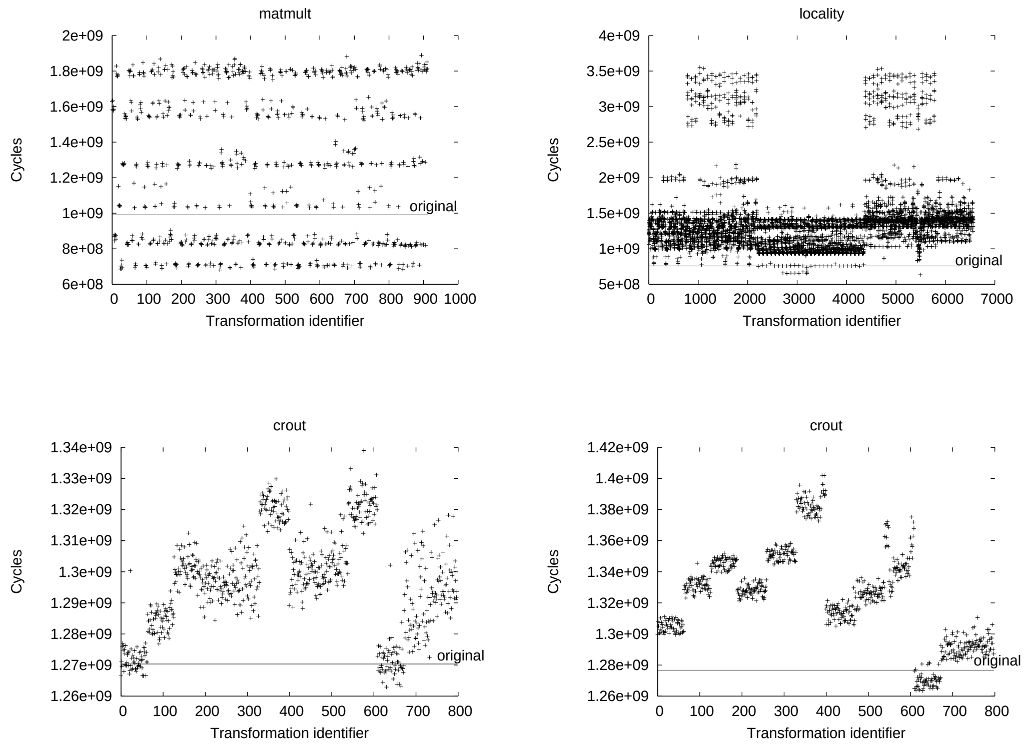


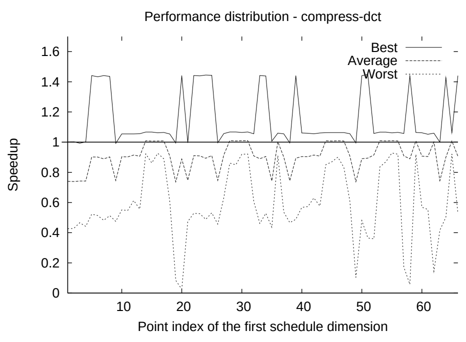
Figure 4.9: Performance distributions for `matmult` and `locality` with GCC -O3, and for `crout` with ICC -fast and GCC -O3 on Intel Xeon 3.2GHz

`compress-dct` benchmark, on AMD Athlon on Figure 4.10(A). We exhaustively enumerate and evaluate all 66 points with a distinct value for the $\vec{\imath}+\vec{p}$ coefficients of the first schedule dimension, combined with all points with a distinct $\vec{\imath}$ value for the second one; a total of 1.29×10^6 schedules are evaluated. For each distinct value of the first schedule dimension (plotted in the horizontal axis), we report the performance of the Best schedule, the Worst one, and the Average for all tested values of the second schedule dimension. We can note the symmetry of performance peaks. Hence significant performance degradation can be used to find the coefficients that matter as well[10].

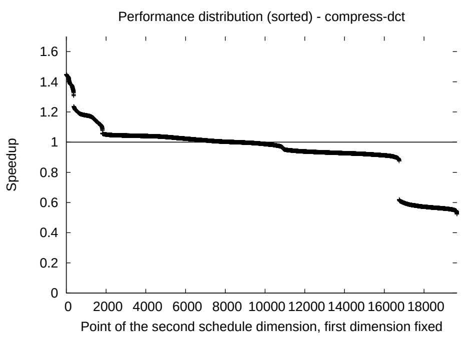
4.5.8 Random Search Is Not Likely to Provide Good Results

Our experiments show that the density of very good mappings is quite low in general. As an example, Figure 4.10(B) shows the performance for all the 19683 possible points of θ^2 once the best possible θ^1 has been selected for the `compress-dct` benchmark. Mappings are sorted from the best to the worse performance. There is an extremely low proportion of good mappings: only 0.14% of points achieve at least 80% of the maximal performance improvement, while only 0.02% achieve 95% and more. Conversely, 61.11% degrade performance of the original code, while in total 10.88% degrade the performance by a factor 2. We can also see in Figure 4.9 that the number of point below the horizontal line showing the

[10]This property is also an illustration of the fragility of the best transformations: a slight change on an important coefficient can lead to major performance drops.



(A) All θ^1 and representatives of θ^2



(B) Best θ^1 fixed and all θ^2

Figure 4.10: Performance Distribution of `compress-dct`, AMD Athlon X64, GCC 4.1.1 options `-O3 -msse2 -ftree-vectorize`

original performance (lower is better) is small compared to the total number of points (with the exception of `matmult`. As a result random search is not a good policy to find the best points. Figure 4.8 shows some examples where the decoupling heuristic outperforms a random strategy, including when the good mapping density is relatively high as with `matmult`.

4.5.9 Benefits Are Significant

We tested our decoupling heuristic and our evolutionary algorithm to traverse the legal mapping space within the LeTSeE framework described in Section 4.4 on a set of benchmarks representative of static control kernels (8 codes from the UTDSP benchmark suite [Lee98], plus two larger programs including an industrial radar detection code), on several architectures. The genetic algorithm was able to improve the performance by **37.6%** in average on an AMD Athlon X64 3700+ with GCC 4.1.1 (25.1% for the decoupling heuristic), **17.9%** in average on an AMD Au1500 500MHz (embedded SoC) with GCC 3.2.1 (11.4% for the decoupling heuristic) and **13.3%** on an STMicroelectronics ST231 400 MHz (embedded SoC) with st200cc 1.9.0B (10.4% for the decoupling heuristic).

Our interpretation of these results is that the more complex the architecture or its compiler are, the better the results obtained using our approach. Let us recall that our approach generates sequential codes: up to that point, we did not expose parallelism.

4.6 Coupling Model-Based and Iterative-Based Optimizations

Some optimizations demonstrate their effectiveness when driven by a model-based heuristics, such as vectorization as shown by Trifunovic et al. [TNC+09]. On the contrary, some other transformations cannot be optimized without a combinatorial search. The loop fusion/distribution is such a transformation as shown by Darte [Dar00]. Affine mapping encompasses both kind of transformations. To reduce the legal mapping space, we consider as equivalent all the “points” that could be compared by a robust performance model. Hence, iterative search can focus on selecting only the “points” for which it is complex to decide.

Using this line of reasoning, we designed a multi-stage optimization scheme combining model-driven and iterative techniques. On one hand, we rely on iterative techniques to drive the choice of loop fusion/distribution. On the other hand, we rely on a model-driven tiling-based optimizing and parallelizing algorithm and on a model-driven vectorization algorithm. The complete scheme is summarized as follows (see our paper and Pouchet's PhD. dissertation for the complete details [CB-PBB$^+$10, Pou10]):

1. We build a search space containing all the legal *partitions* of the various statements, where two statements belong to the same partition if they share at least one common loop in the generated code. This space proves to be very small in practice with respect to the total number of possible partitions if legality is not considered (e.g., from 10^{12} solutions to only 8 legal ones for the `ludcmp` benchmark of the PolyBench suite) and to allow an exhaustive scan.

2. For a given "point" of the legal partition space:

   (a) Apply Bondhugula et al.'s Tiling Hyperplane Method for parallelizing and optimizing locality (which embeds one cost model to choose the way to make a loop nest tilable, and a second one to decide whether to apply tiling or not) [BHRS08].

   (b) Use the model by Trifunovic et al. [TNC$^+$09] to select the order of the innermost loops to expose the best vectorizable loop to the low-level compiler.

The coupled approach obtained good speedups on the PolyBench test suite over the state of the art model-based high-level compiler Pluto [BHRS08]: $2\times$ in average on a 24 cores Xeon E7450 and $2.5\times$ in average on a 4 cores Opteron 8380, and up to $8\times$ with respect to ICC 11.1.

4.7 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 scenarios that may be very difficult to understand. In the late Nineties, several works started to explore an optimization space and demonstrated that performing an extensive search, it is possible to beat the model-based optimization strategies of the compilers. Bodin et al. explore tile size, unrolling factor and padding through a sampling of the optimization space [BKK$^+$98]. Kisuki et al. also focus on tile size and unrolling factor, and evaluated 5 search algorithms to reduce the number of runs, including genetic algorithm, simulated annealing and random search [KKO00]. Cooper et al. relies on genetic algorithm to choose the best sequence of 10 compiler passes to minimize the target code size [CSS99]. Chow and Wu look for the best set of compiler switches and reduce the space by reasoning on sets of switches [CW99]. Those observations and first techniques led to two trends: self-optimizing libraries and iterative search of compilation parameters.

Self-optimizing libraries are a "high-level" part of iterative optimization area where libraries autotune themselves with respect to predefined optimisations (e.g., tiling and unrolling) by generating multiple version of themselves, then empirically choose the best version for the target architecture. ATLAS [WPD00], FFTW [FJ05] and SPIRAL [PSX$^+$04] are efficient implementations of this strategy.

Many works demonstrated the potential for iterative optimization on a large range of optimizations, affecting compiler optimization flags (switches), parameters (e.g., loop unrolling, tiling), phase ordering, the heuristic itself, or the hybridation of multiple heuristics [ACG$^+$04, KHW$^+$05, ABC$^+$06, McBQ02, SAMO03, CMO06]. As a result, the search spaces can be huge and necessitate fast evaluation techniques. Kulkarni et al. [KHW$^+$05] introduce the VISTA system, an interactive compilation system which concentrates on reducing the time to find good solutions. Another system that attempted to speedup iterative compilation was introduced by Cooper et al. called ACME [CGH$^+$05]. Triantafyllis et al. [TVA05] develop an alternative approach to reduce the total number of evaluations of a new program. Here the space of compiler options is examined off-line on a per function basis and the best performing ones are classified into a small tree of options. Because iterative compilation relies on multiple, costly "runs" (including compilation and execution), the current emphasis is on improving the profiling cost of each program version [KHW$^+$05, FCOT05], or the total number of runs, using, e.g., genetic algorithms [KZM$^+$03] or machine learning [ABC$^+$06, CMO06].

Our work studies a different search space: instead of relying on existing 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. Building an actual optimization phase out of this search space is much easier than from the composition of multiple search spaces arising from short-sighted, local transformations. Our method is also complementary to other forms of iterative optimization which address the orchestration of existing heuristics. Furthermore, it is completely independent from the compiler back-end.

Although multidimensional affine scheduling is an obvious target for iterative optimization, its profitability is one of the most difficult to assess, due to (1) the model's intrinsic expressiveness (the downside of its effectiveness) and (2) its lack of analytical models for the impact of transformations on the target architecture. Hence, related work has been very limited up to this point. To the best of our knowledge, Nisbet pioneered research in the area with one of the very first papers in iterative optimization. He developed the GAPS framework [Nis98] which used a genetic algorithm to traverse a search space of affine schedules for automatic parallelization. In addition, Long et al. [LO04, LF05, LF06] considered a search space of transformation sequences represented in the UTF framework [Kel96]. Both of these approaches suffer from under-constraining the search space by considering all possible schedules, including illegal ones. Downstream filtering approaches do not scale, due to the exponentially diminishing proportion of legal schedules with respect to the program size. For instance, Nisbet obtains only $3-5\%$ of legal schedules for the ADI benchmark (6 statements). Moreover, under-constraining the search space limits the possibility to narrow the search to the most promising subspaces.

4.8 Conclusion

Present day compilers fail to model the complex interplay between different optimizations and their effect on all the different processor architecture components. Also, the complexity of current hardware has made it impossible for compilers to accurately model architectures analytically. Thus, empirical search has become essential to achieve portable high performance in spite of the analytically intractable hardware complexity. Most iterative compilation techniques target compiler optimization flags, parameters, decision heuristics, or phase ordering [BKK$^+$98, CST02, SAMO03, KHW$^+$05, TVA05, ABC$^+$06]. This chapter presents a more aggressive stand, aiming for the construction and tuning of complex sequences of high-level transformations.

As demonstrated in previous chapters, affine schedules build a very expressive search space, since a single schedule can represent an arbitrarily complex sequence of loop transformations. The first attempts to traverse such a space faced legality problems and showed poor results because only few legal affine schedules were found [Nis98, LO04]. Our work targets *all* static control programs and, by construction, enables iterative optimization in a closed space of semantics-preserving transformations. Building on Feautrier's work [Fea92a, Fea92b], we showed how to construct a practical legal mapping space in such a way that a traversal becomes possible.

To overcome the combinatorial nature of the optimization search space, we designed heuristics and a genetic algorithm with specialized operators that leverage the algebraic properties of this space, embedding the legality constraints into the operators themselves. We simultaneously demonstrate good performance gains and excellent convergence speed on huge search spaces, even on larger loop nests where fully iterative affine scheduling has never been attempted before. Finally, we designed an efficient coupling between model-based algorithms and our iterative strategy that provided better results than the purely model-based algorithms.

Chapter 5

Scalability: Facing the Real World

Compiler performance has long been quantified through the number of processed code lines per time unit. Compile time used to be (almost) linear in the code length. In order to find the best possible optimizations, present day compilers must rely on higher complexity (possibly worst-case exponential) methods. The polyhedral model is a striking example. Many advances in program restructuring have been achieved through this model. Most of the underlying methods, as data dependence analysis [Fea91, Pug91a], transformation computation [LL97, Gri04, Fea92b, BHRS08] or code generation [KPR95, QRW00, CB-Bas04a] exhibit worst-case exponential complexity.

It is not easy to conclude about the scalability of such techniques. The literature is full of algorithms with high complexity which present a very good practical behavior. The *simplex* algorithm is probably the most famous example [Dan51]. It may be used for instance-wise data dependence analysis or to compute a mapping as a linear programming problem, e.g., when relying on the PIP algorithm [Fea91, Fea92a] which is based on the dual-simplex and Gomory cuts. Building artificially large and complex problems is a poor approach to decide about scalability: small randomly generated problems are likely to be unfeasible while they do not provide much information about how the techniques behave in practice. As a result, our interpretation of the scalability problem is a matter of (1) computing a solution in a reasonable amount of time for any real case, and (2) the ability to compute a convenient enough solution for the most complex real problems.

In this chapter, we describe our efforts and results on facing real problems on two parts of a polyhedral framework. First, Section 5.1 addresses the data dependence analysis problem, showing how to a achieve a fast but conservative violation checking, how to remove transitively covered dependences and presenting an empirical study of exact data dependence analysis on some SPEC benchmarks. Second, Section 5.2 presents solutions to improve the code generation time and to preserve generated code quality on various situations. We present some related work on that matter in Section 5.2.4.

5.1 Data Dependence Analysis Scalability

5.1.1 Fast Data Dependence Violation Check

We presented in Section 3.2.2 page 38 the characterization of the exact sets of violated dependences at depth v. For a dependence at depth d from a source reference r_S of a statement S with mapping θ_S to a target reference r_T of a statement T with mapping θ_T, defined by the constraint matrix $\Delta_{S,r_S \xrightarrow{d} T,r_T}$ the set of violated dependences is described with the following set of constraints:

$$\Upsilon_{S,r_S \xrightarrow{d,v} T,r_T} : \left[\begin{array}{cc:c:cc:c:c:c} \multicolumn{8}{c}{\Delta^{+}_{S,r_S \xrightarrow{d} T,r_T}} \\ \hdashline T_S^{\vec{i}_S} & 0 & T_S^{\vec{t}_S} & 0 & 0 & 0 & T_S^{\vec{p}} & T_S^{c} \\ \hdashline 0 & 0 & 0 & T_T^{\vec{i}_T} & 0 & T_T^{\vec{t}_T} & T_T^{\vec{p}} & T_T^{c} \\ \hdashline 0 & 0 & I_{1..v-1,\bullet} & 0 & 0 & -I_{1..v-1,\bullet} & 0 & 0 \\ \hdashline 0 & 0 & I_{v,\bullet} & 0 & 0 & -I_{v,\bullet} & 0 & 0 \text{ or } -1 \end{array}\right] \left(\begin{array}{c} \vec{i}_S \\ \hdashline \vec{a}_{S,r_S} \\ \hdashline \vec{t}_S \\ \hdashline \vec{i}_T \\ \hdashline \vec{a}_{T,r_T} \\ \hdashline \vec{t}_T \\ \hdashline \vec{p} \\ \hdashline 1 \end{array}\right) \begin{array}{c} = or \geq \\ \hdashline \geq \\ \hdashline \geq \\ \hdashline = \\ \hdashline \geq \end{array} \vec{0},$$

where $\Delta^{+}_{S,r_S \xrightarrow{d} T,r_T}$ is derived from the dependence constraint matrix $\Delta_{S,r_S \xrightarrow{d} T,r_T}$ detailed in Section 2.2.2 page 25, where additional columns set to zero have been added for the corresponding $\vec{t}_S$ and $\vec{t}_T$ dimensions. The existence of a violated dependence can be checked using parametric integer programming [Fea88a] and, e.g., the PIP tool [CB-FcCB02]. This is an overkill for legality checking. This section describes a fast dependence test that may be used to check that a given mapping is legal. This test can also be used to quickly filter out satisfied dependences when computing violated dependence polyhedra.

In practice, dependence violation can be checked efficiently without relying on costly polyhedral operations. Consider a violation depth $v > 0$. The left-hand side of the inequality in the last row represents the violation amount on depth v, i.e., the amount of time by which S is late with respect to T after mapping; we shall denote it by $\Gamma_{S \xrightarrow{v} T}$. The problem is to determine whether or not, $\Gamma_{S \xrightarrow{v} T}$ can be positive on $\Upsilon_{S,r_S \xrightarrow{d,v-1} T,r_T}$, the violated dependence candidates polyhedron at depth $v-1$ defined by all rows except the last one in the violated dependences characterization. This is solved by application of the affine form of the Farkas Lemma we already used in Section 4.1.1 and that we recall here:

Lemma 5.1 *(Affine form of Farkas Lemma [Fea92a, Sch86]) 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:*

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

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

The existence of a set of positive Farkas multipliers, for a given constraints polyhedron, guarantees the function is positive on this polyhedron. In our case, no violation exists iff $\Gamma_{S \xrightarrow{v} T} \leq 0$, or equivalently iff $-\Gamma_{S \xrightarrow{v} T} - 1$ is non-negative everywhere in $\Upsilon_{S,r_S \xrightarrow{d,v-1} T,r_T}$. This translates into finding a set of *rational* positive solutions in a system made of (1) the equalities when equating the various coefficients in both sides of the formula and (2) the constraints specifying that each Farkas multipliers is non-negative. This problem is not parametric anymore and may be solved by (non-integral) linear programming in polynomial time. If no such solution can be found, we consider violated dependences exist. On the other hand, if a solution is found, there are no violated dependences at depth v for $\Delta_{S,r_S \xrightarrow{d} T,r_T}$. Notice this test is associated with a rational relaxation of the integral constraints on $\mathcal{D}$. This may, in rare cases, lead to a conservative results. Finally, we can reuse the system built at depth v for checking at depth $v+1$ since $\Upsilon_{S,r_S \xrightarrow{d,v} T,r_T}$ is simply $\Upsilon_{S,r_S \xrightarrow{d,v-1} T,r_T} \cap \{\Gamma_{S \xrightarrow{v} T} = 0\}$.

Consider the following example:

```
    for (i = 0; i <= N; i ++) {
S:    A[i+1] = A[i];
    }
```

Reversing this loop corresponds to setting $\theta_S = -i$. It is obviously illegal. Let us verify this through our fast legality check. The (simplified) dependence polyhedron is:

$$\delta_{S,1 \xrightarrow{1} S,1}(N) = \left\{ i \geq 0, i \leq N, i' \geq 1, i' \leq i+1, i' \geq i+1 \right\}.$$

After reversal $-\Gamma_{S \xrightarrow{1} S} - 1 \geq 0$ is written $i - i' - 1 \geq 0$. Through the Farkas lemma, a necessary and sufficient condition is

$$i - i' - 1 \;=\; \lambda_0 + \lambda_1 i + \lambda_2 (N - i) + \lambda_3 (i' - 1) + \lambda_4 (i' - i - 1) + \lambda_5 (i - i' + 1).$$

Identifying the coefficients of the i, i', N and constant terms yields the following system:

$$\left\{ \begin{array}{rcl} -1 & = & \lambda_0 - \lambda_3 - \lambda_4 + \lambda_5 \\ 1 & = & \lambda_1 - \lambda_2 - \lambda_4 + \lambda_5 \\ -1 & = & \lambda_3 + \lambda_4 - \lambda_5 \\ 0 & = & \lambda_2 \end{array} \right.$$

The reader may check that this system has no non-negative solution.

5.1.2 Transitively-Covered Dependences

In general, the full dependence graph contains redundant information associated with transitively covered dependences. This incurs computational overhead in subsequent optimization, scheduling or legality checking phases.

The standard technique to eliminate redundant information consists in removing all memory-based dependences by converting the SCoP to (dynamic) single assignment. This transformation amounts to array renaming and expansion (a generalization of array privatization), using the array data-flow analysis technique proposed by Feautrier [Fea88b, Fea91]. This method only considers flow dependences and computes for each statement and each reference it *reads* the last producer of the value read. The algorithm walks the code backwards, calling the PIP library [CB-FcCB02] to incrementally build the result. The solution is a quasi affine selection tree (generalization of a “last write tree” [MAL93]) implementing the case distinctions for pertinent values of the target statement’s iterators and invariant parameters associated with distinct producers (or distinct affine forms). As a dependence graph compression, the major drawback of this approach is the need to operate on a single-assignment program, hence to resort to complex array contraction and storage mapping optimization techniques to ultimately reduce the memory footprint [LF98, QR00].

Our method does not require conversion to single assignment form. Instead, for each target instance, we do identify the last source of a dependence targeting this precise instance, in order to remove transitively covered dependences, but we consider all dependences including the memory based anti (write-after-read) and output (write-after-write) ones. The result is a simplified dependence graph for each depth level, bearing the exact simplified dependence relation. Consider a dependence from S to T at depth d on a given memory location x. The key to our approach is to determine, which are all the possible statements that can be interleaved between the time of execution of a source iteration and the time of execution of its corresponding target iteration(s) (possibly many). We consider a candidate covering statement C that *writes* to x and reason in the Cartesian product of the three former spaces (sharing parameters and the constant dimension though). C must satisfy the following conditions:

- it must be the target of a dependence of depth d from S;
- it must be the source of a dependence of depth d to T.

The layout of the covering polyhedron corresponds to Figure 5.1.

$$
\left[\begin{array}{c:c:c:c:c:c:c:c}
D_S^{\vec{\imath}_S} & 0 & 0 & 0 & 0 & 0 & D_S^{\vec{p}} & D_S^c \\ \hdashline
0 & 0 & D_C^{\vec{\imath}_C} & 0 & 0 & 0 & D_C^{\vec{p}} & D_C^c \\ \hdashline
0 & 0 & 0 & 0 & D_T^{\vec{\imath}_T} & 0 & D_T^{\vec{p}} & D_T^c \\ \hdashline
A_{S,r_S}^{\vec{\imath}_S} & A_{S,r_S}^{\vec{a}_{S,r_S}} & 0 & 0 & 0 & 0 & A_{S,r_S}^{\vec{p}} & A_{S,r_S}^c \\ \hdashline
0 & 0 & A_{C,r_C}^{\vec{\imath}_C} & A_{C,r_C}^{\vec{a}_{C,r_C}} & 0 & 0 & A_{C,r_C}^{\vec{p}} & A_{C,r_C}^c \\ \hdashline
0 & 0 & 0 & 0 & A_{T,r_T}^{\vec{\imath}_T} & A_{T,r_T}^{\vec{a}_{T,r_T}} & A_{T,r_T}^{\vec{p}} & A_{T,r_T}^c \\ \hdashline
0 & I & 0 & -I & 0 & 0 & 0 & 0 \\ \hdashline
0 & I & 0 & 0 & 0 & -I & 0 & 0 \\ \hdashline
I^{1..d-1,\bullet} & 0 & -I^{1..d-1,\bullet} & 0 & 0 & 0 & 0 & 0 \\ \hdashline
I^{d,\bullet} & 0 & -I^{d,\bullet} & 0 & 0 & 0 & 0 & x \\ \hdashline
I^{1..d-1,\bullet} & 0 & 0 & 0 & -I^{1..d-1,\bullet} & 0 & 0 & 0 \\ \hdashline
I^{d,\bullet} & 0 & 0 & 0 & -I^{d,\bullet} & 0 & 0 & x
\end{array}\right]
\begin{pmatrix} \vec{\imath}_S \\ \hdashline \vec{a}_{S,r_S} \\ \hdashline \vec{\imath}_C \\ \hdashline \vec{a}_{T,r_C} \\ \hdashline \vec{\imath}_T \\ \hdashline \vec{a}_{T,r_T} \\ \hdashline \vec{p} \\ \hdashline 1 \end{pmatrix}
\begin{array}{c} \geq \\ \hdashline \geq \\ \hdashline \geq \\ \hdashline \geq \\ \hdashline \geq \\ \hdashline \geq \\ \hdashline = \\ \hdashline = \\ \hdashline = \\ \hdashline \geq \\ \hdashline = \\ \hdashline \geq \end{array}
\vec{0}
$$

where $x = 0$ or 1 depending on the dependence depth.

Figure 5.1: Covering Dependences for $\delta_{S,r_S \xrightarrow{d} T,r_T}(\vec{p})$ and statement C

Due to the transitivity of the equality and inequality relations, access and schedule relations between S and T can be omitted, contributing to lowering the computational cost of the covering polyhedron.

Once a covering polyhedron has been computed and deemed not empty, it is necessary to relate it to the points in $\delta_{S,r_S \xrightarrow{d} T,r_T}(\vec{p})$ that it shadows. The shadowed polyhedron is the projection of the covering polyhedron on $\delta_{S,r_S \xrightarrow{d} T,r_T}(\vec{p})$ dimensions. The projection being a standard PolyLib operation, we will not get into further details here. The last step is then to remove the shadowed polyhedron from the original dependence polyhedron. Once again this is a standard PolyLib operation that may however return a non-convex list of convex polyhedra. If removing some transitively covered dependences makes the data dependence graph more complex, we can simply abort their removal.

Finally, special care must be taken when dealing with conservative dependence approximations associated with non-affine array subscripts. Our current implementation preserves every transitively covered dependence arc when the source or target of this arc is a non-affine reference. More precise methods have been proposed but their practical evaluation is left for future work [Won95, Bar98].

5.1.3 Scalability

Our dependence analysis is implemented within the modern infrastructure of Open64 and PathScale EKOPath [Cho04]. This compiler family provides key interprocedural analyses and pre-optimization phases such as inlining, interprocedural constant propagation, loop normalization, integer comparison normalization, dead-code and `goto` elimination, as well as induction variable substitution. Our tool extracts large and representative SCoPs for SPEC CPU2000 fp benchmarks: on average, 88% of the statements belong to a SCoP containing at least one loop. See [CB-GVB$^+$06] for detailed static and dynamic SCoP coverage.

In this section, we exercise this implementation on 6 full SPEC CPU2000 fp benchmarks. These codes were selected because of their large SCoPs, within the set of 8 benchmarks that our tool could handle without instabilities (largely due to the underlying Open64 platform). In the most challenging examples, the biggest SCoP almost contains the whole program after inlining.

Figure 5.2 summarizes our experimental results. To stress the analyzer, we performed aggressive inlining to favor the formation of the largest possible SCoPs. These statistics are often associated with aggregated SCoPs from multiple functions whose names and line numbers are listed in the second and third columns. #Params gives the number of global parameters, and #Refs gives the total number of array references (read and write) in the SCoP. The next two blocks of columns summarize the properties of the dependence graph, first considering all dependences, second after elimination of transitively covered dependences: #Matrices gives the number of dependence matrices, #Columns give the average number of columns in all dependence matrices (i.e., the average dimension of dependence polyhedra, a good indication of the complexity of the problem), and Analysis Time corresponds the computation time in seconds to compute the dependence graph on a 2.4GHz Pentium 4 (Northwood) workstation.

The selected SCoPs account for the majority of the execution time in all benchmarks. The smaller SCoPs have been omitted to focus the experiments on the most time-consuming ones. The SCoPs in 168.wupwise feature non-affine array accesses due to conservative induction variable substitutions (the actual references are affine but Open64 could not figure it): covered dependences could not be removed in this case.

The full instance-wise dependence analysis takes up to 37.512 seconds, for the largest SCoP in 173.applu. This is an extreme case with huge iteration spaces (more than 13 dimensions on average, and up to 19 dimensions). This may sound quite costly, but it still shows that the analysis is compatible with the typical execution time of aggressive optimizers (typically more than ten seconds for Open64 with interprocedural optimization and aggressive inlining and loop-nest optimizations). In all other cases, it takes less than 5 seconds, despite thousands polyhedral operations with close to 10 dimensions on average.

These results are quite compelling since we compute very large dependence graphs, taking all pairs of references into account. Many implementation details can also be improved, using cheap dependence tests as filters for full polyhedral operations, performing on-demand computations on part of the dependence graph only, and improving the polyhedral computation cache to catch a wider scope of operations. These improvements can bring an additional order of magnitude acceleration, as shown in previous experiments [Won95].

According to these results, removing covered dependences is slightly more expensive. This should not be taken for a definite result: to simplify the implementation, we used polyhedral differences and calls to the PolyLib (following an algorithm closer to the one proposed by Pugh [Pug91a]), yet we anticipate that an implementation based on Feautrier's PIP would have a much lower cost [Fea91, Gri04]. Notice removing covered dependences may sometimes *increase* the total number of matrices, due to domain decompositions to represent non-convex iteration spaces.

					All Dependences			w/o Covered Dependences		
	Function	Source Lines	#Params	#Refs	#Matrices	#Columns	Analysis / Time (s)	#Matrices	#Columns	Analysis / Time (s)
168.wupwise	zaxpy zcopy	11–32 11–24	5 5	16 12	62 30	7.5 7.0	0.008 0.005	62 30	7.5 7.0	0.008 0.005
171.swim	main + calc1 + calc2 + calc3	114–119 + 261–269 + 315–325 + 397–405	5	216	813	10.5	0.895	624	10.4	2.630
172.mgrid	psinv + resid + interp	149–166 + 189–206 + 270–314	2	191	870	8.4	0.735	962	8.4	1.894
173.applu 1st SCoP 2nd SCoP	blts + buts + jacld + jacu + rhs	553–624 + 659–735 + 1669–2013 + 2088–2336 + 2610–3068	4 2	562 1983	3507 12814	11.3 13.5	4.420 37.512	3188 10418	11.2 13.5	14.865 115.439
200.sixtrack	thin6d	560–588	7	86	158	11.1	0.044	110	11.1	0.117
301.apsi 1st SCoP 2nd SCoP	dcdtz + dtdtz + dudtz + dvdtz + wcont + smth	1326–1354 1476–1499 1637–1688 1779–1833 1878–1889 3443–3448	1 7	275 198	4264 216	2.0 12.9	0.211 0.133	203 207	2.0 12.8	1.750 0.726

Figure 5.2: Scalability of instance-wise dependence analysis

A direct computation of the violated dependence graph takes approximately the same amount of time as computing the dependence graph itself. When verifying very complex transformation sequences, it may at most become twice as expensive: this is the case when optimizing the 171.swim benchmark as described in our previous work [CGP$^+$05] (leading to 38% speed-up with respect to the peak SPEC performance with the best optimization flags on Athlon64). As described in Section 5.1.1, if violations are only associated with a limited number of dependences, it is much more practical to apply the fast Farkas-based dependence test and compute violated dependence polyhedra only when a possible violation is detected. This fast dependence test takes a negligible amount of time compared to the actual operations on polyhedra since it considers the same non-negativity constraints but solves (relaxed) rational linear programming problems instead of integral ones.

5.1.4 Related Work

Many tests have been designed for dependence checking between different statements or between different executions of the same statement. It has been extensively shown that this problem amounts to detecting whether or not a system of equations has an integer solution inside a region of $\mathbb{Z}^n$ [Ban88].

Most of the dependence tests try to find efficiently a reliable, approximative but conservative (they overestimate data dependences) solutions. The GCD-test [Ban76] has been the very first practical solution, it is still present in many implementations as a first check with low computational cost. This test assumes that if the greatest common divisor of the coefficients of an equation divides the constant term, then a solution exists. A generalized GCD-test has been proposed to handle multi-dimensional array references [Ban88]. The Banerjee test uses the intermediate value theorem to disprove a dependence: it computes the upper and lower bounds of an equation and checks if the constant part lies in that range [WB87]. The λ-test is an extension to this test that handles multi-dimensional array references [LYZ89]. Some other important solutions are a combination of GCD and Banerjee tests called I-test [KKP90], the Δ-

test [GKT91] that gives an exact solution when there is at most one variable in the subscript functions, and the Power-test which uses the Fourier-Motzkin variable elimination method [Sch86] to prove or disprove dependences [WT92]. Beside their approximative nature, these dependence tests suffer from many other major limitations. The most stringent one is their inability to precisely handle `if` conditionals, loops with parametric bounds, triangular loops (a loop bound depends on an outer loop counter), coupled subscripts (two different array subscripts refer the same loop counter), or parametric subscripts.

On the opposite, a few methods allow to find an exact solution to the dependence problem, but at a higher computational cost. The Omega-test is an extension to the Fourier-Motzkin variable elimination method to find integral solutions [Pug91a]. On one hand, once a variable is eliminated, the original system has an integer solution only if the new system has an integer solution (if this is not the case there is no solution). On the other hand, if an integer point exists in a space computed from the new system, then there exists an integer point in the original system (if this is the case, there is a solution). The PIP-test uses a parametric version of the dual-simplex method with Gomory cuts to find an integral solution [Fea91]. These two tests not only give an exact answer, they are also able to deal with complex loop structures and (affine) array subscripts. The PIP-test is more precise than the Omega-test when dealing with parametric codes (when one or more integer symbolic constant are present). Both tests have worst-case exponential complexities but behave quite well in practice as shown by Pugh for the Omega-test [Pug91a]. Other costly exact tests exist in the literature [MHL91, ES92] but are often not able to handle complex control in spite of their cost.

In this work, we do not advocate for the use of any of these tests, but rather for the computation of *instance-wise* dependence information as precisely as possible, i.e., for intensionally describing the statically unbounded set of all pairs of dependent statement instances. Dependence tests are statement-wise decision problems associated with the existence of a pair of dependent instances, while instance-wise dependence analysis provides additional information that can enable finer program transformations, like affine scheduling [Fea92a, Fea92b, LL97, Gri04, BHRS08, CB-PBB$^+$11]. The intensional characterization of instance-wise dependences can take the form of multiple *dependence abstractions*, depending on the precision of the analysis and on the requirements of the user [YAI95]. The simplest and least precise one is called *dependence levels*, it specifies for a given loop nest which loop carry the dependence. It has been introduced in the Allen and Kennedy parallelization algorithm [AK87]. The *direction vectors* is a more precise abstraction where the i-th element approximates the value of all the i-th elements of the *distance vectors* (which shows the difference of the loop counters of two dependent instances). It has been introduced by Lamport [Lam74] and formalized by Wolfe [Wol95] and is clearly the most widely used representation. The most precise abstraction is the *dependences between iterations* [IT87] which is able to determine exactly the set of statement instances in data dependence relation. The choice of a given dependence abstraction is crucial for further study. Simple abstractions can provide the exact information on some simple cases. But in general, choosing an imprecise abstraction can result in invalidating interesting transformations. Because we are dealing with any code that can be represented using polyhedral relations (including, e.g., imperfectly nested loops), dependences between iterations is the only abstraction which always encode the exact information [YAI95, Iri11]. As a result, this document relies only on this abstraction, also referred in our work as dependence relations.

5.2 Code Generation Scalability

Polyhedral code generation has an intrinsic worst-case complexity of $3^{n\rho}$ polyhedral operations (themselves associated with NP-complete problems), where n is the number of statements, and ρ the maximum number of (non-parameter) dimensions. Nevertheless, input programs are not randomly generated. Most of the time, human-written codes show simple control, loop nests with low depth and which enclose few statements. Such properties make it possible to regenerate, through the whole source-to-polyhedra-to-source framework, well known benchmark codes with hundreds of statements per static control compute kernel (in the SPECfp2000 benchmarks) in an acceptable amount of time [CB-Bas04a].

Complex transformations may be automatically computed by a given optimizing compiler [BDSV98, LL97, CB-BF03a, Gri04, BHRS08, CB-PBCC08a] or discovered by a programmer with the help of an optimization environment [MPNT04, CGP+05, Kel96, BDD+07, CCH08]. Their application diminishes the input program regularity and lead to a challenging code generation problem. The challenge may come either from the ability to compute any solution (because of a complexity explosion) or from the ability to find a satisfactory solution (because of a high resulting control overhead). To solve these problems *in practice*, we carried out an experiment-driven study, starting from a state-of-the-art code generation tool [CB-Bas04a]. We analyzed in depth a complex optimizing transformation sequence of the SPECfp2000 benchmark `Swim` that has been found by an optimization expert with the help of the URUK framework [CGP+05]. Our goal was to find properties of the transformations themselves that may be exploited to reduce the complexity problem, and to improve the generated code quality.

To validate our approach, we studied and applied our methods to other complex problems that have been submitted by various teams from both industry and academia. Each of them uses its own strategy to compute transformations, which encourage the search for common transformation properties. `QR` has been provided by Reservoir Labs Inc. which develop the high level R-Stream compiler [MVW+11]. `Classen` has been submitted by the FMI laboratory of the University of Passau which develops the high level parallelization tool LooPo [Len93, Gri04]. `DreamupT3` has been supplied by the RNTL Project DREAM-UP between Thales Research, Thomson R&D and École des Mines de Paris [HAI+05]. General properties of these reference problems are shown in Figure 5.3. They proved to be quite different, spanning all typical sources of complexity in polyhedral code generation: each benchmark has its own reason to be challenging, e.g. high statement number for `Swim`, deep loop nests for `Classen`, big values that need multi-precision arithmetic to be to manipulated with `DreamupT3`.

	Reference problems			
Properties	`Swim`	`QR`	`Classen`	`DreamupT3`
Statement number	199	10	8	3
Maximum loop depth	5	3	8	2
Number of parameters	5	2	1	0
Scheduling dimensionality	11	7	7	1
Maximum coefficient value	60	5	4	1919

Figure 5.3: General properties of reference problems

We investigate two aspects of code generation for complex problems. Section 5.2.1 investigates algorithmic scalability challenges and our solutions, driven by experimental evaluations of the four reference benchmarks. Section 5.2.2 addresses additional code generation challenges associated with code size reduction and efficiency; in particular, it presents the first modulo-condition elimination technique that succeeds for a large class of real-world schedules while avoiding code bloat due to multi-versioning. Finally, Section 5.2.4 positions this study among related works.

5.2.1 Reducing Code Generation Time

This section analyzes three important properties of affine schedules used in real-world program generation problems, then for each property, proposes an algorithmic solution to improve scalability.

Scalar Dimensions

There are many ways to specify a given transformation (or a given sequence of transformations) using affine schedules. Basically we can divide them in two families. The first kind, mono-dimensional schedules, describe the execution order thanks to functions with only one dimension. The second kind, multi-dimensional schedules, use several dimensions to express the ordering. Most of the time, the original domains are parametric, i.e., are bounded by (statically) unknown constants. For the first kind, this variety amounts to manipulating non-affine expressions. This is not the case with multi-dimensional schedules, when using at least as many dimensions as the original domain [Fea92b]. Moreover, using additional dimensions to explicitly order different statements onto a given dimension makes transformation manipulation easier as shown in Section 3.3.2 [Kel96, CGP+05]. As a result, multi-dimensional schedules with more dimensions than iteration domains are quite often used to specify transformations. Figure 5.4 shows an example of a loop interchange transformation applied to the polynomial multiply kernel shown in Figure 2.1 that may be achieved thanks to different schedules. $\rho(S)$ is the depth of the original statement, i.e., the number of dimensions of its original iteration domain.

Scheduling policy	θ_{S1}	θ_{S2}
Mono-dimensional	(i)	$(n+j*n+i)$
$(\rho(S)+1)$-dimensional	$(i,0)^T$	$(n+j,i,0)^T$
$(2*\rho(S)+1)$-dimensional	$(0,i,0)^T$	$(1,j,0,i,0)^T$

(a) Possible scheduling functions for loop interchange

```
     for (i = 0; i < 2 * degree - 1; i++)
S1:    z[i] = 0;
     for (j = 0; j <= degree; j++)
       for (i = 0; i <= degree; i++)
S2:      z[i+j] += x[i] * y[j];
```

(b) Target code

Figure 5.4: Loop interchange for polynomial multiplication using different schedules

Unified transformation frameworks like UTF [Kel96] or URUK [CGP+05] or CHiLL [CCH08] or the one presented in Section 3.3.2 are good examples of multi-dimensional schedule policies. All ask for $(2\rho(S)+1)$ dimensions which allow them to be much more flexible (in the case of the technique detailed in this document, this is a minimum). Nevertheless, using additional dimensions has a cost. In time: because each dimension needs costly polyhedral operations (projection/separation/sorting). In space: each dimension implies (1) a new column in the constraint matrix, (2) as many rows as new constraints and (3) a new level in the generated code tree.

Most of the time, additional dimensions are *scalar*, i.e. they are constant for every scheduling functions. Because polyhedral operations on such dimensions are trivial, we systematically *remove them from the constraint matrix*, storing the scalar values in ad-hoc vectors. In the following, scalar dimensions will be implicitly stripped away from the schedule matrices. Polyhedral operations as usual with the additional provision that, before each separation step, we order the polyhedra according to the appropriate scalar vector components. Further steps of the code generation algorithm are applied onto lists of polyhedra having the same values for these components.

This optimization benefits from schedule properties without impacting expressiveness. It may dramatically reduce the number of polyhedral operations, improving both time and space complexity. Moreover, it also reduces the cost (in time and space) of every single polyhedral operation, by reducing matrix size. In practice, the actual benefits depend on the transformation policy: the more the constant scalar dimensions, the better the results. Also, this step has a very low complexity and thus does not degrade computation time even in worst case scenarios. Figure 5.5 shows the results when applying this optimization to our reference code generation problems. The scalar ratio gives the number of scalar dimensions with respect to the total number of dimensions, showing that the different teams which provided their problems do use scalar dimensions. This results into significant time and space improvement, except for the last program.

		Time			Space		
Benchmark	Scalar ratio	Original(s)	Scalar(s)	Speedup	Original(KB)	Scalar(KB)	Reduction
Swim	6/11	41.20	10.33	3.99×	17480	8128	2.15×
QR	4/7	19.47	2.44	7.98×	3012	988	3.05×
Classen	3/7	1.12	0.69	1.62×	1092	672	1.62×
DreamupT3	0/1	0.49	0.49	1.00×	160	160	1.00×

Figure 5.5: Experimental results for scalar dimension removal

Node Fusion

When specifying transformations for a program with many statements, the processing is often similar for several statements, at least for some dimensions. For instance, applying a given transformation (same schedules) to some statements of a given loop nest (same domains) allow to consider only one statement block. The modified version of the QRW algorithm [CB-Bas04a] is given in Figure 5.6 and exploits the similarities of the transformations on certain dimensions for different statements.

```
CodeGeneration: builds an AST (Abstract Syntax Tree) scanning a list of polyhedra
Input:
  node: flat AST holding the domains to scan
  context: static context (known constraints met by the parameters)
  depth: the nesting level
Output: An AST scanning the polyhedra in the lexicographic order

   AST ← ∅
   while node has successors
1    Intersect node.domain with the context
2    Project intersected domain on the depth outermost dimensions and on parameters
3    node ← node.next

   if nodes have scalar values at depth and they are different
4    Sort nodes according to their scalar values at depth
5  worklist ← partition nodes by scalar values

   foreach job in worklist
6    fusedlist ← Fuse nodes of job with the same projected intersected domain
7    separatedlist ← Apply QRW's separation step to fusedlist
8    sortedlist ← Sort separatedlist according to the lexicographic order
     foreach ASTnode in sortedlist
       if ASTnode.domain dimensionality > depth
9        ASTnode.inner = CodeGeneration(ASTnode.node, context, depth+1)
10     Enqueue ASTnode to AST

   return AST
```

Figure 5.6: Extended Code Generation Algorithm

Steps 4 and 5 create work-lists that fully take advantage of the detection of scalar dimensions described in Section 5.2.1. Step 6 examines nodes of each job of the work-list and tries to fuse them into sub-work-lists to reduce the number of elements given to the QRW algorithm as much as possible. Node fusion occurs at current depth on the projected domains and is guaranteed to exploit similarities between schedules at each nesting level independently. The complexity gain of Steps 4, 5 and 6 is difficult to quantify as it depends on the shape of the generated code itself and transformation similarities across different statements.

Considering a simple case with n statements in a loop nest level that can be blocked into c chunks of s_c statements with same scalar components. Suppose each chunk can further be blocked into b_c blocks of $l_{b_c} \leq s_c$ statements with same projected domains. This translates to $\sum_{b_c} (\text{QRW}(l_{b_c}))$ instead of $\text{QRW}(n)$ which stands for a call to the QRW separation algorithm that has a worst-case complexity of 3^n. Furthermore, Step 8 also benefits from the reduction above and allows for $\sum_{b_c} (\text{Sort}(l_{b_c}))$ instead of $\text{Sort}(n)$ which stands for a call to a function sorting n polyhedra that also has an exponential worst case complexity. Experimental results are summarized in Figure 5.7. As expected, this technique is quite useful for large problems like `Swim`.

	Time			Space		
Benchmark	Original(s)	Fused(s)	Speedup	Original(KB)	Fused(KB)	Reduction
Swim	41.20	5.90	6.98×	17480	5048	3.46×
QR	19.47	19.17	1.02×	3012	2992	1.01×
Classen	1.12	1.03	1.09×	1092	1060	1.03×
DreamupT3	0.49	0.49	1.00×	160	160	1.00×

Figure 5.7: Experimental results on node fusion

Domain Iterators

It is well known that code generation is easier when restricting the problem to invertible schedules [Xue94, QRW00]. CLooG was the first tool to seamlessly manage non-invertible schedules, at the cost of additional recursion steps, polyhedral projections and larger matrix sizes in the QRW algorithm [CB-Bas03, CB-Bas04a]. For scalability reasons, we propose to detect non-singularity conditions and refine the recursive AST traversal automatically. Indeed, when considering invertible transformations, the value of the original domain iterators (used, e.g., in the statement bodies) according to the target space iterators can be efficiently obtained via matrix inversion (instead of recursive polyhedral projections).

Let $\theta(\vec{\imath}) = T \cdot \vec{\imath} + T_p \cdot \vec{p}$ be a schedule transformation where T is invertible, and consider an iteration domain $\mathcal{D} : A \cdot \vec{\imath} + A_p \cdot \vec{p} \geq 0$. The mapping $\mathcal{T}$ can be broken down into two distinct components:

- a polyhedron to scan (Figure 5.8) obtained by projecting $\mathcal{T}$ on time iterators and parameters only;
- an *inverted scatter matrix* (ISM) that associates, locally to each statement, the expression of the domain iterators as invertible functions of time iterators and parameters. When T is non-unimodular, T^{-1} has rational coefficients. Let $(d_{i,j})$ be the denominators of T^{-1}, by taking $\lambda_i = \text{lcm}(d_{i,\bullet})$ we define $\Lambda = \text{Diag}(\lambda_i)$ as the diagonal matrix where the diagonal element of the i^{th} line is λ_i. The left multiplication of the matrix representation of $\mathcal{T}$ by $(\Lambda T^{-1} | 0)$ yields an integral matrix, the ISM in Figure 5.9.

The benefits brought to the separation algorithm are threefold and contribute to possibly exponential complexity gains:

$$\mathcal{T}\perp\left(\begin{array}{c|c|c} \vec{t} & 0 & 0 \\ \hline 0 & 0 & \vec{p} \end{array}\right)$$

Figure 5.8: Simplified time-extended domain

$$\left(\begin{array}{c|c} \Lambda T^{-1} & -\Lambda \end{array}\right)\left(\begin{array}{c} \vec{t} \\ \hline \vec{\imath} \end{array}\right) - \Lambda T^{-1} T_p \vec{p} = 0$$

Figure 5.9: ISM to recover the domain iterators

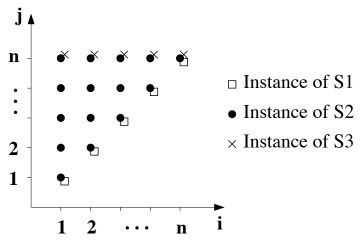
- it is straightforward to write domain iterators as expressions of time iterators and parameters from Figure 5.9 instead of performing costly polyhedral projections on each domain iterator;
- the column number of each polyhedron to scan is reduced by the number of domain iterators (potentially half the original size if there are no parameters);
- the height of the generated AST is reduced on each path to every statement by the same amount above. However the paths subject to reduction are linear and save no branches from the original AST but still save polyhedral projections.

The `Swim` benchmark has invertible schedules only (this is a strong assumption of the URUK framework [CGP$^+$05]), but this is not the case for the other benchmarks. We could therefore evaluate this optimization to `Swim` only, yielding 36% reduction in code generation time and 57% reduction in memory usage. We are working on extending this domain iterator elimination technique to all kinds of non-invertible schedules, combining Gaussian elimination steps with polyhedral projections.

If Conditional Hoisting

Under complex transformation sequences, the top-down part of the QRW code generation algorithm [QRW00] yields `if` conditionals that greatly hamper the quality of the generated code and thus, its execution time. Figure 5.10 exhibits this behavior on a basic example: generating a code for scanning the polyhedra of Figure 5.10(a) using the algorithm in Figure 5.6 would lead to the code in Figure 5.10(b). This figure shows internal guards leading to a high control-overhead.

The approach presented in [QRW00] for removing inner `if` conditionals and generating a better code such as the one in Figure 5.10(c) consists of a backtracking call to the separation procedure. Although it proved successful at performing its primary task, its side effects can yield unnecessary computation and code bloating. The aforementioned algorithm lacks the capability of factorizing similar conditionals. Examine a node at depth d after the separation phase. Assume the separation has generated an inner conditional c which depends only on the i first dimension iterators, $i < d$. During the backtracking called at depth d, the original algorithm [QRW00] and its first extension in [CB-Bas04a] perform separation regardless of the condition c. Therefore, costly polyhedral operations have been made while only a separation at depth i was necessary. Focusing only on conditionals also avoids to version triangular loops which may not execute only for specific values of the outer loop counters. For instance, in Figure 5.10(c)



(a) Domains to scan

```
for (i=1; i<=n; i++) {
  if (i==n) {
    S1(i,n);
    S2(i,n);
    S3(i,n);
  }
  if (i<=n-1) {
    S1(i,i);
    S2(i,i);
  }
  for (j=i+1; j<=n-1; j++)
    S2(i,j);
  if (i<=n-1) {
    S2(i,n);
    S3(i,n);
  }
}
```

(b) Before if-hoisting

```
for (i=1; i<=n-1; i++) {
  S1(i,i);
  S2(i,i);
  for (j=i+1; j<=n-1; j++)
    S2(i,j);
  S2(j,n);
  S3(j,n);
}
S1(n,n);
S2(n,n);
S3(n,n);
```

(c) After if-hoisting

Figure 5.10: Removing internal guards with if-hoisting

the j-loop does not iterate for $i = n - 1$; removing this negligible control overhead would increase code size by 50%.

Our solution boils down to a depth-first traversal of the AST, fetching all the conditionals of subsequent domains for the current nesting level, *factorizing* them by performing polyhedral separation (intersection and difference) *on conditionals relevant to the current depth only*, and intersecting these newfound conditionals with the current domain, duplicating the underlying AST structure. The algorithm, which intervenes as a post pass after separation guarantees no unnecessary cuts are performed and therefore avoids unnecessary code explosion. Figure 5.11 shows the duplication factor results on the four reference benchmarks, i.e., the number of computational statements in the generated code divided by the number of statements in the polyhedral representation, a reasonable metric for code quality [CB-Bas03]. These results show strong code size reductions can be achieved through our improved `if`-hoisting phase. The relatively low duplication factor for `Swim` (2.5) is also a very good indication of the applicability and scalability of polyhedral techniques to larger optimization and parallelization problems. Eventually, to better isolate the effect of this optimization, the last row (Figure 5.11) reports results for the simple one-statement matrix multiplication, applying three-dimensional tiling and shifting through the URUK framework [CGP+05]. It incurs major (yet unavoidable) code bloat, but our technique reduces it by a factor of 2.5.

Benchmark	Original dup. factor	`if`-hoisting dup. factor	Reduction
Swim	2.5	2.5	1
QR	107	35	3
Classen	11.5	9.6	1.2
DreamupT3	23.3	4	5.8
MxM	175	69	2.5

Figure 5.11: Experimental results with `if`-hoisting

5.2.2 Preserving Generated Code Quality

Beyond code generation performance, addressing real-world problems raises generated code quality issues that may not directly emerge from smaller, academic examples. This section investigates four of them: first, extending code generation to implement a smarter loop unrolling strategy, and second building on this extension to achieve a major step in code generation for strided domains and re-indexed schedules. The last two solutions are examples of code generation-time optimization that have been implemented within the Reservoir Labs Inc. R-Stream compiler (see Appendix A) for an efficient mapping to architectures where control or access function overhead is critical such as ClearSpeed CSX700 [Cle08b, CB-BVL$^+$09]: mapping simplification and full-tile extraction.

Enabling Strip-Mining for Unrolling

In most cases, loop unrolling can be implemented as a combination of *strip-mining* and *full unrolling* [Wol95]. Strip-mining itself may be implemented in several ways in a polyhedral setting. Calling b the strip-mining factor, we choose to model a strip-mined loop by dividing the iteration span of the outer loop by b instead of leaving the bounds unchanged and inserting a non-unit stride b:

```
for (t=$\ell(\vec{\imath})$; t<=$u(\vec{\imath})$; t++)
```

$\Downarrow$ strip-mine(b)

```
for (t1=$\left\lceil \frac{\ell(\vec{\imath})}{b} \right\rceil$; t1<=$\left\lfloor \frac{u(\vec{\imath})}{b} \right\rfloor$; t1++)
  for (t2=max($\ell(\vec{\imath})$,b*t1); t2<=min($u(\vec{\imath})$,b*t1+b-1); t2++)
```

This design preserves the convexity of the polyhedra representing the transformed code, alleviating the need for specific stride-recognition mechanisms (based, e.g., on the Hermite normal form).

In Figure 5.12(b) we can see how strip-mining by a factor of 2 the original code of Figure 5.12(a) yields an internal loop with non-trivial bounds. It can be very useful to unroll the innermost loop to exhibit register reuse (a.k.a. register tiling), relax scheduling constraints and diminish the impact of control on useful code. However, unrolling requires to cut the domains so that `min` and `max` constraints disappear from loop bounds. Our method derives from our if-hoisting strategy; the difference lies in the selection of conditionals. For the purpose of if-hoisting (see Section 5.2.1), we just had to pick the constraints that did not concern the node at current depth. Here we focus on finding conditionals (lower bound and upper bound) for the current depth, *such that their difference is a non-parametric constant*: the unrolling factor. Hoisting these conditionals actually amounts to splitting the outer strip-mined loop into a kernel part where the inner strip-mined loop will be fully unrolled, and a remainder part (not unrollable) spanning at most as many iterations as the strip-mining factor. In our example, the conditions associated with a constant trip-count (equal to 2) are `t2>=2*t1` and `t2<=2*t1+1` and are associated with the kernel, separated from the prologue where `2*t1<M` and from the epilogue where `2*t1+1>N`. This separation leads to the more desirable form of Figure 5.12(c).

Finally, instead of implementing loop unrolling in the intermediate representation of our framework, we delay it to the code generation phase and perform full loop unrolling in a lazy way, avoiding the added (exponential) complexity on the separation algorithm. This approach relies on a preliminary strip-mine step that determines the amount of partial unrolling.

Removing Modulo Conditions

When mappings $\mathcal{T}$ are $\mathbb{Z}$-polyhedra (a.k.a. lattice polyhedra), the generated code shows modulo conditions. The modulo guards guarantee that only the iterations that belong to the original domain are

```
for (t=M; t<=N; t++)
  S1(t);
```

(a) Original code

```
for (t1=M/2; t1<=(N+1)/2; t1++)
  for (t2=max(M,2*t1);
       t2<=min(N,2*t1+1); t2++)
    S1(t2);
```

(b) Strip-mining of 2

```
if (M%2==1)
  S1(M);
for (t1=(M+1)/2;
     t1<=(N-1)/2; t1++)
  S1(2*t1);
  S1(2*t1+1);
if (N%2==0)
  S1(N);
```

(c) Separation & unrolling

Figure 5.12: Strip-mining and unrolling transformation

scanned in the generated code. For instance, if the ISM of a statement S (see section 5.2.1) that gives the value of the original domain iterators (e.g., i) according to the transformed space iterators (e.g., t) gives $2i = t$, the execution of the statement S will be guarded with `if` (t%2 == 0). This situation happens either when the mapping corresponds to a non-unimodular transformation or when the original domains $\mathcal{D}$ are $\mathbb{Z}$-polyhedra. Both cases boil down to the same code generation problem. We will only detail the solution in the case of invertible, non-unimodular schedules, for a complete modulo condition removal scheme, the reader should refer to Vasilache's thesis [Vas07].

The consequence of generating modulo guards is to introduce a high control overhead. Many solutions have been suggested to avoid them. The first idea was to compute an appropriate loop stride. At first it was done using the Hermite Normal Form [LP94, Xue94, DR94, Ram95], but this was limited to only one domain, then by considering the transformation expression itself [KPR95, CB-Bas03], but some guards cannot be removed in this way. More recent methods suggest to use strip-mining for one domain [FO05], or to find equivalent transformations with convenient additional dimensions when this is possible [Gri04], or to unroll the loops according to a convenient unroll factor in the case where modulo guards depend on only one loop counter [Gri04]. Here we give a general algorithm to drastically reduce the number of modulo guards inside the loops and even void them all in the *loop kernels*.

Consider a simple example with two statements, where S1 has the one-dimensional schedule $2t - 5$ and S2 has the one-dimensional schedule $3t$. In other words, the rate of S1 is 50% higher than S2 and is shifted ahead by 5 iterations. This example is derived from the low-level scheduling and code generation for a software-pipelined FIR filter, where one functional unit (a multiplier in S1) is needed at a 50% higher rate than a another one (an adder in S2), and S2 depends on S1. Due to the combined reindexing (factors 2 and 3 in the schedule) and shifting (by 5 iterations), traditional techniques to avoid modulo expressions cannot be applied [CB-Bas03], and existing code generators yield the inefficient code of Figure 5.13. Our technique eliminates modulo expressions completely from the kernel part (the hot path) of the generated code, without code bloat, and generates the much more efficient version in Figure 5.14. On this simple example, our technique achieves a 67% reduction in generated code execution-time, with respect to the more naive one with modulo expressions.

In the general case, the main problem resides in the lower bound of the mapped domain [DR94, Ram95, Xue94] whose value *modulo the stride factor* must be known in order to exhibit a regular pattern in the loop body. This lower bound can be viewed as a *pattern alignment synchronization barrier* for S1 and S2. Indeed, parametric schedules with non-unit stride factors may generate as many different loop body patterns as the least common multiplier of these strides; notice these patterns are *not* identical (in general) up to loop body "rotations". The only solution to thoroughly eliminate modulo conditions is multi-versioning, but it results in severe code bloat for stride factors over 2 or 3.

Our approach consists in forcing *pattern synchronization* by strip-mining the original loop by a factor

```
(...)
// software pipeline kernel
for (t=5; t<=2*N-2; t++) {
  if ((t-5)%3 == 0)
    S2(i = (t-5)/3);
  if (t%2 == 0)
    S1(i = t/2);
}
(...)
```

Figure 5.13: Usual Solution

```
// prologue
S2(0);
// kernel code with S1 and S2 synchronized modulo 6
for (t1=1; t1<=floord(N-4,3); t1++) {
  S1(i = 3*t1);                      // t2%6=0
  S2(i = 2*t1 -1);                   //
  S1(i = 3*t1 +1);                   // t2%6=2
  S1(i = 3*t1 +2);                   // t2%6=3
  S2(i = 2*t1);                      // t2%6=4
}
// epilogue
for (t1=ceild(N-3,3); t1<=floord(N-1,3); t1++) {
  for (t2=6*t1; t2<=2*N-2; t2++) {
    if ((-t2+5)%3 == 0)
      S2(i = (t2 -5)/3);
    if (-t2%2 == 0)
      S1(i = t2/2);
  }
}
```

Figure 5.14: Software-Pipelined Solution

that is yet to determine. This amounts to extracting a prologue and an epilogue from the unrollable kernel, yielding the much more efficient solution of Figure 5.14. Using this method, the prologue and epilogue still contain internal modulo conditions whereas the kernel (where the vast majority of the execution time is spent) can be unrolled. This approach is effective on a large class of “well-behaved” schedules. We will argue at the end of this section that the other “ill-behaved” schedules are intrinsically code-bloating if modulo expression elimination is to be attempted.

The previous case having the sole purpose of stating the problem simply, we now outline the general algorithm. This step takes place after the separation, `if`-hoisting, and lazy unrolling steps. From the *Inverse Scatter Matrix* (ISM) shown in Figure 5.9, we can derive that the i^{th} original loop iterator x_i corresponding to a given statement S can be expressed thanks to the i^{th} line of its ISM formula: $\lambda_i \cdot x_i = \left(\sum_j (k_{i,j} \cdot t_j) + C\right)$, where C is the constant parametric part. It follows, a modulo condition that rules the execution of S is $\left(\sum_j (k_{i,j} \cdot t_j) + C\right) \bmod \lambda_i = 0$. Let us first assume that C is known at compile time. The point is to statically determine the values of $(k_{i,j} \cdot t_j) \bmod \lambda_i$ for all i and j to be able to remove all the modulo guards. For that purpose, for each node of the AST at depth j, the time dimension t_j will be unrolled by the least common multiplier over all statements under this node (at depth j) of

$$\mathrm{lcm}_j = \operatorname*{lcm}_{\{i|k_{i,j}\neq 0\}} \left(\lambda_i / \gcd(k_{i,j}, \lambda_i)\right).$$

Unrolling by this factor yields as many instances of t_j for which we statically know the value modulo λ_i. For a given loop node at depth d, the least common multiplier of all such unrolling factors yields the global unrolling factor lcm_j that is *necessary* for static elimination of all internal modulo conditions. To enable unrolling, a new time dimension is introduced by strip-mining by lcm_j. This new dimension scans the same points as the old time dimension, with the additional property that its first iteration is divisible by lcm_j, thus achieving the required *synchronization of all statements to a statically known pattern*. Building on the strip-mining method introduced in Section 5.2.2, the strip-mined loop is actually split into a prologue, a so-called *zero-aligned kernel*, and an epilogue. By construction, the zero-aligned kernel has the important property that *its outer strip-mined loop scans multiples of* lcm_j *only*. Thanks to this property, and having fully unrolled the inner strip-mined loop, we may *statically evaluate* the remainder of the division of the *inner strip-mined loop's iterator* by lcm_j. Applying this systematically to all depths where lcm_j is greater than 1 allows all modulo conditions to be removed *from the zero-aligned kernel only*.

```
RemoveModuloGuards: removes modulo conditionals from loop kernels
Input:
  node: AST root node
  depth: the depth of the modulo conditional
Output: an AST without modulo conditionals in loop nest kernel

  nodelist ← empty list
  while node has successors
    if node is a for loop
1     compute $\text{lcm}_{\text{depth}}$
      if $\text{lcm}_{\text{depth}} > 1$
2       kernel.inner ← new time dimension between $t_{\text{depth}}$ and $t_{\text{depth}+1}$ with constraints
          $\text{lcm}_{\text{depth}} \times t_{\text{depth}} \leq t_{\text{new}} \leq \text{lcm}_{\text{depth}} \times t_{\text{depth}} + (\text{lcm}_{\text{depth}} - 1)$
3       Update all the statement informations (domains and ISMs) with the new dimension
4       Strip-mine and partition node.domain in prologue, zero-aligned kernel, and epilogue
5       Enqueue prologue, kernel and epilogue to nodelist
6       Unroll kernel with respect to $t_{\text{new}}$
7       RemoveModuloGuards(kernel.inner.inner, depth+2)
      else
8       RemoveModuloGuards(node.inner, depth+1)
    else node is a statement
9       Prune node off the AST if needed
    node ← node.next

  return nodelist
```

Figure 5.15: RemoveModuloGuards Algorithm

The algorithm in Figure 5.15 describes how to introduce new time dimensions and unroll them so as to eliminate modulo conditions. Step 9 is actually not trivial. When reaching the leaves of the AST, we need to determine which modulo guards have been simplified, which ones are still necessary and which ones have become unfeasible. Having strip-mined (and unrolled) by the factor $\text{lcm}_{\text{depth}}$, we have forced newly created time iterators on the path to the innermost kernel to be divisible by λ_i. If all the components of an ISM line i are divisible by λ_i, then the modulo condition is always true and needs not to be printed. If all the components are divisible by λ_i but not the constant part, the modulo condition is always false and the statement should be pruned. In the last case, the modulo condition for line i needs to be printed, but at least its expression simpler (and faster to evaluate) than it would have been without strip-mining and unrolling.

Had we wished to fully unroll and had we used versioning, we could have generated an unreasonable number of versions (up to the factorial of $\text{lcm}_{\text{depth}}$). Our algorithm manages to fully unroll the kernel only, where most computation time is spent, while the prologues and epilogues (with modulo conditions) hold at most $\text{lcm}_{\text{depth}} - 1$ iterations.

When the value of *constant parametric shift component* C modulo $\text{lcm}_{\text{depth}}$ is not statically known, it is impossible to statically determine an interleaving pattern. Synchronizing the values of time iterators modulo $\text{lcm}_{\text{depth}}$ does not help and even leads to the insertion of internal modulo conditions. Nonetheless, one can argue on the interest of schedules that do not exhibit a regular pattern: the interleaving of statements itself totally changes with the values of parameters, hence is intrinsically tied to multi-versioning.

Mapping Simplification

Once a transformation has been computed to optimize a program, it is possible to find another, equivalent transformation, such that the relative order between the various iterations is not modified [Vas07, CB-Bas04b]. A simple example is the *shifting* transformation: if all the iterations of all statements are shifted in the same way, the relative order is not changed, thus the original and the shifted transformation are equivalent. The R-Stream compiler (see Appendix A) uses this property to iteratively compute an equivalent transformations (based on compositions of shifting and skewing transformations) in order to simplify the subscript functions of the array accessed inside a loop. This phase relies on a cost model to find the best simplification. This may be critical for some target architecture. For instance on ClearSpeed, there is a significant penalty when an expression involves the processing element number [Cle08b, Cle08a]. R-Stream extends its cost model to avoid such expressions. As an example, Figure 5.16 shows the internal tile kernel of a matrix multiplication after mapping by R-Stream, before simplification in Figure 5.16(a), where `PROC0` stands for the processing element number. Thanks to the simplification, it can be transformed to the one in Figure 5.16(b).

```
parallel for (t = 384*i+4*PROC0; t <= 384*i+4*PROC0+3; t++)
  parallel for (m = 8*j; m <= 8*j+7; m++)
    reduction for (n = 64*k; n <= 64*k+63; n++)
      C_t[-384*i+t-4*PROC0][-8*j+m] += A_t[-384*i+t-4*PROC0][-64*k+n] *
                                       B_t[-8*j+m][-64*k+n];
```

(a) Internal Kernel Without Simplification

```
parallel for (t = 0; t <= 3; t++)
  parallel for (m = 0; m <= 7; m++)
    reduction for (n = 0; n <= 63; n++)
      C_t[t,m] += A_t[t,n] * B_t[m,n];
```

(b) Simplified Internal Kernel

Figure 5.16: Example of Transformation Simplification For Matrix-Multiply Internal Kernel

Full-Tile Extraction & Normalization

When applying the tiling optimization [Wol87, IT88], it may not always be possible to generate tiles of the same shape. This may happen because either the original iteration domain is complex or the algorithm used to enable tiling cannot guarantee this property. When tiles have to be distributed between the processing elements of an hardware accelerator such as a GPGPU or ClearSpeed, it may cause performance penalty because a processing element (PE) may have to execute a tile of a different shape than other processing elements, i.e., introducing costly expressions involving the processing element coordinates. Using ClearSpeed terminology, PE coordinates are *poly variables*, and expressions using them are *poly expressions* [Cle08a].

Such a situation is illustrated in Figure 5.17. The figure shows the polynomial multiplication kernel of Figure 2.1 with a rectangular iteration domain (Figure 5.17(a)) that requires a transformation of its iteration space called *skewing* to expose parallelism (Figure 5.17(b)). Then we use the extracted parallelism to create blocks of workload to be distributed across the processing elements thanks to the tiling transformation, in our example for the 96 PEs of the ClearSpeed CSX700 architecture [Cle08b, CB-BVL$^+$09]. The tiled code is shown in Figure 5.17(c) (for clarity reasons we keep the outer loop on the PEs visible and we do not show explicit memory transfers). The outermost loop on the iterator `i` is the parallel loop on the 96 PEs. Hence `i` is a poly variable and all expressions using `i` are poly expressions. It follows that due to a snowball effect, all loops in this tiled code are inefficient poly loops.

The reason for this situation is that the same code is used to perform the workload of the full tiles (that have a fixed size determined by the compiler) as well as the partial tiles. Because the compiler knows which loops iterate over the tiles and which loops actually achieve the workload inside a tile, it is possible to separate the processing of the full tiles and the partial tiles.

This separation is done at the code generation step. For each statement, we start by determining a set of conditions (a lower bound and an upper bound) on each intra-tile dimension that ensures a constant number of iterations are spanned by the given intra-tile dimension. We perform this extraction for each intra-tile dimension. We subsequently perform projections to derive the necessary conditions exclusively on the inter-tile dimensions to derive a full tile. These necessary conditions are attached to each statement as a predicate at the start of polyhedra scanning. When the first inter-tile dimension is reached during scanning, the predicates for all statements are combined and separated. The combined predicates are intersected into each statement's domain and a clone of the statement is created with the negation of this predicate. Polyhedral domain simplification occurs to cleanup redundancies. After this transformation, the trip count of every intra-tile loop is constant but their bounds may still depend on the PE number to reflect their position in the original iteration space. A last step of normalization to 0 moves the offset to the statement expressions and allows the intra-tile loops to be non-poly.

Applying this code generation scheme to our example leads to the code in Figure 5.17(d). In this code, there are two parts. The first one corresponds to the full tiles where the inner loops (intra-tile) have constant bounds and are non poly. The second part is devoted to scanning the partial tiles and have poly intra-tile loops. Unfortunately because of the normalization, the array subscripts in the statements are more complex. However (without other positive memory transfer effects), the final code in Figure 5.17(d) is 30% faster than the one in Figure 5.17(c). The construction of full-tiles has already been used, for instance when considering parametric tiling [CB-HBB$^+$09], however, to the best of our knowledge, this is the first solution proposed as a code generation step in the polyhedral model and applied to the control overhead minimization problem.

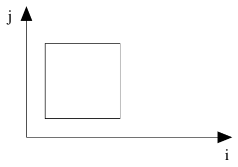
5.2.3 Putting it All Together

Let us combine all the previous optimizations and summarize the total improvements in code generation time, memory usage and generated code size. To further stress the scalability of our tool, we added a more complex optimization of the `Swim` benchmark, called `Swim+`, in its most general setting with 5 parameters (without context). Those results present strong and consistent improvements on code generation time and memory footprint while reducing the generated code size significantly.

5.2.4 Related Work

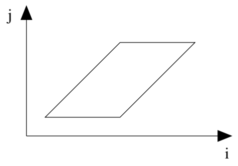
The history of code generation in the polyhedral model shows a constant growth in transformation complexity, from basic schedules for a single statement to general affine transformations for wide code regions. In their seminal work, Ancourt and Irigoin limited transformations to unimodular functions (the transformation matrix has determinant 1 or -1) and the code generation process was applicable for only

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    z[i+j] += x[i] * y[j];
```



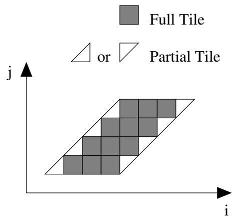
(a) Original Kernel with Rectangular Iteration Domain

```
parallel for (i = 0; i <= 2*N-2; i++)
  for (j = max(0, i-N+1); j <= min(N-1, i); j++)
    z[i] += x[j] * y[i - j];
```



(b) Extraction of a Parallel Loop by Iteration Domain Skewing

```
parallel for (i = 0; i <= 95; i++)
  parallel for (j = (95 - i) / 96;
                j <= floord(-116 * i + N - 1, 11136); j++)
    for (k = max((232*i + 22272*j - N + 1) / 152, 0);
         k <= min(floord(N - 1, 152),
                  (29*i + 2784*j + 2821) / 19 - 147); k++)
      parallel for (l = max(232*i + 22272*j, 152*k);
                    l <= min(2*N + -2, 152*k + N + 150,
                             232*i + 22272*j + 231); l++)
        for (m = max(152*k, l - N + 1);
             m <= min(N - 1, 152*k + 151, l); m++)
          z[l] += x[m] * y[l - m];
```



(c) Tiling of 232x152 With an Outer Loop On the PEs Number

```
parallel for (i = 0; i <= 95; i++) {
  parallel for (j = (95 - i) / 96; j <= floord(-52*i + N + -1, 4992); j++) {
    for (k = max(0, (104*i + 9984*j - N + 1) / 152) ;
         k <= min(floord(N - 1, 152), (13*i + 1248*j + 1285) / 19 - 67); k++) {
      if (-104*i -9984*j + 152*k + N >= 104) {
        if (104*i + 9984*j - N >= -1 && -152*k + N >= 152) {
          parallel for (l = 0; l <= 103; l++) {
            for (m = 0; m <= 151; m++) {
              z[l - 104*i - 9984*j] += x[m - 152*k] * y[l - 104*i - 9984*j - m - 152*k];
            }
          }
        }
        if (-104*i -9984*j + N >= 2 && 13*i + 1248*j -19*k >= 19) {
          parallel for (l = 0; l <= 103; l++) {
            for (m = 0; m <= 151; m++) {
              z[l - 104*i - 9984*j] += x[m - 152*k] * y[l - 104*i - 9984*j - m - 152*k];
            }
          }
        }
      }
      if (!(-152*k + N >= 152 && -104*i -9984*j + 152*k + N >= 104 &&
            104*i + 9984*j - N >= -1 || 13*i + 1248*j - 19*k >= 19 &&
          -104*i - 9984*j + N >= 2 && -104*i -9984*j + 152*k + N >= 104)) {
        parallel for (l = max(152*k, 104*i + 9984*j);
                      l <= min(104*i + 9984*j + 103, 2*N - 2, 152*k + N + 150); l++) {
          for (m = max(l - N + 1, 152*k); m <= min(N - 1, 152*k + 151, l); m++) {
            z[l] += x[m] * y[l - m];
          }
        }
      }
    }
  }
}
```

(d) Code After Full Tile Extraction & Normalization

Figure 5.17: Full Tile Extraction & Normalization for Polynomial Multiplication

	Time			Space			Code size		
Benchmark	Orig.(s)	Opt.(s)	Speedup	Orig.(KB)	Opt.(KB)	Reduction	Orig.(Lines)	Opt.(Lines)	Reduction
Swim	41.20	2.41	17.09×	17480	2380	7.34×	830	764	1.09×
Swim+	1219.67	21.62	56.41×	322624	22180	14.55×	17791	12041	1.48×
QR	19.47	2.42	8.05×	3012	988	3.05×	4733	1432	3.33×
Classen	1.12	0.25	4.48×	1092	272	4.01×	130	105	1.24×
DreamupT3	0.49	0.20	2.45×	160	160	1.00×	382	68	5.62×

Figure 5.18: Summary of experimental results

one domain at once [AI91]. Several works succeeded in relaxing the unimodularity constraint to invertibility (the transformation matrix has to be invertible), enlarging the set of possible transformations [LP94, Xue94, DR94, Ram95]. They all used the Hermite Normal Form [Sch86] to avoid scanning integers points that have no corresponding points in the initial space. Griebl, Lengauer and Wetzel [GLW98] relaxed the constraints of code generation further to transformation matrices with non-full rank, relying on a completion algorithm, and also presented preliminary techniques for scanning several polyhedra using a single loop nest. A further step has been achieved by Kelly et al. by considering more than one domain and multiple scheduling functions at the same time [KP95]. This methods has been recently revisited by Chen to reduce significantly the control overhead in generated codes [Che12].

All these methods relies on the Fourier-Motzkin elimination method [Sch86] to build the target code. Two alternative have been suggested. First, Boulet and Feautrier presented a code generation technique based on Parametric Integer Programming [BF98]. Second, Le Verge, Van Dongen and Wilde showed how to use polyhedral operations based on the Chernikova Algorithm [Le 92] instead, to benefit from its practical efficiency to handle bigger problems. Quilleré et al. showed a generalization of this work to several polyhedra [QRW00], known as the QRW algorithm. I presented a transformation policy to allow general non-invertible, non-uniform, non-integral affine transformations and several extensions to the QRW algorithm to build a robust implementation of this algorithm and to minimize generated control overhead [CB-Bas03, CB-Bas04a]. Further extensions have been proposed by Vasilache et al. [CB-VBC06]. Such freedom allowed to apply polyhedral techniques to much larger programs with very sophisticated transformations, and led to novel complexity, scalability and code quality challenges discussed in this chapter.

5.3 Conclusion

The polyhedral model is a powerful framework to reason about high level loop transformations. However, it relies on worst-case exponential algorithms which may hamper its use in production compilers or may prevent the computation of a (good enough) solution in a reasonable amount of time. In this chapter we addressed two critical parts of a polyhedral compilation framework: data dependence anaysis and code generation.

Instance-wise array dependence analysis computes a finite representation of the set of all pairs of dependent iterations of all statements. This problem has always been considered non-scalable or an overkill compared to less expressive but faster dependence tests. We presented technical contributions to instance-wise array dependence analysis, and compelling experimental evidence of its scalability through the validation on full SPEC CPU2000 benchmarks. This relieves the compiler of the expensive and cumbersome task of implementing specific legality checks for each single transformation. It also allows, in the case of invalid transformations, to precisely determine the violated dependences that need to be corrected.

For a long time, scheduling and placement techniques were many steps forward code generation capabilities. In 1992, Feautrier provided a general scheduling technique for multiple polyhedra and general affine functions [Fea92a]. At this time, the only code generation algorithm available had been designed in 1991 by Ancourt and Irigoin and supported only one polyhedron and unimodular scheduling functions [AI91]. Some scheduling functions had to wait for nearly one decade to be successfully applied by a code generator. Recent advances on code generation algorithms [QRW00, CB-Bas04a] made it possible to compute the target code for hundreds of statements while this code generation step was expected not to be scalable [GLW98]. Unfortunately, these improvements allowed the exploration of larger, more complex optimization and parallelization problems, which in turn raised several scalability and code quality challenges. We presented scalable code generation methods that make possible the application of complex program transformations to real-world computation kernels with up to 199 statements. By studying the transformations themselves, we show how it is possible to benefit from their properties to dramatically improve both code generation quality and space/time complexity. Moreover, building on these algorithmic improvements, we proposed new techniques to generate more efficient code for complex mappings involving, e.g., non-unit strides or tiling.

We believe these improvements (already available in some tools, like R-Stream, CLooG and Candl, see Appendix A) are initiating another virtuous cycle towards allowing polyhedral techniques to bring dramatic improvements in the effectiveness of optimizing and parallelizing compilers.

Chapter 6

Applicability: Beyond Static Control

The ability to perform complex loop nest restructuring is required for optimizing and parallelizing tools, to cope with the complexity of modern architectures. The widespread adoption of multicore processors and massively parallel hardware accelerators (GPUs) urges production compilers to provide such capability. The polyhedral model has demonstrated its potential to achieve portability of performance over a variety of targets. So far, these successes have been limited to static-control, regular loop nests.

Compilers based on the Polyhedral model, including recent research tools like PoCC [Pou10] or Pluto [BHRS08] or CHiLL [CCH08], target code parts that exactly fit the affine constraints of the model. Only loop nests with affine bounds and conditional expressions can be translated to a polyhedral representation. The reason behind this limitation is not that exact dependence analysis is required to make use of the polyhedral model, but rather that there is no general scheme to support dynamic control flow in the program transformation and code generation algorithms. To fight a common misunderstanding, *the power of the polyhedral model is not to achieve exact data dependence analysis, but to implement compositions of complex transformations as a single algebraic operation, and to model these transformations in a convex optimization space* [Fea92b, Lim01, CB-GVB+06, CB-PBB+10, CB-PBB+11].

In this chapter, we expand the application domain of the polyhedral model. We present slight extensions to the representation itself, based on the notions of *exit* and *control* predicates that allow to consider general `while` loops and `if` conditions. We revisit the whole framework, from input code analysis to output code generation, while taking care of preserving expressiveness and flexibility. We present experimental evidence that this extended framework offers new optimization opportunities for existing optimization algorithms, and opens the door to novel techniques targeting full functions.

The chapter is organized as follows. Section 6.1 introduces extensions to the classical polyhedral representation of programs to support irregular control flow. Section 6.2 revisits the polyhedral model to target full functions, from analysis to code generation. Section 6.3 discusses control overhead and some solutions. Section 6.4 presents experimental results in the extended framework. Section 6.5 discusses related work, before the conclusion in Section 6.6.

6.1 Relaxing the Static Control Constraints

The program model we target is general functions where the only control statements are `for` loops, `while` loops and `if` conditionals. This means function calls have to be inlined[1] and `goto`, `continue` and `break` statements have been removed thanks to some preprocessing. To move from static control parts to such general control flow we need to address two issues: (1) modeling loop structures with arbitrary bounds (typically `while` loops); and (2) modeling arbitrary conditionals (typically data-dependent ones). In both cases, it implies to not be anymore able to exactly characterize statically the iteration domain of statements, which remains the privilege of Static Control Parts.

First, we demonstrate that it is possible to express safe over-approximations of the iteration domains to allow the construction of a polyhedral representation in the case of arbitrary control-flow.

6.1.1 Modeling Arbitrary Loop Structure

Any arbitrarily iterative structure such as `for` loops with non-affine bounds or `while` loops is amenable to polyhedral representation. As explained in Section 2.1.2, the iteration domain of a statement is a subset of $\mathbb{Z}^n$. The convex hull of all executed instances of any statement, even with a non-polyhedral iteration domain, is a subset of $\mathbb{Z}^n$. Thus, an over-approximation that fits the polyhedral model for the iteration domain of any statement enclosed in a non-static loop is $\mathbb{Z}^n$ itself. We actually choose to over-approximate it as $\mathbb{N}^n$ to match the standard loop normalization scheme, represented by the non-negative half-space polyhedron[2]. This translates to over-approximate any non-static loop with a static loop iterating from 0 to infinity. Such over-estimate have been used in the same way by Griebl and Collard for `while` loop parallelization [GC95].

To guarantee that the program semantics will be preserved, we introduce an **exit predication** statement which bears the loop bound check. This statement is executed at the beginning of any iteration of the infinite loop, and exits the loop thanks to a `break` instruction if the loop conditional is no longer satisfied. This is summarized in Figure 6.1: we consider the original code in Figure 6.1(a) as the equivalent code in Figure 6.1(b) with the exit predicate `ep(i)`. As shown on the figure, the exit predicate does not depend on i on the equivalent code. However, we write `ep(i)` for consistency with the mathematical representation. In the case of arbitrary `for` loops, initialization statements are inserted just before the loop and at the end of the loop body for the increment. Note that all statements in the body of the loop depend on the exit predication statement.

Formally, an exit predicate `ep(i)` is a non-affine constraint such that there exists a value v such that `ep(i)` is true for $i < v$ and false otherwise. Each statement S has a set of exit predicates, $\mathcal{E}_S$. The exit predicate is attached to the iteration domain of the predicated statements as illustrated in the example in Figure 6.1(c).

6.1.2 Modeling Arbitrary Conditionals

We apply a similar reasoning to represent non-affine conditionals. To model such a conditionally executed statement in the polyhedral representation we decouple the regular part of the iteration domain and the irregular conditional. Again, the polyhedral iteration domain is over-approximated and we need

[1]Pure functions (without side effects) can be called without being inlined but they will not be optimized using our approach.

[2]This is reported on the constraint part so we accept values in $\mathbb{Z}^n$ for consistency with the relation definition shown in Section 2.1.1, page 17.

```
while (condition) {
  S();
}
```

(a) Original Code

```
for (i = 0;; i++) {
  ep(i) = condition;
  if (ep(i))
    S(i);
  else
    break;
}
```

(b) Equivalent Code

$$\mathcal{D}_S() = \left\{ () \to \left(\; i \; \right) \in \mathbb{Z} \middle| ep(i) \in \mathcal{E}_S, \left[\; 1 \; \vdots \; 0 \; \right] \begin{pmatrix} i \\ \hdashline 1 \end{pmatrix} \geq \vec{0} \wedge ep(i) \right\}$$

(c) Iteration Domain of S

Figure 6.1: Exit predication

to ensure the semantics is preserved. To do so we introduce a **control predication** which consists in predicating individually each statement dominated by the non-static conditional by its condition (similar to if-conversion).

Formally, a control predicate `cp(i)` is a non-affine constraint that may be true or false depending on i. Each statement S has a set of control predicates, $\mathcal{C}_S$. This is summarized in Figure 6.2: we consider the code in Figure 6.2(a) as the equivalent code in Figure 6.2(b) with the control predicate `cp(i)`. This predicate is attached to the iteration domain of the predicated statements as shown in Figure 6.2(c).

```
for (i = 0; i < N; i++) {
  if (condition(i))
    S(i);
}
```

(a) Original Code

```
for (i = 0; i < N; i++) {
  cp(i) = condition(i);
  if (cp(i))
    S(i);
}
```

(b) Equivalent Code

$$\mathcal{D}_S(N) = \left\{ () \to \left(\; i \; \right) \in \mathbb{Z} \middle| cp(i) \in \mathcal{C}_S, \left[\begin{array}{r:r:r} 1 & 0 & 0 \\ -1 & 1 & -1 \end{array} \right] \begin{pmatrix} i \\ \hdashline N \\ \hdashline 1 \end{pmatrix} \geq \vec{0} \wedge cp(i) \right\}$$

(c) Iteration Domain of S

Figure 6.2: Control predication

Being able to safely describe (from the iteration domain point of view) the convex hull of the dynamic control flow is only the first step towards supporting full functions. The following section presents necessary and sufficient modifications of the framework that allow to *transform* general codes with polyhedral techniques. Our goal is to show that, provided a suitable dependence analysis (static, dynamic or both), only the code generation step needs to be altered to enable any polyhedral optimization technique on full functions.

6.2 Revisiting the Polyhedral Framework

Restructuring programs using the polyhedral model is a three steps framework. First, the Program Analysis phase aims at translating high level codes to their polyhedral representation and to provide data dependence analysis based on this representation. Second, some optimizing or parallelizing algorithm use the analysis to restructure the programs in the polyhedral model. This is the Program Transformation step. Lastly, the Code Generation step returns back from the polyhedral representation to a high level program. Targeting full functions requires revisiting the whole framework, from analysis to code generation.

6.2.1 Program Analysis

Once a function has been translated to the polyhedral model with the predicate extensions described in Section 6.1, data dependence analysis must be performed. Two statements are said to be in dependence if they access the same memory reference, and at least one of these accesses is a write. When restricting the study to SCoPs and to array references with affine subscripts — we talk about *static references* — it is possible to compute on which instance (iteration) of a given statement any other instance depends as detailed in Section 2.2.2 [Fea91, Fea92a].

As we broaden the set of handled programs, we have to deal with dynamic behavior (e.g., `while` loops) and structural complexity (e.g. subscript of subscript, as in `A[B[i]]`). As a result, an exact analysis is no more possible statically. Instead, we rely on a *conservative* policy, over-estimating data dependences, preventing some optimizations when semantics safety is unsure.

Conservative policies are widely used in compilation to achieve an approximate analysis of programs without slowing down the compiler. GCD-test [Ban76] or I-test [KKP90] are popular examples of such analysis for array references: they can state thanks to a fast GCD computation that two references do not depend on each other, then safely consider a dependence relation exists otherwise (for instance, GCC 4.4 relies on a multi-dimensional GCD-test for production and on a more costly but exact Omega-test [Pug91a] for testing). When dedicated preprocessing techniques fail to simplify complex array references (typically subscript of subscript or linearized subscripts) it is usual to consider the reference as an access to a single variable, i.e., to suppose that the whole array is read or written. In the same way, when *array recovery* fails to translate pointer-based accesses to explicit array references [FO03], it is usual to consider a dependence between the pointer access and every previously accessed references. Overall, it is possible to handle any kind of data access in a conservative way.

A conservative approach for irregular data dependence analysis is adding new statements or new statement iterations because the only effect is adding extra data dependences, without modifying pre-existing data-dependences[3]. It is possible to add any additional statement, as long as it does not modify immediately the control flow (as `break`, `continue` or `goto` statements). Therefore for data dependence analysis, it is safe to consider that irregular conditions (from `while` loops as well as `if` conditionals) are always true. A convenient data dependence analysis for our purpose is described by Feautrier [Fea92a, Fea92b]. This transformation is not compatible with all analyses because assuming predicates are always true may not be conservative. For instance, it is not convenient for dead code analysis: in the example in Figure 6.3(a), if both branches are considered to be executed, the first branch would be considered dead (data are totally over-written by the second branch). In the same way,

[3]As long as they are not considered to simplify pre-existing data dependences, e.g., when removing transitively covered data dependences.

Feautrier's data-flow analysis [Fea91] that relies on last writer computation is not directly suitable for our conservative approach.

In this chapter, we transform the program control structures in such a way we only have to deal with `for` loops, `if` conditionals and infinite `for` loops that fits the polyhedral model. Irregularity has been spread thanks to control and exit predicates to the iteration domains of irregular-control-surrounded statements. One can achieve a naive but simple conservative analysis by considering an altered representation of the input irregular program called *abstract program*. We build this representation from the original program in this way:

1. Introduce control and exit predicates as described in Section 6.1.
2. Predicate evaluations are considered as statements that write the predicate, and read the necessary data to compute the predicate.
3. Irregular data accesses are modeled conservatively (an array with a complex subscript is considered as a single variable).
4. Predicated statements are considered to read their predicates.

Writing and reading predicates ensure the semantics is preserved when a statement modifies an element necessary for the predicate evaluation. Ultimately we may perform on this representation usual data dependence elimination techniques like array privatization [AK02] then exact data dependence analysis [Fea92a].

We illustrate the construction of the abstract program for conservative data dependence analysis in Figure 6.3. The considered program in Figure 6.3(a) is an optimized version of the Outer Product Kernel in the case one vector contains some zeros. The conditional introduces irregular control flow that usually prevents considering such kernel in the polyhedral model (only the inside of the loop i could be processed, considering i and $x[i]$ as parameters). The first step is to introduce a control predicate and to attach it to the predicated statements. The predicate evaluation is considered as a new statement as shown in Figure 6.3(b). Lastly, we consider the value of the predicate is read by each predicated statement and that the predicate is always true for conservative data dependence analysis as shown in Figure 6.3(c). Figure 6.3(c) presents the information sent to the data dependence algorithm (everything is regular): for each statement, its iteration domain and the sets of written and read references. We may use well known techniques to remove some dependences. In this example we can privatize `p` to remove loop-carried dependence and parallelize the code or even interchange the loops using existing polyhedral techniques.

Discussion

Previous work aim at providing less naive and conservative solutions to avoid, as much as possible, to consider additional dependences. Griebl and Collard proposed a solution in the context of `while` loops parallelization, focusing on control flow [GC95]. Collard et al. extended this approach to support complex data references [CBF95]. Other techniques aim at removing some dependences as, e.g., Value-based Array Data Dependence Analysis [PW93], Array Region Analysis [CI96] Array SSA [KS98] or Maximal Static Expansion [BCC98]. These techniques would expose their full potential in the context of manipulating full functions in the polyhedral model to minimize the unavoidable conservative aspects. Combining these static analyses with dynamic dependence tests [RP95, RRH03, RR03, RPR07, RVRA08] into hybrid polyhedral/dynamic analyses remains to be investigated.

```
for (i = 0; i < N; i++) {
  if (x[i] == 0) {
    for (j=0; j < M; j++)
      A[i][j] = 0;
  }
  else {
    for (j=0; j < M; j++)
      A[i][j] = x[i] * y[j];
  }
}
```

(a) Outer product kernel

```
for (i = 0; i < N; i++) {
  p = (x[i] == 0);
  for (j=0; j < M; j++) {
    if (p)
      A[i][j] = 0;
  }
  for (j=0; j < M; j++) {
    if (!p)
      A[i][j] = x[i] * y[j];
  }
}
```

(b) Using a control predicate

```
for (i = 0; i < N; i++) {
  S0: Written = {p}, Read = {x[i]}
  for (j=0; j < M; j++)
    S1: Written = {A[i][j]}, Read = {p}
  for (j=0; j < M; j++)
    S2: Written = {A[i][j]}, Read = {x[i], y[j], p}
}
```

(c) Abstract program for conservative data dependence analysis

Figure 6.3: Abstract program representation for the irregular outer product

6.2.2 Program Transformation

A (sequence of) program transformation(s) in the polyhedral model is represented by a set of mapping relations (see Section 2.1.1), most of the time restricted to affine functions, one for each statement, called scheduling, allocation, chunking, etc. depending on the technique. Mapping relations depend on the counters of the loops surrounding their corresponding statement; they map each run-time statement instance to a logical execution date. The literature is full of algorithms to find such relations dedicated to parallelization, data locality or global performance improvement [Fea92b, Lim01, Gri04, BHRS08], see Chapter 4 for an iterative technique to discover optimizing mappings. Our approach allows to reuse most existing techniques based on the polyhedral model and multi-dimensional mapping *directly*.

However, managing `while` loops, that are translated into unbounded `for` loops requires a slight adaptation to preserve the expressiveness of affine mapping relations. This is particularly important in the context of one-dimensional affine functions, where it is necessary to know the upper bounds of the loops to be able to reorder them. For instance let us consider the pseudo-code in Figure 6.4(a) composed of two loops enclosing two statements, `S1` and `S2`. To implement a transformation such that the loop enclosing `S2` will be executed before the loop enclosing `S1`, we need the logical dates of the instances of `S1` to be *higher* than those of the instances of `S2`. Such transformation may be implemented by the mapping functions $\theta_{S1}(i) = i + Up2$ and $\theta_{S2}(i) = i$. In these functions, the i part ensures the instances of a given statement are executed in the same order as in the original code, and the upper bound *Up2* of the second loop is used to ensure the loop of `S1` *starts* after the end of the loop of `S2`. The target code is shown in Figure 6.4(b), where variable t represents logical time.

In this work, we may consider `for` loops with no upper bounds. It is not possible in this way to reorder those loops respectively to other loops (bounded or unbounded) using one-dimensional schedules

```
for (i = 0; i < Up1; i++)
  S1(i);
for (i = 0; i < Up2; i++)
  S2(i);
```

(a) Original program

```
for (t = 0; t < Up2; t++) {
  i = t;
  S2(i);
}
for (t = Up2; i < Up2 + Up1; i++) {
  i = t - Up2;
  S1(i);
}
```

(b) Loop reordering with mapping $\theta_{S1}(i) = i + Up2$ and $\theta_{S2}(i) = i$

Figure 6.4: Loop reordering using one-dimensional mapping

only.[4] Thus, we introduce a virtual parametric upper bound `W`, the same for all unbounded `for` loops with the constraint that `W` is strictly greater than all upper bounds of bounded `for` loops. The `W`-parameter will be considered during the program transformation and code generation steps. It will be removed during a dedicated stage of code generation as detailed in Section 6.2.3. This parameter has to be chosen strictly greater than other loop bounds to ensure a fusion between a bounded and an unbounded loop will always be partial (hence the code generation step will always be able to re-create the unbounded part). A single `W`-parameter for multiple unbounded loops is enough to be able to reorder them relatively to each other by using coefficients of this parameter (e.g., to reorder three unbounded loops, we can use mapping functions like $\theta_{S1}(i) = i$, $\theta_{S2}(i) = i + W$ and $\theta_{S3}(i) = i + 2W$). The `W`-parameter allows to reuse any of the existing algorithms supporting parameters to compute mapping functions in our irregular context.

6.2.3 Code Generation

Once a transformation (i.e., a mapping relation) has been computed by an optimization heuristic, applying it in the polyhedral model is straightforward and leads to a new coordinate system for each iteration domain [CB-Bas04a]. The last step consists in translating the transformed program from its polyhedral representation back to a syntactic representation. This phase amounts to finding a set of nested loops visiting each integral point of each polyhedron once and only once. This is a critical step in the polyhedral framework since the final program effectiveness highly depends on the target code quality. In particular, we must ensure that a bad control management does not spoil performance, for instance by producing redundant conditions, complex loop bounds or under-used iterations. On the other hand, we have to avoid code explosion typically because a large code may pollute the instruction cache.

Among existing methods to scan polyhedra and generate code, the extended QRW algorithm is considered now as the most efficient algorithm [QRW00, CB-Bas04a]. This algorithm is not able in its original form to generate semantically correct code for our extended polyhedral representation, as special care is needed to handle properly predicates and their impact on the generated control-flow. Nevertheless, it is possible to extend this algorithm to scan and generate regular codes corresponding to the over-estimates of the iteration domains then to post-process its output to guarantee semantically correct code generation. See Section 2.3.3 for details about this algorithm.

Previous approaches to model irregular codes (see Section 6.5) were based on complex representations that did not allow any easy modification of the extended Quilleré et al. algorithm to generate the

[4]It is easy to remove the limitation using more dimensions, but several algorithms to compute mapping functions are based on one-dimensional mapping only, and some others rely on the full expressiveness of each dimension.

code. Instead they rely on ad-hoc, mostly syntactic, code generation schemes. By relaxing the static constraints thanks to exit and control predication, we make possible, and even natural, the adjustment of the QRW code generation algorithm. This adaptation takes into account the additional data dependences on control predicates. The price to pay is displacing the problem of modeling data dependent non-affine conditions into legality constraints. There is no alteration of the core QRW algorithm: we apply it on the polyhedral over-estimated iteration domains, leaving predicates attached to each statement. Then we post-process the result to handle the predicates. There are two tasks to perform: (1) to achieve a semantically-correct generation of control predicates and exit predicates, and (2) to reconstruct while loops in the generated code.

Generation of Arbitrary Conditionals Generating arbitrary conditionals is straightforward: the control predicate is available as a statement information, attached to the polyhedral iteration domain. The only task is to generate the `if` instruction containing the predicate around the convenient statement.

Generation of `while` Loop Structure The task of generating `while` loops starts by identifying loops with the `W` parameter introduced in Section 6.2.2 as an upper bound. Next, we have to identify exit predicates corresponding to each `while` loop. Again, this information can be easily extracted because it is attached to the polyhedral iteration domain of each statement that belongs to a `while` loop in the original program.

However, due to the separation step of the extended QRW algorithm, several statements with different exit predicates could be found in the same iteration domain without corresponding to the same `while` loop. So we need to separate these statements and generate the appropriate `while` loops. we distinguish three main cases of separation that involve exit predicates:

1. If all statements of the loop have the same exit predicate, no case distinction is needed during the separation phase. The predicate is therefore considered as the exit predicate of the generated `while` loop. Figure 6.5(a) is an example of such a case.

2. If statements or block of statements have different exit predicates, this means (1) they belong to different `while` loops; and (2) these statements can be executed in any order (the semantics of `while` loops transformations is particular, as discussed later in the section). For this second case, we can proceed to a separation quite similar to the separation of polyhedra in the regular case. More exactly, it consists in scanning the domain where both predicates are true at the same time, thanks to the intersection of two polyhedra, i.e., the space of common points. Then, we scan domains where only one of the two predicates is true, thanks to the differences between polyhedra. Figure 6.5(b) shows separation of while loops based on exit predicates attached to statements `s1` and `s2`.

3. If some statements have exit predicates while some others do not have any, this means a regular `for` loop has been fused with a part of a `while` loop. In such a case, we find a statement with an exit predicate attached to it without identifying the `while` loop (by identifying the `W` parameter). The exit predicate is transformed here into a control predicate plus an exit Boolean (false at the start of the program). Figure 6.5(c) illustrates this case.

Re-injecting irregular control inside the generated code is simply a matter of replacing predicates with their expressions taken from the original source code. Unfortunately, this is likely to bring high control overhead as it is inserted close to the statement, at the innermost level.

```
for (i = 0; i < W; i++) {           while (ep1) {
  S1(i); ep1                    ⇒     S1();
  S2(i); ep1                          S2();
}                                   }
```

(a) Same Exit Predicates

```
                                    while(ep1 && ep2) {
                                      S1();
for (i = 0; i < W; i++) {             S2();
  S1(i); ep1                    ⇒   }
  S2(i); ep2                        while(ep1)
}                                     S1();
                                    while(ep2)
                                      S2();
```

(b) Different Exit Predicates

```
                                    exit = false;
                                    for (i = 0; i < N; i++) {
for (i = 0; i < N; i++) {             S1(i);
  S1(i);                        ⇒     if (ep1 && !exit)
  S2(i); ep1                            S2(i);
}                                     else
                                        exit = true;
                                    }
```

(c) An Exit Predicate Inside A Regular Loop

Figure 6.5: Separation of `while` loops

Discussion

The semantics of transformations involving `while` loops is particular: fusion of such loops should be performed only if the loops can be executed in any order (in Figure 6.5(b), the order of the last two `while` loops is arbitrary) and `while` loop reversal is clearly not supported by our extended framework. Also, when the transformation states the loop may be run in parallel (e.g., no mapping functions means all loops are parallel) it means that, except what is necessary for the predicate evaluation, iterations of the loop may be run in parallel (this allows basic parallelization, e.g., a process devoted to the predicate computation that spread bundles of full iterations to different processors).

6.3 Reducing Control Overhead

The underlying principle of converting programs to the extended polyhedral representation is to conditionally execute statements depending on the value of a given predicate, which is not necessarily statically computable. To put the program into the model, we extensively predicate statements regardless of the control overhead we introduce. We rely on post-pass optimizations to limit this overhead for the generation of efficient code.

We discuss two main optimizations, namely the *computation of the predicate value* and the *placement of control predicates*. A preliminary for those optimizations to be performed is the gathering of the set of *read* and *written* variables, for each statement and each predicate. It means we have to analyze the statement content to extract the variables involved. Obviously, the optimality of our optimization processes is constrained by the accuracy of this analysis. For instance we do not deal yet with control overhead optimizations of codes containing non-strictly aliased pointers, as the problem of pointer aliasing makes very difficult to compute correctly the set of read and written variables.

6.3.1 Computing the Value of Predicates

The main overhead induced by predication is the re-computation of the p predicate when its value has not been modified. To address this problem we decouple the computation stages of the predicate from its evaluation. We first define the set of variables used to compute the predicate value. Let p be a predicate used to guard a statement, $\mathcal{V}_p$ is the set of variables used to compute p. For instance, if we consider the predicate `p = x + 2 * y + b[i]` (where `i` is the *generated* iterator name), then $\mathcal{V}_p = \{x, y, b, i\}$.

The algorithm operates on the generated abstract syntax tree (AST), in a two-step process. The first step consists in identifying the statements in the AST which compute the value of p, for each predicated statement. To guarantee the optimality of the predicate computation placement, we ensure it is not possible to execute `p` less frequently while preserving the program semantics. This is done by putting the statement `p` at the highest tree level such that no statement dominated by `p` modifies any of the variables in $\mathcal{V}_p$. The second step consists in eliminating duplicated predicate computations when a given predicate is used from multiple calling sites. We proceed by inspecting the AST for all `p` statements (involving the same predicate p), and checking if any of the variables in $\mathcal{V}_p$ is ever assigned in any execution path between two occurrences of `p`. If not, then the second occurrence can be safely removed.

As a result of this optimization, the computation of the value of each predicate is minimized in terms of number of executions — again given the accuracy of $\mathcal{V}_p$ computation. The check of the predicate value before each executed instance of a predicated statement is reduced to a simple test instruction over a scalar, as shown in Figure 6.2(b).

6.3.2 Predicate Placement

The second critical optimization is to reduce the number of executed checks on the value of a predicate. To do so, we hoist the conditional `if (p)` to the highest possible level in the AST, provided the location of the computation of `p`. A typical example is the case of all reachable instances of a given loop being predicated by the same p, which is never modified during the loop execution. The instruction `if (p)` can then be hoisted outside the loop, dramatically reducing the control overhead. We proceed by merging under a common conditional all consecutive statements (under the same loop) which involve the same predicate, such that none of the statements modify the predicate value. Then, if all statements inside a loop are under the same conditional and this conditional does not depend on neither the loop iterator nor any of the statements under it, then the conditional can be safely moved around the loop instead. This optimization is reminiscent of classical if-hoisting compiler techniques, and it is efficiently performed as a code generation optimization pass. We extended the code generation tool CLooG [CB-Bas04a] to support these extensions.

6.4 Experimental Results

6.4.1 Methodology and Setup

The extension and the associated algorithms presented in this chapter have been implemented in the *Polyhedral Compiler Collection* framework PoCC (see Appendix A for information about tools). Specifically, the implementation consisted in upgrading two modules: the *ir*regulat extension of the polyhedral model has been implemented in *ir*Clan, an extended version of the Clan tool to extract the polyhedral representation, and in *ir*CLooG built on the code generator CLooG.

To show the impact of our approach, we illustrate it with two of the state-of-the-art polyhedral optimizers.

- LeTSeE is a complete platform for iterative compilation in the polyhedral model [CB-PBCC08a] presented in Chapter 4. It leverages the algebraic properties of the polyhedral model to build an expressive search space of affine schedules, encompassing only *legal and distinct* program versions. It uses multiple heuristics to prune and search for a best program version within this space. Its optimization goal is fine-grain parallelism for vectorization and locality enhancement, see Appendix A.
- Pluto is an automatic parallelization tool based on the polyhedral model [BHRS08]. It optimizes for coarse-grain parallelism and locality simultaneously, looking for complex affine transformations based on rectangular time-tiling [Gri04] and fusion. OpenMP parallel code can be automatically generated from sequential C, together with finer grain register tiling and transformations to enable automatic vectorization, see Appendix A.

Our goal is to experiment these existing optimization tools *without any modification*, demonstrating the effectiveness of our extended approach on a set of irregular benchmarks. We also compare the performance improvements considering only the regular parts of these programs, when applicable. Notice that we did not implement a sophisticated analysis of the predicates themselves or a dynamic parallelization scheme; this may significantly reduce conservativeness and allow to find better transformations. Hence, we consider the following results as a *lower bound* of the extended framework's potential.

Our experimental setup is a 2-socket Intel Quad-Core E5430 at 2.66 GHz with 16 GB of RAM, running Linux. We used the ICC compiler version 11.0, the best performing compiler on the benchmarks considered. All programs were compiled with `icc -fast -parallel -openmp` (i.e., the baseline includes automatic parallelization in ICC). Because our goal is to demonstrate the relevance of the extended model and not to present a new optimizing technique, we did not investigate various architectures.

6.4.2 Results

We studied typical kernels solving real computational problems that are not (partially or totally) amenable to standard polyhedral representation because of control flow irregularities. 2strings is a program counting the occurrences of two different strings in another string. It features a very data-dependent `while`-loop typical of search and pattern-matching programs. sat-add is a saturated addition of two images deblurred thanks to two stencil-based filters. It represents an example of saturated arithmetic, a very common source of irregularity in numerical or image processing programs. QR is a QR decomposition computed by Householder reflections on real data, featuring dynamic control flow in outer loops like the outer product example in Figure 6.3 page 106. Other forms of outer loop irregularity are exhibited in two additional benchmarks: ShortPath and TransClos, respectively a shortest-path and a transitive closure kernel based on adjacency matrices. We also provide larger loop nests to exercise search space construction and code generation scalability: the Givens benchmark computes the R matrix of the QR decomposition using Givens rotations on complex numbers; Dither is a kernel for error-distribution dithering; Svdvar computes a covariance matrix; Svbksb solves $A\vec{x} = B$ for a vector $\vec{x}$ where A is on a singular value decomposition; Gauss-J is a Gauss-Jordan elimination finding a maximum pivot, pivoting being a relevant source of data-dependent control flow; and PtIncluded checks if an integer point is included in a polyhedron, involving a linked list traversal, another usual source of control-flow irregularity.

Figure 6.6 lists the main properties of these programs: their number of loops, their number of array references, the maximum loop depth, the maximum loop depth of strictly affine SCoPs in the program (to quantify the extra expressiveness offered by our extension), and the data-set size.

	#loops	#refs	Max Depth	SCoP Depth	Data Size
2strings	4	15	2	0	1M
Dither	2	12	2	0	1024x1024
Gauss-J	4	14	2	1	1024x1024
Givens	5	64	3	1	1024x1024
PtIncluded	3	19	3	1	350 vars, 15000 csts
QR	6	29	3	2	1024x1024
Sat-add	6	27	2	2	1920x1080
ShortPath	3	6	3	0	1000 nodes
Svdksb	5	10	2	2	1024x1024
Svdvar	4	10	3	3	1024x1024
TransClos	3	3	3	0	1000 nodes

Figure 6.6: Kernel description

Our results are summarized in Figure 6.7. For each kernel, we provide the speedup achieved by LeTSeE and Pluto[5] when considering only the regular parts of the program, then when using the extended representation. We also provide the compilation time penalty when considering the extended representation. N/A means that the benchmark cannot be handled in the specific context.

	Speedup regular		Speedup extended		Compilation time penalty	
	LetSee	Pluto	LetSee	Pluto	LetSee	Pluto
2strings	N/A	N/A	1.18×	1×	N/A	N/A
Dither	N/A	N/A	1×	5.42×	N/A	N/A
Gauss-J	1×	1.46×	1×	1.77×	2.51×	1.22×
Givens	1×	1×	1.03×	7.02×	21.23×	15.39×
PtIncluded	1×	1×	1×	1.44×	10.12×	1.44×
QR	1.04×	1.09×	1.04×	8.66×	9.56×	2.10×
Sat-add	1×	1.08×	1.51×	1.61×	1.22×	1.35×
ShortPath	N/A	N/A	1.53×	5.88×	N/A	N/A
Svbksb	1×	1×	1×	1.96×	2×	1.66×
Svdvar	1×	3.54×	1×	3.82×	1.93×	1.33×
TransClos	N/A	N/A	1.43×	2.27×	N/A	N/A

Figure 6.7: Performance and compilation time

The results show that for the programs we considered — spanning representative sources of irregularity in loop-based computations — we are able to significantly improve performance.

[5]With or without tiling, whatever performs best.

On our target platform, applying existing polyhedral optimizers with the help of the proposed extension allows to achieve up to a 1.53× speedup for ShortPath when applying LeTSeE (single-threaded), and up to a 8.66× speedup for QR when applying Pluto (multithreaded, on 8 cores). We were also able to significantly improve performance for codes that were already partially regular.[6] For those programs, we obtained speedup reaching 1.51× using LeTSeE, and from 1.09× to 8.66× using Pluto.

Typically, the performance achieved using the LeTSeE algorithm comes from a better locality of the memory accesses (with carefully crafted loop fusions) and compiler optimizations that have been *enabled* (e.g. vectorization). On the other hand, our approach also exposes parallelization opportunities that are exploited by Pluto with efficient tiling and coarse-grain parallelization, to combine both parallelization and locality improvement.

We summarize our findings with more detailed insight about the transformations obtained by LeTSeE and Pluto for our benchmark suite:

- 2Strings is composed of two distinct non-dependent `while` loops. Using our approach, LeTSeE is able to fuse them leading to performance improvements. Pluto did not manage to parallelize the benchmark.

- Dither is a code composed of a perfectly nested loop of depth 2 and with all the statements guarded with various non-affine `if` conditionals. Relying on the extension, Pluto is able to identify parallelism and to tile the loops, achieving a 5.42× speedup over the original code.

- Gauss-J is another code where parallelization and tiling of non SCoP part become possible on Pluto with our approach, but where the SCoP part holds most of the computation time.

- Givens features at depth 2 a sequence of data-dependent conditions to separate different cases of complex sine/cosine computations for Givens rotations. These conditions may prevent optimization. Using the extensions discussed in this chapter, Pluto is able to parallelize the code. We show in the appendix the result of the optimization achieved by Pluto with the help of our extended framework. The result may be understood as a sequence of basic transformations such as skewing, tiling or index-set-splitting to extract coarse grain parallelism and to improve data locality [BHRS08]. The parallelism has been made explicit through OpenMP pragmas. The target code shows a 7.02× speedup over the original code.

- PtIncluded is a routine that checks if an integer point is included in a polyhedron. It relies on a linked list traversal, another usual source of control flow irregularity. Since there is no dependence between iterations of the loop traversing the linked list, Pluto is able to parallelize it.

- QR is a code where most of the inner loops are guarded by non-affine `if` conditionals. All these loops are regular, hence LeTSeE and Pluto are able to optimize and parallelize some of them, leading to 1.04× and 1.09× speedup respectively. Nevertheless, the best performance is achieved when relying on our extension, as Pluto may now parallelize and tile the full code. The super-linear speedup is a consequence of SIMDization that has been enabled by the transformation.

- Sat-add could be divided into two parts, a static control part and a non-static control part. Both parts are parallel. Without our approach, Pluto is able to detect parallelism in the static control part only, yielding a performance improvement of 1.08× compared to the original code. Note that

[6]We call a program partially regular if it contains a SCoP depth of at least 1, i.e., if it has at least one purely static loop.

this parallelism was already found by ICC. However, through our extension, Pluto can handle and parallelize the whole code, with a speedup of 1.61×.

- ShortPath is composed by a perfectly nested loop of depth 3 without any SCoP, dealing with 2-dimensional matrices. Using our approach, Pluto is able to parallelize the outer loop, hence a significant 5.88× speedup; LeTSeE applies a loop interchange transformation on the original code. These two optimizations were performed as well on TransClos providing 1.43× and 2.27× speedup on LeTSeE and Pluto respectively.
- On Svbksb, on the extended framework, Pluto is able to parallelize the outermost loop, leading to a speedup of 1.96× over the original code.
- Svdvar is a code composed of two perfectly nested loops, one of them is a SCoP. With the regular framework, Pluto is able to parallelize this SCoP only. Using the extended framework, Pluto performs a parallelization on both loops. Nevertheless, the same performance is achieved. This is due to the amount of calculations the SCoP carries out in this code.

These results were achieved without modifying either LeTSeE or Pluto and using a conservative dependence analysis. They demonstrate the power of this approach, finding new or better opportunities for deep optimizations in the polyhedral model.

The price to pay for these improvements is a longer compilation time as we consider larger kernels, up to a factor 20 for LeTSeE due to its iterative nature. This remains practical in our experiments as the compilation time is at worse a matter of seconds. As the applicability of the polyhedral grows with our extended framework, so is the problem size for the optimizations. Our extended model raises the question of designing novel, highly scalable polyhedral optimization algorithms, which is ongoing work.

6.5 Related Work

Most irregular polyhedral techniques were developed in the context of `while` loop parallelization. Collard explored a speculative approach to parallelize loops nests with `while` loops [Col95, Col94]. The idea is to allow a speculative execution of iterations which are not in the iteration domain of the original program. This method leads to more potential parallelism than with traditional polyhedral methods, at the expense of an invalid space-time mapping, which is fixed thanks to a backtracking policy. In contrast to the speculative approach, Griebl et al. explore a conservative one. They try to enumerate a superset of the target execution space, and propose solutions to eliminate iterations that are not in the target execution space and to take care of the termination condition. For the first problem, they define what they call *execution determination* where they introduce a predicate to determine if a point in the iteration space can be executed or not. For the second point, they define and compute *termination detection*. Griebl and Lengauer [GL94] propose another solution using a communication scheme in a distributed-memory model to determine the upper bounds of the target loops, but this solution increases the execution time of the scanning. For the same problem, but on shared-memory models, Griebl and Collard [GC95] describe a so-called counter scheme. Griebl et al. [GGL98, GGL99] present another one called maximum scheme.

Other authors concentrated on extending the expressiveness of the polyhedral model in special cases; these efforts are complementary to our conservative yet general approach. Palkovič wrote the most comprehensive monograph on the topic [Pal07]. In contrast, our approach handles any function body and transparently inherits all existing optimization and parallelization techniques based on the polyhedral model.

In addition, our extended model opens the door to important loop transformations targeted to data-dependent control flow. For example, Decoupled Software Pipelining (DSWP) [RVVA04] extracts and exploits pipeline parallelism from irregular codes involving complex control flow and data structures. Full automation of DSWP remains a challenge, due to the intricacy of the transformations involved and their interplay with other optimizations. Another example is Deep Jam [CCJ05], a generalization of loop fusion and unroll-and-jam to dynamic control flow, targeted at instruction-level and vector parallelism. Deep Jam is at least as complex as DSWP to automate.

Previously related work and our approach are static techniques, which analyse programs at compile time. Several works aim at performing runtime analyses, e.g., to detect parallel loops [RP95, RRH03, RR03, RPR07, RVRA08]. Unfortunately, they are mostly used to *expose* parallelism without transforming the code, while restructuring is in general useful either to *extract* parallelism or to translate it to performance. Jimborean et al. proposed the very first dynamic approach based on the polyhedral model to speculate at runtime the existence of a static control loop and to transform it using a state-of-the-art algorithm [JCP+12]. We are currently investigating such a dynamic approach to complete our conservative analysis at runtime.

6.6 Conclusion

The study presented in this chapter overcomes the control-flow limitations of the polyhedral model in an intraprocedural setting. The solution comes from a sleek and natural modeling of control-flow predicates at all stages of a polyhedral compilation framework. This extension goes far beyond the state-of-the-art which only addresses special non-affine cases as they misleaded the control-flow management from the data-flow one. The main difficulty resides in the design of an extended code generation algorithm supporting those extensions while limiting control-flow overhead. Several subtle difficulties also trickle down to the extraction of the polyhedral representation and the storage mapping of control predicates (privatization). We experimentally validated our approach, demonstrating new optimization opportunities for irregular programs as well as improvements over previous results on partially-regular applications.

The static control limitations of the polyhedral model are now history. Research may now concentrate on accurate static/dynamic analysis, and complementing speculative optimization and parallelization techniques with aggressive program transformations. However, an important limitation stressed by this approach is the high complexity of the algorithms supporting polyhedral operations — typically exponential in the number of statements and/or the number of array references and/or the loop nesting depth, see Chapter 5. Enlarging its application domain stresses the scalability of these algorithms even further. In this context, we are working on macro-block and region formation heuristics, as well as novel polyhedral optimizations that could scale to full functions.

Chapter 7

Conclusion and Perspectives

Programmers should not be heroes for reaching a decent part of an architecture's peak performance. Unfortunately this is the situation today as chipmakers are relying on users to find ways to exploit the many hardware resources they are providing. There is little hope for this to change anytime soon as next generation processors are planned to bring more and more cores while there is no general easy way to exploit them efficiently yet. Hence, software industry is under pressure to provide better tools, languages, libraries and compilers to help the users to make an effective use of the available resources. This thesis presents some contributions towards meeting this challenge.

Our strategy relies on high-level compilation. It is the most promising since it does not require the user to learn about a new language or new abstractions or the target architecture details. But it is also the most challenging since an automatic tool has to deeply analyse programmer's code and to take the best decisions for optimizing and parallelizing.

We rely on the polyhedral model to benefit from its analysis power and its transformation expressiveness. We contributed to the collective effort on generalizing the model to unions of relations. We designed scalable techniques for code generation and data dependence analysis and demonstrated they could be applied on complex real-world situations. We extended the polyhedral model to support irregular extensions for both access and control structures. All those advances enabled the support for polyhedral techniques on more general programs requiring more complex transformations. We designed two optimization strategies. The first approach is semi-automatic and is based on the violated dependence analysis to provide semantic feedback to the user. The second approach is a unique high-level iterative compilation technique based on the space of legal transformations. Reported experimental results consistently outperform production compilers on sequential codes. We also coupled this technique to state-of-the-art model-driven parallelization techniques to achieve, to the best of our knowledge, unprecedented performance for a fully automatic system.

Significant steps forward have been made during the last decade on automatic parallelization and optimization in the polyhedral model, on both the theoretical and the practical sides. The polyhedral model is now considered as a viable strategy and is reaching production compilers as demonstrated by related frameworks on GCC, LLVM, IBM XL or ACE CoSy. However, many problems remain open or need better solutions. Perspectives with respect to the work presented in this thesis aim at making high-level compilers better at their primary task of automatic optimization and at making them play a key role to help users to design their own optimizations. First, many other dimensions can be investigated in our high-level iterative compilation framework, such as low-level compiler options or data layout. Another follow-up is to build on our experience at optimizing with different approaches to target architectures with multiple parallelism levels explicitly. Next, our extension to irregular codes suggests to investigate combined static/dynamic approaches for a more precise analysis of the properties of the programs and to allow them to adapt to their environment. Finally, research must be continued on better approaches for semi-automatic optimization, from new optimization-centric directives to long term graphical manipulation of programs.

New Dimensions for Iterative Compilation Our high-level iterative compilation approach presented in Chapter 4 is based on mapping selection. Others complementary dimensions can be explored *together* with the mapping. First, all elements of the target platform, i.e., the architecture, the low-level compiler and its options [KHW+05, CGH+05, TVA05] can be investigated. Another promising dimension is the data layout.

Current architectures include complex memory features such as deep memory hierarchies, shared caches between cores, memory controller parallelism, data alignment constraints etc. As a result, choosing a good data layout is critical to achieve high performance on such systems. Many data layout transformation techniques have been developed for that purpose, including recent efforts on improving data locality [LAB+09], performing vectorization [HSP+11, JMS+10], benefiting from interleaved DRAM banks [SSH10] or managing of-chip memory bandwidth [BBK+08a]. Because data layout transformations are always legal, the layout can be used together with the mapping selection just as a promising additional dimension of the search space.

Hierarchical Optimization Both supercomputers and modern workstation architectures show several parallelism layers. Basically, they have at least three levels. The first one corresponds to vector units (SIMD), able to apply simultaneously the same processing to a small data set. The second one corresponds to processor cores, which may execute several threads or processes in parallel while sharing a part of the memory hierarchy. The third level corresponds to multi-processor, distributed memory architectures which are able to execute several tasks in parallel relying on distributed memory.

Those parallelism levels have major differences from the compilation point of view. The first difference is programming. Vector units are typically used through dedicated instructions called *intrinsics*. Multicore programming is done using threads directly or through OpenMP. Lastly, MPI is typically used for the last level. The second difference comes from the performance properties. Optimizations on the first level are very fragile: a slight difference in the source code can have a dramatic impact on performances [CB-PBCV07]. The second level is more robust, but complex interactions between architecture features and the complexity of the compilers makes the performance hardly predictable (see Chapter 4). The third level is more stable on this respect.

Most polyhedral compilation techniques target directly only one parallelism level. Most of the time, they focus on the intermediate level, i.e., thread-level parallelism. Benefiting from the composition properties of the polyhedral mappings, and building on our experience on different optimization approaches, new hierarchical strategies can be derived. Aggressive iterative techniques [CB-PBCV07] should be used to find the very best internal loops, since the overall performance is bounded by the fastest element. Machine learning strategy [CB-PBCC08b] is appropriate to design the intermediate level and model-driven techniques [CB-BF03a, CB-PBB+10] are convenient for the most stable performance. Hence, mapping composition using well suited strategies, is a promising way to address the global optimization problem.

Speculative Parallelization Static control programs are an optimal input for optimization techniques in the polyhedral model. However, we have shown in Chapter 6 that despite the use of conservative analyses, polyhedral mapping is keeping its expressiveness on irregular programs. The consequence is, it is possible to benefit from polyhedral optimization techniques in the general case, and to apply adequate optimizations, provided the conservative information from static analysis is known more precisely at some point. This can be done by the programmer, e.g., on request by the compiler, or during execution, using a dynamic analysis that remains to be built.

Usual speculative parallelization approaches optimistically decide that a given loop is parallel when static analyses are not sufficient to make a decision. It is possible to go further, relying on the polyhedral model to transform the irregular codes, ignoring some well chosen conservative dependences, and to generate most interesting program versions. The challenge is to design an efficient runtime analysis computing the missing information to decide which code version can be used. This problem is currently under investigation.

Dynamic Program Tuning Parallel applications used to be executed alone until their termination on partitions of supercomputers. The recent shift to multicore architectures for desktop and embedded systems is raising the problem of the coexistence of several parallel programs. Operating systems already take into account the *affinity* mechanism to ensure a thread will run only onto a subset of available processors (e.g., to reuse data remaining in the cache since its previous execution). But this is not enough, as demonstrated by the large performance gaps between executions of a given parallel program on desktop computers (running several processes). To support many parallel applications, advances must be made on the system side (scheduling policies, runtimes, memory management...). However, automatic optimization and parallelization can play a role by generating programs with dynamic-auto-tuning capabilities to adapt themselves to the system load.

Human-Compiler Interface Using the polyhedral model directly to optimize a code, e.g., through mappings, is very counter-intuitive for a programmer. The reason is that the structure of the original program is totally removed. There are no more loops or syntactic ordering between statements, only the data dependences remain. We are developing two strategies to let a user benefit from its power. First, fully automatic techniques as the iterative approach described in Chapter 4. This approach is successful with respect to performance improvements [CB-PBB$^+$10, CB-PBCC08a] and to ongoing integration onto production compilers. The second strategy is to provide high-level directives, allowing the user to use a polyhedral engine transparently, e.g., using *URUK* [CB-GVB$^+$06], Clay (see Chapter 2.1.4) or other alternatives such as UTF [Kel96] or CHiLL [CCH08]. Unfortunately those semi-automatic methods are arduous and still require a significant expertise from the programmer. All in all, high-level transformation script systems have not generated a significant interest from users yet.

Existing scripting systems try to mimic classical loop transformations. While this approach may be useful in some cases, loop transformations are usually not the goal, but the *consequence* of an optimizing action. For instance, loop fusion is often the consequence of the action of bringing several accesses to the same reference closer to each other. Hence, we need to design a new set of directives, based on the optimization the user is looking for, rather than manipulating syntactic constructions. Because long transformation scripts are complex to generate, automatic parallelization can be used as a first step by translating a set of mapping relations generated with an optimizing algorithm to a transformation script. Then, the user can modify this first “optimization draft” to improve it.

Another approach is to exploit the geometrical representation of the polyhedral model which has never been used as a support for designing optimizations. It could abstract programs graphically and could allow to achieve restructuring through *interactions* with this graphical representation. Using this side of the polyhedral model would require less expertise from the user while using the polyhedral engine with an unprecedented ease. Such interactions necessitate a deep study on the various ways to interact and to expose useful information to the user. We believe this approach has potential to enable semi-automatic optimization where transformation scripts partially failed.

Appendix A

Related Tools

There is a lot of pragmatism and no result without a corresponding implementation in this document, some of them with a significant number of users. I believe that tools are a good support for disseminating our research work and, above all, that like in other domains where Science is close to Engineering, better tools lead to new research directions and new research results. Hence there is both Science and Engineering in this thesis. As a matter of a fact, the availability of CLooG that I first released during my PhD thesis had a significant impact on the high-level compilation community. It was suddenly possible to apply easily complex mappings to complex programs at a new milestone of efficiency and robustness. Hence we could build on the many fundamental studies on, e.g., optimal parallelism extraction or data reuse, to convert parallelism and data reuse into performance using more complex and more numerous mappings, putting more stress on code generation in return. Pluto [BHRS08] (up to fifteen mapping dimensions and counting) and LeTSeE [CB-PBCV07, CB-PBCC08b] (millions of generated codes for the empirical studies) are great examples of this virtuous circle.

This appendix provides information about the tools and libraries related in this document, with more details on those where I have been involved for research and/or development.

A.1 Candl

Address	`http://www.lri.fr/~bastoul/development/candl`
References	[CB-BP12]
Description	Candl is a free software and a library devoted to data dependences computation. It has been developed to be a basic bloc of our optimizing compilation tool chain in the polyhedral model. From a polyhedral representation of a static control part of a program, it is able to compute exactly the set of statement instances in dependence relation. Hence, its output is useful to build program transformations respecting the original program semantics. This tool has been designed to be robust and precise. It implements some usual techniques for data dependence removal, as array privatization or array expansion, offers simplified abstractions like dependence vectors and performs violation dependence analysis. Main authors include Cédric Bastoul and Louis-Noël Pouchet.

A.2 Clan

Address	`http://www.lri.fr/~bastoul/development/clan`
References	[CB-Bas08]
Description	Clan is a free software and library that translates some particular parts of high level programs written in C, C++, C# or Java into a polyhedral representation (strict or extended to irregular control flow –still experimental at this document redaction time). This representation may be manipulated by other tools to, e.g., achieve complex program restructuring (for optimization, parallelization or any other kind of manipulation). Main authors include Cédric Bastoul and Louis-Noël Pouchet, irregular extensions by Mohamed-Walid Benabderrahmane.

A.3 Clay

Address	`http://www.lri.fr/~bastoul/development/clay`
References	[CB-Bas12]
Description	Clay is a free software and library devoted to semi-automatic optimization using the polyhedral model. It can input a high-level program or its polyhedral representation and transform it according to a transformation script. Classic loop transformations primitives are provided. Clay is able to check for the legality of the complete sequence of transformation and to suggest corrections to the user if the original semantics is not preserved (experimental at this document redaction time). Main authors include Joël Poudroux and Cédric Bastoul.

A.4 CLooG

Address	`http://www.cloog.org`
References	[CB-Bas03, CB-Bas04a, CB-VBC06, CB-Bas02a]
Description	CLooG is a free software and library to generate code for scanning Z-polyhedra. That is, it finds a code (e.g. in C, FORTRAN...) that reaches each integral point of one or more parameterized polyhedra. CLooG has been originally written to solve the code generation problem for optimizing compilers based on the polytope model. Nevertheless it is used now in various area e.g. to build control automata for high-level synthesis or to find the best polynomial approximation of a function. CLooG may help in any situation where scanning polyhedra matters. While the user has full control on generated code quality, CLooG is designed to avoid control overhead and to produce a very effective code. Main authors include Cédric Bastoul and Sven Verdoolaege, irregular extensions by Mohamed-Walid Benabderrahmane.

A.5 LeTSeE

Address	`http://www.cse.ohio-state.edu/~pouchet/software/letsee`
References	[CB-PBCV07, CB-PBCC08b, CB-PBCC08a]
Description	LeTSeE is a platform dedicated to computing and exploring the legal affine scheduling space of a statically controlled program. It is programmed as a library, offering services such as: (1) a tunable algorithm for legal transformation space construction, (2) various heuristics to traverse legal spaces, (3) many auxiliary functions (graph manipulation, transformation generation, etc.). Author is Louis-Noël Pouchet.

A.6 OpenScop (Formerly SCoPLib)

Address	`http://www.lri.fr/~bastoul/development/openscop`
References	[CB-Bas11b]
Description	OpenScop is an open specification that defines a file format and a set of data structures to represent a static control part (SCoP for short), i.e., a program part that can be represented in the polyhedral model. The goal of OpenScop is to provide a common interface to the different polyhedral compilation tools in order to simplify their interaction. To help the tool developers to adopt this specification, OpenScop comes with an example library (under 3-clause BSD license) that provides an implementation of the most important functionalities necessary to work with OpenScop. Main authors include Cédric Bastoul and Louis-Noël Pouchet.

A.7 PipLib

Address	`http://www.piplib.org`
References	[Fea88a, CB-FcCB02]
Description	PIP/PipLib is Paul Feautrier's parametric integer linear programming solver. PIP is a software that finds the lexicographic minimum (or maximum) in the set of integer points belonging to a convex polyhedron. The very big difference with well known integer programming tools like lp_solve or CPLEX is the polyhedron may depend linearly on one or more integral parameters. If the user asks for a non integral solution, PIP can give the exact solution as an integral quotient. The heart of PIP is the parametrized Gomory's cuts algorithm followed by the parameterized dual simplex method. The PIP Library (PipLib for short) was implemented to allow the user to call PIP directly from his programs, without file accesses or system calls. The user only needs to link his programs with C libraries. Main authors include Paul Feautrier, Cédric Bastoul and Sven Verdoolaege.

A.8 PoCC

Address	`http://pocc.sf.net`
References	[CB-PBB^{+}10, CB-PBB^{+}11, PPC^{+}11]
Description	PoCC is a flexible source-to-source iterative and model-driven compiler, embedding most of the state-of-the-art tools for polyhedral compilation. The main features are: (1) a full-flavored compiler, from C code to optimized C code, and to binary; (2) the state-of-the art techniques in polyhedral optimization (iterative search among legal schedules, powerful model-driven tiling and parallelization); (3) a flexible platform to quickly prototype and develop optimizations leveraging the polyhedral model; (4) modular design, configuration files-oriented. PoCC embeds powerful Free software for polyhedral compilation. This software is accessible from the main driver, and several IR conversion functions allows to communicate easily between passes of the compiler. Main author is Louis-Noël Pouchet.

A.9 R-Stream

Address	`https://www.reservoir.com/?q=rstream`
References	[MVW^{+}11, CB-MLV^{+}09, CB-BVL^{+}09, CB-LVM^{+}10]
Description	R-Stream is a multi-target high-level compiler developed by Reservoir Labs Inc. and specialized in the efficient mapping of high-performance applications to modern parallel architectures. R-Stream supports heterogeneous multi and manycore architectures with several levels of parallelism, deep memory hierarchies, explicitly managed memory management and communications. It offers high retargetability through user-defined *machine models*. R-Stream relies on a polyhedral mapping engine to automatically translate sequential programs written in C to parallel high-level codes e.g., in C with OpenMP for shared memory architectures, C with convenient APIs for Cell or Tilera, CUDA for GPGPUs, C^{n} for ClearSpeed accelerators or dataflow assembly for FPGA targets.

A.10 Other Related Tools

GRAPHITE

Address	`http://gcc.gnu.org/wiki/Graphite`
References	[CB-PCB^{+}06, TCE^{+}10]
Description	Polyhedral compilation framework for GCC

FM

Address	`http://www.cse.ohio-state.edu/~pouchet/software/fm`
References	[Pou10]
Description	Fourier-Motzkin library

isl

Address	`http://freecode.com/projects/isl`
References	[Ver10]
Description	Integet set library, data dependence analysis and mapping

PolyLib

Address	`http://icps.u-strasbg.fr/polylib`
References	[Wil93, LW97]
Description	General polyhedral library

Omega Library

Address	`http://www.cs.umd.edu/projects/omega`
References	[KMP+96]
Description	General polyhedral library, data dependence analysis and code generation

Pet

Address	`http://freecode.com/projects/libpet`
References	[Ver12]
Description	Direct raising tool and library

Pluto

Address	`http://pluto-compiler.sf.net`
References	[BHRS08, BBK+08b, BGDR10]
Description	High-level optimizing and parallelizing compiler

PPL

Address	`http://bugseng.com/products/ppl`
References	[BHZ08]
Description	General polyhedral library

Polly

Address	`http://polly.llvm.org`
References	[GZA+11]
Description	Polyhedral compilation framework for LLVM

PolyBench

Address	`http://www.cse.ohio-state.edu/~pouchet/software/polybench`
References	Pouchet 2011
Description	Static control benchmark collection

WRAP-IT

Address	`N/A`
References	[CB-BCG$^+$03a, CB-GVB$^+$06, Gir05]
Description	Polyhedral compilation framework for ORC

Appendix B

CV (French)

Informations personnelles		**Informations de contact**	
Date de naissance	26 novembre 1975	Adresse	12 rue Hélène Boucher
Lieu de naissance	Saint Dizier, France		78960 Voisins-le-Bretonneux
Nationalité	Française		France
Situation familiale	Marié, trois enfants	Téléphone	+33 1 72 92 59 65
Page web	`www.lri.fr/~bastoul`	Email	`Cedric.Bastoul@u-psud.fr`

B.1 Expérience professionnelle

2005-Présent — MAÎTRE DE CONFÉRENCES, UNIVERSITÉ PARIS-SUD

Chercheur dans l'équipe de compilation et d'architecture du laboratoire LRI, enseignant en informatique à l'IUT d'Orsay, délégation CNRS en 2011-2012

▷ *Addresse :Faculté des sciences, F-91405 Orsay Cedex*

▷ *Dates d'emploi : du 1er octobre 2005 à nos jours*

▷ *Superviseur : Pr. Brigitte Rozoy*

2009 (mission longue durée) — VISITING PROFESSOR, RESERVOIR LABS INC.

Recherche et développement de techniques d'optimisation à la compilation dans le compilateur R-Stream, développé par la société

▷ *Addresse : Reservoir Labs Inc., 632 Broadway Suite 803, New York, NY 10012*

▷ *Dates d'emploi : du 2 février 2009 au 22 décembre 2009*

▷ *Superviseur : Dr. Richard Lethin*

2004-2005 — ASSISTANT TEMPORAIRE D'ENSEIGNEMENT ET DE RECHERCHE, UNIVERSITÉ D'AUVERGNE

Chercheur dans l'équipe d'algorithmique du laboratoire LAIC, enseignant à l'IUT de Clermont-Ferrand

▷ *Addresse : Ensemble universitaire des Cézeaux, F-63172 Aubière*

▷ *Dates d'emploi : du 1er septembre 2004 au 31 août 2005*

▷ *Pr. Jean-Pierre Reveillès*

2003-2004	ASSISTANT TEMPORAIRE D'ENSEIGNEMENT ET DE RECHERCHE, UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES Chercheur dans l'équipe d'architecture et parallélisme du laboratoire PRiSM, enseignant au département informatique de l'UFR Sciences ▷ *Addresse : UFR Sciences, 45 avenue des États-Unis, 78035 Versailles Cedex* ▷ *Dates d'emploi : du 1er octobre 2003 au 31 octobre 2004* ▷ *Superviseur : Pr. William Jalby*

B.2 Formation

2000-2004 — DOCTORAT SYSTÈMES INFORMATIQUES RÉPARTIS, UNIVERSITÉ PARIS 6 PIERRE ET MARIE CURIE

Dirigé par Pr. Paul Feautrier. Titre de la thèse : *amélioration de la localité dans les programmes à contrôle statique*, soutenue le 7 décembre 2004

▷ *Titulaire d'une bourse MESR, moniteur de l'université Paris 6*

▷ *Jury de thèse :*
- Claude GIRAULT, professeur université Paris 6, président
- Philippe CLAUSS, professeur université de Strasbourg, rapporteur
- Christian LENGAUER, professeur universität Passau, rapporteur
- Paul FEAUTRIER, professeur ENS Lyon, directeur
- Nathalie DRACH-TEMAM, professeur université Paris 6, examinateur
- Patrice QUINTON, professeur IRISA Rennes, examinateur

▷ *Mention "Très Honorable" (plus haute mention décernée à l'université Paris 6)*

1999-2000 — DEA SYSTÈMES INFORMATIQUES RÉPARTIS, UNIVERSITÉ PARIS 6 PIERRE ET MARIE CURIE

Obtenu avec mention Bien

▷ *Stage de 6 mois au laboratoire PRiSM, université de Versailles Saint-Quentin-en-Yvelines, dirigé par Pr. Paul Feautrier sur les techniques d'optimisation à la compilation*

1998-1999 — MAÎTRISE INFORMATIQUE, UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Obtenu avec mention Bien, major de promotion

▷ *Stage de deux mois à l'institut Gaspard Monge, université de Marne-la-Vallée, dirigé par Pr. Éric Laporte sur la manipulation d'automates finis pour la synthèse de langage parlé*

B.3 Compétences techniques

Programmation	C/C++, Java, OpenMP, Cuda, C^n, Programmation système, Shell, PHP, SQL
Système	Linux / Unix, OSX, Windows XP/7
Bureautique	LaTeX, Microsoft Office

B.4 Langues

Français	Langue maternelle
Anglais	Lu, écrit, parlé. Pratique quotidienne, travail aux USA durant un an

B.5 Recherche et développement

Spécialité	Optimisation et parallélisation automatiques des programmes à la compilation ▷ Représentation formelle des programmes à travers le modèle polyédrique ▷ Optimisation de la localité des données ▷ Extraction de parallélisme de haut niveau dans les programmes séquentiels ▷ Génération de code dans le modèle polyédrique ▷ Compilation pour les GPGPUs ▷ Compilation itérative et par machine learning ▷ Optimisation assistée par ordinateur ▷ Calcul polyédrique
Publications	Auteur ou co-auteur de plus de 30 publications parmi les journaux et conférences internationaux les plus prestigieux en compilation (liste complète en section *Bibliography*) ▷ ACM PLDI, ACM PoPL, ACM CGO, ACM ICS, IEEE PACT, Kluwer IJPP etc.
CLooG `www.cloog.org`	Créateur du générateur de code polyédrique le plus avancé actuellement, reconnu comme un standard de fait pour la compilation de haut niveau ▷ Plusieurs dizaines de sociétés et laboratoires comme utilisateurs directs ▷ Utilisé comme bibliothèque par le compilateur GCC
PipLib `www.piplib.org`	Co-mainteneur du résolveur de problèmes de programmation linéaires paramétriques en nombres entiers PIP ▷ Plus d'une dizaine de sociétés et laboratoires comme utilisateurs directs

B.6 Distinctions

HiPEAC 2011	HiPEAC European Network of Excellence Paper Award pour le papier *Loop Transformations: Convexity, Pruning and Optimization* à ACM PoPL 2011
HiPEAC 2008	HiPEAC European Network of Excellence Paper Award pour le papier *Iterative optimization in the polyhedral model: Part II, multidimensional time* à la conférence ACM PLDI 2008
Euro-Par 2004	Distinguished Paper Award pour le papier *More legal transformations for locality* à la conférence Euro-Par 2004
PACT 2004	Student Award pour le papier *Code generation in the polyhedral model is easier than you think* à la conférence ACM PACT 2004

B.7 Coordination et participation à des projets scientifiques

MANY
2011-2014

Projet ITEA2 de recherche d'outils pour la programmation et la gestion des ressources sur les architectures multicœurs embarquées haute performance

▷ Rôle : coordinateur local

▷ Large consortium européen à dominante industrielle

▷ `http://www.eurekamany.org`

OpenGPU
2010-2012

Projet Systematics pour le développement d'une plateforme logicielle libre d'aide à la parallélisation et à l'évaluation des architectures GPGPU

▷ Rôle : coordinateur local à la suite d'Albert Cohen en 2011

▷ Large consortium représentant les acteurs français industriels ou académiques

▷ `http://www.opengpu.net`

PolyMorph
2008-2009

Projet interne INRIA (suport ingénieur durant deux ans) pour le développement d'un système de manipulation graphique des programmes

▷ Rôle : coordinateur

Participations

▷ ANR PetaQCD 2008-2012, création d'outils et de matériels pour atteindre le petaflop pour la simulation de chromodynamique quantique.

▷ FP6 ACOTES 2006-2009, recherche de techniques de compilation avancée pour le streaming embarqué.

▷ ANR PARA 2006-2008, optimisation à la compilation des codes de chromodynamique quantique.

B.8 Enseignement

2005-Présent

Enseignant titulaire à l'IUT d'Orsay (243h/an en moyenne, encadrement compris), vacataire à l'École Polytechnique (36h/an de 2007 à 2008, interventions en 2009), échange de service à l'UFR Sciences de l'Université Paris-Sud (13,5h/an de 2005 à 2008)

▷ Thèmes principaux : système, Java, architecture et réseaux. Niveau licence (IUT et UFR) et niveau master (système et architecture à l'École Polytechnique)

▷ Responsable de trois cours de système (1ère et seconde année DUT Informatique)

▷ Responsable cours de Java (seconde année par apprentissage DUT Informatique)

▷ Responsable cours de programmation orientée objet (L2 Math-Info)

▷ Responsable cours de sécurite informatique (formation continue)

2004-2005

ATER à temps complet à l'IUT de Clermont-Ferrand (192h)

▷ Thèmes principaux : système, réseaux et structures de données. Niveau licence

▷ Responsable cours de structures de données (seconde année DUT Informatique)

2003-2004

ATER à mi-temps à l'UFR Sciences de l'université de Versailles Saint-Quentin-en-Yvelines (96h)

▷ Thème principal : administration système. Niveau master

2000-2003

Moniteur à l'université Paris 6 Pierre et Marie Curie (64h/an)

▷ Thèmes principaux : algorithmique et programmation. Niveau licence

B.9 Encadrement

Doctorants

▷ Lénaïc Bagnères, sujet *optimisation et parallélisation automatiques pour les architectures many-cœurs* début octobre 2012, soutenance prévue en décembre 2015.

▷ Mohamed-Walid Benabderrahmane (direction par dérogation de HDR), sujet *manipulation de codes irréguliers dans le modèle polyédrique*, début octobre 2008, arrêt pour raisons personnelles en 2010, 1 publication co-signé (rang A). Actuellement ingénieur sécurité informatique.

▷ Louis-Noël Pouchet, sujet *optimisation itérative dans le modèle polyédrique*, début septembre 2006, soutenue en janvier 2010, 10 publications co-signées (7 de rang A). Actuellement chercheur à University of California at Los Angeles (UCLA).

▷ Nicolas Vasilache, sujet *techniques scalables d'optimisation des programmes dans le modèle polyédrique*, début septembre 2004, soutenue en septembre 2007, 11 publications co-signées (5 de rang A). Actuellement ingénieur de recherche à Reservoir Labs Inc.

Ingénieurs

▷ Taj Khan (encadrement 100%), ingénieur spécialiste recherche et développement, financé sur projet européen (ITEA2 MANY), CDD de septembre 2012 à août 2014.

▷ Abdelfetteh Louati (encadrement 100%), ingénieur de recherche et développement sur projet, CDD d'octobre 2008 à février 2010.

Master Recherche

▷ Stage M2R Soufiane Baghdadi et Gilles Duboscq, mars à août 2011.

▷ Stage M2R Mohamed-Walid Benabderrahmane, mars à septembre 2008.

▷ Stage M2R Louis-Noël Pouchet, mars à août 2006.

B.10 Fonctions d'intérêt collectif

Expertise

Membre du Conseil Consultatif des Spécialistes de l'Université Paris-Sud pour la 27ème section depuis 2010

▷ *Membre de la Commission Mixte de Spécialistes à l'IUT d'Orsay de 2006 à 2009*

▷ *Membre du jury ITRF BAP E à l'université Paris-Sud 2010*

Vie du laboratoire

Responsable de l'équipe Architectures Parallèles au LRI en 2011 et 2012

▷ *Membre catégorie B au Conseil de Laboratoire LRI depuis 2006*

▷ *Membre au LRI des commissions Web, Valorisation et Locaux travaillant à différents aspects de la vie et la politique du laboratoire*

Vie scientifique

- ▷ Membre des comités de programme des conférences internationales PACT (Parallel Architectures and Compilation Techniques), Computing Frontiers, DATE (Design Automation and Test in Europe) et SSS (Symposium on Stabilization, Safety and Security of Distributed Systems), ainsi que des workshops internationaux IMPACT (International Workshop on Polyhedral Compilation Techniques) et PARMA (Parallel Programming and Run-Time Management Techniques for Many-core Architectures)
- ▷ Organisateur et co-chair du premier workshop sur les techniques d'optimisation polyédriques `http://impact2011.inrialpes.fr`
- ▷ Relectures pour 20+ journaux et conférences internationaux et plusieurs dizaines d'articles

Responsabilités pédagogiques

- ▷ Responsable de la Licence Professionnelle de Programmation en Environnement Réparti à l'IUT d'Orsay depuis 2012 (formation par apprentissage accueillant chaque année 24 étudiants)
- ▷ Responsable de la Licence Professionnelle de Sécurité des Réseaux et Systèmes Informatiques à l'IUT d'Orsay de 2007 à 2011 (formation par apprentissage accueillant chaque année 30 étudiants)
- ▷ Membre du Bureau du Département Informatique de 2007 à 2011
- ▷ Interface équipe enseignante – Centre Commun de Ressources Informatiques de 2006 à 2010
- ▷ Responsable de plusieurs cours à l'IUT d'Orsay (animation d'équipes pédagogiques jusqu'à une dizaine d'intervenants)

B.11 Intérêts personnels

Sports	Judo
Loisirs	Programmation, peinture sur figurines
Vie associative	Membre fondateur et ancien membre du bureau de l'ATSIR : Association des Troisième Cycle en Systèmes Informatiques Répartis

B.12 Publications

La section *Bibliography* de ce document commence par ma bibliographie personnelle. Les différentes publications sont annotées par le classement ARC (classement arrêté en 2011 mais utilisé ici à titre d'information : `http://www.arc.gov.au/era/journal_list_dev.htm`) du journal ou de la conférence correspondante.

Bibliography

Personal Bibliography

— PhD Thesis —

[CB-Bas04b] Cédric Bastoul. *Improving Data Locality in Static Control Programs*. PhD thesis, University Paris 6, Pierre et Marie Curie, France, December 2004.

— Patents —

[CB-LMV$^+$10] Allen Leung, Benoît Meister, Nicolas Vasilache, David Wohlford, Cédric Bastoul, Peter Szilagyi, and Richard Lethin. System, methods and apparatus for program optimization for multi-threaded processor architectures. In *Patent number US20100218196*, June 2010.

— International Journals and Book Chapters —

[CB-Bas11a] Cédric Bastoul. *Encyclopedia of Parallel Computing*, chapter Parallel Code Generation. Springer-Verlag, 2011. To appear, 8 double column pages.

[CB-BF05] Cédric Bastoul and Paul Feautrier. Adjusting a program transformation for legality. *Parallel processing letters*, 15(1):3–17, March 2005. 15 pages. ARC Ranking: B.

[CB-GVB$^+$06] 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. *International Journal of Parallel Programming*, 34(3):261–317, June 2006. 57 pages. ARC Ranking: A.

[CB-MAB$^+$10] Harm Munk, Eduard Ayguadé, Cédric Bastoul, Paul Carpenter, Zbigniew Chamski, Albert Cohen, Marco Cornero, Philippe Dumont, Marc Duranton, Mohammed Fellahi, Roger Ferrer, Razya Ladelsky, Menno Lindwer, Xavier Martorell, Cupertino Miranda, Dorit Nuzman, Andrea Ornstein, Antoniu Pop, Sebastian Pop, Louis-Noël Pouchet, Alex Ramírez, David Ródenas, Erven Rohou, Ira Rosen, Uzi Shvadron, Konrad Trifunović, and Ayal Zaks. Acotes project: Advanced compiler technologies for embedded streaming. *International Journal of Parallel Programming*, 39(3):397–450, June 2010. 54 pages. ARC Ranking: A.

[CB-PCP$^+$12] Eunjung Park, John Cavazos, Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *International*

Journal of Parallel Programming, 2012. Accepted for publication. 39 pages. ARC Ranking: A.

— International Conferences with Program Committee —

[CB-Bas03] Cédric Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30. Ljubljana, Slovenia, October 2003. 8 double column pages. Accepted papers: 40, Submitted: 60 (rate 67%). ARC Ranking: C.

[CB-Bas04a] Cédric 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, France, September 2004. 10 double column pages. Accepted papers: 23, Submitted: 122 (rate 19%). ARC Ranking: A. Student Award.

[CB-BF03a] Cédric Bastoul and Paul Feautrier. Improving data locality by chunking. In *CC'12 International Conference on Compiler Construction, LNCS 2622*, pages 320–335. Warsaw, Poland, April 2003. 15 pages. Accepted papers: 21, Submitted: 83 (rate 25%). ARC Ranking: A.

[CB-BF04] Cédric Bastoul and Paul Feautrier. More legal transformations for locality. In *Euro-Par'10 International Euro-Par conference, LNCS 3149*, pages 272–283. Springer-Verlag, Pisa, Italy, August 2004. 12 pages. Accepted papers: 124, Submitted: 352 (rate 35%). ARC Ranking: A. Distinguished Paper Award.

[CB-BPCB10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, LNCS, pages 283–303. Springer-Verlag, Paphos, Cyprus, March 2010. 20 pages. Accepted papers: 16, Submitted: 56 (rate 28%). ARC Ranking: A.

[CB-HBB$^+$09] Albert Hartono, Muthu Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the ACM International Conference on Supercomputing (ICS'09)*, pages 147–157. Yorktown Heights, New York, June 2009. 11 double column pages. Accepted papers: 47, Submitted: 191 (rate 24%). ARC Ranking: A.

[CB-PBB$^+$10] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Conference on Supercomputing (SC'10)*. New Orleans, LA, November 2010. 11 double column pages. Accepted papers: 51, Submitted: 253 (rate 20%). ARC Ranking: A.

[CB-PBB$^+$11] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: Convexity, pruning and optimization. In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (PoPL'11)*, pages 549–562. Austin, TX, January 2011. 14 double column pages. Accepted papers: 49, Submitted: 209 (rate 23%). ARC Ranking: A. HiPEAC Award.

[CB-PBCC08a] L.-N. Pouchet, C. Bastoul, A. Cohen, and S. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100. Tucson, Arizona, June 2008. 11 double column pages. Accepted papers: 34, Submitted: 184 (rate 18%). ARC Ranking: A. HiPEAC Award.

[CB-PBCV07] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *ACM International Conference on Code Generation and Optimization (CGO'07)*, pages 144–156. San Jose, California, March 2007. 13 double column pages. Accepted papers: 27, Submitted: 84 (rate 32%). ARC Ranking: A.

[CB-PCB$^+$06] 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*, pages 179–198. Ottawa, Canada, June 2006. 18 double column pages. ARC Ranking: unrated.

[CB-VBC06] 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 3923, pages 185–201. Springer-Verlag, Vienna, Austria, March 2006. 15 pages. Accepted papers: 20, Submitted: 71 (rate 28%). ARC Ranking: A.

[CB-VBGC06] Nicolas Vasilache, Cédric Bastoul, Sylvain Girbal, and Albert Cohen. Violated dependence analysis. In *Proceedings of the ACM International Conference on Supercomputing (ICS'06)*, pages 335–344. Cairns, Australia, June 2006. 10 double column pages. Accepted papers: 37, Submitted: 141 (rate 26%). ARC Ranking: A.

— International Workshops with Program Committee —

[CB-BCB$^+$10] Riyadh Baghdadi, Albert Cohen, Cédric Bastoul, Louis-Noël Pouchet, and Lawrence Rauchwerger. The potential of synergistic static, dynamic and speculative loop nest optimizations for automatic parallelization. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA'10)*. Saint-Malo, France, June 2010. 5 double column pages. ARC Ranking: unrated.

[CB-BCG$^+$03a] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, LNCS, pages 209–225. Springer-Verlag, College Station, Texas, October 2003. 15 pages. Accepted papers: 35, Submitted: 48 (rate 73%). ARC Ranking: unrated.

[CB-BVL$^+$09] Cédric Bastoul, Nicolas Vasilache, Allen Leung, Benoît Meister, David Wohlford, and Richard Lethin. Extended static control programs as a programming model for accelerators, a case study: Targetting ClearSpeed CSX700 with the R-Stream Compiler. In *PMEA'09 Workshop on Programming Models for Emerging Architectures*, pages 45–52. Raleigh, North Carolina, September 2009. 8 double column pages. Accepted papers: 10, Submitted: 18 (rate 55%). ARC Ranking: unrated.

[CB-HVB+10] Albert Hartono, Nicolas Vasilache, Cédric Bastoul, Allen Leung, Benoît Meister, Richard Lethin, and Peter Vouras. Automatic parallelization and locality optimization of beamforming algorithms. In *High Performance Embedded Computing Workshop (HPEC)*. MIT Lincoln Laboratory, Lexington, Massachusetts, September 2010. 2 pages doble colonnes. ARC Ranking: unrated.

[CB-LVM+10] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Manikandan Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010*, volume 425 of *ACM International Conference Proceeding Series*, pages 51–61. Pittsburgh, Pennsylvania, March 2010. 11 double column pages. ARC Ranking: unrated.

[CB-MLV+09] Benoît Meister, Allen Leung, Nicolas Vasilache, David Wohlford, Cédric Bastoul, and Richard Lethin. Productivity via automatic code generation for PGAS platforms with the R-Stream Compiler. In *APGAS'09 Workshop on Asynchrony in the PGAS Programming Model*. Yorktown Heights, New York, June 2009. 15 pages. ARC Ranking: unrated.

[CB-PBCC08b] Louis-Noël Pouchet, Cédric Bastoul, John Cavazos, and Albert Cohen. A note on the performance distribution of affine schedules. In *2nd Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion (SMART'08)*. Göteborg, Sweden, January 2008. 15 pages. ARC Ranking: unrated.

[CB-SDB07] Sébastien Salva, Clément Delamare, and Cédric Bastoul. Web service call parallelization using OpenMP. In *3rd International Workshop on OpenMP*, LNCS, pages 185–194. Springer-Verlag, Beijing, China, June 2007. 10 pages. Accepted papers: 14, Submitted: 28 (rate 50%), ARC Ranking: unrated.

— International Workshops without Program Committee —

[CB-BF03b] Cédric Bastoul and Paul Feautrier. Reordering methods for data locality improvement. In *CPC'10 Compilers for Parallel Computers*, pages 187–196. Amsterdam, The Netherlands, January 2003. 10 double column pages. ARC Ranking: unrated.

[CB-PBC07] Louis-Noël Pouchet, Cédric Bastoul, and Albert Cohen. LetSee: the LEgal Transformation SpacE Explorator. Third International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES'07), L'Aquila, Italia, July 2007. pages 247–251. ARC Ranking: unrated.

— National Conferences with Program Committee —

[CB-Bas02b] Cédric Bastoul. Une méthode d'amélioration de la localité basée sur des estimations asymptotiques du trafic. In *RENPAR'14*, pages 127–134. Hammamet, Tunisia, Avril 2002. 8 pages. ARC Ranking: unrated.

— Tool-Related Technical Reports —

[CB-ABB+01] Jaume Abella, Cédric Bastoul, Jean-Luc Béchennec, Nathalie Drach, Christine Eisenbeis, Paul Feautrier, Bjoern Franke, Grigori Fursin, Antonio Gonzalez, Toru Kisku, Peter Knijnenburg, Josep Llosa, Michael O'Boyle, Julien Sebot, and Xavier Vera. Guided transformations. Technical Report M3.D2, MHAOTEU ESPRIT project No 24942, february 2001. Related to the MHAOTEU toolset.

[CB-Bas02a] Cédric Bastoul. Generating loops for scanning polyhedra. Technical Report 2002/23, PRiSM, Versailles University, 2002. Related to the CLooG tool.

[CB-Bas08] Cédric Bastoul. Extracting polyhedral representation from high level languages. Technical report, LRI, Paris-Sud University, 2008. Related to the Clan tool.

[CB-Bas11b] Cédric Bastoul. OpenScop: A specification and a library for data exchange in polyhedral compilation tools. Technical report, LRI, Paris-Sud University, France, September 2011.

[CB-Bas12] Cédric Bastoul. Clay: the chunky loop alteration wizardry. Technical report, LRI, Paris-Sud University, 2012. Related to the Clay tool.

[CB-BCG+03b] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. Technical Report 4902, INRIA Rocquencourt, 2003. Related to the WRAP-IT tool.

[CB-BP12] Cédric Bastoul and Louis-Noël Pouchet. Candl: the chunky analyzer for dependences in loops. Technical report, LRI, Paris-Sud University, 2012. Related to the Candl tool.

[CB-FcCB02] Paul Feautrier, Jean-François Collard, and Cédric Bastoul. Solving systems of affine (in)equalities. Technical report, PRiSM, Versailles University, 2002. Related to the PIP/PipLib tool.

General Bibliography

[ABC+06] 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 *IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO'06)*, pages 295–305. IEEE Computer Society, Washington, DC, USA, 2006.

[ACG+04] 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.

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, June 1991.

[AK87] J. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AK02] J. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[Ban76] U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1976.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, 1988.

[Ban90] U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*, pages 192–219. Irvine, august 1990.

[Ban93] U. Banerjee. *Loop Transformations for Restructuring Compilers*. Kluwer Academic, 1993.

[Bar98] D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints*. PhD thesis, Université de Versailles Saint-Quentin, France, February 1998.

[BBK$^+$08a] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS'08, pages 225–234. Island of Kos, Greece, 2008.

[BBK$^+$08b] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Intl. Conf. on Compiler Construction (ETAPS CC 17)*. Budapest, Hungary, April 2008.

[BCC98] Denis Barthou, Albert Cohen, and Jean-François Collard. Maximal static expansion. In *ACM Symp. on Principles of Programming Languages (PoPL'98)*. San Diego, California, 1998.

[BDD$^+$07] Denis Barthou, Sebastien Donadio, Alexandre Duchateau, Patrick Carribault, and William Jalby. Loop optimization using adaptive compilation and kernel decomposition. In *ACM/IEEE Int. Symp. on Code Optimization and Generation*, pages 170–184. IEEE Computer Society, San Jose, California, March 2007.

[BDSV98] P. Boulet, A. Darte, G-A. Silber, and F. Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3):421–444, 1998.

[Ber66] A. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on Electronic Computers*, 15(5):757–763, October 1966.

[BF98] Pierre Boulet and Paul Feautrier. Scanning polyhedra without do-loops. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 4–11. Paris, France, October 1998.

[BGDR10] Uday Bondhugula, Oktay Günlük, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT'10, pages 343–352. ACM, Vienna, Austria, September 2010.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. of the 2008 ACM Conf. on Programming language design and implementation (PLDI'08)*. Tucson, AZ, USA, June 2008.

[BHZ08] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[BKK$^+$98] 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 *W. on Profile and Feedback Directed Compilation*. Paris, October 1998.

[CBF95] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. In *ACM Symp. on Principles and practice of parallel programming (PPoPP'95)*, pages 92–101. Santa Barbara, California, 1995.

[CCH08] Chun Chen, Jacqueline Chame, and Mary Hall. A framework for composing high-level loop transformations. Technical Report 08-897, USC Computer Science, June 2008.

[CCJ05] P. Carribault, A. Cohen, and W. Jalby. Deep Jam: Conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 291–300. St-Louis, Missouri, September 2005.

[CGH$^+$05] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. ACME: adaptive compilation made efficient. In *ACM SIGLPAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, pages 69–77. ACM Press, Chicago, IL, USA, 2005.

[CGP$^+$05] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ICS'05*, pages 151–160. Boston, Massachusetts, June 2005.

[Che12] Chun Chen. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 499–508. ACM, Beijing, China, 2012.

[Cho04] F. Chow. Maximizing application performance through interprocedural optimization with the PathScale EKO compiler suite, August 2004.

[CI96] Béatrice Creusillet and François Irigoin. Exact versus approximate array region analyses, lncs 1239. In *LCPC'96 9th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, pages 86–100, 1996.

[Cle08a] ClearSpeed Inc. *Cn Standard Libraries Reference Manual, Document 06-RM-1139.* 2008.

[Cle08b] ClearSpeed Inc. *CSX700 Floating Point Processor Datasheet, Document 06-PD-1425.* 2008.

[CMO06] J. Cavazos, J. E. Moss, and M. F. P. O'Boyle. Hybrid optimizations: Which optimization algorithm to use. In *(ETAPS CC 16)*. Vienna, Austria, April 2006.

[Col94] Jean-François Collard. Space-time transformation of while-loops using speculative execution. In *In Proc. of the 1994 Scalable High Performance Computing Conf*, 1994.

[Col95] Jean-François Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. Parallel Program.*, 23(2):191–219, 1995.

[CSS99] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9. ACM Press, Atlanta, GA, USA, July 1999.

[CST02] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomputing*, 23(1):7–22, 2002.

[CW99] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *2nd Workshop on Feedback-Directed Optimization*. Israel, November 1999.

[Dan51] G. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In T.C. Koopmans, editor, *Activity Analysis of Production and Allocation, Cowles Commission Monograph No. 13*, pages 339–347. John Wiley & Sons, Inc., New York, 1951.

[Dar00] A. Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.

[DR94] A. Darte and Y. Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20(5):679–710, 1994.

[DRV00] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser, 2000.

[ES92] Christine Eisenbeis and Jean-Claude Sogno. A general algorithm for data dependence analysis. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 292–302. ACM Press, Washington, D. C., United States, 1992.

[FCOT05] 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. Springer-Verlag, Barcelona, November 2005.

[Fea88a] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22 (3):243–268, 1988.

[Fea88b] Paul Feautrier. Array expansion. In *ICS*, pages 429–441. St Malo, France, July 1988.

[Fea91] P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[Fea92a] P. Feautrier. Some efficient solutions to the affine scheduling problem: one dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, october 1992.

[Fea92b] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[FJ05] Matteo Frigo and Steven G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[FO03] B. Franke and M. O'Boyle. Array recovery and high level transformations for dsp applications. In *CPC'10 Intl. Workshop on Compilers for Parallel Computers*, pages 29–38. Amsterdam, January 2003.

[FO05] B. Franke and M. O'Boyle. A complete compiler approach to auto-parallelizing c programs for Multi-DSP systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 16(3):234–245, march 2005.

[GC95] Martin Griebl and Jean-François Collard. Generation of synchronous code for automatic parallelization of while loops. In *Euro-PAR'95, LNCS 966*, pages 315–326, 1995.

[GFL00] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *Int. Journal of Parallel Programming*, 28(6):607–631, 2000.

[GFL04] M. Griebl, P. Faber, and C. Lengauer. Space-time mapping and tiling – a helpful combination. *Concurrency and Computation: Practice and Experience*, 16(3):221–246, March 2004.

[GGL98] Max Geigl, Martin Griebl, and Christian Lengauer. A scheme for detecting the termination of a parallel loop nest. In *Proc. GI/ITG FG PARS'98*, 1998.

[GGL99] Max Geigl, Martin Griebl, and Christian Lengauer. Termination detection in parallel loop nests with while loops. *Parallel Comput.*, 25(12):1489–1510, 1999.

[Gir05] Sylvain Girbal. *Optimisation d'applications - Composition de transformations de programme: modle et outils*. Phd thesis, University Paris-Sud 11, Orsay, France, September 2005.

[GKT91] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 15–29. New York, june 1991.

[GL94] Martin Griebl and Christian Lengauer. On scanning space-time mapped while loops. In *CONPAR 94 - VAPP VI: Proceedings of the Third Joint International Conference on Vector and Parallel Processing*, pages 677–688. London, UK, 1994.

[GLW98] Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 106–111, 1998.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1989.

[GR07] Gautam Gupta and Sanjay V. Rajopadhye. The z-polyhedral model. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'07*, pages 237–248. ACM, San Jose, California, USA, March 2007.

[Gri04] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. Facultät für Mathematik und Informatik, Universität Passau, 2004.

[GZA$^+$11] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *IMPACT 2011 First International Workshop on Polyhedral Compilation Techniques*. Chamonix, France, 2011.

[HAI$^+$05] I. Hurbain, C. Ancourt, F. Irigoin, M. Barreteau, J. Mattioli, and F. Paquier. A case study of design space exploration for embedded multimedia applications in SoCs. Technical Report A-361, CRI – École des Mines de Paris, february 2005.

[HKL73] R. J. Hanson, F. T. Krogh, and C. L. Lawson. A proposal for standard linear algebra subprograms. *ACM Signum Newsletter*, 8(16), 1973.

[HSP$^+$11] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, pages 225–245. Saarbrücken, Germany, 2011.

[Iri11] François Irigoin. Dependence abstractions. In *Encyclopedia of Parallel Computing*, pages 552–556. 2011.

[IT87] F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report ENSMP-CAI-87-E94, Ecole des Mines de Paris, Fontainebleau (France), 1987.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 319–329. San Diego, California, United States, 1988.

[JCP$^+$12] Alexandra Jimborean, Philippe Clauss, Benoît Pradelle, Luis Mastrangelo, and Vincent Loechner. Adapting the polyhedral model as a framework for efficient speculative parallelization. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 295–296. New Orleans, Louisiana, USA, 2012.

[JMS$^+$10] Byunghyun Jang, Perhaad Mistry, Dana Schaa, Rodrigo Dominguez, and David Kaeli. Data transformations enabling loop vectorization on multithreaded data parallel architectures. *SIGPLAN Not.*, 45(5):353–354, January 2010.

[KAP97] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN'97 Conf. on Programming Language Design and Implementation*, pages 346–357. Las Vegas, June 1997.

[Kel96] W. Kelly. *Optimization within a Unified Transformation Framework*. doctoral thesis, University of Maryland, 1996.

[KHW$^+$05] 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. on Architecture and Code Optimization*, 2(2):165–198, 2005.

[KKO00] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 237–246. IEEE Computer Society, Philadelphia, PA, USA, 2000.

[KKP90] X. Kong, D. Klappholz, and K. Psarris. The i test: A new test for subscript data dependence. In *ICPP'90 International Conference on Parallel Processing*, pages 204–211. St. Charles, August 1990.

[KMP$^+$96] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library. Technical report, University of Maryland, November 1996.

[KMW67] R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, july 1967.

[KP93] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126.1, University of Maryland Institute for Advanced Computer Studies, 1993.

[KP95] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. In *IEEE Intl. Conf. on Algorithms and Architectures for Parallel Processing (ICAPP'95)*, pages 153–162, April 1995.

[KPR95] 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.

[KS98] Kathleen Knobe and Vivek Sarkar. Array ssa form and its use in parallelization. In *ACM Symp. on Principles of Programming Languages (PoPL'98)*. California, 1998.

[KZM$^+$03] 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: Proc. of the 2003 ACM SIGPLAN Conf. on Language, compiler, and tool for embedded systems*, pages 12–23. ACM Press, San Diego, California, USA, 2003.

[LAB$^+$09] Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, P. Sadayappan, Yongjian Chen, Haibo Lin, and Tin fook Ngai. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 348–357. Raleigh, North Carolina, USA, September 2009.

[Lam74] Leslie Lamport. The parallel execution of do loops. *Communications of ACM*, 17(2): 83–93, 1974.

[Le 92] H. Le Verge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.

[Le 95] H. Le Verge. Recurrences on lattice polyhedra and their applications, April 1995. Unpublished work based on a manuscript written by H. Le Verge just before his untimely death in 1994.

[Le 96] M. Le Fur. Scanning parameterized polyhedron using Fourier-Motzkin elimination. *Concurrency - Practice and Experience*, 8(6):445–460, 1996.

[Lee98] Corinna Lee. UTDSP benchmark suite, 1998. `http://www.eecg.toronto.edu/~corinna/DSP`.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In *Int. Conf. on Concurrency Theory, LNCS 715*, pages 398–416. Hildesheim, August 1993.

[LF98] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3–4):649–671, 1998.

[LF05] Shun Long and Grigori Fursin. A heuristic search algorithm based on unified transformation framework. In *ICPPW '05: Proc. of the 2005 Intl. Conf. on Parallel Processing Workshops (ICPPW'05)*, pages 137–144. IEEE Computer Society, Washington, DC, USA, 2005.

[LF06] Shun Long and Grigori Fursin. Systematic search within an optimisation space based on unified transformation framework. *IJCSE Intl. J. of Computational Science and Engineering*, 2006.

[Lim01] A. Lim. *Improving Parallelism and Data Locality with Affine Partitioning*. PhD thesis, Stanford University, 2001.

[LL97] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *24th ACM Symposium on Principles of Programming Languages*, pages 201–214. Paris, France, January 1997.

[LO04] S. Long and M.F.P. O'Boyle. Adaptive Java optimisation using instance-based learning. In *ACM Intl. Conf. on Supercomputing (ICS'04)*, pages 237–246. Saint-Malo, France, June 2004.

[LP94] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, April 1994.

[LVW94] H. Le Verge, V. Van Dongen, and D. Wilde. Loop nest synthesis using the polyhedral library. Technical Report 830, IRISA, 1994.

[LW97] Vincent Loechner and Doran K. Wilde. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6):525–549, 1997.

[LYZ89] Z. Li, P. Yew, and C. Zhu. Data dependence analysis on multi-dimensional array references. In *ICS'3 ACM International Conference on Supercomputing*, pages 215–224. Heraklion, June 1989.

[MAL93] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *20þACM Symp. on Principles of Programming Languages (PoPL)*, pages 2–15. Charleston, South Carolina, January 1993.

[McBQ02] Antoine Monsifrot, François Bodin, and Rene Quiniou. A machine learning approach to automatic production of compiler heuristics. In *AIMSA '02: Proc. of the 10th Intl. Conf. on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50. Springer-Verlag, London, UK, 2002.

[MHL91] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 1–14. New York, NY, USA, 1991.

[MPNT04] R. MÃ$\frac{1}{4}$ller-Pfefferkorn, W. Nagel, and B. Trenkler. Optimizing cache access: A tool for source-to-source transformations and real-life compiler tests. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference*, pages 72–81. Pisa, august 2004.

[MVW$^+$11] Benoît Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. R-stream compiler. In *Encyclopedia of Parallel Computing*, pages 1756–1765. 2011.

[Nis98] Andy Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In *HPCN Europe 1998: Proc. of the Intl. Conf. and Exhibition on High-Performance Computing and Networking*, pages 987–989. Springer-Verlag, London, UK, 1998.

[nVi12] nVidia. *CUDA Compute Unified Device Architecture C Programming Guide (version 4.2)*. 2012.

[Pal07] M. Palkovič. *Enhanced Applicability of Loop Transformations*. PhD thesis, T. U. Eindhoven, The Netherlands, September 2007.

[PK04] K. Psarris and K. Kyriakopoulos. An experimental evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems*, 15(3): 196–213, March 2004.

[Pop06] Sebastian Pop. *The SSA Representation Framework: Semantics, Analyses and GCC Implementation*. doctoral thesis, École des Mines de Paris, 2006.

[Pou10] Louis-Noël Pouchet. *Interative Optimization in the Polyhedral Model*. PhD thesis, University of Paris-Sud 11, Orsay, France, January 2010.

[PP91] P. Petersen and D. Padua. Experimental evaluation of some data dependence tests. Technical Report CSRD 1080, University of Illinois at Urbana-Champaign, February 1991.

[PP96] P. Petersen and D. Padua. Static and dynamic evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1121–1132, november 1996.

[PPC$^+$11] Eunjung Park, Louis-Noël Pouchet, John Cavazos, Albert Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *CGO'11 Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 119–129. Chamonix, France, 2011.

[PSX$^+$04] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *J. of High Performance Computing and Applications, special issue on Automatic Performance Tuning*, 18(1):21–45, 2004.

[Pug91a] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the third ACM/IEEE conference on Supercomputing*, pages 4–13. Albuquerque, August 1991.

[Pug91b] W. Pugh. Uniform techniques for loop optimization. In *ICS'91*, pages 341–352. Cologne, Germany, June 1991.

[PW93] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *LCPC'93 Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing, LNCS 768*, pages 546–566, 1993.

[QD89] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *The Journal of VLSI Signal Processing*, 1(2):95–113, October 1989.

[QR00] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, September 2000.

[QRW00] 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.

[Qui84] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of the 11th Annual Symposium on Computer Architecture*, pages 208–214. Ann Arbor, USA, June 1984.

[Ram95] J. Ramanujam. Beyond unimodular transformations. *J. of Supercomputing*, 9(4): 365–389, 1995.

[RK88] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementations on processor arrays. *Proc. IEEE*, 76(3):259–282, March 1988.

[RKRS07] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. *SIGPLAN Notices, Proc. of the 2007 PLDI Conf.*, 42(6):405–414, 2007.

[RP95] Lawrence Rauchwerger and David A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *ACM Conf. on Programming Language Design and Implementation (PLDI'95)*, June 1995.

[RPF86] Sanjay V. Rajopadhye, S. Purushothaman, and Richard Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *FSTTCS Foundations of Software Technology and Theoretical Computer Science, Sixth Conference*, pages 488–503. New Delhi, India, December 1986.

[RPR07] Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *ACM Intl. Conf. Supercomputing (ICS'07)*, 2007.

[RR03] Silvius Rus and Lawrence Rauchwerger. Hybrid dependence analysis for automatic parallelization. Technical report, Parasol Laboratory, Texas A&M University, 2003.

[RRH03] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: Static & dynamic memory reference analysis. *Intl. J. of Parallel Programming*, 31(4), 2003.

[RVRA08] Easwaran Raman, Neil Vachharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 175–184. Boston, MA, USA, 2008.

[RVVA04] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, September 2004.

[SAMO03] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Notices*, 38(5):77–90, 2003.

[Sch86] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.

[SLM12] Rachid Seghir, Vincent Loechner, and Benoît Meister. Integer affine transformations of parametric z-polytopes and applications to loop nest optimization. *TACO*, 9(2):8, 2012.

[SSH10] I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 513–522. Vienna, Austria, 2010.

[ST92] Vivek Sarkar and Radhika Thekkath. A general framework for iteration-reordering loop transformations. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 175–187. San Francisco, California, United States, 1992.

[TCE$^+$10] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*. Pisa, Italy, 2010.

[TNC$^+$09] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 327–337. IEEE Computer Society, Washington, DC, USA, 2009.

[Tri11] Konrad Trifunovic. *Efficient search-based strategies for polyhedral compilation: algorithms and experience in a production compiler*. doctoral thesis, University of Paris-Sud, 2011.

[TVA05] S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *J. of Instruction-level Parallelism*, volume 7, January 2005.

[TVVA03] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *IEEE/ACM Intl. Symp. on Code generation and optimization (CGO'03)*, pages 204–215. IEEE Computer Society, Washington, DC, USA, 2003.

[Vas07] Nicolas Vasilache. *Scalable optimization techniques in the polyhedral model*. Phd thesis, University Paris-Sud 11, Orsay, France, September 2007.

[VCP07] Nicolas Vasilache, Albert Cohen, and Louis-Noël Pouchet. Automatic correction of loop transformations. In *PACT*, pages 292–304. Brasov, Romania, September 2007.

[Ver10] Sven Verdoolaege. *isl*: An integer set library for the polyhedral model. In *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software*, pages 299–302. Kobe, Japan, September 2010.

[Ver12] Sven Verdoolaege. Polyhedral extraction tool. In *IMPACT'12 International Workshop on Polyhedral Compilation Techniques*. Paris, France, January 2012.

[Viv02] Frédéric Vivien. On the optimality of Feautrier's scheduling algorithm. In *Intl. Euro-Par Conf. on Parallel Processing (EURO-PAR'02)*, pages 299–308. Springer-Verlag, London, UK, 2002.

[WB87] Michael Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.

[Wil93] D. Wilde. A library for doing polyhedral operations. Technical report, IRISA, 1993.

[WL91] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, October 1991.

[Wol82] Michael Joseph Wolfe. *Optimizing supercompilers for supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, October 1982.

[Wol87] M. Wolfe. Iteration space tiling for memory hierarchies. In *3rd SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, december 1987.

[Wol92] M. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of computer science, Stanford University, California, 1992.

[Wol95] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, 1995.

[Won95] D. G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, University of Maryland, 1995.

[WPD00] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27(1–2):3–35, 2000.

[WT92] M. Wolfe and C. W. Tseng. The power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, 1992.

[Xue94] J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, 1994.

[YAI95] Yi-Qing Yang, Corinne Ancourt, and Francois Irigoin. Minimal data dependence abstractions for loop transformations. *International Journal of Parallel Programming*, 23 (4):359–388, 1995.