Efficient search-based strategies for polyhedral compilation : algorithms and experience in a production compiler

Konrad Trifunovic

UNIVERSITÉ DE PARIS-SUD 11
U.F.R. SCIENTIFIQUE D'ORSAY

In Partial Fulfillment of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Discipline: Computing science

Konrad TRIFUNOVIC

Subject:

EFFICIENT SEARCH-BASED STRATEGIES FOR POLYHEDRAL COMPILATION: ALGORITHMS AND EXPERIENCE IN A PRODUCTION COMPILER

Thesis supervisor: Dr. Albert Cohen

Thesis committee:

M. John Cavazos	University of Delaware
M. Philippe Clauss	University of Strasbourg
M. Albert Cohen	Senior Research Scientist at INRIA
M. Yannis Manoussakis	University Paris Sud
M. Ayal Zaks	IBM Haifa Research Labs

Contents

Notation

Mathematical	
$\mathbb{R}$	a set of real numbers
$\mathbb{Z}$	a set of integer numbers
$\mathbb{Q}$	a set of rational numbers
$(i_1, i_2, \ldots, i_n)^T$	a column vector represented in a row
Polyhedral model	
$\mathbf{i}$	iteration vector
$\mathbf{t}$	timestamp vector
$\mathcal{S}$	statement set
S	a polyhedral statement
$(S, \mathbf{i})$	statement instance
$\mathcal{D}^S$	an iteration domain of statement S
$\lvert\mathcal{D}^S\rvert$	iteration volume
$\mathrm{A} \in \mathcal{A}$	an array in an array set
θ^S	a scheduling function for statement S
R	a data reference
$f(\mathbf{i})$	access function
$\ell(\mathbf{i})$	linearized access function
$G = (V, E)$	dependence graph
$\mathcal{P}_e$	dependence polyhedron

Chapter 1

Introduction

The responsibility of a *compiler* is to translate the programs written in a language suitable for use by human programmer into the machine language of the target machine. Naturally, the compiler is required to produce the *correct* translation to the machine code.

In 1944 John von Neumann has laid the foundation to almost all widely used architectures today - that of decoupling the program memory from data memory. This also implied a sequential execution model, in which the machine instructions are executed sequentially, changing the global state of the memory through stores and loads.

Besides producing the correct code, compilers were enhanced with *optimizations*, so that the translated machine code performance could match the performance of the hand-crafted one. Contributions in scalar optimizations, interprocedural analysis, instruction scheduling and register allocation mainly achieved this goal.

The major driving forces for the sustained computing performance increase were the advancements in instruction level parallelism (ILP) and ever increasing clock rates of uniprocessors - according to the Moore's law. This obviated the need for advanced compiler optimizations, since the performance boost could have been obtained by running non-modified sequential codes on the new machines. Advanced compiler optimizations were considered as a domain of specialized, scientific computing community, specializing in the expensive parallel and vector machines programmed mainly in FORTRAN.

End of uniprocessor era

The trends have changed in the early 2000s, since it was noticed that increasing the ILP brought diminishing returns, altogether with physical limitations and power dissipation that have put an end to the increase in the clock-speeds of uniprocessors. The capacity to increase the number of transistors has since then been used in another direction: providing *multiple cores* on the single chip.

The era of multi-core

Multi-core processors are now in widespread use in almost all areas of the computing: in desktop or laptop machines, accelerators like the Cell Broadband Engine, GPUs, GPGPUs and in mobile embedded devices implemented as MP-SoCs (Multiprocessor System-on-Chip).

In the scientific computing field, where specialized multiprocessors have been used since 1960's, the commodity multicore and GPGPUs are entering very rapidly, as main building blocks of the computational clusters.

Optimizing compilers are again in demand

To harness the power of multiple cores and complex memory hierarchies, the need for powerful compiler optimizations is now again in high demand. Compilers are now required to efficiently map the semantics of the, mostly imperative, high-level programming languages into the efficient use of target architecture resources [82].

Increasing the number of cores on a chip does not improve the performance of the legacy, single-threaded applications. What is more, the sequential programming paradigm, due to the simplicity of the von Neumann computational model, is still used to program the new applications. Those applications do not benefit from the performance improvements of the new architectures.

In order to utilize the peak performance of the current multi-core architectures, the optimizing compiler is responsible for finding the intrinsic *parallelism* within the computations of the source program. Also, the memory hierarchy has to be utilized carefully, in order to narrow down the gap between latencies of the main memory and the computing core - the *memory wall* problem.

Semantic gap

There is a notorious semantic gap between the semantics of the currently used imperative programming languages and the target platform execution model. Nevertheless, while more than 200 parallel programming paradigms and environments have been proposed, few of them are in a widespread use - the majority of the applications are still written in imperative sequential languages such as C, C++ or FORTRAN.

Most current applications are written using a single thread of execution. There is a good reason for this - writing single-threaded code is intuitive and practical: one has to manage only a single state space, it is easily analysable, the proof of correctness could be obtained in an inductive way, the execution trace is predictable and reproducible.

On the other hand, writing the parallel programs introduces the problems such as communication, synchronization and data transfers. Writing hand-crafted parallel code is costly, error-prone and, mostly importantly, impossible to debug, since the race conditions in parallel program are generally non-reproducible. Hand-crafted parallel programs are *non-portable*, since the synchronization mechanisms, communication costs, scheduling policies and other issues are highly platform specific. The non-portability manifests itself in two ways: platform non-portability and *performance non-portability*.

Automatic program transformations

Very promising solution to the mentioned semantic gap problem is to enable the *automatic transformation* of sequential code into the form that best fits the target execution architecture.

The task of the compiler is now to understand the semantics of the input sequential source program and to perform the transformation, while preserving the semantics, into the form that best fits the target platform execution model, e.g., parallel threads, vectorized instructions, memory accesses clustered into blocks. This is a challenging task, since all the responsibility of translating the semantics into performance now relies on the compiler.

Loop Transformations

Focusing the program transformations on loop nests is particularly interesting, since the majority of compute-intensive applications spend most of their execution time in loops. Loop transformations have a long history and they have been used in restructuring compilers for enabling automatic parallelization and vectorization of the scientific FORTRAN codes.

Loop transformations potentially have a very powerful performance impact, since they might enable parallelism, vectorization or memory locality. The major challenge for efficiently employing the loop transformations is to find the best *composition* of loop transformations and to *predict* the performance impact of those transformations.

The Polyhedral Model

The polyhedral model is a powerful algebraic abstraction for reasoning about loop transformations expressed as *schedules*. The polyhedral model might be viewed as an *abstract interpretation* framework for reasoning about dynamic program execution statically - at compile time. The foundations of the polyhedral model have their roots in operational research and systolic array design.

An execution trace of the sequential program is modelled as an ordered sequence of *statement instances*. It is assumed that each statement of the input program might be executed multiple times in a loop, giving rise to statement instances - one instance for each execution of the loop.

Single-Shot Optimization Approach

Traditionally, the program transformation using the polyhedral model is done in a sequence of three steps: (1) getting the program into the polyhedral model, (2) computing the scheduling transformation expressed in the polyhedral model, and (3) generating the transformed code from the polyhedral model.

The crucial step is the *scheduling* step that provides the actual program transformation. The automatic scheduling algorithms [67, 34] based on integer linear programming exist, and they are based on the *best effort* heuristics.

The advantage of the best effort scheduling heuristic is that the transformation is *computed* within a single step. Hopefully, the best possible transformation within the space of legal transformations is chosen.

The space of legal (loop) transformations is huge, and selecting the best transformation within this space is still an open problem. The best effort heuristics rely on a limited performance predicting cost-functions, abstracting away the architectural details of the target platform. The optimal transformation might easily be mispredicted.

Iterative Optimization Approach

Due to the intrinsic difficulty of selecting the best program transformation within a *single step*, a viable alternative approach is the *iterative* feedback-directed compilation. Iterative optimization resorts to *testing* the different program transformations. For each transformation, the code is generated. The code is then executed and the runtime impact of the transformation is evaluated.

A carefully crafted heuristic for generating the multiple program transformations is the key component of this approach. Experimental evidence confirms that iterative approach can be much better at adapting the semantics of the program to the execution platform than the single-shot, model based heuristics. Obviously - by trying more times, there is a better chance of getting the optimal transformation.

While the iterative approach might give better results in terms of the output program performance, it has one fundamental drawback: the whole process might be very *time consuming*. It is not uncommon for the iterative approach to take hours or days to converge to the best solution.

The iterative approach breaks the traditional view of the compilation: the compiler is now a component of the optimizing feedback loop, and not a self-standing tool that transforms the input source code to the binary. While the iterative compilation is a good approach to specialized program optimization, the two characteristics of this approach - that of having an unpredictable running time and reliance on the actual generated program execution - preclude it from being incorporated into the *general purpose* compiler.

Problem Statement and Proposed Solution

Our view is that the single-shot, best effort heuristics for loop optimizationsare not precise enough to model the complex performance characteristics of the modern architectures. On the other hand, the iterative compilation approach, due to its unpredictable runtime and reliance on experimental program evaluation is not directly employable in the general purpose compiler.
The **problem statement** could be summarized as follows:

> We want to keep the *effectiveness* of the model-based transformation approach, while bridging the gap between transformation prediction *precision* between single-shot and iterative approaches.

In order to achieve this, we propose a **solution** in the form of a new, search-based transformation approach, that could be summarized as follows:

> *A new transformation search approach is based on the controlled enumeration of the transformation candidates, each transformation candidate being evaluated at the compilation time, according to a precise cost function based on the target machine model.*

We postulate that the new proposed solution brings us closer to solving the stated problem. The new approach takes the benefits of both model-based and iterative approaches. The *effectiveness* is achieved by having an expressive but limited search space, whose complexity could be controlled. The fact that the cost function is evaluated at compile-time, and not through experimental evaluation, brings the *predictability* of the running time of the transformation search, allowing it to be incorporated into the general purpose compiler.

The *precision* is achieved by the fact that we use a machine dependent cost function that directly evaluates the execution cost on a specified machine. This function is much more precise than the simplistic linear cost functions, that are used in the current model-based approaches. Though, this precision comes with a price: non-linear cost functions cannot be optimized using linear programming machinery, as it is done in the current model-based approaches - our solution uses a *search based* strategy to evaluate those non-linear cost functions.

Contributions

The main contributions of this thesis are to provide algorithms, methodologies and the compilation framework that enable our new search-based strategy. The relevant contributions are detailed in the respective chapters.

Our contributions are built on top of the polyhedral model, which serves as the theoretical foundation of the work. The polyhedral model is used to express the data dependences, the cost-model functions and schedules.

Contrary to the current source-to-source polyhedral frameworks, we have investigated the direct polyhedral compilation on the low-level three address code. While being a challenging task, this low-level of abstraction is the foundation for our precise performance cost-modelling that is out of reach of the current source-to-source polyhedral compilers.

We take a fresh look at the problem of finding the program transformations. Contrary to the current linear cost-function based scheduling algorithms, we propose a *search strategy* that evaluates the complex cost-model function on the discrete set of legal transformations represented as *decision diagrams*. By doing so, we take the benefits of the iterative optimization, while not requiring the generation of the transformed programs and measuring their actual runtime.

Thesis overview

The thesis is organized in 3 parts and a conclusion. Contributions and related work are provided in each chapter.

The first part of the thesis gives the background material, starting from the basic mathematical definitions. The polyhedral model is defined, together with the detailed discussion of this abstract representation. Later, we discuss the topics of representing the program semantics, polyhedral transformations and legality of those transformations. The mathematical background is included so that the manuscript is self-contained - a reader familiar with the topic of linear optimization might directly skip to the sections devoted to the polyhedral program model.

The second part of the thesis is a detailed description of our three-address-code polyhedral compilation framework - GRAPHITE. The crucial design decisions were explained, together with our contributions in not so well investigated topics. A novel approach for efficiently handling the restrictions imposed by memory-based dependences is explained as well.

The third part is the core subject of the thesis. It is dedicated to the description of the search-based transformation methodology. A detailed explanation of our precise, machine-level cost-model is provided. The final part of the thesis is our proposal for the new search strategy, starting with a review of the most important related work in this field. We show how our search strategy is built on an enumeration of the discrete sets of legal transformations, represented as decision diagrams.

Our techniques are implemented in GRAPHITE polyhedral framework, that is a part of the widely used GCC compiler.

Part I

Basic Concepts

Chapter 2

Background

The polyhedral model representation is based on linear algebra, linear programming and convex optimization theory.

In this chapter we present the polyhedral model and the mathematical notations we use in the rest of the dissertation. The basic linear algebra and convex optimization background is given as well.

The Section 2.1 gives the necessary mathematical background and it is expected to be self-contained. A complete coverage of the material could be found in [143]. The Section 2.2 gives an introduction to the polyhedral model and places it in the context of the compilers. The Section 2.3 goes into the details of the polyhedral representation and covers the basic components of this representation. An extensive coverage of the literature and an historical overview is given in Section 2.4.

2.1 Mathematical background

We denote by $\mathbb{R}$, $\mathbb{Z}$ and $\mathbb{Q}$ a field of real, integer and rational numbers respectively. A column vector w is denoted by $\mathbf{w} = (w_1, \ldots, w_m)^T$.

Given two vectors $\mathbf{a} \in \mathbb{R}^n$ and $\mathbf{b} \in \mathbb{R}^m$ we will denote a *concatenation* of two vectors: $\mathbf{c} = (\mathbf{a}, \mathbf{b})^T \in \mathbb{R}^{n+m}$. A concatenated vector c if formed by taking n components from a vector a and m components from a vector b:

$$\mathbf{c} = (c_1, \ldots, c_n, c_{n+1}, \ldots, c_{n+m})^T = (a_1, \ldots, a_n, b_1, \ldots, b_m)^T$$

The vector concatenation notation will be used very often to distinguish several parts of a vector.

In a similar way, given a matrix $A \in \mathbb{R}^{k \times n}$ and a matrix $B \in \mathbb{R}^{k \times k}$, we can *compose* two matrices to get: $C = (A|B)$. A composed matrix C is obtained by taking n columns from the matrix A and m columns form the matrix B:

$$C = (C_{\bullet,1} \ldots C_{\bullet,m+n})$$

where $C_{\bullet,i}$ denotes i-th column of the matrix C. In the same way $C_{k,\bullet}$ denotes k-th rows of the matrix C.

Definition 2.1.1 (Affine function)**.** A n-dimensional function $f : \mathbb{R}^m \to \mathbb{R}^n$ is affine if and only if it can be expressed as follows:

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

where $\mathbf{b} = (b_1, \ldots, b_n)^T \in \mathbb{R}^n$ is a column vector and $A \in \mathbb{R}^{n \times m}$ is a matrix with n rows and m columns.

Definition 2.1.2 (Affine relation)**.** A relation $F \subset \mathbb{R}^m \times \mathbb{R}^n$ is affine if and only if:

$$\forall \mathbf{v} \in \mathbb{R}^m, \forall \mathbf{w} \in \mathbb{R}^n : (\mathbf{v}, \mathbf{w}) \in F \iff \mathbf{w} = A\mathbf{v} + \mathbf{b}$$

where $\mathbf{b} = (b_1, \ldots, b_n)^T \in \mathbb{R}^n$ is a column vector and $A \in \mathbb{R}^{n \times m}$ is a matrix with n rows and m columns.

Definition 2.1.3 (Affine hyperplane)**.** For each vector $\mathbf{a} \in \mathbb{R}^n$ and scalar $b \in \mathbb{R}$ the set X of all vectors $\mathbf{x} \in \mathbb{R}^n$ such that:

$$\mathbf{a}^T\mathbf{x} = b$$

defines an affine hyperplane.

Definition 2.1.4 (Affine half-space)**.** For each vector $\mathbf{a} \in \mathbb{R}^n$ and scalar $b \in \mathbb{R}$ the set X of all vectors $\mathbf{x} \in \mathbb{R}^n$ satisfying the following affine inequality:

$$\mathbf{a}^T\mathbf{x} \geq b$$

defines a topologically closed affine halfspace. If we replace $\geq$ by $>$ we will get a definition of topologically open affine halfspace.

Definition 2.1.5 (Convex polyhedron)**.** The set $\mathcal{P} \subseteq \mathbb{R}^n$ is *convex polyhedron* if it can be represented as an intersection of a finite number of affine half-spaces of $\mathbb{R}^n$.
Each half-space is called a *face* of the polyhedron. The set of all affine inequalities representing faces of the polyhedron can be compactly represented by an matrix A and a vector $\mathbf{b}$:

$$\mathcal{P} = \{\mathbf{x} \in \mathbb{R}^n | A\mathbf{x} + \mathbf{b} \geq \mathbf{0}\}$$

where $A \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$, m being the number of affine inequalities.

Definition 2.1.6 (Parametric polyhedron)**.** The set $\mathcal{P}(\mathbf{g}) \subseteq \mathbb{R}^n$ is *parametric polyhedron*, parametrized by a vector $\mathbf{g} \in \mathbb{R}^k$, if it is a convex polyhedron that can be represented as:

$$\mathcal{P}(\mathbf{g}) = \{\mathbf{x} \in \mathbb{R}^n | A\mathbf{x} + B\mathbf{g} + \mathbf{b} \geq \mathbf{0}\}$$

where $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{m \times k}$ and $\mathbf{b} \in \mathbb{R}^m$. The number of parameters in a parameter vector is k and m is the number of affine inequalities.

Definition 2.1.7 (Polytope, bounded polyhedron)**.** A polyhedron $\mathcal{P} \subseteq \mathbb{R}^n$ is called a *bounded polyhedron* if there exists a $\lambda \in \mathbb{R}_+$ such that the polyhedron $\mathcal{P}$ is contained in a bounding box:

$$\mathcal{P} \subseteq \{\mathbf{x} \in \mathbb{R}^n | -\lambda \leq x_j \leq \lambda, 1 \leq j \leq n\}$$

For the program analysis problems that we are interested in, we will consider only integer solutions, so we restrict all vectors to contain only integer components: $\mathbf{x} \in \mathbb{Z}^n$. Also, $A \in \mathbb{Z}^{n \times m}$ and $\mathbf{b} \in \mathbb{Z}^n$. For more details please refer to [143]. An efficient implementation of the presented concepts is given by [12].

2.2 The polyhedral model

Classical compiler internal representations are based on a syntax and semantics of imperative programs: abstract syntax trees, control-flow graph, def-reach chains, SSA form, three-address-code, basic-blocks [3, 114]. Those representations capture a syntax, control and data dependences in a program.

In such reduced representations of the dynamic execution trace, each statement of a high-level program occurs only once, even if it is executed many times (e.g., when enclosed within a loop). Representing a program this way is not convenient for aggressive loop optimizations which operate at the level of dynamic *statement instances*.

Compile-time constraints and lack of adequate algebraic representation of loop nest semantics prevent traditional compilers from adapting the schedule of statement instances of a program to best exploit architecture resources. For example, compilers typically cannot apply loop transformations if data dependences are non-uniform [163] or simply because profitability is too unpredictable.

A well known alternative approach, facilitating complex loop transformations, represents programs in the *polyhedral model*. This model is a flexible and expressive representation for loop nests with statically predictable control flow. Such loop nests, amenable to algebraic representation, are called *static control parts* (SCoP) [64, 74].

The polyhedral model captures an exact execution semantics of a statically predictable control flow program. We can think of the polyhedral model as an *abstract interpretation* framework [51] in which we can precisely reason about a dynamic execution of a program at a compilation time.

2.2.1 Static Control Parts

Providing a static analysis of a dynamic execution trace of an arbitrary program is undecidable problem [67]. In order to apply the polyhedral model to real computing problems, we have to restrict the scope of the analysis to program parts with statically determinable control flow. Those program parts are contained in *affine loops*.

Definition 2.2.1 (Affine loops)**.** Affine loop nests are nests of perfectly or imperfectly [1] nested loops with constant strides, having their loop bounds and memory accesses expressed as affine functions of outer loop induction variables and global parameters.

The scope of the polyhedral program analysis and manipulation are *Static Control Parts* containing sequences of affine loop nests and conditionals with boolean expressions of affine inequalities. This class of programs includes non-perfectly nested loops, non-rectangular loops and conditionals with boolean expressions of affine inequalities [67]. Global parameters are symbolic variables that are invariant inside a loop.

We will use an abbreviation SCoP (Static Control Part) to denote the parts of a program amenable for polyhedral analysis [74]. In Chapter 4 we will show an algorithmic approach that is used in our tool to extract SCoP parts from an arbitrary imperative program. An example of the program part that fits in SCoP category is shown in Figure 2.1. The global parameters for this affine loop nest are `N` and `K`.

2.3 Polyhedral representation

The polyhedral program representation is based on the concept of dynamic *statement instances*, which is the basic atomic unit of a program execution. The set of those instances is modelled in a linear algebraic framework, which we are going to present in the subsequent parts of this section.

We will define a concept of the *polyhedral statement* and the three components of the polyhedral representation:

– Iteration domains - capturing the set of dynamic statement instances
– Data access functions - capturing the memory access patterns expressed as affine functions of loop induction variables
– Scheduling functions - capturing the relative execution order of the statement instances

Building on those primitives, we will show in Section 3.2 how to statically extract the semantics of the SCoP programs. Also, we will show how to express the transformations in the polyhedral representation in Section 3.3.

2.3.1 Polyhedral statements

The polyhedral model is an abstraction that expresses the program execution trace as an execution of the dynamic *polyhedral statements*.

This abstraction is built on top of some internal representation used in a compiler. Depending on the level of abstraction, one polyhedral statement could correspond to:

1. syntactic statement - a source-level statement in a source program.
2. basic block of a low-level internal compiler representation
3. three-address code instruction

The polyhedral model is mostly used in the context of source-to-source compilers [74, 34, 89, 113, 85]. Thus, the first case, where a polyhedral statement directly corresponds to a source-level statement in a source program, is the most common. It is also the most natural correspondence for presentation purposes.

In this dissertation, we investigate the application of the polyhedral model to the low-level, three address code compilation. Thus, in Chapter 4, we will also use an abstraction level where a polyhedral statement corresponds to a basic block of low-level internal compiler representation.

1. A *perfect loop nest* is a set of nested loops if all statements are contained in an innermost nested loop. Otherwise a loop nest is called *imperfect loop nest*. In this work we consider a general class of imperfectly nested loops, assuming that perfectly nested loops are contained in this class.

In the rest of this dissertation we will implicitly use a term statement to refer to an abstract polyhedral statement of the polyhedral model. Depending on the context, the term *statement* might refer to a polyhedral statement corresponding to the source-level statement[2], or to the basic block in a low-level internal representation. We will try to make this distinction clear in the following chapters.

```
    for (v = 0; v < N; v++)
      for (h = 0; h < N; h++) {
S1:     s = 0;
        for (i = 0; i < K; i++)
          for (j = 0; j < K; j++)
S2:         s += image[v+i][h+j] * filter[i][j];
S3:     out[v][h] = s >> factor;
      }
```

Figure 2.1 – Main loop kernel in Convolve

2.3.2 Iteration domains

Once we have defined an abstract polyhedral statement, we are interested in analyzing dynamic *statement instances*. An execution of an abstract sequential program might be seen as a totally-ordered interleaving of statement instances. Each statement might be executed multiple times in a loop.

Definition 2.3.1 (Iteration vector)**.** The *iteration vector* $\mathbf{i}$ of a statement S consists of values of the induction variables of all loops surrounding the statement.

Each statement S is executed once for each possible value of a surrounding loop induction variables We denote a statement by S and a statement instance by $(S,\mathbf{i})$, where $\mathbf{i}$ is an iteration vector of a statement.

If a statement S belongs to a SCoP then the set of all iteration vectors $\mathbf{i}$ relevant for S can be represented by a parametric polytope:

$$\mathcal{D}^S(\mathbf{g}) = \left\{\mathbf{i} \in \mathbb{Z}^n \mid A \cdot \mathbf{i} + B \cdot \mathbf{g} + \mathbf{b} \geq \mathbf{0}\right\}$$

which is called the *iteration domain* of S, where $\mathbf{g}$ is the vector of *global parameters* whose dimension is n_g. Global parameters are invariants inside the SCoP, but their values are not known at compile time (parameters representing loop bounds for example).

We will combine matrices A, B and a column vector $\mathbf{b}$ in one matrix $D = (A \mid B \mid \mathbf{b})$. We will also concatenate vectors $\mathbf{i}$, $\mathbf{g}$ to form an *homogeneous* column vector $(\mathbf{i},\mathbf{g},1)^T$, where last component is a scalar constant 1. We can then provide a more compact representation of a polyhedron:

$$\mathcal{D}^S(\mathbf{g}) = \left\{\mathbf{i} \in \mathbb{Z}^n \mid \mathrm{D} \cdot (\mathbf{i},\mathbf{g},1)^T \geq \mathbf{0}\right\}$$

Let us consider a computational kernel given in Figure 2.1. We will name the components of the iteration vector according to loop induction variables and global parameters given in a source code. The resulting iteration domain polyhedron for statement S_1 is then represented as:

2. Mainly used for presentation purposes

$$\left[\begin{array}{cc|cc|c} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & -1 \end{array}\right] \begin{pmatrix} v \\ h \\ N \\ K \\ 1 \end{pmatrix} \geq \mathbf{0} \quad \left| \begin{array}{l} v \geq 0 \\ v \leq N-1 \\ h \geq 0 \\ h \leq N-1 \end{array} \right.$$

We can represent the same iteration domain polyhedron in a more concise way:
$\mathcal{D}^{S_1} = \left\{(v,h)^T \in \mathbb{Z}^2 \mid 0 \leq v,h \leq N-1\right\}$
The iteration domains for statements S_2 and S_3 are represented as follows:
$\mathcal{D}^{S_2} = \left\{(v,h,i,j)^T \in \mathbb{Z}^4 \mid 0 \leq v,h \leq N-1 \wedge 0 \leq i,j \leq K-1\right\}$
$\mathcal{D}^{S_3} = \left\{(v,h)^T \in \mathbb{Z}^2 \mid 0 \leq v,h \leq N-1\right\}$

2.3.3 Data access functions

Given an affine program, all accesses to memory locations must be expressed through array references. We will treat accesses to scalar values as accesses to single-element arrays `A[0]`.

A data reference to an array $\mathrm{A} \in \mathcal{A}$ is denoted by $R = \langle \mathrm{A}, f \rangle$, where f is a *subscript function*. Together, they describe an array access of the form $\mathrm{A}[f(\mathbf{i})]$. A subscript function is a function that maps an iteration vector $\mathbf{i}$ to an array subscripts:

$$f(\mathbf{i}) = F \cdot (\mathbf{i}, \mathbf{g}, 1)^T$$

where $F \in \mathbb{Z}^{m \times (n+n_g)}$ is a matrix containing m rows - m being the number of array subscripts, n and n_g being the iteration vector and global parameters vector sizes respectively.

In affine programs all subscript functions must be expressed as affine functions. A given statement S can contain multiple data references. A set of write and read data references of a statement S is denoted as $\mathcal{W}^S$ and $\mathcal{R}^S$ respectively.

As an example, let us take the statement S_2 from the example in Figure 2.1:

```
S2:       s += image[v+i][h+j] * filter[i][j];
```

The set of write and read references is the following:
$\mathcal{W}^{S_2} = \{\langle \mathtt{s}, f(\mathbf{i}) = 0 \rangle\}$
$\mathcal{R}^{S_2} = \{\langle \mathtt{s}, f(\mathbf{i}) = 0 \rangle, \langle \mathtt{image}, f_1 \rangle, \langle \mathtt{filter}, f_2 \rangle\}$
The access functions that map the iteration vectors to array subscripts have the following form:

$$f_1((v,h,i,j)^T) = \begin{pmatrix} v+i \\ h+j \end{pmatrix}$$

$$f_2((v,h,i,j)^T) = \begin{pmatrix} i \\ j \end{pmatrix}$$

2.3.4 Schedules

An execution trace of a sequential program could be described as a *total order* relation on a set of statement instances $\{(S,\mathbf{i}) : \mathbf{i} \in \mathcal{D}^S\}$. In order to impose a total execution order on a set of statement instances, each statement instance $(S,\mathbf{i})$ is assigned a *timestamp* denoted by a vector $\mathbf{t}$.

The $\mathbf{t}$ vector represent a multidimensional time. In order to impose a temporal ordering of time vectors, an ordering relation between two time vectors has to be defined. We use a lexicographic order between two vectors to impose their temporal order. Given two vectors $\mathbf{a} = (a_1, \ldots, a_n)^T$ and $\mathbf{b} = (b_1, \ldots, b_n)^T$, we define a lexicographic order relation $\mathbf{a} \ll \mathbf{b}$ as follows:

$$(a_1, \ldots, a_n)^T \ll (b_1, \ldots, b_n)^T \iff \exists i : 1 \leq i < n, a_i = b_i \wedge a_{i+1} < b_{i+1}$$

In other words, there exist an integer i such that all vector components up to i are equal, while the vector component at the $i+1$ position of the vector **b** is greater than respective component of the **a** vector. The idea of using multidimensional timestamp vectors was proposed by Feautrier [68] and later by Kelly and Pugh [93].

For each polyhedral statement S we define an affine *scheduling function* as a mapping between an iteration vector **i** and a timestamp vector **t**:

$$\mathbf{t} = \theta^S(\mathbf{i}) = \Theta^S \cdot (\mathbf{i}, \mathbf{g}, 1)^T$$

where $\Theta^S \in \mathbb{Z}^{p \times (n+n_g+1)}$ is a *scheduling matrix* having p rows, where p is a dimensionality of a timestamp vector **t**.

Once we have defined the scheduling function and the lexicographic ordering of the timestamp vectors, we can formally define an execution order of statement instances:

Definition 2.3.2 (Execution order)**.** A given statement instance $(S, \mathbf{i})$ is executed before a statement instance $(S', \mathbf{i}')$ if and only if:

$$\theta^S(\mathbf{i}) \ll \theta^{S'}(\mathbf{i}')$$

2.4 Related work and an historical overview

The polyhedral model is well-established theoretical foundation for reasoning about program semantics and transformations of static-control programs.

The seminal work of Karp, Miller and Winograd [88] on scheduling the computations of uniform recurrence equations can be considered as an origin of the polyhedral model theory. It introduced many concepts such as dependences, dependence graph, scheduling, iteration vectors.

Later work on automatic systolic array synthesis by Rajopadhye introduced the terms of *timing function*, *allocation function*, affine mappings and *polyhedral domains*. The PolyLib library for performing operations on rational polyhedra represented as a system of affine inequalities was conceived by Le Verge and Wilde [160, 108].

Different authors have used different notations to refer to the same concepts relating to polyhedral compilation. The most recent attempt to unify the notation and introduce the canonical form was that of Girbal [74]. When referring to *the polyhedral model* we are referring to the notation introduced in this chapter, which is mainly based on the notation coming from the work of Girbal [74].

A group at INRIA [74] has demonstrated that one can consider the polyhedral model as a fully-capable internal representation of the compiler. WRaP-IT [73] project has shown an internal representation based on the polyhedral model incorporated into production quality compiler Open64 [44].

As the last note, we remind that the polyhedral analysis is restricted to *affine programs* – programs that have static control and data access patterns described by the system of affine constraints. There are works that try to widen the scope of applicability of the polyhedral model to programs going beyond static-control [26, 18]. They mainly rely on the conservative approximations of the program semantics.

2.5 Summary

In this chapter we have given a self-contained mathematical background that forms a foundation of the polyhedral model. We have formally defined the affine functions, affine hyperplanes, polyhedra and polytopes. We have shown the place of the polyhedral model in the general compilation context. We have defined the class of the affine programs for which the polyhedral model representation could be obtained.

In Section 2.3 the basic building block of the model are defined : 1) iteration domains, 2) data access functions and 3) schedules.

Given the basic components of the polyhedral model, we define the way to express the multidimensional program transformations and to assess their legality by introducing the data dependence concepts in the next chapter.

Chapter 3

Program Transformations in the Polyhedral Model

3.1 Canonical Form of the Scheduling Functions

We have given the definition of the scheduling function in Section 2.3.4, defining it as as a function that expresses a mapping from statement iteration vectors to timestamp vectors. While the representation of the iteration domains and access functions (shown in Sections 2.3.2 and 2.3.3) follows straightforwardly from their definition, the question of representing the scheduling functions leaves many degrees of freedom.

One might restrict the scheduling functions to one-dimensional timestamps. But as Feautrier [68] has shown, one-dimensional timestamps cannot express the schedule of some class of the programs. Feautrier [68] has also shown that multidimensional schedules are expressive enough to encompass the full class of affine programs.

Within the class of multidimensional schedules, the following question arises: how many dimensions should a timestamp vector contain? Feautrier [68] provides a scheduling algorithm that minimizes the necessary number of dimensions of the timestamp vector. Bondhugula [34] provides a scheduling approach that requires at least *d* scheduling dimensions, *d* being the maximal loop depth of the statements within SCoP.

For the compiler construction purposes, it is desirable to have a canonical form of the scheduling functions. Girbal et al. [47, 23, 46] have proposed the *canonical form* of the scheduling functions that unifies the format of the scheduling matrices representing those functions.

We will discuss the motivation for providing the canonical form of the scheduling matrices, then we will define the form and discuss some of its properties.

3.1.1 Motivation

Given the scheduling function $\theta^S(\mathbf{i})$ represented as $\theta^S(\mathbf{i}) = \Theta^S \cdot (\mathbf{i}, \mathbf{g}, 1)^T$, the goal is to provide a canonical form of the scheduling matrix Θ^S.

Even if a matrix $\Theta^S \in \mathbb{Z}^{d\times(n+n_g+1)}$ has a full rank (requiring $d \geq n$), a direct mapping from an iteration vector $\mathbf{i}$ to a timestamp $\mathbf{t}$ is not enough to distinguish instances of two different statements.

Let us take an example in Figure 3.1. Given the two statements S_1 and S_2, if we simply take a full-rank identity matrix as a scheduling matrix for both statements, we would get:

$\theta^{S_1}((i_1,i_2)^T) = (i_1,i_2)^T$

$\theta^{S_2}((i'_1,i'_2)^T) = (i'_1,i'_2)^T$

```
    for (i1=0; i1<N; i1++) {
        for (i2 = 1; i2 < N; i2++) {
S1:             X[i1][i2] = X[i1][i2] - X[i1][i2-1] * A[i1][i2] / B[i1][i2-1];
S2:             B[i1][i2] = B[i1][i2] - A[i1][i2] * A[i1][i2] / B[i1][i2-1];
        }
    }
```

Figure 3.1 – An excerpt from ADI numerical kernel

There is no way to express that a given instance of statement $(S_1,(i_1,i_2)^T)$ happens before an instance $(S_2,(i'_1,i'_2)^T)$ for $(i_1,i_2)^T = (i'_1,i'_2)^T$, since both statement instances are mapped to the same timestamp vector.

An easy solution to this problem is to extend the timestamp vectors with constant entries that disambiguate different static statements in an original program. Those constant entries corresponds to relative execution order of statement instances inside a common loop nest. Now we have:

$\theta^{S_1}((i_1,i_2)^T) = (i_1,i_2,0)^T$

$\theta^{S_2}((i'_1,i'_2)^T) = (i'_1,i'_2,1)^T$

and it is always true that $(i_1,i_2,0)^T \ll (i'_1,i'_2,1)^T$ for $(i_1,i_2)^T = (i'_1,i'_2)^T$, since the lexicographic order is disambiguated at the last component of the timestamp.

Since the polyhedral model allows non-perfectly nested loops, a similar problem might occur when different statements, say S_1 and S_2 have different loop depths.

Aforementioned problems were limiting factors in early works on polyhedral program transformations [9, 103, 138, 165]. This was mainly due to the limitations of code generation algorithms that required the scheduling matrices to be unimodular [9] or at least invertible [103, 138, 165]. The recent developments [94, 77, 22, 155] in the polyhedral code generation have obviated those restrictions.

3.1.2 Canonical form of the scheduling matrix

Feautrier [68] and later Kelly and Pugh [93] have proposed a timestamp encoding for characterizing an execution order of statement instances within a non-perfectly nested loops.

Girbal et al. [47, 23, 46] have defined the *canonical form* of the scheduling matrix Θ^S. This encoding is generalized to handle arbitrary compositions of affine transformations. The canonical form encodes a mapping from iteration vector $\mathbf{i}$ to timestamp vector $\mathbf{t}$, as well as a static statement order within the common loop. The scheduling matrix Θ^S in the canonical form has the following format:

$$\Theta^S = \left[\begin{array}{c|c|c} 0 \cdots 0 & 0 \cdots 0 & \beta_0^S \\ A_{1,1}^S \cdots A_{1,n}^S & \Gamma_{1,1}^S \cdots \Gamma_{1,n_g}^S & \Gamma_{1,n_g+1}^S \\ 0 \cdots 0 & 0 \cdots 0 & \beta_1^S \\ A_{2,1}^S \cdots A_{2,n}^S & \Gamma_{2,1}^S \cdots \Gamma_{2,n_g}^S & \Gamma_{2,n_g+1}^S \\ \vdots \ddots \vdots & \vdots \ddots \vdots & \vdots \\ A_{n,1}^S \cdots A_{n,n}^S & \Gamma_{n,1}^S \cdots \Gamma_{n,n_g}^S & \Gamma_{n,n_g+1}^S \\ 0 \cdots 0 & 0 \cdots 0 & \beta_n^S \end{array}\right] \quad (3.1)$$

The scheduling matrix Θ^S is composed of three components:

– Component A is an invertible (full rank) matrix capturing the relative ordering of iteration vectors. Changing coefficients of this component corresponds to loop interchange, skewing and other *unimodular* transformations.

– Column β reschedules statements statically, at all nesting levels. It expresses code motion, loop fusion and fission.
– Component Γ captures loop shifting (pipelining) effects.

We call the A and Γ matrix components a *dynamic* schedule components – since those components are used to reschedule dynamic statement instances, while β vector component is called a *static* schedule component – since it corresponds to a static statement order inside a program.

The matrix $\Theta^S \in \mathbb{Z}^{(2n+1)\times(n+n_g+1)}$ has $2n+1$ rows, n being the loop depth of the statement S. Odd rows correspond to static execution order of statements that are enclosed in a common loop nest. Each such a row contains a single integer constant. All the odd rows could be summarized by β vector having $n+1$ components. We call this vector a *static scheduling* vector, since it represents the constant part of the timestamp.

Rows at even positions represent an affine mapping from iteration vectors **i** and global parameters **g** to their respective timestamp component. Affine mappings enable an *instance-wise* multidimensional affine scheduling [67, 68, 105, 34, 132]. Changing the coefficients of A and Γ matrix components enables the representation of arbitrary affine schedules which encompass classical *loop transformations* like loop interchange, skewing, shifting or unimodular transformations [5, 7, 163, 15, 54] and much more.

An example

The scheduling matrix representing the original program schedule has the A matrix initialized to the identity matrix, the Γ matrix is all zeros and the β vector encodes relative statement order inside a program source-code (or other program intermediate representation).

As an example, let us take a non-perfectly nested loop shown in Figure 2.1. The scheduling matrix that corresponds to an original execution order of statement S_1 is given as follows:

$$\Theta^{S_1} = \left[\begin{array}{cc|cc|c} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

this canonical scheduling matrix is decomposed into following components:

$$\mathrm{A}^{S_1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Gamma^{S_1} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \beta^{S_1} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The matrix Θ^{S_1} represents the scheduling function: $\theta^{S_1}((v,h)^T) = (0,v,0,h,0)^T$. In the same way, the scheduling matrix Θ^{S_2} is decomposed into the following components (the statement S_2 is nested within 4 nested loops):

$$\mathrm{A}^{S_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \Gamma^{S_2} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \beta^{S_2} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

that represent the scheduling function: $\theta^{S_2}(v,h,i,j)^T = (0,v,0,h,1,i,0,j,0)^T$. The scheduling function for statement S_3 is: $\theta^{S_3}((v,h)^T) = (0,v,0,h,2)^T$.

Note that the timestamp components at *odd* positions are integer constants, whereas the components at *even* positions correspond to affine mapping of loop induction variables to their respective timestamp components. The fifth components of all time vectors are integer constants $0,1,2$ – they correspond to textual order of statements S_1, S_2 and S_3 respectively.

Multidimensional time vectors are compared lexicographically. If we take statements S_1 and S_2 with their respective scheduling functions, we can say that:

$$\theta^{S_1}((v,h)^T) \ll \theta^{S_2}((v',h',i',j')^T)$$

i.f.f.

$$v < v' \vee (v = v' \wedge h < h') \vee (v = v' \wedge h = h' \wedge 0 < 1)$$

In other words, an instance of statement S_1 occurs before an instance of statement S_2 when one of the following conditions is satisfied: $v < v'$, $(v = v' \wedge h < h')$ or $(v = v' \wedge h = h' \wedge 0 < 1)$. The first two conditions are obvious: if a statement instance is scheduled at an earlier iteration of the outermost loop then it happens before. Likewise, if a statement instance is scheduled at the same outermost loop iteration, but at an earlier innermost loop iteration, then it happens before. The last condition states: if two statement instances are scheduled at the same iteration of their common outer loops, a statement instance $(S_1,(v,h)^T)$ is going to be executed earlier than a statement instance $(S_1,(v',h',i',j')^T)$, because it is statically scheduled to happen earlier at the fifth timestamp component ($0 < 1$).

3.2 Capturing semantical constraints

We have shown a powerful way to express the program transformations - by changing the schedule of the executed statement instances. Obviously, if we want to perform a *legal* program transformation, the semantics of the transformed program must match the semantics of the original program.

There are numerous ways to define the program semantics. But for the purpose of the polyhedral transformations, it is enough to define a semantical *equivalence* of the transformed program with respect to an original program. To determine this equivalence it is not necessary to define the exact semantics of each computational step.

The classical polyhedral model does not model the actual computations that are performed inside the polyhedral statements $S \in \mathcal{S}$. Polyhedral statement is treated as a *black box* that reads from and/or stores to a memory - thus changing the visible state. A given sequential program can be seen as a sequence of reads from and writes to a memory, irrelevant of the computations that are performed within the statements [7].

The transformation is legal if the transformed program computes the same values as the original program. This definition allows the reordering of the statement instances, as long as the transformed program would provide the equivalent results as the original one.

We use a well defined concept of *data dependence* as a semantical constraint that has to be preserved in order for a transformation to be defined as a *legal*. [27, 7].

A good survey of data dependence abstractions was given in [167, 61, 159, 14]. Naturally, the more precise the data dependence abstraction is, the more legal transformations are exploitable. The most precise data dependence abstraction is that of Feautrier [66] expressed as a *dependence polyhedra*. The similar approach was proposed by Pugh [133, 136, 137].

A practical and efficient implementation of instance-wise dependence analysis was provided by Vasilache [156]. Barthou [18] has extended the scope of the application of the polyhedral model to non-affine programs by providing the fuzzy array dataflow analysis - an approach that handles the non-affine program parts in a conservative way.

We will proceed by presenting the data dependence representation and analysis. Later we will introduce a refinement of dependence analysis that captures only the dataflow dependences.

source	target	dependence type	other names	symbol
write	read	true dependence	flow, RAW	δ
read	write	anti dependence	WAR	δ^{-1}
write	write	output dependence	WAW	δ^{o}
read	read	(input) no dependence	-	-

Figure 3.2 – Read-Write classification of the dependences

3.2.1 Data dependences

Two statement instances $(S_i, \mathbf{i}_{S_i})$ and $(S_j, \mathbf{i}_{S_j})$ are in a *data dependence* relation if they access the same memory location and at least one of those accesses is a *write* memory access. We designate a statement instance that happens earlier as the *source* instance, and a statement instance that happens later as the *target* instance. If a source instance is a write and a target instance is a read then the data dependence is a *true dependence*. If a source instance is a read and a target instance is a write then the dependence is a *antidependence*. Lastly, if a source instance is a write and a target instance is a write then the data dependence is an *output dependence*. All the possible types of dependences are summarized in Table 3.2.

It is not practical to compute data dependences between all statement instances in a program. What is more, if a loop iteration count depends on a parameter that is unknown at the compilation time, it is not even possible to enumerate all the data dependences in a program.

A compact way to summarize all the data dependences between all statement instances in a program is to use a structure called the *Reduced Data Dependence Graph* (RDDG) [57].

A RDDG is a directed multi-graph, $G = (V, E)$, whose vertex set V is a set of all program statements, i.e., $V = \mathcal{S}$. There is an edge $e \in E$ from a vertex S_i to a vertex S_j if there is a data dependence relation between an access in some instance of S_i and an access in some instance of S_j. Each edge $e \in E$ is labelled by a *dependence polyhedron* $\mathcal{P}_e$.

The dependence polyhedron $\mathcal{P}_e$ that labels an edge from S_i to S_j represents the set of iteration vector pairs denoting statement instances that are in data dependence relation:

$$\mathcal{P}_e = \{(\mathbf{i}_{S_i}, \mathbf{i}_{S_j}) \mid (S_i, \mathbf{i}_{S_i}) \text{ and } (S_j, \mathbf{i}_{S_j}) \text{ are data dependent } \}$$

In order to compute such an *instancewise* dependence polyhedron, one has to construct an intersection of the following affine constraints [156]:

Conflict condition both statement instances access the same memory location. This is equivalent to stating that their subscript functions have all their subscripts equal: $f^{S_i}(\mathbf{i}_{S_i}) = f^{S_j}(\mathbf{i}_{S_j})$

Causality condition the instance $(S_i, \mathbf{i}_{S_i})$ happens before the instance $(S_j, \mathbf{i}_{S_j})$ in the original program execution order: $\theta^{S_i}(\mathbf{i}_{S_i}) \ll \theta^{S_j}(\mathbf{i}_{S_j})$

Execution condition both statement instances are actually executed, i.e., both instances belong to statement iteration domains: $\mathbf{i}_{S_i} \in \mathcal{D}^{S_i} \wedge \mathbf{i}_{S_j} \in \mathcal{D}^{S_j}$

Since all the conditions could be expressed as a system of affine equalities or inequalities, the resulting affine constraint system represents an affine relation that can be easily manipulated by an integer linear programming and linear algebra tools.

The causality condition is a lexicographical comparison of vectors and could be represented as a disjunction of $N_{S_iS_j}$ constraint components, where $N_{S_iS_j}$ is a common loop depth of statements S_i and S_j.

An example

As an example, let us take a matrix-vector multiplication kernel shown in Figure 3.3. There is a true data dependence between an instance of the statement S_1 and an instance of the statement S_2, denoted

```
    for (i = 0; i < N; i++) {
S1: b[i] = 0;
      for (j = 0; j < N; j++)
S2:     b[i] = b[i] + A[i][j] * x[j];
    }
```

Figure 3.3 – Matvect - matrix vector multiplication

name	condition	affine constraints
execution condition	$(i)^T \in \mathcal{D}^{S_1}$	$0 \leq i < N$
execution condition	$(i', j')^T \in \mathcal{D}^{S_2}$	$0 \leq i' < N \wedge 0 \leq j' < N$
conflict condition	$f^{S_1}((i)^T) = f^{S_2}((i', j')^T)$	$i = i'$
causality condition	$\theta^{S_1}((i)^T) \ll \theta^{S_2}((i', j')^T)$	$i \leq i'$

Table 3.1 – A summary of affine conditions for a dependence edge $S_1 \to S_2$

$(S_1, (i)^T)$ and $(S_2, (i', j')^T)$ respectively.

As the source of the dependence we consider the statement S_1 containing a write to an array `b[i]` whose corresponding subscript function is $f^{S_1}(i)^T = (i)$. As the target of the dependence we consider the statement S_2 containing a read from an array `b[i]` whose corresponding subscript function is $f^{S_2}(i, j)^T = (i)$. We summarize the necessary affine constraints in Table 3.1. Taking an intersection of those affine constraints, we end up with the following dependence polyhedron [1] :

$$\mathcal{P}_e : \left[\begin{array}{ccc|c|c} 1 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & -1 \end{array}\right] \begin{pmatrix} i \\ i' \\ j' \\ N \\ 1 \end{pmatrix} \geq \mathbf{0} \quad \left| \begin{array}{l} i \geq i' \\ i \leq i' \\ i' \geq 0 \\ i' \leq N-1 \\ j' \geq 0 \\ j' \leq N-1 \end{array} \right.$$

A dependence polyhedron $\mathcal{P}_e$ is a relation that describes all pairs of iteration vectors (i) and (i', j') of statement instances that are in a dependence relation. The pairs of iteration vectors are concatenated in a single vector (i, i', j'), so the relation is a subset of Cartesian product space:

$$\mathcal{P}_e = \{(i, i', j') \in \mathbb{Z}^{dim(S_1)+dim(S_2)} | (S_2, (i', j')) \text{ is dependent on } (S_1, (i))\}$$

thus a dependence polyhedron could be concisely represented as:

$\mathcal{P}_{e_1} = \{(i, i', j') | i = i' \wedge 0 \leq i' \leq N-1 \wedge 0 \leq j' \leq N-1\}$

In our example, the pair is decomposed in the first component $(i)^T$ that corresponds to the source iteration, while the second component $(i', j')^T$ corresponds to the target iteration. In other words, if the value was stored in `b[i]` in the i-th iteration of the statement S_1, it is read by all the iterations of the statement S_2 for which $i' = i$. There is no constraint on the j', since for $i = i'$, the statement instance S_2 is always executed after an instance of the statement S_1.

A true data dependence between S_1 and S_2 is not the only data dependence that exists in a program given in Figure 3.3. For example, there exists a write-after-write (output) dependence between statements S_1 and S_2, described by the same polyhedron as a true dependence. The reason for the output data dependence is the fact that the write to an array `b[i]` at S_1 is subsequently overwritten by a write to an array

1. An equality $i = i'$ is expressed as two inequalities $i \geq i' \wedge i \leq i'$

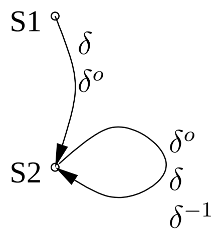


Figure 3.4 – A RDDG for the matvect kernel

source	sink	type
S_1	S_2	true
S_1	S_2	output
S_2	S_2	true
S_2	S_2	output
S_2	S_2	antidependence

Table 3.2 – A summary of dependences in matvect kernel

at S_2. Figure 3.4 shows a full dependence graph for `matvect` kernel in Figure 3.3. The dependences are summarized in Table 3.2.

3.2.2 Memory based dependences

It is a well known fact [102] that anti and output data dependences can always be removed from the dependence graph by a technique called an *array expansion* [110, 37]. The anti and output dependences are called *memory based dependences* [137] since they are induced by the fact that the same memory cell is reused for storing values. One can remove those dependences by attributing a separate memory cell to each statement instance, so that the same memory cell will not be reused - an array expansion.

On the contrary, the true dependences have to be preserved, since they capture the relation between the producer and a consumer of the data. The subset of the true dependences, named *dataflow* dependences has to be always preserved - those dependences cannot be removed from the dependence graph.

In the next subsection we will discuss the technique for computing the set of dataflow dependences - an array dataflow analysis. The whole Chapter 5 will be devoted to a technique for efficiently handling memory based dependences.

3.2.3 Array data-flow analysis

Standard data dependence relation gives an information on whether there exists an *aliasing* of memory locations. If a statement instance $(S_i, \mathbf{i}_{S_i})$ writes to a memory location $\mathrm{A}[f^{S_i}(\mathbf{i}_{S_i})]$, and a statement instance $(S_j, \mathbf{i}_{S_j})$ reads from the same memory location, they are considered to be in a true data dependence relation, even if there was an intervening write coming from some other statement instance $(S_k, \mathbf{i}_{S_k})$.

This notion captures the fact of the potential memory conflict in the case the statement instances are rescheduled, but it does not provide an information on the flow of values: a read in the statement instance $(S_j, \mathbf{i}_{S_j})$ will only see a value that was stored in statement instance that was scheduled as the latest before statement instance $(S_j, \mathbf{i}_{S_j})$.

In order to capture the flow of values for each array element, an instancewise array data-flow analysis based on the polyhedral model was provided by Feautrier [66] and Pugh [137]. It has also been studied by Maydan and Lam [110].

The purpose of array data-flow analysis is to give an exact characterization of the source statement instance that defines *value* stored in a given array element that is seen by the given read statement instance. array data-flow analysis is also known under the name of value-based array data dependence analysis [137], since it captures the data dependences induced by the flow of values, and not merely by the *aliasing* of memory locations.

We will define a true data-flow dependence between two statement instances as a true data dependence satisfying an additional constraint stating that an element written in a source statement instance must not be overwritten by some other statement that is executed between source and target statement instances. The additional constraint is called a *liveness condition*. If we put together all the affine constraints together, we will get the following set of affine constraints:

Conflict condition both statement instances access the same memory location. This is equivalent to stating that they access the same memory array A and their subscript functions have all their subscripts equal: $f^{S_i}(\mathbf{i}_{S_i}) = f^{S_j}(\mathbf{i}_{S_j})$

Causality condition the instance $(S_i, \mathbf{i}_{S_i})$ happens before the instance $(S_j, \mathbf{i}_{S_j})$ in the original program execution order: $\theta^{S_i}(\mathbf{i}_{S_i}) \ll \theta^{S_j}(\mathbf{i}_{S_j})$

Execution condition both statement instances are actually executed, i.e., both instances belong to statement iteration domains: $\mathbf{i}_{S_i} \in \mathcal{D}^{S_i} \wedge \mathbf{i}_{S_j} \in \mathcal{D}^{S_j}$

Liveness condition there is no overwriting statement instance that happens between source and target statement:

$$\forall S_k \in \mathcal{S}_{kill} : \neg\exists \mathbf{i}_{S_k} : \theta^{S_i}(\mathbf{i}_{S_i}) \ll \theta^{S_k}(\mathbf{i}_{S_k}) \ll \theta^{S_j}(\mathbf{i}_{S_j}) \wedge f^{S_i}(\mathbf{i}_{S_i}) = f^{S_k}(\mathbf{i}_{S_k})$$

$\mathcal{S}_{kill}$ is the set of statements containing array references that might write to the array A. Note that both S_i and S_j are included in $\mathcal{S}_{kill}$.

While this definition is a denotational description of how to compute value-based (data-flow) dependences, it does not express those constraints directly as a set of affine constraints. Indeed, an existential quantifier and a negation operator might induce a solution that contains a set of non-convex polyhedra. Pugh [137] shows different heuristics that control the complexity of the solution.

An ILP based formulation

Featurier [66] has shown that the previously mentioned liveness condition might be equivalently expressed as the following problem:

Given a read statement instance $(S_j, \mathbf{i}_{S_j})$ containing a read data reference $\mathrm{A}[f^{S_j}(\mathbf{i}_{S_j})]$, compute an *unique* statement S_i and its iteration vector $\mathbf{i}_{S_i}$ that form a *source* statement instance $(S_i, \mathbf{i}_{S_i})$ being the *source of the value* read in $\mathrm{A}[f^{S_j}(\mathbf{i}_{S_j})]$.

This provides a notion of *source function* $H : \langle \mathrm{A}[f^{S_j}(\mathbf{i}_{S_j})], (S_j, \mathbf{i}_{S_j})\rangle \rightarrow (S_i, \mathbf{i}_{S_i})$: an argument is a data reference $\mathrm{A}[f^{S_j}(\mathbf{i}_{S_j})]$ and its associated statement instance $(S_j, \mathbf{i}_{S_j})$ while the result of the function is *source* statement instance $(S_i, \mathbf{i}_{S_i})$ that produced a value in $\mathrm{A}[f^{S_j}(\mathbf{i}_{S_j})]$.

Among the set of possible *source* statement instances only the one that is executed as *the latest* is the true source of the given value, since all values written by other sources are overwritten by the latest executed instance.

In order to compute the unique value of the source function for each possible data reference, Feautrier [66] proposes to use a PIP [65] (Parametric Integer Programming) algorithm to compute the lexicographically largest element of the parametric set Q.

The parametric set $Q\langle \mathrm{A}[f^{S_j}(\mathbf{i}_{S_j})], (S_j, \mathbf{i}_{S_j})\rangle = \{(S_k, \mathbf{i}_{S_k})\}$, contains the set of all executed statement instances $(S_k, \mathbf{i}_{S_k})$ that write to the given array element $\mathrm{A}[f^{S_j}(\mathbf{i}_{S_j})]$ and that happen before a statement

```
(S_1,(i)^T)          b[i] = 0;
(S_2,(i,0)^T)        b[i] = b[i] + A[i][0] * x[0];
(S_2,(i,1)^T)        b[i] = b[i] + A[i][1] * x[1];
...                  ...
(S_2,(i,N-1)^T)      b[i] = b[i] + A[i][N-1] * x[N-1];
```

Figure 3.5 – A slice of the execution trace of matvect kernel

instance $(S_j, \mathbf{i}_{S_j})$ (instances that satisfy conflict, execution and causality condition). An unique solution is obtained by solving the following parametric integer programming problem:

$$max_{\ll}\theta(S_k, \mathbf{i}_{S_k}), (S_k, \mathbf{i}_{S_k}) \in Q\langle \mathrm{A}[f^{S_j}(\mathbf{i}_{S_j})], (S_j, \mathbf{i}_{S_j})\rangle$$

In our approach, we will use an implementation of the array dataflow analysis as proposed by Feautrier and implemented in an ISL [158] library. An extension of array data-flow analysis to non-affine programs proposed by Barthou [18] follows the same principle, but it introduces an additional predicate variables that model the non-affine access functions and non-affine control flow in the program.

An Example

As an illustration, let us consider a kernel in Figure 3.3. Figure 3.5 illustrates a slice of the sequential execution trace for some fixed outer loop induction variable i. For a given outer loop induction variable i one instance of statement S_1 is executed: $(S_1,(i)^T)$. Subsequently $N-1$ instances of S_2 are executed: first an instance $(S_2,(i,0)^T)$ is executed, next an instance $(S_2,(i,1)^T)$ and so on.

By definition, the *alias* based dependence analysis considers all the pairs of dependences between statements S_1 and S_2: $(S_1,(i)^T) \rightarrow (S_2,(i,0)^T)$, $(S_1,(i)^T) \rightarrow (S_2,(i,1)^T)$, ..., $(S_1,(i)^T) \rightarrow (S_2,(i,N-1)^T)$. Those dependences are induced by a write `b[i] = 0` in statement S_1 and subsequent reads `... = b[i]` in statement instances of statement S_2.

The array data-flow analysis will only consider the true value-based dependence $(S_1,(i)^T) \rightarrow (S_2,(i,0)^T)$. There is no value-based dependence $(S_1,(i)^T) \rightarrow (S_2,(i,1)^T)$, since the value written in statement instance $(S_1,(i)^T)$ is overwritten in statement instance $(S_2,(i,0)^T)$ and thus not visible in instances $(S_2,(i,1)^T),\ldots,(S_2,(i,N-1)^T)$.

3.3 Transformations and legality

A program transformation in the polyhedral model is expressed as an reordering transformation on statement instances. One expresses a transformation as a set of scheduling matrices. For each statement $S_i \in \mathcal{S}$ one provides a scheduling matrix θ^{S_i} that completely describes a multidimensional schedule of statement instances.

Definition 3.3.1 (Dependence satisfaction)**.** A RDDG edge $e \in E$ from S_i to S_j, labelled by dependence polyhedron $\mathcal{P}_e$, is *satisfied* by schedules θ^{S_i} and θ^{S_i} iff:

$$\forall (\mathbf{i}_{S_i}, \mathbf{i}_{S_j})^T \in \mathcal{P}_e : \theta^{S_i}(\mathbf{i}_{S_i}) \ll \theta^{S_j}(\mathbf{i}_{S_j}) \tag{3.2}$$

A dependence satisfaction condition for a dependence edge $e \in E$ simply states that for each pair of dependent statement instances, the source instance $(S_i, \mathbf{i}_{S_i})$ is scheduled before the target instance $(S_j, \mathbf{i}_{S_j})$. This is expressed as a lexicographical comparison of their timestamps which are transformed through their respective scheduling functions $\theta^{S_i}(\mathbf{i}_{S_i})$ and $\theta^{S_j}(\mathbf{i}_{S_j})$.

A program transformation is expressed as the set of scheduling matrices:

$$T = \{\theta^{S_1}, \ldots, \theta^{S_p}\}$$

where p is the number of statements inside the program statement set $\mathcal{S}$. We say that a transformation T is *legal* for a given RDDG G=(V, E) if all the edges $e \in E$ are satisfied by their respective statement schedules.

A priori versus a posteriori legality

Depending on the usage scenario, there are two major approaches to assuring the legality of transformations in the polyhedral model:

a posteriori checking the legality of the *given* transformation. A transformation might be given by an user or obtained semi-automatically [41, 74]. A violation analysis [156] is performed to check the legality of the proposed transformation.

a priori computing the automatic transformations that are guaranteed to be legal. All the affine scheduling approaches fall into this category [67, 68, 76, 34, 107, 106].

A computational procedure for a posteriori legality check is based on a legality condition given in Equation 3.2. But in order to express this condition in a way that could be handled by the polyhedral techniques, it has to be rephrased into an equivalent condition:

$$\exists (\mathbf{i}_{S_i}, \mathbf{i}_{S_j})^T \in \mathcal{P}_e : \theta^{S_j}(\mathbf{i}_{S_j}) \ll \theta^{S_i}(\mathbf{i}_{S_i})$$

This set represents those pairs of statement instances that are in dependence relation, but whose order is inversed (please note that lexicographical comparison is an inversion of the one shown in Equation 3.2). If this set is empty, there are no illegal inversed statement instances, and the given dependence relation is preserver. The same check is performed for each dependence edge $e \in E$.

An emptiness check is done by performing a Fourier-Motzkin elimination [143] restricted to integer solutions only [133]. An efficient implementation of the violation check was proposed by Vasilache [156].

In the case there is some pair of dependent statement instances that is inverted it is not a dead-end: one might resort to *correct* the schedule by applying a corrective shifting. This technique was proposed by Vasilache [157].

An another approach is to automatically build the legal schedules a priori. This is the approach taken by all the affine scheduling algorithms [67, 68, 76, 34, 107, 106].

One can build the *space of legal transformations* by putting together the dependence satisfaction constraints for all the dependence edges $e \in E$ in RDDG. As a solution, one gets the space of legal scheduling matrices, expressed as the set of possible matrix coefficients that give the legal transformation.

Building and solving such a constraint system is based on affine form of Farkas lemma [143] (to be defined in Chapter 7) and Fourier-Motzkin elimination [67]. The related problem that arises is that of selecting the best schedule among the set of possible legal schedules. That problem is the core of the Part III of this dissertation.

An Example

As an example, consider a matvect kernel in Figure 3.3. The original scheduling matrices Θ^{S_1} and Θ^{S_2} are shown in Figure 3.6. By definition, the original scheduling is always legal – indeed a RDDG is computed from the original scheduling matrices.

$$\Theta^{S_1} = \left[\begin{array}{c|c|c} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right] \begin{pmatrix} i \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ i \\ 0 \end{pmatrix} \qquad \Theta^{S_2} = \left[\begin{array}{cc|c|c} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ i \\ 1 \\ j \\ 0 \end{pmatrix}$$

Figure 3.6 – Original scheduling matrices

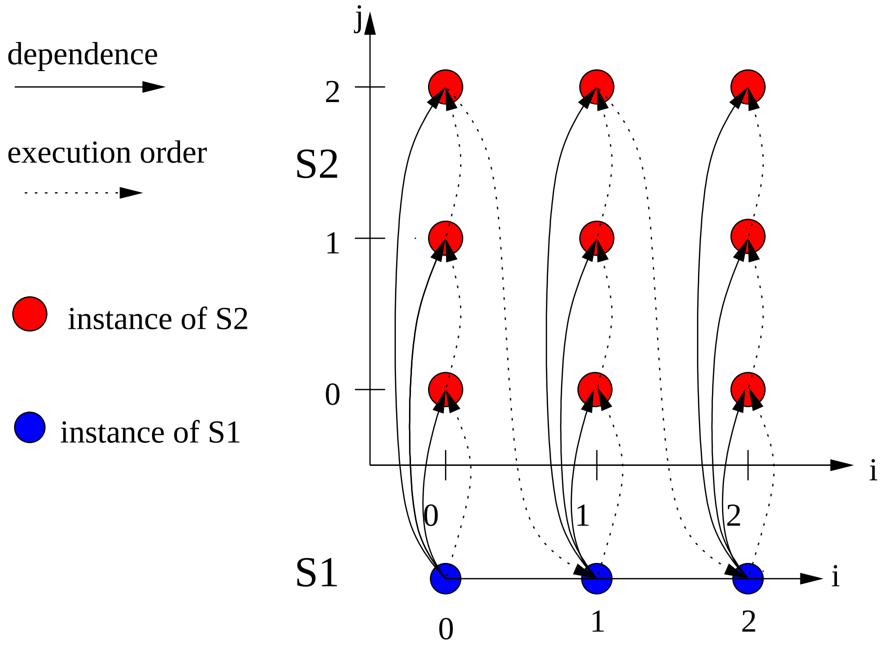


Figure 3.7 – Matvect: an original sequential execution order

Let us consider a true data dependence edge from S_1 to S_2, whose dependence instances are summarized by the following dependence polyhedron:

$\mathcal{P}_e = \{(i,i',j')^T \mid 0 \leq i < N \wedge 0 \leq i' < N \wedge 0 \leq j' < N \wedge i = i'\}$.

An illustration of the original execution order and data dependence instances for a dependence edge $S_1 \rightarrow S_2$ is shown in Figure 3.7.

As an instance of a possible reordering transformation, let us change the scheduling matrix of statement S_2 by interchanging the original scheduling matrix columns. A program transformation is represented by a set of new scheduling matrices: $T = \{\Theta'^{S_1}, \Theta'^{S_2}\}$. The scheduling matrix of statement S_1 is left intact. The scheduling is illustrated in Figure 3.8.

The transformed schedule does not satisfy the dependence edge e from S_1 to S_2 summarized by the polyhedron $\mathcal{P}_e$. This is illustrated in Figure 3.11 showing a new (rescheduled) execution order, together with an instance of dependence pair that is not satisfied. Indeed, a true data dependence from S_1 to S_2 requires that an instance $(S_1,(2)^T)$ is executed before an instance $(S_2,(2,0)^T)$, but in a transformed schedule an instance $(S_2,(2,0)^T)$ is executed first – a data dependence is *inversed*.

On the other hand, Figure 3.9 shows a scheduling transformation that is legal. As in the previous example, the scheduling matrix of the statement S_1 is left intact. The scheduling matrix of the statement S_2 is formed by putting an integer constant in the first row and leaving the rest of the rows intact. In this way, we have changed the β (see Section 3.1) component of the scheduling matrix.

$$\Theta'^{S_1} = \left[\begin{array}{c|c|c} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right] \begin{pmatrix} i \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ i \\ 0 \end{pmatrix} \qquad \Theta'^{S_2} = \left[\begin{array}{cc|c|c} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ j \\ 1 \\ i \\ 0 \end{pmatrix}$$

Figure 3.8 – Illegal transformation matrices

$$\Theta'^{S_1} = \left[\begin{array}{c|c|c} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right] \begin{pmatrix} i \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ i \\ 0 \end{pmatrix} \qquad \Theta'^{S_2} = \left[\begin{array}{cc|c|c} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ i \\ 0 \\ j \\ 0 \end{pmatrix}$$

Figure 3.9 – Legal transformation matrices

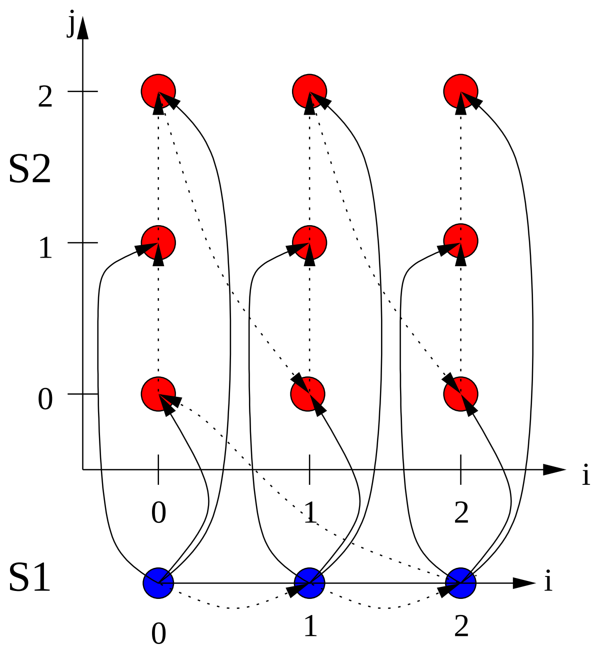


Figure 3.10 – A transformed and legal sequential execution order

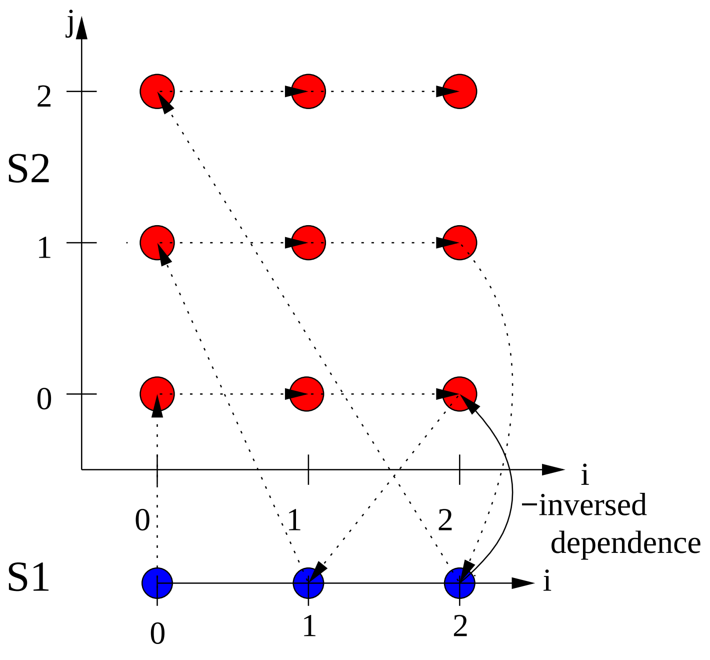


Figure 3.11 – A transformed and illegal sequential execution order

Figure 3.10 illustrates the new execution order – first, all statement instances of the statement S_1 are executed, and then all instances of S_2 are executed. This execution order preserves all dependence pairs for a true data dependence from S_1 to S_2.

Please note that in order to prove *legality* of a transformation, one has to prove that all dependence pairs are satisfied. On the other hand, in order to prove that a transformation is *illegal* it is enough to demonstrate that there exists at least one dependence pair that is *inversed*.

3.4 Related work

A loop transformation theory based on data dependences was developed by researchers working on optimizing compilers for high performance computing [13, 98, 6]. The data dependence graph was used as a central concept for ensuring the legality of transformations. The loop transformations were performed on a *syntax* representation.

The first attempts to model the imperative program loop transformations through linear algebraic representation were those of *unimodular transformations* [99, 14, 161]. The loop transformations ware represented by unimodular matrices. A sequence of several loop transformations is represented in a single unimodular matrix and transformed source code is generated automatically. The drawback of unimodular transformations is the fact that they are amenable to perfectly nested loops only.

Some works were trying to circumvent the limitation of unimodular transformations so that they could be applied to non-perfectly nested loops [103, 166]. But they have limitations in code generation phase - requiring the transformation matrices to be invertible.

A unifying theory that could handle non-perfectly nested loops was developed [134, 67, 68, 105]. It relies on multidimensional schedules of statement instances. They also show how to automatically search for a schedule that optimizes program parallelization or data locality.

Feautrier [64] has demonstrated the feasibility of handling static control program analysis by using integer linear programming, and he proposed a Parametrized Integer Programming (PIP) algorithm. Later, Feautrier has shown how to handle the problem of instancewise array dataflow analysis [66] and minimal latency scheduling problem [67] – two seminal works in the history of theoretical polyhedral compilation.

Another technique developed by Pugh [133] is based on an extension of Fourier-Motzkin variable elimination which led to the development of library for manipulating integer sets and relations represented as systems of affine constraints called Omega [90]. Pugh has also solved a problem of array dataflow analysis under the name of *value based array dependence analysis* [137].

More recent advances in the scalable and general code generation techniques [21, 22, 155] enabled the polyhedral model to be integrated into the real production compiler.

3.5 Summary

In this chapter we have summarized the state of the art in program transformations expressed in the polyhedral model. The concepts of data dependences and array dataflow analysis were introduced. The notion of transformation *legality* was briefly discussed. Some implementational issues of array dataflow analysis and transformation legality check were presented.

We will extend the concepts that were presented in this chapter with our new contributions. In Chapter 5 we will show that we can relax the conditions for determining the legality of the transformation. In Chapter 6 we will show that we can construct a precise cost models based on the polyhedral representation, provided that we do not treat the polyhedral statement as a black box. Chapter 7 provides our contribution to the schedule construction problem.

Part II

The framework

Chapter 4

Graphite framework

Despite several decades of research into the polyhedral model, there is still no general-purpose production compiler using the polyhedral model internally. The situation is changing with the demonstration of the scalability of polyhedral algorithms and with the widespread dissemination of multicore processors and hardware accelerators.

This chapter describes GRAPHITE framework that incorporates the polyhedral analyses and transformations into GCC [1] compiler. GCC compiler is one of the most widely used open-source compilers for a variety of imperative programming languages (C, C++, ADA, Fortran) and platforms (x86, ARM, PowerPC, MIPS).

The original motivation for GRAPHITE project is the study of loop optimizations on a low-level three-address code [3] intermediate representation. Polyhedral representation is extracted from the GCC three-address code intermediate representation that is in the *static single assignment* (SSA) [52] form.

This is a major difference with traditional source-to-source polyhedral compilers that operate on high-level abstract syntax level. Operating directly on the three-address code brings in new challenges but also new opportunities: we can leverage existing scalar analyses in the compiler and interact directly with a wealth of optimizations for extracting coarse and fine-grained parallelism and improving memory locality.

We will first discuss the related work on different approaches to polyhedral compilation. A general overview of the compilation flow in GRAPHITE will be presented. Later, a discussion of the relevant and interesting design issues follows, together with the conclusion.

4.1 Related Work

There have been many efforts in designing an advanced loop-nest transformation infrastructure. Most loop restructuring compilers introduced syntax-based models and intermediate representations. ParaScope [50] and Polaris [31] are dependence-based, source-to-source parallelizers for Fortran. KAP [87] is closely related to these academic tools.

SUIF [81] is a platform for implementing advanced compiler prototypes. PIPS [85] is one of the most complete loop restructuring compilers, implementing polyhedral analyses and transformations (including affine scheduling) and interprocedural analyses (array regions, alias). Both of them use a syntax tree extended with polyhedral annotations, but not a unified polyhedral representation.

The MARS compiler [119] unifies classical dependence-based loop transformations with data storage optimizations. However, the MARS intermediate representation only captures part of the loop in-

1. https:/gcc.gnu.org

formation (domains and access functions): it lacks the characterization of iteration orderings through multidimensional affine schedules.

The first thorough application of the polyhedral representation was the Petit tool [89], based on the Omega library [94]. It provides space-time mappings for iteration reordering, and it shares our emphasis on per-statement transformations, but it is intended as a research tool for small kernels only. We also use a code generation technique that is significantly more robust than the code generation in Omega [22].

Semi-automatic polyhedral frameworks have been designed as building blocks for compiler construction or (auto-tuned) library generation systems [89, 46, 157, 41, 147]. They do not define automatic methods or integrate a model-based heuristic to construct profitable optimization strategies.

The GRAPHITE project was first announced by Pop et al. in 2006, [125]. The design of GRAPHITE is largely borrowed from the WRaP-IT polyhedral interface to Open64 and its URUK loop nest optimizer [74]. The CHiLL project from Chen et al. revisited the URUK approach, focusing on source-to-source transformation scripting [41, 147].

Unlike URUK and CHiLL, GRAPHITE aims at complete automation, resorting to iterative search and cost modeling of the profitability of program transformations. In addition, unexpected design and algorithmic issues have been discovered, partly due to the design of GCC itself, but mostly due to the integration of the polyhedral representation in a three-address code in SSA form.

To the best of our knowledge, two proprietary polyhedral compilers based on a low-level internal representation are currently in development: the R-Stream compiler from Reservoir Labs [113], and IBM's polyhedral extension of its XL compiler suite [140]. Little work has been published on compilation issues of three-address code based polyhedral model abstraction.

4.2 An overview of the compilation flow

Traditionally the polyhedral model is used in the source-to-source [123, 124] translators and may be viewed in terms of the three step process: (1) extraction of the polyhedral representation of the static control program part, (2) transformation on the polyhedral abstraction and (3) generation of the equivalent source code fragment from the abstraction.

On the other hand, GRAPHITE is the polyhedral transformation engine that is only a small part of the complete compiler toolchain – it is implemented as an optimization pass of GCC compiler. As such, GRAPHITE operates on a three-address code internal representation. A place of GRAPHITE inside GCC compiler is illustrated in Figure 4.1.

GCC compiler is decoupled into three principal components: front-end, middle-end and back-end, as shown in Figure 4.1. Front-end is language dependent. It is responsible for parsing the source language [2] into GENERIC [70] high-level abstract syntax tree. GENERIC representation is lowered into GIMPLE [70] intermediate three-address code code used in compiler middle-end.

Middle-end is language independent optimization stage. Inside this stage multiple optimization passes are executed [3]. Intraprocedural analyses and optimizations are performed. GIMPLE three-address code is transformed into SSA form. Various optimizations based on SSA form are performed, including DCE (Dead Code Elimination), forward propagation and copy propagation. Loop based optimizations such as loop invariant motion, loop unswitching (moving conditional statements out of the loop), PRE(Partial Redundancy Elimination) are also performed.

GRAPHITE is scheduled as a loop optimization pass operating on a transformed GIMPLE code in SSA form. GRAPHITE itself is composed of several steps that will be explained in the following section.

2. we will be principally interested in optimizing C, C++ and Fortran codes
3. more than 200 optimization passes in the current GCCimplementation

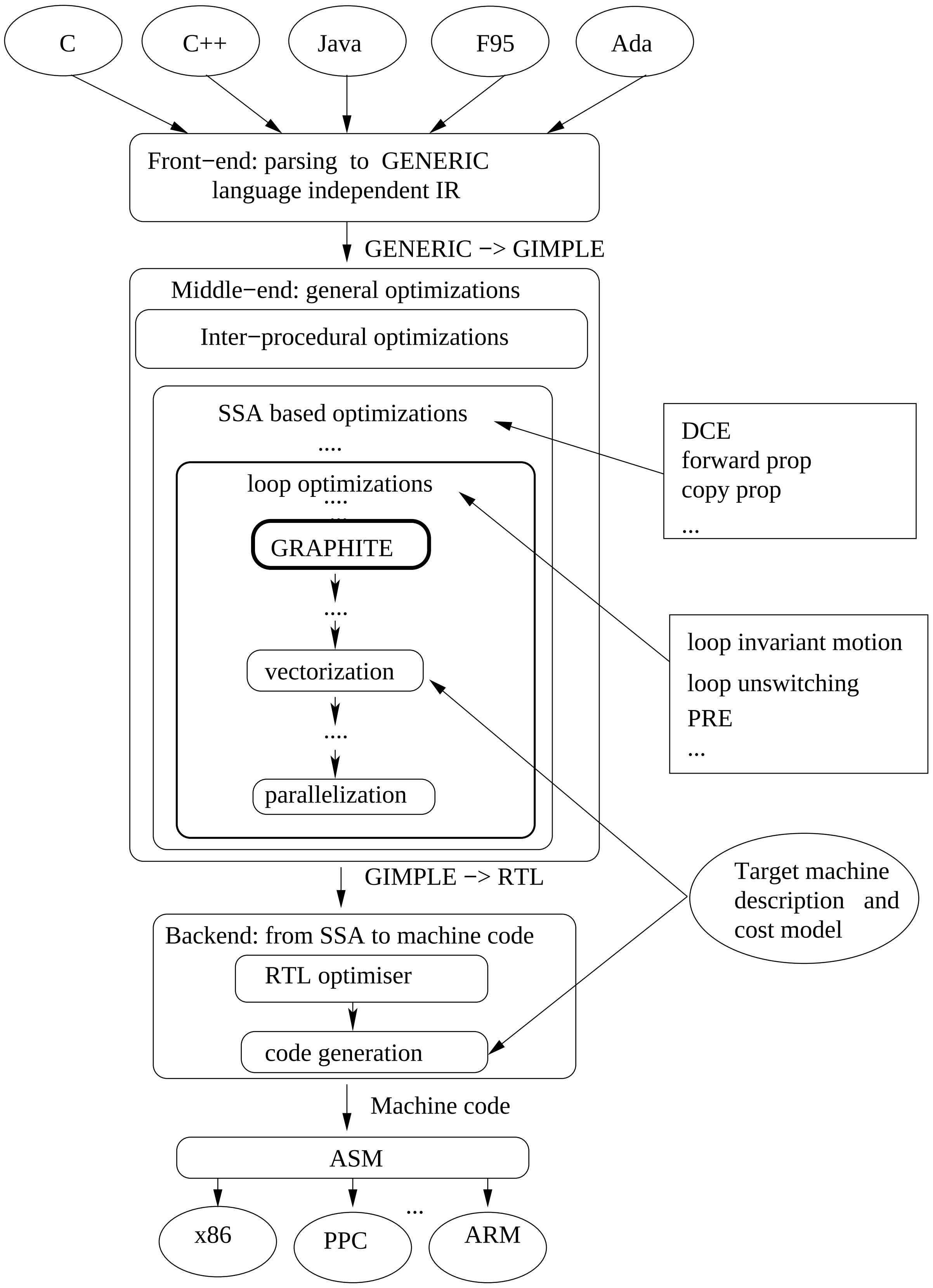


Figure 4.1 – A gcc compilation flow

GRAPHITE performs complex loop transformations and produces a regenerated GIMPLE code that is further transformed by automatic vectorization and automatic parallelization passes.

After middle-end optimization passes, the transformed GIMPLE code is in turn lowered into RTL (Register Transfer Language) [70] which is used in the GCC back-end. Back-end is target machine dependent and can produce machine code for different architectures (x86, x86_64, PowerPC, MIPS, ARM).

Further in this work we will not consider neither front-end neither back-end, since we are operating on a language and architecture neutral middle-end stage. The only architecture dependent part of the GRAPHITE loop transformation search algorithm is the dependence on the target architecture cost model as shown in Figure 4.1.

4.2.1 GIMPLE internal representation

GRAPHITE is operating on a three-address code internal representation in SSA (Single Static Assignment) form. An internal representation called GIMPLE is composed of basic blocks containing three-address code statements. Each statement has no more than 3 operands. Necessary temporaries are introduced to store intermediate values needed to compute complex expressions. Each statement could contain at most one data access to a memory location, either read or write. All the high level control structures are expressed as conditional jumps and the lexical scopes are removed [70].

GCC keeps track of control flow between basic blocks in CFG (Control Flow Graph) graph, whose vertices are the basic blocks and whose directed edges show the control-flow. Loops are strongly connected components of the CFG.

SSA [52] form ensures uniqueness of a variable definition: each variable is assigned only once. When transforming into SSA form, each original variable is split into *versions* if there are multiple definitions of the same variable. So, for example, if there are two distinct assignments to the variable x, it would be split into two variables x_1 and x_2.

GIMPLE in SSA form contains ϕ-functions [52] of the form $\phi(x_1, x_2, \ldots, x_n)$, where $x_1, x_2, \ldots, x_n$ are variable versions and the number of arguments of the ϕ-function is the number of distinct control flow paths that reach the ϕ-function. ϕ-functions are inserted at the places where multiple distinct variable versions $x_1, x_2, \ldots, x_n$ might reach the use, depending on the control flow.

An excerpt of GIMPLE internal representation and the corresponding source code is shown in Figure 4.7.

4.3 Inside GRAPHITE

GRAPHITEis the polyhedral compilation framework that operates on GIMPLE code in SSA form. As such, it has to first extract the polyhedral representation out of three-address code. Also, since it is a GCC pass, it has to produce GIMPLE three-address code.

This two tasks are more complicated when operating on three-address code than when operating on high-level abstract syntax trees. In this section we will discuss an approach of mapping three-address GIMPLE code into the polyhedral model, and mapping the polyhedral model back into the three-address GIMPLE code. Actual polyhedral transformations are the topic of subsequent chapters. GRAPHITE is itself composed of several sequential stages. Figure 4.2 shows the steps inside GRAPHITE framework:

- **SCoP outlining.** The maximal subgraphs of CFG that have static control property and statically analysable affine data access patterns are extracted from the GIMPLE three-address code.
- **Mapping three-address code into the polyhedra.** For the outlined SCoPs (static control parts) the polyhedral representation is reconstructed and mapped to the GIMPLE intermediate representation.

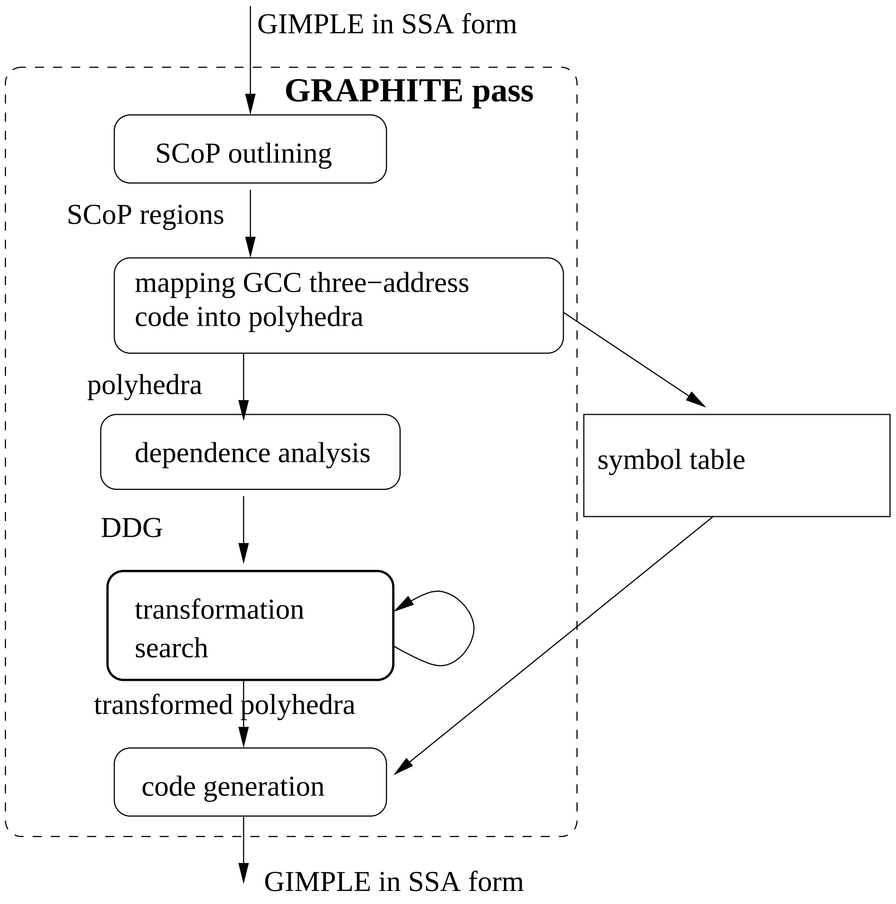


Figure 4.2 – Inside Graphite

- **Data dependence analysis.** Data-dependence analysis is performed to construct RDDG. Special treatment for scalar dependences and reductions is applied.
- **Transformation search.** A search for a legal loop transformation sequence based on a cost model is performed. This is the topic of chapters 6 and 7.
- **GIMPLE code generation.** Three-address GIMPLE code corresponding to the transformation expressed in the polyhedral model is generated.

We will discuss the technical details of GRAPHITE steps in the subsequent sections. We will not be covering the actual loop transformations, since the search for efficient loop transformations is extensively covered in Chapters 6, 5 and 7. We will start by briefly discussing an induction variable recognition, which is an essential building block for the operation of GRAPHITE.

Induction variable recognition

Induction variable analysis is essential part of any loop optimizing compiler [63]. GCC implements induction variable analysis by constructing TREC(Tree of RECurrences) [126] – a closed form expressions that capture the evolution of induction variable as a function of iteration counts:

$$\chi(\mathbf{i}) = \chi(i_1, i_2, \ldots, i_n)$$

TREC expressions are either constants $\chi = c$ or they are defined recursively as $\chi = \{\chi_a, +, \chi_b\}_k$, where χ_a and χ_b are trees of recurrences, c is an integer constant or variable name, and subscript k is the loop dimension along which we are evaluating an induction variable [126]. If we consider an evolution of induction variable in the loop at level k, χ_a is an initial value and χ_b is an increment at each loop iteration.

TREC expressions can represent linear, affine, polynomial and exponential functions as well. We will not provide details on construction and evaluation of those expressions, for the reference please see [126].

The use of TREC expressions in GRAPHITE is threefold: (1) detection of static control part regions, (2) construction of the polyhedral information for the given CFG subgraph and (3) code generation.

4.3.1 SCoP outlining

The first step in GRAPHITE is the extraction of the maximal subgraphs of CFG that have static control described by the affine loop bounds, affine conditionals and affine data access patterns. Each such a subgraph is called SCoP(Static Control Part) in GRAPHITE terminology.

Since GRAPHITE is operating on a low level three-address code scattered in a CFG, high level syntactical information is lost: loop structure, loop induction variables, loop bounds, conditionals, data accesses and reductions. Extraction of maximal SCoP regions proceeds in two steps:

1. Construct a *region tree* that represents the nesting of reducible [38] regions in CFG.
2. Traverse the tree, starting from the topmost root region. For each region (in the traversal order):
   (a) Check if all the basic-blocks belonging to the region satisfy the *affine program* conditions. If yes:
      i. Mark the region as a SCoP

      else

      i. Mark the region as nonSCoP. Repeat the procedure for child regions.

In order to check whether a basic block satisfies affine program conditions several checks are performed:

1. if a basic block is a loop header block:

   (a) iteration count is symbolically determinable (but it might be unknown at compilation time if it depends on the global parameter)
   (b) loop bounds are constants or affine expressions of the outer loop induction variables and global parameters
   (c) loop is *single exit loop*
2. otherwise, all the statements inside a basic block are checked:
   (a) calls to functions with side effects are not allowed (`pure` and `const` function calls are allowed)
   (b) the only memory references that are allowed are accesses through arrays with affine subscript functions
   (c) the conditional statements are expressed as affine functions of induction variables and global parameters

If some basic-block does not satisfy affine program conditions, the region that contains this basic block is marked as a non-SCoP. In such a case, the sub-regions (child nodes in a region tree) are checked recursively.

The most favorable case is when the outermost region is proved to be a SCoP. This is obvious, especially if the outer SCoP contains the outermost loop in a function. In a less favorable case, some child nodes in the region tree are marked as a SCoP and some as a non-SCoP. It is less favorable, since inner SCoPs contain inner-loops and also because inner SCoPs are fragmented. The least favorable case happens when a SCoP region is composed of a single basic block only - such SCoPs are called *degenerate* and they do not contain any loop. Degenerate SCoPs are not optimized.

As an example, a CFG with marked reducible regions is shown in Figure 4.3. Regions themselves are either nested within each other or they are ordered in a sequence. A relative order and nesting of regions is represented as a region-tree and it is shown in Figure 4.4. Regions containing whole loops are marked ($L1$ and $L2$ respectively). Regions that contain basic blocks numbered 2 and 8 are degenerate regions, since they are not part of any loop. The region marked as $L1$ includes a subregion $L2$ and three subregions executed in sequence. If a region $L1$ is a SCoP then it would contain two nested loops, which is the best

On the other hand, let us consider an example in Figure 4.6. The corresponding region tree is shown in Figure 4.5. An original CFG represents a single nest loop with one basic block scheduled before the loop, and another one scheduled after the loop. The loop header basic block (number 3) contains non-affine loop bounds, so it is marked as not satisfying affine conditions. Thus, the region that contains the loop is not marked as a SCoP. The remaining regions form three separate SCoPs: SCoP1, SCoP2 and SCoP3. Those SCoPs are degenerate and are not amenable to loop optimizations.

4.3.2 Mapping three-address intermediate representation into polyhedra

After outlining SCoP regions, each SCoP is processed so to map the three-address code contained inside region into the polyhedral representation. Only SCoPs that are containing at least one loop are processed.

As shown in Figure 4.7, the strongly connected components of the CFG subgraph that belongs to the region form the loops. Given any two loops, one of them is either completely nested in the other or they are disjoint. Each basic block is enclosed in at least one loop region. The nesting of the loop regions determines the *loop depth* of the basic block.

Three components (Section 2.3) of the polyhedral model representation are extracted from three-address code IR:

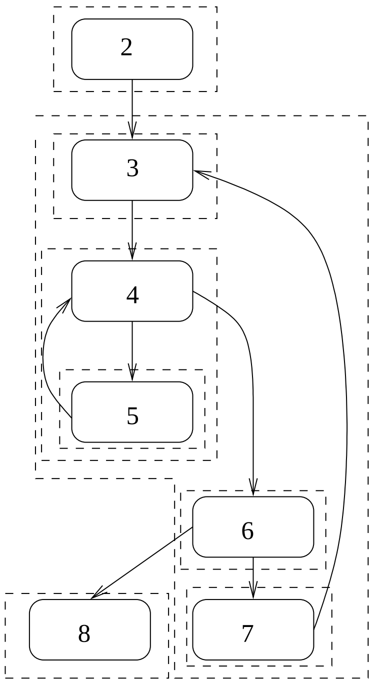


Figure 4.3 – CFG graph with region nesting

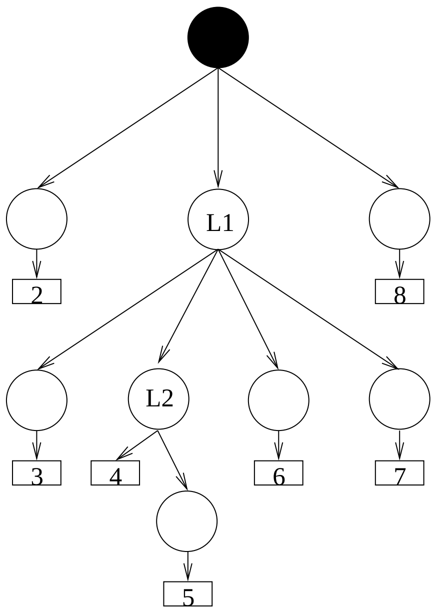


Figure 4.4 – Region nesting tree

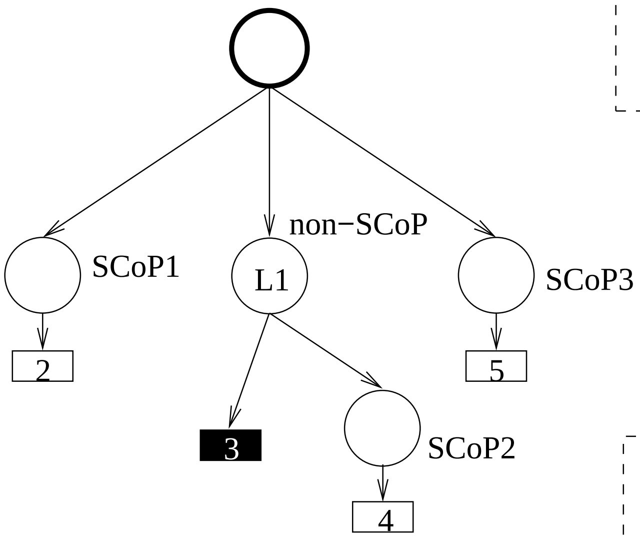


Figure 4.5 – Region tree for splitted SCoPs

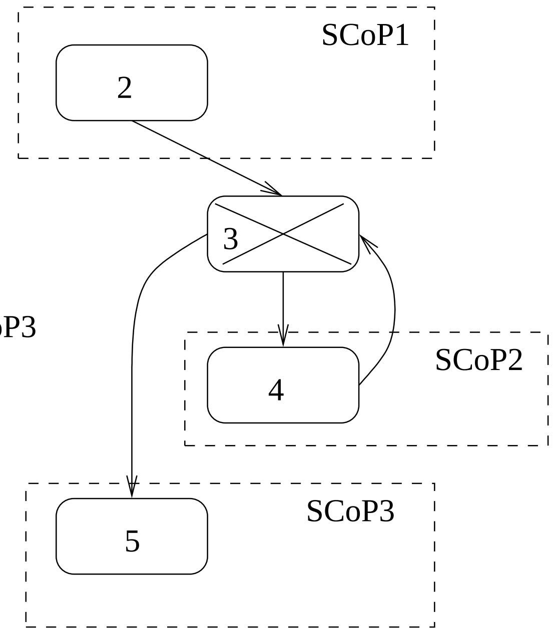


Figure 4.6 – Breaking SCoP

1. **Iteration domains.** For each basic block B that belongs to a SCoP the iteration domain $\mathcal{D}^B$ is built. The dimensionality of the iteration space is equal to the number of loops belonging to a SCoP and enclosing the basic block.
2. **Schedules.** The *initial* scheduling function θ^B for each basic block B belonging to a SCoP is built. The initial scheduling function encodes the execution order of the basic block in an original program – before a transformation is applied.
3. **Data references.** Each basic block might contain memory data accesses in its statements. In a region complying with affine program constraints the only memory accesses are through array accesses whose access functions could be expressed as affine functions of loop induction variables. For each basic block B the sets of write and read data references, $\mathcal{W}^B$ and $\mathcal{R}^B$ respectively, are constructed.

We will discuss the technical details of the polyhedral model representation extraction in the rest of this subsection. A discussion is based on an example CFG shown in Figure 4.7, obtained during the compilation of a matrix vector multiplication kernel shown in the same figure.

Reconstructing loop domains

SCoP extraction detects the static control region containing two nested loop regions: $L1$ and $L2$. Basic block B_4 is the loop header of the loop $L2$ in which the variable j is used to controls the exit condition of the loop. Since the code is in SSA form, variable j has two versions: j_1 and j_2, corresponding to different assignments to the original j variable.

In order to detect the evolution of the variable j in the consecutive loop iterations of the loop $L2$, the TREC construction algorithm is used. TREC expression construction is done by following def-use chains of the induction variable. In SSA form, def-use chains are explicit, since there is an unique definition for each use.

The variable version j_1 that is used in a loop exit condition is defined in a statement S_7: $j_1 = j_2 + 1$. This statement uses a variable version j_2 that is in turn defined in a ϕ-function in the statement S_1: $j_2 = \phi(j_1, 0)$. The statement S_1 uses the variable version j_1 that is defined in the statement S_7. This def-use chain closes a cycle and a TREC expression of the form $\{0, +, 1\}_2$ is deduced. This is illustrated by dashed arrows inside the basic block B_4 in Figure 4.7.

A TREC expression of the form $\{0, +, 1\}_2$ means that the induction variable initial value is the integer constant 0 and that the induction variable is incremented by the step of 1 at each iteration of the loop $L2$.

The variable version j_1 is used in the conditional statement S_8: $j_1 < N$. Combining this (affine) condition with a TREC an affine constraint on loop induction variable could be deduced: $0 \leq j \leq N - 1$. The same induction variable analysis is performed for the loop $L1$.

Affine expressions defining loop domains for all the basic blocks of the SCoP are shown in Figure 4.7.

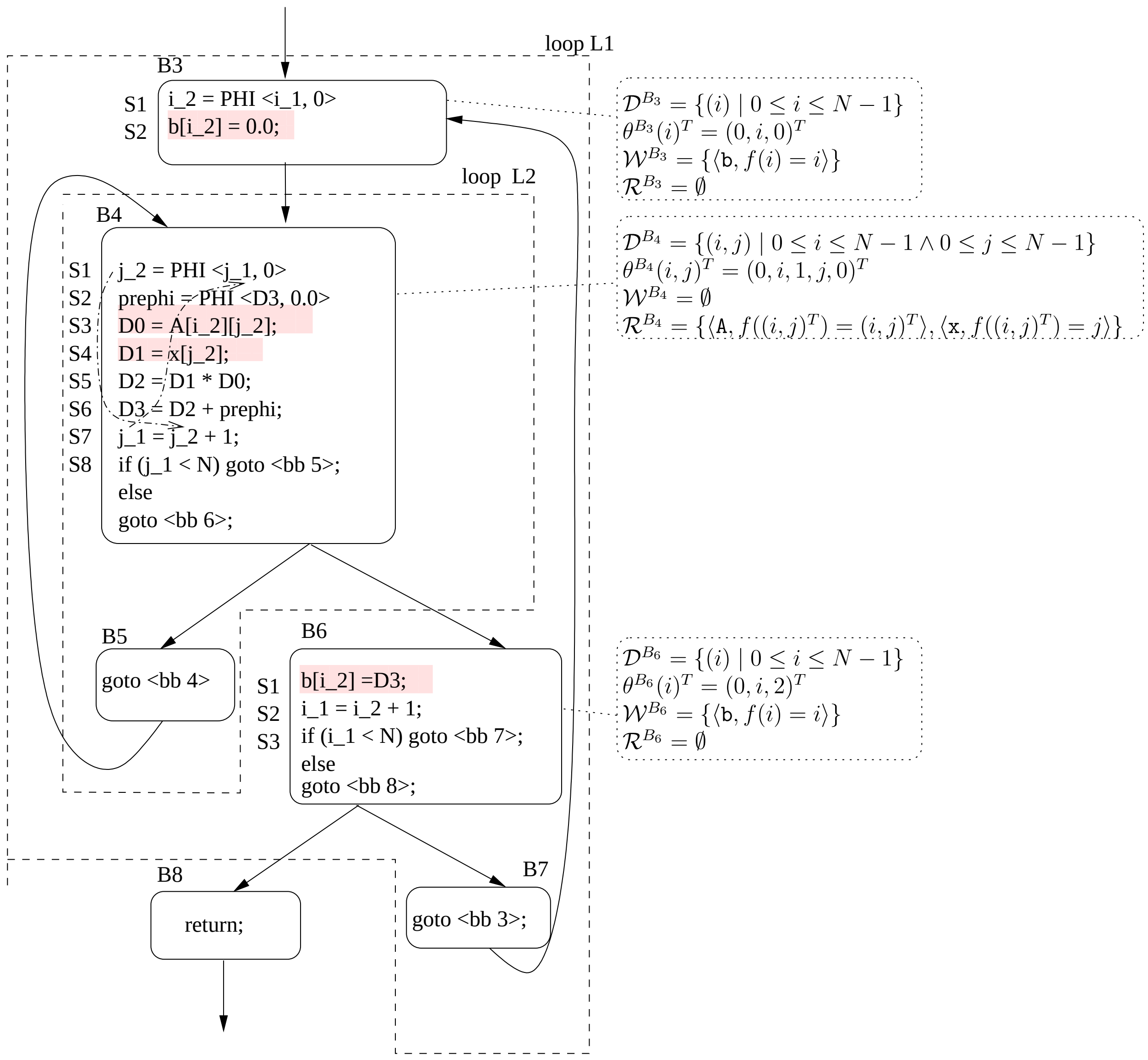
Reconstructing data references

The property of the three-address code used in GCC is that each statement might contain at most one read or write data reference to a memory location. Furthermore the affine program constraint mandates that all the memory accesses could only be performed through arrays whose subscript functions are affine functions.

For each basic block that belongs to the SCoP all the statements are scanned. For each memory access statement the subscript functions are reconstructed by evaluating the evolution of induction variables that are used as array indices.

For the reconstruction of *access functions* the TREC expressions of the array indices are evaluated. For the basic block B_3 there is one memory access statement S_2: $\mathtt{b}[i_2] = 0.0$. The basic block B_3 belongs

```
  for (i = 0; i < N; i++) {
S1: b[i] = 0;
    for (j = 0; j < N; j++)
S2:   b[i] = b[i] + A[i][j] * x[j];
  }
```



```
B3
S1  i_2 = PHI <i_1, 0>
S2  b[i_2] = 0.0;

B4
S1  j_2 = PHI <j_1, 0>
S2  prephi = PHI <D3, 0.0>
S3  D0 = A[i_2][j_2];
S4  D1 = x[j_2];
S5  D2 = D1 * D0;
S6  D3 = D2 + prephi;
S7  j_1 = j_2 + 1;
S8  if (j_1 < N) goto <bb 5>;
    else
    goto <bb 6>;

B5
goto <bb 4>

B6
S1  b[i_2] =D3;
S2  i_1 = i_2 + 1;
S3  if (i_1 < N) goto <bb 7>;
    else
    goto <bb 8>;

B7
goto <bb 3>;

B8
return;
```

$\mathcal{D}^{B_3} = \{(i) \mid 0 \leq i \leq N-1\}$
$\theta^{B_3}(i)^T = (0, i, 0)^T$
$\mathcal{W}^{B_3} = \{\langle \texttt{b}, f(i) = i \rangle\}$
$\mathcal{R}^{B_3} = \emptyset$

$\mathcal{D}^{B_4} = \{(i,j) \mid 0 \leq i \leq N-1 \wedge 0 \leq j \leq N-1\}$
$\theta^{B_4}(i,j)^T = (0, i, 1, j, 0)^T$
$\mathcal{W}^{B_4} = \emptyset$
$\mathcal{R}^{B_4} = \{\langle \texttt{A}, f((i,j)^T) = (i,j)^T \rangle, \langle \texttt{x}, f((i,j)^T) = j \rangle\}$

$\mathcal{D}^{B_6} = \{(i) \mid 0 \leq i \leq N-1\}$
$\theta^{B_6}(i)^T = (0, i, 2)^T$
$\mathcal{W}^{B_6} = \{\langle \texttt{b}, f(i) = i \rangle\}$
$\mathcal{R}^{B_6} = \emptyset$

Figure 4.7 – Matvect kernel - source code and internal representation

to the loop $L1$ and the evolution of the array index variable i_1 is expressed as a TREC $\{0,+,1\}_1$. This expression is interpreted as an affine access function $f(i) = i$. The complete set of write references of the basic block B_3 is: $\mathcal{W}^{B_3} = \{\langle \mathtt{b}, f(i) = i\rangle\}$. Symbol `b` is a symbolic reference to the base address of the `b` array.

The basic block B_4 belongs to the loop $L2$ (with iteration vector is $(i,j)^T$). Its statement S_3: $\mathtt{D0} = \mathtt{A}[i_2][j_2]$ represents a read memory array access. The TREC of the first subscript is $\{0,+,1\}_1$, while the TREC of the second subscript is $\{0,+,1\}_2$. Combining those two expression we get a multidimensional affine access function : $f((i,j)^T) = (i,j)^T$. The complete set of read references of B_4 is: $\mathcal{R}^{B_4} = \{\langle \mathtt{A}, f((i,j)^T) = (i,j)^T\rangle, \langle \mathtt{x}, f((i,j)^T) = j\rangle\}$.

Writes/reads to/from scalar variables are not represented. Writes and reads to temporary scalar variables are treated as an internal state of the basic block. Since they are not producing visible side-effects they do not have to be captured. We will discuss this further in the section dealing with data dependence analysis. In Figure 4.7 the represented data references are marked (the statements that access the memory are emphasized).

Construction of the original schedules

The scheduling functions θ^B encoding the original program execution order are constructed for each basic block belonging to a SCoP region of a CFG.

As it was defined in section 2.3.4, the scheduling matrix Θ^B is decomposed into dynamic (A and Γ matrices) and static (β vector) scheduling components. For the scheduling matrices Θ^B corresponding to the original execution order, their dynamic scheduling components, A and Γ matrices, are an identity matrix and all-zero matrices respectively. The static scheduling β vector for each basic block B is deduced from the region tree.

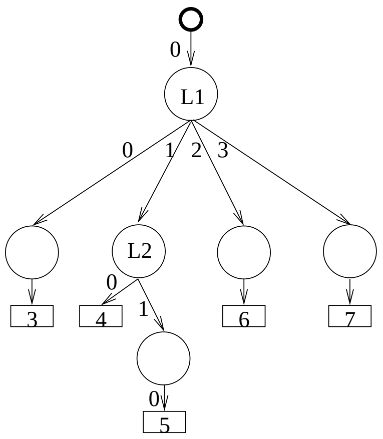


Figure 4.8 – Relative nesting and ordering of loops and basic blocks

Let us assume that our SCoP region includes the whole $L1$ loop region. The region tree for the CFG in Figure 4.7 is shown in Figure 4.8. The child nodes of each region are numbered. By following a path from the SCoP root region down to the given basic block, we construct a β vector that consists of the numbers labelling the child positions. This idea is described in [22]. For example, the β vector for the basic block B_3 is $(0,0)$. The β vector for the basic block B_4 is $(0,1,0)$. The construction of static scheduling components (β vectors) based on a region tree guarantees an uniqueness of the scheduling time-stamps for each basic block instance.

As a side note: the basic blocks that do not contain any computational statements are not considered for the polyhedral representation. Empty basic blocks are simply discarded from the region tree. In our example, the basic blocks [4] B_5 and B_7 are not considered as a part of the polyhedral representation. Since the polyhedral model captures all the necessary information to reconstruct the control-flow according to the new schedule, the basic blocks containing only `goto` statements are ignored, since they are going to be recreated in the code generation stage of the GRAPHITE.

Mapping IR symbols into the polyhedra

The GIMPLE intermediate representation statements contain symbolic variable names. Variables are either scalars or arrays. On the other hand, the polyhedral representation is a mathematical abstraction representing sets of integer vectors [5] . The mapping between symbolic names of the three-address code and the variables of the polyhedra has to be maintained. For that purpose a mapping, similar to compiler *symbol table*, is used. An example of this correspondence is seen in Figure 4.7.

Differences from the source-to-source compilers

The source-to-source compilers [124, 123] directly map the source code level statements into the polyhedral model. For example, the polyhedral representation for the source code in Figure 4.7 would contain two domains $\mathcal{D}^{S_1}$ and $\mathcal{D}^{S_2}$ corresponding to the two syntactical statements S_1 and S_2. Also, the scheduling functions are provided per source level statement: θ^{S_1}, θ^{S_2}.

After the source code is translated into GIMPLE three-address code, the high-level statements are broken down into several three-address code statements - temporary variables are inserted to hold the intermediate results. Inside GRAPHITE we are storing the polyhedral representation per each basic block and not per each GIMPLE statement. Thus, we have three domains, one for each basic block: $\mathcal{D}^{B_1}$, $\mathcal{D}^{B_2}$ and $\mathcal{D}^{B_3}$. The same holds true for scheduling functions. We use a superscript B_i to emphasize this fact.

The fact that we store the scheduling functions per basic block and not per three-address code statement contained inside basic block, means that all the statements inside one basic block share the same scheduling function. Statements inside the basic block are not rescheduled. In section 4.5.1 we will discuss the implications of this decision.

In the source code of a typical imperative language (C, fortran), one assignment statement contains one write (left hand side) and several reads (right hand side). After transforming the source level statement into three-address code the original write and read data references are scattered into several statements. The original data references might be scattered among several basic blocks, or even optimized out completely.

For example, the original source code shown in Figure 4.7 contains one write data reference (`b[i]`) in textual statement S_1. The corresponding data reference in the three-address code (shown in Figure 4.7) is contained inside basic block B_3 in the statement S_2. The two read data references contained in the statement S_2 (`A[i][j]` and `x[j]`) have their corresponding references inside the basic block B_4 (in statements S_3 and S_4). On the other hand, the write data reference (`b[i]`) from source-level statement S_2 appears in the basic block B_6 as statement S_1. The original read from array `b` inside source-level statement S_2 does not appear in three-address code at all - it has been optimized out and replaced by a scalar access [6] .

4. so called 'latch' [70] basic blocks that jump back to the loop header

5. If we want to emphasize the different domains of variable definitions, we will use `i` for the symbol names in IR and the mathematical notation i for the occurrence of the variable in the polyhedral model.

6. this happens in the PRE (Partial Redundancy Elimination) optimization pass that is scheduled to run before GRAPHITE

$\mathcal{D}^{B_3} = \{(i) \mid 0 \le i \le N-1\}$ $\theta^{B_3}(i)^T = (0,i,0)^T$ $\mathcal{W}^{B_3} = \{\langle \mathtt{b}, f(i) = i\rangle\}$ $\mathcal{R}^{B_3} = \emptyset$
$\mathcal{D}^{B_4} = \{(i,j) \mid 0 \le i \le N-1 \land 0 \le j \le N-1\}$ $\mathcal{W}^{B_4} = \emptyset$ $\mathcal{R}^{B_4} = \{\langle \mathtt{A}, f((i,j)^T) = (i,j)^T\rangle, \langle \mathtt{x}, f((i,j)^T) = j\rangle\}$ $\theta^{B_4}(i,j)^T = (0,i,1,j,0)^T$
$\mathcal{D}^{B_6} = \{(i) \mid 0 \le i \le N-1\}$ $\mathcal{W}^{B_6} = \{\langle \mathtt{b}, f(i) = i\rangle\}$ $\mathcal{R}^{B_6} = \emptyset$ $\theta^{B_6}(i)^T = (0,i,2)^T$

Figure 4.9 – Polyhedral representation of three-address code

$\mathcal{D}^{S_1} = \{(i) \mid 0 \le i \le N-1\}$ $\theta^{S_1}(i)^T = (0,i,0)^T$ $\mathcal{W}^{S_1} = \{\langle \mathtt{b}, f(i) = i\rangle\}$ $\mathcal{R}^{S_1} = \emptyset$
$\mathcal{D}^{S_2} = \{(i,j) \mid 0 \le i \le N-1 \land 0 \le j \le N-1\}$ $\theta^{S_2}(i,j)^T = (0,i,1,j,0)^T$ $\mathcal{W}^{S_2} = \{\langle \mathtt{b}, f(i) = i\rangle\}$ $\mathcal{R}^{S_2} = \{\langle \mathtt{A}, f((i,j)^T) = (i,j)^T\rangle, \langle \mathtt{x}, f((i,j)^T) = j\rangle, \langle \mathtt{b}, f(i) = i\rangle\}$

Figure 4.10 – Polyhedral representation the origianl source code

Putting it all together

The complete polyhedral representation of the three-address code fragment contained inside the SCoP from Figure 4.7 is shown in Table 4.9. Figure 4.7 shows the same information attached to the respective basic blocks.

As a comparison, Table 4.10 shows the (hypothetical) polyhedral representation of the respective source-level code in Figure 4.7.

4.4 The data dependence analysis of three-address code

The polyhedral model used in the source-to-source compilers [34], [67] only allows memory references through array accesses. An access to a scalar value is treated as an access to an one-element array: `a = ...` is rewritten as `a[0] = ...`.

While this approach is practical for the source-to-source compilers, transforming the numerical kernels where array accesses constitute the majority of data references, it has prohibitive cost if used in the three-address code compilation context.

The lowering process from the source code down to the three-address code introduces (many) temporary scalar variables. Treating all the scalar temporary variables as single-element arrays and taking them into an account for the dependence analysis might lead to an explosion in the size of data dependence graph.

This is of particular concern, if we take into an account that the scheduling algorithms [34, 67], based on the polyhedral model, have a complexity that is dependent on the number of edges of a data dependence graph $G = (V, E)$.

In order to efficiently manage the data dependences between scalar variables in the three-address code, the smallest necessary subset of the scalar variables is taken into a consideration for the dependence analysis.

Internally in GRAPHITE, those scalars whose data dependences must be exposed are marked as if they were single-element arrays. As a preprocessing step, GRAPHITE is marking those scalars that must to be processed by the dependence analysis as single-element arrays. Scalars that are not marked are not taken into the consideration for the dependence analysis.

In order to mark the necessary scalars for the dependence analysis, GRAPHITE is classifying the scalar data dependences into four categories:

1. Intra basic block scalar data dependences
2. Cross basic block scalar data dependences
3. Inductions
4. Reductions

Only the *cross basic block* and the *reduction* scalar dependences have to be explicitly represented. This is done by introducing the *shadow* single-element arrays for those scalars that are involved in dependences. Scalar data dependences that are induced by the computation of the induction variables and scalar data dependences that are contained within a single basic block are not represented explicitly in the polyhedral data dependence graph.

Given the three-address code in Figure 4.11, the effect of the preprocessing step that introduces the shadow arrays is shown.We will provide the brief discussion of the details related to handling each kind of scalar dependences in GRAPHITE. The illustration shown in Figure 4.11 will serve as common base example for all the discussions.

Intra basic block data dependences

As mentioned in Subsection 4.3.2, the basic block is the basic unit of scheduling. The statements inside a basic block are not rescheduled. The transformation process from the source-code to the lower level three-address code introduces temporary scalar variables to store the intermediate computation results.

Definition 4.4.1 (Intra basic block dependence)**.** A scalar data dependence between the scalar value X defined within a basic block B at iteration $\mathbf{i}$ and its subsequent uses within the same iteration $\mathbf{i}$ of the same basic block B are called *intra basic block dependences*

Since the relative order of the statements inside a basic block is not changed by a transformation, intra basic block dependences are always preserved, regardless of the basic block schedules θ^B. As a consequence, intra basic block dependences are not explicitly represented in the polyhedral dependence graph.

For example, consider scalar variables `D0`, `D1`, `D2` defined in B_4 - they are defined and used within the same basic block and within the same iteration. The relative ordering of the statements S_3, S_4, S_5 and S_6 will not be changed in the polyhedral transformation. The flow dependence from statements S_3 and S_4 to statement S_5 would always be preserved. Please note that this approach is similar to the explicit privatization used for dependence removal [7].

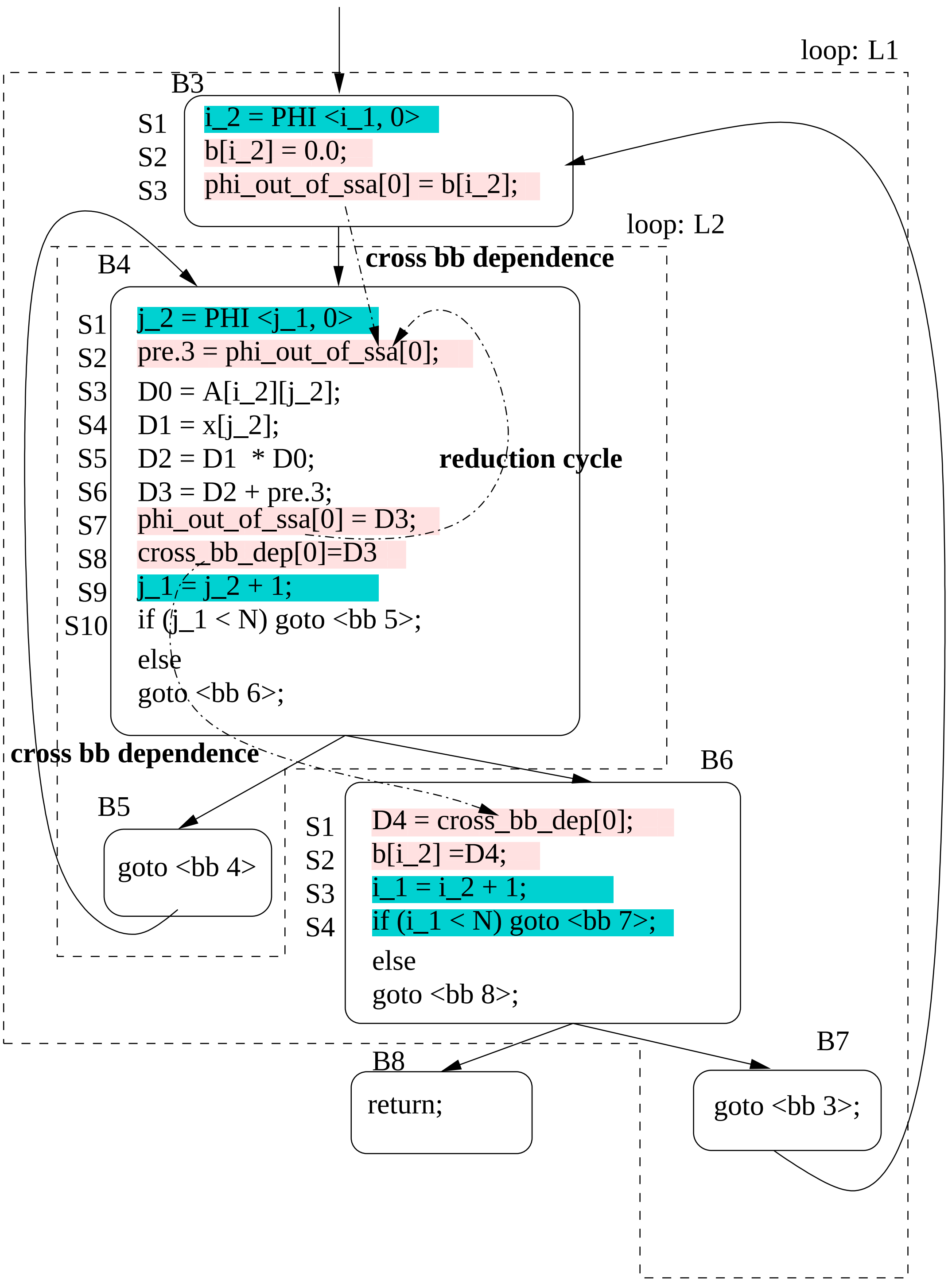


Figure 4.11 – Matvect kernel - cross bb dependences, inductions and reductions

Cross basic block data dependences

Since the different basic block instances might be rescheduled independently, the data dependences that occur between scalar values belonging to two different basic blocks must be captured in order to restrict the possible rescheduling of the basic block instances.

If some scalar variable X is assigned in one instance of the basic block and if it is subsequently used in another basic block or subsequent iteration of the same basic block, that scalar value will get the shadow array that exposes the data dependence to the dependence analyser. Those scalars are treated as single element arrays, so that dependence analysis could capture them in the dependence test.

An example is given in Figure 4.11: there is cross basic block dependence between a scalar write in B_4 in the statement `D.3 = D.2 + pre.3` and its subsequent use in B_6 in the statement `b[i_2] = D.4`. An explicit single-element array `crossu_bb_dep[0]` is introduced to represent this dependence.

Inductions

Scalar variables representing the loop counters or auxiliary induction variables that could be expressed as TREC expressions are ignored for dependence analysis. The data dependences induced by computation of those variables could be ignored, since they could be represented by *scheduling invariant* expressions of the form: $\chi(\mathbf{i}) = \chi(i_1, i_2, \ldots, i_n)$. Those expressions are reconstructed in the GIMPLE during the code generation step, after the transformation has been applied. Since the expression could be correctly reconstructed, regardless of the applied scheduling, there is no need to explicitly keep those dependences in the dependence graph.

Please note that in the source-to-source compilers, the induction variables are implicit and their computation is not taken into an account for the dependence analysis. We base our reasoning on the same principle, but in our case we have to explicitly discover those inductions inside a three-address code.

Reductions

Reductions [139] form a special computational pattern used very often in the scientific codes. Indeed, an effective parallelization of loop kernels relies heavily on an effective detection and parallelization of reductions. Mostly often it is used to perform a summation over the values of an array.

Data dependence theory puts a constraint on the relative scheduling of statement instances. But some operations, if proved to be associative and commutative algebraic operations, might be scheduled in an arbitrary order.

If the operator along which the reduction happens can be proved to be commutative and associative, then the dependences that are induced by variables involved in a reduction are marked as belonging to reduction.

As an illustration, the data dependences that form a *reduction cycle* are shown in Figure 4.11. Indeed, the scalar `D3` accumulates the values computed in the scalar `D2` in each iteration of the loop. An addition operator '$+$' is commutative and associative, if performed on integer data type, so the order in which the summation is performed does not influence the correctness of the final results.

Putting it all together

Taking into an account all the dependences that have to be explicitly represented, we come up with the data dependence graph shown in Figure 4.12. Please note that the set of vertices is composed from the three-address code statements, but those statements are contained within basic blocks, which is rep-

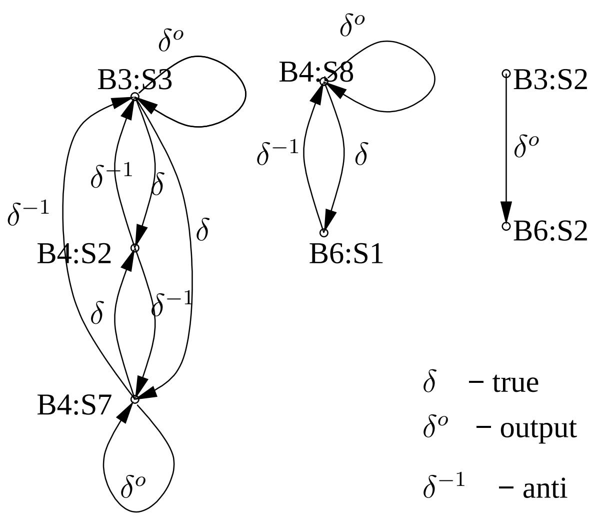


Figure 4.12 – Explicit data dependence graph

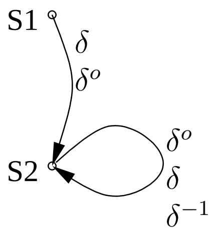


Figure 4.13 – Explicit data dependence graph

resented with the notation $B_i : S_j$. The dependence edges $e \in E$ are connecting the three-address code statements which contain the explicit memory accesses to arrays.

As a comparison, the dependence graph that is built by a source-to-source compiler is shown in Figure 4.13. The dependence graph of the three-address code contains, besides true data dependences, many anti and output data dependences - those are jointly classified as *memory based* dependences. These dependences might significantly constrain the space of legal transformations. Later in Chapter 5 we will show how to efficiently deal with those kind of dependences, such that the space of legal polyhedral transformations on the three-address code is equivalent to the source-level polyhedral compilation.

4.5 Transformations

The loop transformations in the GRAPHITE framework are performed on the polyhedral representation. In order to maintain the *composability* of loop transformations, we perform all the loop transformations through the scheduling matrix of the polyhedral model. This is in contrast to the traditional [46, 156, 34] approach to transformations in the polyhedral model, where some transformations are expressed through scheduling matrices (affine transformations), where the loop tiling and strip-mining transformations require the modification of the iteration domains. We will later show how we do overcome this non-homogeneity in expressing the loop transformations.

Once the analysed program is brought into the polyhedral model representation and after the desired transformations are selected, the output code is generated from the transformed schedules. The *automatic* search for the transformation is conducted according to the cost-model driven search approach shown in chapters 7, 5 and 6.

The state of the art polyhedral code generator CLooG [22, 20] generates the loops that scan the integer vectors contained in polyhedra. The integer vectors are scanned in the lexicographic order specified by the scheduling function. The code generator is not responsible for maintaining the legality of the transformations - iterations are scanned according to the transformed scheduling matrices. It is the duty of the *violated dependence analysis*, discussed in Chapter 5, to guarantee the legality of the transformations.

The CLooG code generator generates an AST (abstract syntax tree) representing the newly generated loop nests, conditionals and affine loop bounds and conditions. This structure is traditionally used in the source-to-source compilers [123, 124] to generate the target source code. In the case of GRAPHITE the AST is used to generate the *output CFG* that corresponds to new loops. The symbol table (as shown in Figure 4.2) that maps the original three-address code into polyhedral model variables is now used to regenerate the target three-address code.

All the loop transformations, including loop tiling, are expressed seamlessly through *scheduling functions* and it is the duty of CLooG to generate the appropriate loop structures. GRAPHITE is interacting with parallelizer and vectorizer (as shown in Figure 4.1) passes of GCC compiler: if the loop is detected to be a parallel DOALL loop, it is marked as a such in the code generation phase. After GRAPHITE, the marked loop will be processed by the parallelizer to generate the multithreaded code. The similar interaction happens with vectorizer pass.

We will now discuss several design issues related to the transformations and code generation in our framework.

4.5.1 Granularity of scheduling

As we have mentioned in Chapter 2 (subsection2.3.1), the basic building block of the polyhedral representation could be chosen according to the abstraction level on which the polyhedral model is to be built.

In the source-to-source compilers, the basic, and the most natural, atomic unit of scheduling is a source-level statement. In a three-address-code based polyhedral compiler, such as GRAPHITE, it is not practical to build a polyhedral representation for each three-address code statement. The most reasonable choice is to have a basic block of the control flow graph as the basic scheduling unit.

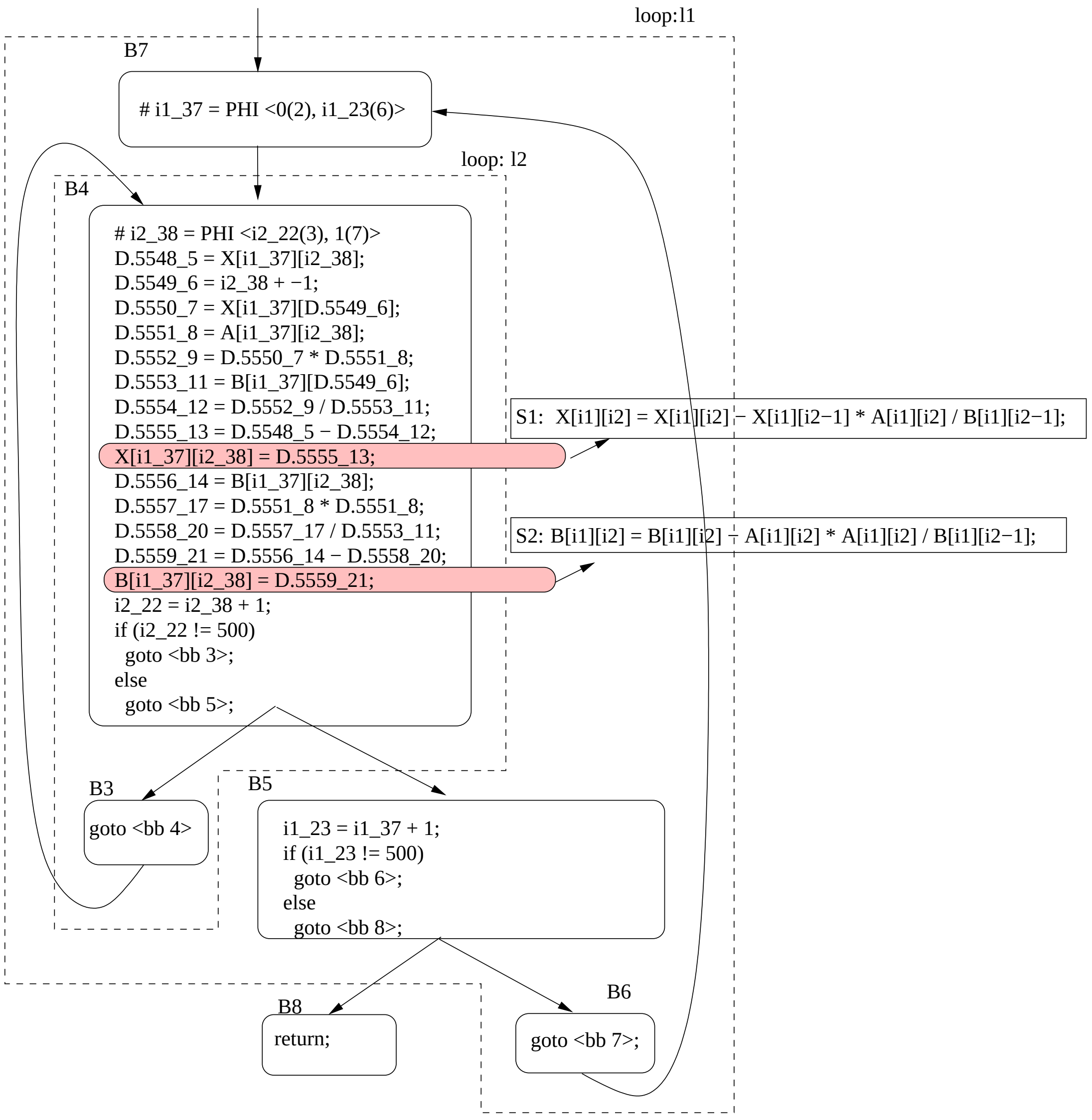
Nevertheless, the choice of having a basic block as the basic unit of scheduling is not always optimal and may be way too restrictive. Let us consider an illustration in Figure 4.14. The source-level code for the ADI[7] kernel is shown. It contains two syntactic statements: S_1 and S_2. The corresponding three-address code is shown as well. One can observe that all the three-address statements are clustered in one basic block, the basic block B_4.

Since the design choice is to have a scheduling function per basic block, all the three-address code statements contained within the same basic block would get the same schedule. Obviously, this is not desirable, especially in the case when we want to apply *per statement* affine scheduling [34]. But having one scheduling function per each three-address code statement is non-practical, for the scheduling scalability reasons.

A solution that we have applied is to *split* such a basic block into *clusters*, such that each cluster would contain one write statement (highlighted in Figure 4.14) and all the necessary temporary variables that are produced within the same basic block. This could easily be achieved by following use-def chains in the SSA form. After this preprocessing, each cluster becomes a separate basic block that could be scheduled independently.

7. taken from Polybench benchmark suite

```
    for (i1=0; i1<N; i1++) {
        for (i2 = 1; i2 < N; i2++) {
S1:         X[i1][i2] = X[i1][i2] − X[i1][i2−1] ∗ A[i1][i2] / B[i1][i2−1];
S2:         B[i1][i2] = B[i1][i2] − A[i1][i2] ∗ A[i1][i2] / B[i1][i2−1];
        }
    }
```



```
# i2_38 = PHI <i2_22(3), 1(7)>
D.5548_5 = X[i1_37][i2_38];
D.5549_6 = i2_38 + −1;
D.5550_7 = X[i1_37][D.5549_6];
D.5551_8 = A[i1_37][i2_38];
D.5552_9 = D.5550_7 * D.5551_8;
D.5553_11 = B[i1_37][D.5549_6];
D.5554_12 = D.5552_9 / D.5553_11;
D.5555_13 = D.5548_5 − D.5554_12;
X[i1_37][i2_38] = D.5555_13;
D.5556_14 = B[i1_37][i2_38];
D.5557_17 = D.5551_8 * D.5551_8;
D.5558_20 = D.5557_17 / D.5553_11;
D.5559_21 = D.5556_14 − D.5558_20;
B[i1_37][i2_38] = D.5559_21;
i2_22 = i2_38 + 1;
if (i2_22 != 500)
  goto <bb 3>;
else
  goto <bb 5>;
```

```
S1:  X[i1][i2] = X[i1][i2] − X[i1][i2−1] * A[i1][i2] / B[i1][i2−1];
S2: B[i1][i2] = B[i1][i2] − A[i1][i2] * A[i1][i2] / B[i1][i2−1];
```

```
i1_23 = i1_37 + 1;
if (i1_23 != 500)
  goto <bb 6>;
else
  goto <bb 8>;
```

Figure 4.14 – source code and IR representation for ADI kernel

4.5.2 Functions represented as relations

The classical definition (Chapter 2, subsection 2.3.3) of the polyhedral model components states that the data accesses and schedules are expressed as data access *functions* and scheduling *functions*:

$f : \mathbf{i} \in \mathcal{D} \rightarrow \mathbf{s} \in \mathcal{D}$

$\theta : \mathbf{i} \in \mathcal{D} \rightarrow \mathbf{t} \in \mathcal{D}$

The data access function is the one-to-one mapping from an iteration space ($\mathbf{i} \in \mathcal{D}$) to a *data space* of array subscripts ($\mathbf{s} \in \mathcal{D}$). The scheduling function is the one-to-one mapping from an iteration space ($\mathbf{i} \in \mathcal{D}$) to a *time domain* of multidimensional timestamps ($\mathbf{t} \in \mathcal{D}$).
In GRAPHITE we extend this definition with a notion of *access relation* and *scheduling relation*:

$\mathcal{F} : (\mathbf{i}, \mathbf{s}) \in \mathcal{D} \times \mathcal{D}$

$\theta : (\mathbf{i}, \mathbf{t}) \in \mathcal{D} \times \mathcal{D}$

The mapping from the iteration vector to the subscript function is not anymore restricted to one-to-one mappings. This enables us to represent the full *access regions* – when the data reference information could not be expressed as a direct affine function. This is the case if only the approximation of the memory access is available, coming from an interprocedural analysis for example. The notion of access regions is a known term used in interprocedural compilation [151]. The access relation is now expressed by an *access polyhedron*:

$\mathcal{F} = \left\{ (\mathbf{i}, a, \mathbf{s}) \mid \mathrm{F} \times (\mathbf{i}, a, \mathbf{s}, \mathbf{g}, 1)^T \geq \mathbf{0} \right\}$

An additional component, the *alias set number 'a'*, captures points-to information (pointer aliasing); it allows us to extend the notion of array memory access to pointers accesses, provided that the precise aliasing information [122, 10] could be obtained and that the original access functions (or regions) could be reconstructed, as shown in Section 4.3.2.

Similarly, the scheduling is represented as a *scheduling relation*. [8] represented by an *scheduling polyhedron*:

$\theta = \left\{ (\mathbf{t}, \mathbf{i}) \mid \Theta \times (\mathbf{t}, \mathbf{i}, \mathbf{g}, 1)^T \geq \mathbf{0} \right\}$.

Having the schedules expressed as relations, we can express the *tiling* transformation with a scheduling relation alone, without modifying the statement iteration domain $\mathcal{D}^S$. This is of particular importance for the transparent integration of analytical cost-modelling approach discussed in Chapter 6 and lazy memory expansion scheme shown in Chapter 5.

4.5.3 Adding new scheduling dimensions - loop tiling

Loop tiling or loop blocking is an extremely important loop transformation [162, 86, 2] used both for coarsening the grain of parallelism and data locality improvement [34]. An example of the tiling applied to a 2d iteration domain is shown in Figure 4.15. The new iterators `ii` and `jj` are introduced to express the tile-space, while the intra-tile iterations are constrained to the boundaries of the containing tile.

In the classical polyhedral framework, this transformation is expressed both as an iteration domain transformation (increasing the loop nest depth) and scheduling transformation(permuting the scheduling dimensions) [46, 156, 34]. But this formulation breaks the principle of transformation *composability* - the ability to express an arbitrary sequence of loop transformation through scheduling function only.

We postulate that it is possible to express the loop tiling transformations by modifying the scheduling relation θ alone. Given a timestamp variable t that corresponds to some scheduling dimension of the *dynamic* scheduling component of the scheduling matrix (Section 3.1.2), we introduce a new time variable tt, called *supernode* [86] time variable, by extending the scheduling relation with a new dimension and introducing an inequality:

8. also called *scattering relations* following CLooG [22] code generator terminology

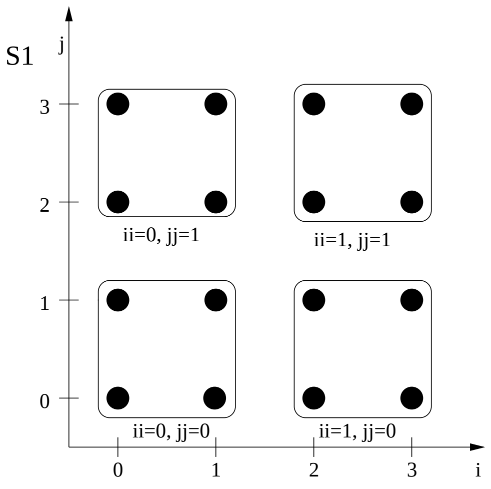


Figure 4.15 – Tiling the iteration space

$$B \cdot tt \leq t \leq B \cdot tt + (B-1)$$

where B is the compile-time [9] known *loop blocking* factor.

The new supernode time variable tt denotes the B-sized blocks of the original iteration domain, while the original time variable t is now constrained to scan the iterations within the block denoted by tt.

As an example, we will consider the representation of the *strip-mining*, which is the basic building block for implementing tiling - in order to get a loop tiling, the strip-mining has to be performed along each dimension considered for tiling. Consider a code snippet:

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

The original scheduling matrix in the canonical form could be decomposed into the following components: $\mathrm{A}^{S_1} = \begin{bmatrix} 1 \end{bmatrix}$ $\Gamma^{S_1} = \begin{bmatrix} 0 \end{bmatrix}$ $\beta^{S_1} = \begin{bmatrix} 0 \end{bmatrix}$
The concise representation of the scheduling relation is:

$$\theta^{S_1} = \left\{(t_1, t_2, t_3, i) \mid t_1 = 0 \wedge t_2 = i \wedge t_3 = 0\right\}$$

The strip-mining of the t_2 time dimension with a factor $B = 64$ is expressed as follows:

$$\theta^{S_1}_{stripmined} = \left\{(t_1, tt_2, t_2, t_3, i) \mid t_1 = 0 \wedge t_2 = i \wedge 64tt_2 \leq t_2 \leq 64tt_2 + 63 \wedge t_3 = 0\right\}$$

The corresponding code generated by CLooG [22] code generator is shown:

9. Thus, we do not support *parametric tiling* as [83] do for example.

```
for (tt2=0;tt2<=floord(N,64);tt2++) {
  for (t2=max(64*tt2,0);t2<=min(64*tt2+63,N);t2++) {
    S1(i = t2) ;
  }
}
```

The `floord` stands for the floor function of integer division. Please note that we show the syntactic code, but only for the presentation purposes. Internally in GRAPHITE, the loops are generated as basic blocks within a CFG. Please note that the original iteration domain $\mathcal{D}^{S_1}$ of the statement S_1 is not modified in any way, even though the iteration domain of the generated code is changed, but this change was expressed solely through scheduling relation.

4.6 Conclusions and future work

In this chapter we have shown a detailed design of the GRAPHITE polyhedral compilation framework which is currently a standard component of the general purpose GCC [71] compiler suite. The author is a direct contributor to this project.

Two proprietary polyhedral compilers based on a low-level internal representation are currently in development: the R-Stream compiler from Reservoir Labs [113], and IBM's polyhedral extension of its XL compiler suite [140]. But little has been published on the specific problems and trade-offs that arise when introducing the polyhedral model based compilation flow into the realm of low-level three-address code based compiler.

Our work is the first widely published contribution [149, 148, 150] on the specifics of the direct polyhedral compilation on a three-address code representation.

4.6.1 Benefits of direct manipulation of the three-address code

The traditional, source-to-source based approach to the polyhedral compilation regards the *syntactical statement* as the basic atomic unit of the computation, abstracting away the internal state of the computation and giving relevance to the loads/stores of the statement only.

On the other hand, the intrinsic nature of the three-address code is the explicit visibility of the state - the intermediate results of all the computations are visible through intermediate temporary variables.

This intrinsic property of the three-address code is the source of many difficulties in applying the polyhedral model concept, but it also gives many degrees of freedom that could be exploited. Those degrees of freedom go along two axes: the granularity of the scheduling (statement versus basic-block) and the granularity of visible state (intra versus inter basic data block dependences).

The low-level nature of the three-address code enables us to consider a precise modelling of the actual instruction costs - a property that is out of reach of the source-to-source compiler [124, 123]. We will heavily exploit this property in Chapter 6, where we present the first-of-a-kind approach to precise cost modelling of SIMD vectorization that is low-level, machine specific in nature.

We will summarize the benefits of providing the direct polyhedral compilation of the three-address code:

Additional degrees of freedom. Since the original high-level syntactic computations are broken down into the atomic three-address code instructions, there are additional degrees of freedom with respect to the scheduling granularity (which statements should be clustered together) and exposing data dependences (whether to keep the data dependence internal to the basic block, or to externalize it).

Tight interaction with a compiler. This benefit is more of a technical nature: since we are operating within the pipeline of the compiler optimization passes, all the analysis information like aliasing, pointer analysis, data alignment, induction variable analysis, interprocedural analysis is available for the use within the polyhedral transformation framework.

Precise modelling of instruction costs. The three-address code is much closer to the machine code of the target architecture than an abstract syntax tree of a source program. Also, the recognition of the intrinsics (for vectorization) and idioms [127] is readily available at the level of three-address code. The actual instruction costs are available from the backend of the compiler, since they are anyway used in the instruction scheduling phase. We will heavily rely on this property in Chapter 6.

Semantical transparency. The three-address code is expressed in SSA [52] form which captures the essential flow of the scalar data in the program, and could be regarded as the functional representation of the computation [11]. Also, the essential scalar optimizations based on the SSA form are performed before the GRAPHITE stage is reached.

This leads to a degree of semantical transparency with respect to the form of the input program : two semantically equivalent kernels would be represented in the same way in SSA form, when they reach the polyhedral framework. This is in contrast with source-to-source compilers, which are *fragile* with respect to syntactical details - if the programmer forgot to bring the program in the form expected by the syntactical parser, the program would simply be ignored.

4.6.2 Future work

The aforementioned benefits of having the polyhedral transformations within the three-address code compiler bring many new research opportunities.

We have used GRAPHITE framework to conduct a research on the iterative search and cost modeling of the profitability of the loop transformations, as explained in subsequent chapters of this dissertation.

But many new possible and interesting research opportunities emerged. Some problems were left unsolved, since their in-depth investigation was not on the main line of this thesis. We will summarize several of those research problems:

Interplay with other compiler optimizations. The GRAPHITE polyhedral framework is used to perform the loop transformations expressed as scheduling relations. But the loop transformations are just a part of the bigger picture. GRAPHITE is a part of the complete compilation pipeline consisting of more than 200 optimization passes - scalar optimizations, instruction selection and scheduling, register allocation, loop unrolling and much more. An in-depth study of the possible interactions between loop optimizations and other optimization seems very interesting, though intractable in its full generality .

Some optimizations might conflict with the polyhedral transformations. A notorious example is *loop unrolling*, that might be performed before GRAPHITE. Obviously, unrolling a loop before going into the polyhedral representation can have a dramatic effect on the scalability of the polyhedral optimizations, and it is better avoided at all.

Data layout transformations. The expressive power of the polyhedral model is used in the context of data-layout transformations [109] as well. At the same time, those transformations are studied independently, as the components of other optimization stages of the compiler [75]. Combining both might be beneficial, as the polyhedral model precisely captures the execution order, while the internal representation of the compiler has a low-level details about the data-layout. Also, the controlled memory expansion schemes like [102] are a key missing part for fully enabling the potential of automatic parallelization with GRAPHITE.

Extending the scope of the analyzable programs. GRAPHITE restricts its scope of analyzable and transformable programs to the static control programs, much in the same way as the traditional polyhedral compilation approaches. Recent works [26] on extending the applicability of the polyhedral model to non-static control programs (loop bounds and conditionals that depend on the input data) seem a promising research direction for the GRAPHITE. This is particularly true for the three-address code based compilers, because of their rich semantical analysis that could be employed.

Chapter 5

Lazy memory expansion scheme

A program transformation needs to be safe – the semantics of the original imperative program cannot be changed. In order to preserve the legality, data-dependences (Section 3.2.1) need to be analyzed. There are essentially two types of data-dependences: *data-flow* dependences and *memory-based* dependences - as discussed in Section 3.2.2. While preserving the data-flow dependences is always mandatory, spurious memory-based dependences can be removed, or simply ignored in certain conditions.

So called *true* or *dataflow* [102] dependences are imposing ordering constraints between write and read operations – they should always be preserved, since this preserves the right producer-consumer ordering, which in turn guarantees the correctness of the computations.

Memory-based dependences [1] are induced by the reuse of the same memory location to store multiple, temporary values. Spurious scalar dependences not only increase the total number of dependences in the RDDG graph, but, most importantly, they reduce the degrees of freedom available to express effective loop transformations and parallelization.

Memory-based dependences could be removed by introducing new memory locations, i.e. *expansion* of the data structures [45]. While the *expansion* approaches can remove spurious memory-based dependences, they have to be avoided whenever possible due to their detrimental impact on cache locality and memory footprint - expanding a scalar value into an array requires extra memory storage, proportional to the size of the iteration space.

Designing a polyhedral compilation framework on a three-address code exacerbates the problem of spurious memory dependences even further, since the *gimplification* [2] process introduces many temporary variables in the internal representation.

In this chapter, we show a technique that detects those memory-based dependences that could be ignored when checking for a legality of the transformation. If a memory-based dependence could not be ignored - it is *violated* in another words - it can be *removed* proposes an expansion or privatization that would enable a transformation.

Using a proposed technique, we can get rid of many memory-based dependences, either those induced by the lowering of a source program into three-address-code or those introduced by a programmer. Only if a dependence cannot be ignored, it is removed through scalar/array expansion.

1. anti and output dependences [7]
2. *gimplification* is a GCC jargon term denoting the lowering of a high-level AST into a low-level GIMPLE internal GCC representation

5.1 State of the art

The problem of eliminating the memory-based (anti and output) dependences has been studied since a long time in the automatic parallelization community [98, 120]. It is a well known fact that memory-based dependences hinder the opportunities for automatic parallelization and other loop transformations. Darte [37] gives a good overview on the removal of the anti and output data dependences.

The simple and most natural solution is to perform a memory expansion - assigning a separate memory location for each executed iteration [64]. A similar approach is that of *privatization* - assigning a private copy to each thread executing in parallel [153].

While the approach of memory expansion or privatization helps the parallelism, it has an inherent cost - an additional memory footprint or additional copy in/copy out operations. The cost of those additional overheads might be prohibitive. What is more - there is no guarantee that removing the memory-based dependences will help extract more parallelism.

In order to control the cost of the array expansion footprint, the two general approaches are:

– Perform a maximal expansion [17], apply a transformation, and then do an array contraction [56, 4, 58, 72] which minimizes the memory footprint. Approaches like [110, 102, 48] fall into this category. This approach gives the maximal degree of freedom for parallelization or loop transformation, but an array contraction phase is not always capable of optimizing the memory footprint.
– Control the memory expansion phase by imposing constraints on the scheduling. Approaches like [45], [146] fall into this category. This category of approaches tries to optimize the memory footprint, but it might restrict schedules, thus loosing optimization opportunities.

5.1.1 Our contribution

There is a trade-off between parallelization and memory usage: if we expand maximally, we will get the maximal degree of freedom for parallelization and loop transformations, but with a possibly huge memory footprint. If we choose not to expand at all, we will save memory, but our parallelization or loop transformation possibilities would be limited.

Our approach takes the lazy-expansion strategy: we do not expand memory before transformation. When checking for transformation legality, we simply ignore all memory based dependences. Only after applying a given transformation, we perform a violation analysis to check which memory based dependences might have been violated, and we propose to expand memory or to change a schedule.

By taking our approach, we are combining the best from the two mentioned state of the art approaches: we do not perform a full expansion before a transformation and we do not restrict the transformation too early.

The lazy-expansion strategy is not used to compute transformations automatically, as it is done in techniques using linear programming approach [68, 34], instead, it is used in an *iterative enumeration* of possible schedules as it is done in [150].

5.2 Motivating example

Consider a matrix multiplication numerical kernel given in a Figure 5.1. If we analyze this kernel at the source-code level, we will get the dependence graph shown in Figure 5.7. It contains both dataflow [3], and memory-based [4] dependences. Those data dependences do not prevent the rescheduling transformation that permutes the `'i'` and `'j'` loops.

3. true, a.k.a. read-after-write
4. write-after-write and write-after-read

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

Figure 5.1 – matrix multiplication kernel

```
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        {
S1:       t = 0;
          for (k = 0; k < N; k++)
          {
S2:         t += B[i][k]*C[k][j];
          }
S3:       A[i][j] = t;
        }
```

Figure 5.2 – matrix multiplication - after PRE

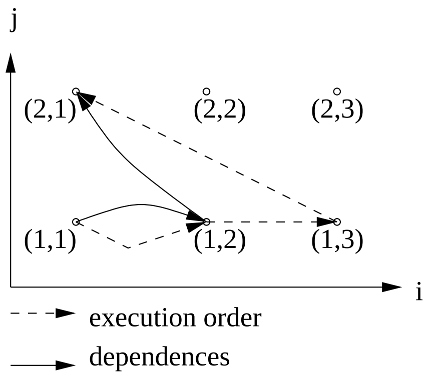


Figure 5.3 – Legal execution order

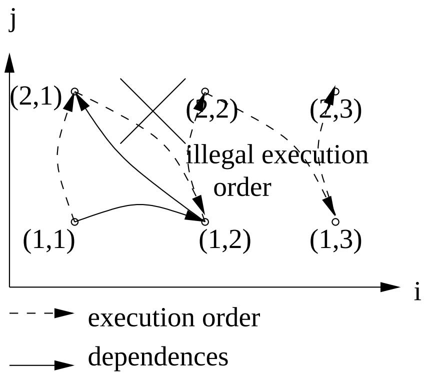


Figure 5.4 – Illegal execution order

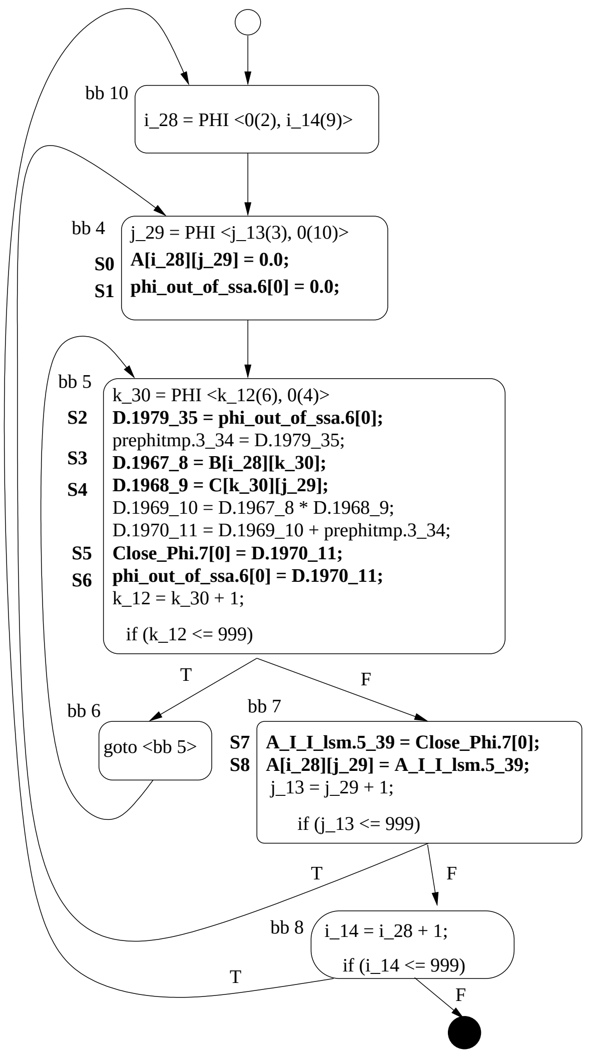
If the same kernel was written in an optimized way, shown in Figure 5.2, the dependence graph would now contain new output dependences due to the reuse of the scalar variable t for storing intermediate results.

The similar thing would happen if a compiler, such as [71], performs a lowering of the high-level source code into a three-address code internal representation. After the source code is transformed into low-level form, the compiler performs many optimization passes. One of those passes is PRE [114] (Partial Redundancy Elimination) which does the following scalar optimization: instead of accumulating a values into an array, it initializes a scalar value, accumulates values into that scalar and then stores the scalar into an array element. Conceptually, the idea is the same as if the user has rewritten the code as shown in Figure 5.2. An equivalent three-address code seen in GCC compiler is shown in Figure 5.5.

A data dependence graph corresponding to code in Figure 5.5 is shown in Figure 5.6. After introducing a scalar into the loop, a new write-after-write dependence on statement S_1 has been introduced: $\delta^{WAW}_{S_1 \to S_1}$. This dependence stems from the fact that the same temporary scalar value is overwritten in each iteration of the containing loop.

This output dependence has to be respected, which forces the sequential execution in the original scheduling order. Figure 5.3 shows that if we execute the code in a sequential manner, according to the original loop nesting (loop i as outermost, loop j as innermost), then the dependences would be preserved. If we try to interchange loops i and j, we would invert a dependence constraint, thus violating the write-after-write dependence on the scalar t. This is shown in Figure 5.4.

But an intuition tells us that it is legal to interchange loops i and j and still have a correct output code. An essential observation is that some memory based dependences (write-after-write and write-after-read) could be ignored when performing some transformations. But how do we determine when it is safe to



```
bb 10:
i_28 = PHI <0(2), i_14(9)>

bb 4:
j_29 = PHI <j_13(3), 0(10)>
S0  A[i_28][j_29] = 0.0;
S1  phi_out_of_ssa.6[0] = 0.0;

bb 5:
k_30 = PHI <k_12(6), 0(4)>
S2  D.1979_35 = phi_out_of_ssa.6[0];
prephitmp.3_34 = D.1979_35;
S3  D.1967_8 = B[i_28][k_30];
S4  D.1968_9 = C[k_30][j_29];
D.1969_10 = D.1967_8 * D.1968_9;
D.1970_11 = D.1969_10 + prephitmp.3_34;
S5  Close_Phi.7[0] = D.1970_11;
S6  phi_out_of_ssa.6[0] = D.1970_11;
k_12 = k_30 + 1;
  if (k_12 <= 999)

bb 6:
goto <bb 5>

bb 7:
S7  A_I_I_lsm.5_39 = Close_Phi.7[0];
S8  A[i_28][j_29] = A_I_I_lsm.5_39;
 j_13 = j_29 + 1;
  if (j_13 <= 999)

bb 8:
i_14 = i_28 + 1;
  if (i_14 <= 999)
```

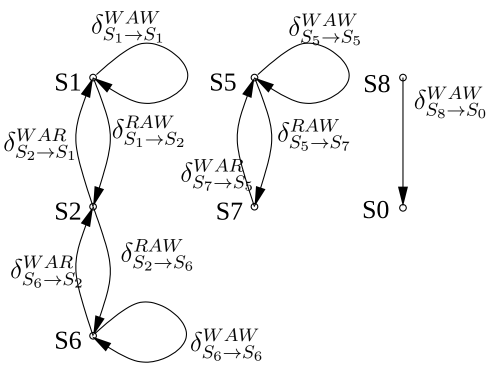

$\delta^{WAW}_{S_1\to S_1}$
$\delta^{WAW}_{S_5\to S_5}$
S1
S5
S8
$\delta^{WAW}_{S_8\to S_0}$
$\delta^{WAR}_{S_2\to S_1}$
$\delta^{RAW}_{S_1\to S_2}$
$\delta^{RAW}_{S_5\to S_7}$
$\delta^{WAR}_{S_7\to S_5}$
S2
S7
S0
$\delta^{RAW}_{S_2\to S_6}$
$\delta^{WAR}_{S_6\to S_2}$
S6
$\delta^{WAW}_{S_6\to S_6}$

Figure 5.6 – Data Dependence Graph

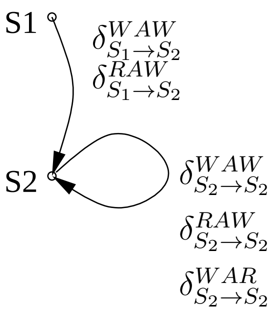

S1
$\delta^{WAW}_{S_1\to S_2}$
$\delta^{RAW}_{S_1\to S_2}$
S2
$\delta^{WAW}_{S_2\to S_2}$
$\delta^{RAW}_{S_2\to S_2}$
$\delta^{WAR}_{S_2\to S_2}$

Figure 5.5 – GIMPLE code with CFG

Figure 5.7 – Matmult Data Dependence Graph

ignore some dependences?

On the other hand, let us consider an example where an user has written a code in Figure 5.2 manually and provided it to the compiler. As an example let us consider the legality of loop distribution transformation shown in Figure 5.8. This transformation would not be legal, and an output dependence on the scalar `t` could not be ignored. Though, an expansion of the scalar `t` could be performed, as shown in Figure 5.9. Expansion removes an output dependence on the scalar `t` and allows the distribution transformation to be legal. But an expansion introduces an additional cost - that of transforming the scalar `t` into an array `t[N][N]`). This is a situation that should be rather avoided.

In the following section we will show how to formally prove which memory-based dependences, given a desired transformation, could be ignored (as in the example of loop interchange) and which must be removed by a memory expansion.

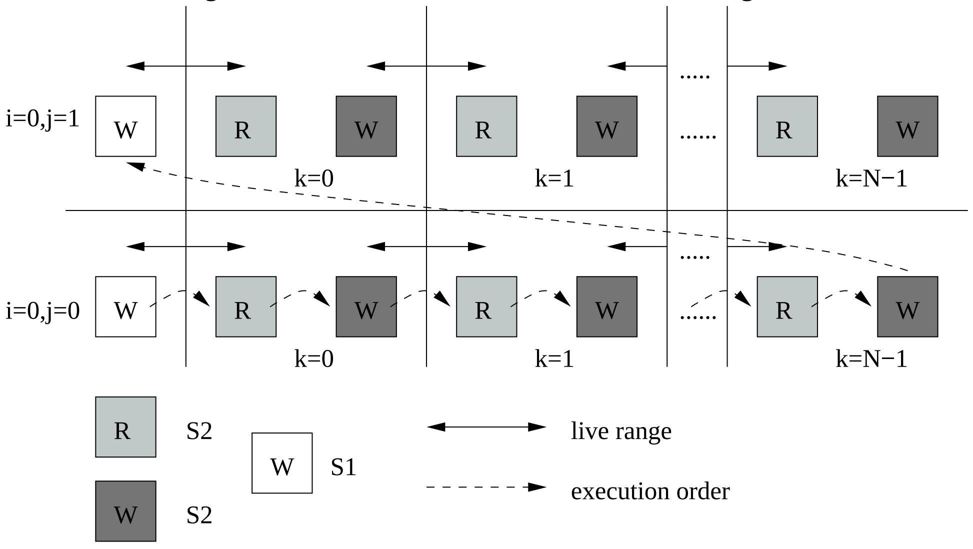
```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    t = 0;
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    {
      for (k = 0; k < N; k++)
      {
        t += B[i][k]*C[k][j];
      }
      A[i][j] = t;
    }
```

Figure 5.8 – illegal - loop distribution without an expansion

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

Figure 5.9 – legal - loop distribution with an expansion

Figure 5.10 – Read/Write instruction interleaving and variable live ranges



5.3 Framework

We will first define the concept of *live range* and we will show the way of computing the closed form expression that summarizes the set of live ranges. The core of the approach will be shown in subsection 5.3.2, where we discuss a computational procedure for checking the potential *violation* of value live range instances: if a value live range instance of some memory location *M* is violated, then the memory based dependence induced by the reuse of the memory location cannot be ignored. A detailed example is shown in subsection 5.3.3.

5.3.1 Live ranges

An execution trace of a sequential program can be seen as an interleaving of read and write instructions. During an execution of a loop, values are produced and stored into memory locations. Those values are then read by subsequent instructions. The value stored in a memory location is *live* until its last read, before it is destroyed by a subsequent write to the same memory location.

We are interested in formally analyzing and modelling value *live ranges* inside loops, using the polyhedral model. Given a code in Figure 5.2 we can represent the execution trace and instances of live ranges as shown graphically in Figure 5.10. A similar concept, named *utility span of a value*, was introduced in [102].

Consider a value V written into a memory cell $M[b]$ by an instruction instance $w = (S_W, \mathbf{i_W})$. We can compute a set of instruction instances $R = \{(S_R, \mathbf{i_R})\}$ such that there is a direct data-flow of value V from instance w to instances in set R.

Each instance of a value live range is a tuple, describing a write and read instruction instances. Let us consider a set of value live range tuples:

$$L = \{ < (S_W, \mathbf{i_W}), (S_R, \mathbf{i_R}) > \}$$

We want to have a closed form expression that summarizes all instances in this set. We can decompose the set L into a finite number of convex polyhedra, where each polyhedron describes live range instances for a pair of statements:

$$\lambda^{S_W \rightarrow S_R} = \{ (\mathbf{i_W}, \mathbf{i_R}) \mid \Lambda \times (\mathbf{i_W}, \mathbf{i_R}, \mathbf{g}, 1)^T \geq \mathbf{0} \}$$

Each convex polyhedron $\lambda^{S_W \rightarrow S_R}$ represents instances of statements that form a definition/use pairs. This polyhedron is constructed by enforcing the following conditions:

conflict condition - write and read statement instances refer to the same memory location:

$$\mathrm{F}_{S_W}(\mathbf{i_W}) = \mathrm{F}_{S_R}(\mathbf{i_R})$$

causality condition - a read instruction is scheduled after a write instruction:

$$\theta^{S_w}(\mathbf{i_W}) \prec \theta^{S_r}(\mathbf{i}))$$

liveness condition - there is no intervening write $w_k = (S_{KW}, \mathbf{i_{KW}})$ happening between $(S_{LW}, \mathbf{i_{LW}})$ and $(S_R, \mathbf{i_{LR}})$ instruction instances.

Described polyhedra satisfies exactly the same conditions that an array dataflow analysis (Section 3.2.3) requires. It is computed by the array dataflow algorithm of Feautrier, detailed in [66]. Given a memory location M, all the read accesses and their *source functions* (Section 3.2.3) are combined into the set L. A practical implementation of this computation is given in [148].

5.3.2 Live range violation analysis

After applying a transformation, the relative execution order of statement instances might change. This change might cause live ranges described in the set L, mapped to the same memory location, to *interfere*.

The classical dependence analysis theory states that a source instruction instance must execute before a sink instruction instance: $\theta'_{S_W}(\mathbf{i_W}) \prec \theta'_{S_R}(\mathbf{i_R})$. Once dataflow dependences are computed, we use a simple dependence violation analysis [156] to check this condition on all dataflow dependences.

Since we propose to *ignore* the memory based dependences, how do we assure that a transformation would not reschedule some statement instance, say $(S_{KW}, \mathbf{i_{KW}})$, to overwrite a value V, produced at instance $(S_W, \mathbf{i_W})$, before it is read by a statement instance $< S_R, \mathbf{i_{LR}} >$, as it was expected in the original program?

In other words: we have to check that such a case may not happen after a transformation. We form the following system of equations, modelled using a polyhedral model:

$$Vio^{S_W \to S_{KW} \to S_R} = \left\{ \begin{array}{l} (\mathbf{i_W}, \mathbf{i_R}) \in \lambda^{S_W \to S_R} : \theta'^{S_W}(\mathbf{i_W}) \prec \theta'^{S_{KW}}(\mathbf{i_{KW}}) \prec \theta'^{S_R}(\mathbf{i_R}) \wedge \\ \mathrm{F}_{S_W}(\mathbf{i_W}) = \mathrm{F}_{S_R}(\mathbf{i_R}) \end{array} \right\}$$

New schedules θ' represent a polyhedral transformation that we check for legality. If a violation set $Vio^{S_W \to S_{KW} \to S_R}$ is empty, there is no violation after a transformation. We have to perform this check for each dataflow dependence polyhedron $\delta^{S_j \to S_i}$ and for each possible write statement S_{KW}.

Legality of loop parallelization: Previously mentioned check takes into an account the fact that that schedule functions θ' are describing a sequential execution order. Sequential execution order has a property that no two different statement instances could be scheduled at the same time: so either $\theta'_{S_i}(\mathbf{i}) \prec \theta'_{S_j}(\mathbf{j})$ or $\theta'_{S_j}(\mathbf{j}) \prec \theta'_{S_i}(\mathbf{i})$.

If we consider loop parallelization transformation, we need to take into an account that some statement instances might be executed simultaneously $\theta'_{S_i}(\mathbf{i}) = \theta'_{S_j}(\mathbf{j})$. Thus, value live range legality violation check used for sequential transformations could not be used.

Given a loop at level k that we consider for parallelization[5] , we proceed as follows: we check that there are no *loop carried* [7] dataflow dependences at level k. Then, we check that no two distinct statement instances $\theta'^{S_i}(\mathbf{i})$ and $\theta'^{S_j}(\mathbf{j})$ sharing the same prefix up to depth $k-1$ write to the same memory location (instances associated with the same iteration of the parallel loop but different iterations of some inner loops can still write to the same memory location).

Supporting array/scalar expansion: Our approach is compatible with well known array/scalar expansion approaches. If a transformation produces at least one violated live range (the set *Vio* is not empty) then we can choose to expand the variable M whose live ranges are violated. A precise characterization of violated live range instances in a set *Vio* could be used to drive the needed degree of expansion as in [102]. If we do not want to perform any expansion, we can iteratively choose a new schedule that does not violate any live ranges.

Supporting privatization: Privatization is a common concept in the loop parallelization community. We can use our framework to automatically detect which scalars/arrays need to be privatized to enable loop parallelization transformation. This is actually how GRAPHITE is driving *autopar* – a GCC loop parallelizer.

5.3.3 An example

Let us take the GIMPLE code from Figure 5.5 and consider a memory location `phi_out_of_ssa`. Figure 5.10 shows an interleaving of writes and reads to this memory location. A slice of the execution trace, for a finite number of iterations, is shown. Value live ranges are shown as well. Some value live range instances contained in a set $L_{\texttt{phi\_out\_of\_ssa}}$:

$$\begin{array}{l} <(S_1,(0,0)),(S_2,(0,0,0))> \\ <(S_6,(0,0,0)),(S_2,(0,0,1))> \\ <(S_6,(0,0,1)),(S_2,(0,0,2))> \\ \ldots \\ <(S_6,(0,0,N-2)),(S_2,(0,0,N-1))> \\ <(S_1,(0,1)),(S_2,(0,1,0))> \\ \ldots \end{array}$$

5. A parallelized loop is a loop whose iterations could be executed by independent threads, as in a DOALL [7] loops.

After array dataflow analysis, we come up with two closed form expressions: $\lambda^{S_1 \to S_2}$ and $\lambda^{S_6 \to S_2}$. These two polyhedra summarize value live range instances between statements S_1 and S_2, and between S_6 and S_2 respectively. They have the following form:

$$\lambda^{S_1 \to S_2} = \{< (i,j),(i',j',k') >: i' = i \wedge j' = j \wedge k' = 0 \wedge 0 \leq i < N \wedge 0 \leq j < N\}$$

$$\lambda^{S_6 \to S_2} = \left\{ \begin{array}{l} < (i,j,k),(i',j',k') >: i' = i \wedge j' = j \wedge k' = k+1 \wedge \\ 0 \leq i < N \wedge 0 \leq j < N \wedge 0 \leq k < N-1 \end{array} \right\}$$

For the purpose of this example, we would like to check whether interchanging loops `i` and `j` is a transformation that preserves non-conflicting condition on all live range instances. Referring again to Figure 5.5, we see that we are interested in the schedule of statements S_1, S_2 and S_6. Their scheduling functions in an original program are as follows:
$\theta^{S_1}(i,j)^T = (0,i,0,j,0)^T$
$\theta^{S_2}(i,j,k)^T = (0,i,0,j,1,k,1)^T$
$\theta^{S_6}(i,j,k)^T = (0,i,0,j,1,k,8)^T$
If we perform a loop interchange transformation, we will get the following transformed scheduling functions:
$\theta'^{S_1}(i,j)^T = (0,j,0,i,0)^T$
$\theta'^{S_2}(i,j,k)^T = (0,j,0,i,1,k,1)^T$
$\theta'^{S_6}(i,j,k)^T = (0,j,0,i,1,k,8)^T$
By applying the sequential version of the violation check, we find that the violation sets are empty:
$Vio^{S_1 \to S_1 \to S_2} = \emptyset$
$Vio^{S_1 \to S_6 \to S_2} = \emptyset$
$Vio^{S_6 \to S_1 \to S_2} = \emptyset$
$Vio^{S_6 \to S_6 \to S_2} = \emptyset$
proving that there are no violations of live range intervals, after an interchange transformation. The conclusion is that the output dependences induced by the scalar variable `phi_out_of_ssa.6[0]` (shown as $\delta^{WAW}_{S_1 \to S_1}$ and $\delta^{WAW}_{S_6 \to S_6}$ in Figure 5.6), could be *safely* ignored when checking the legality of the transformation.

This check has to be performed for other memory accesses as well. In addition, we perform a scheduling violation analysis (Section 3.3) on true dataflow (read-after-write) dependences only: $\delta^{S_1 \to S_2}$ and $\delta^{S_6 \to S_2}$, to check that relative writes/reads are executed in a correct order.

A classical dependence violation analysis [156], that takes into an account the full dependence graph, would consider this transformation to be illegal, thus precluding (obviously) legal interchange.

5.4 Performance benefits

The original motivation for coming up with the lazy expansion scheme was the fact that a three-address-code internal representation of the compiler that we were developing (Chapter 4) inherently contains many scalar variables.

It is a known fact that memory based write-after-write dependences on scalar variables hamper any effort on automatic parallelization. Indeed, all the codes that contain a reduction operation, like the matrix matrix multiplication shown in Figure 5.1, are internally transformed into the form where a scalar value is used to accumulate the values. Also, if an user writes such a code, such as the version of the matmult shown in Figure 5.2 the parallelization is not possible, since there is a loop carried output dependence [7].

Benchmarks	**Classical**		**Lazy**	
	outer	inner	outer	inner
2mm.c	7	0	0	5
3mm.c	10	0	0	7
atax.c	1	2	0	3
bicg.c	1	1	0	2
gemm.c	4	0	0	3
gemver.c	4	1	0	3
gesummv.c	2	0	0	1
gramschmidt.c	3	1	0	4
jacobi-2d-imper.c	3	0	1	1
ludcmp.c	1	0	0	1
seidel.c	1	0	0	1

Table 5.1 – The number of the parallelized loops for Polybench [84] suite

Our scheme enables the *transparent* handling of all the non-harmful output dependences, also in the case of loop parallelization. We ran measurements on a suite of computationally intensive numerical kernels [84]. We compared the parallelizability of the kernels using both dependence models: the classical, all data-dependences analysis, and our approach of combined array data-flow analysis with live range analysis.

After integrating the new approach into the GRAPHITE polyhedral framework of the GCC compiler, we found that the new approach allowed for more loops – and mostly importantly, more outer loops – to get parallelized, as depicted in Table 5.1. This translates naturally into the speedups obtained by running those benchmarks on chip multiprocessor 5.11.

The baseline is the *w/o dataflow* column, standing for a sequential run when the current transformation violation analysis is used. The *dataflow* column represents a sequential run when the new, lazy live range violation analysis is used. As expected, changing the analysis doesn't influence the runtime of sequential runs, since we perform no sequential code transformation, which is why these columns show the same runtimes for all kernels.

Enabling parallelization on top of the current dependence analysis produces only a mild speedup on *jacobi-2d-imper*, while it does not impact or sometimes even slows down the others. This is mainly due to the low speedup potential of *inner loop* parallelization.

When autoparallelization is enabled with the new data-flow and live range analysis, we achieve substantial speedups of up to 3.8x. We experience one degradation in *atax* kernel, which we relate to the overhead incurred by autopar, unrelated to which data dependence model we use, since it happens for both approaches.

These are only the preliminary results, showing the potential of lazy memory expansion scheme combined with privatization and automatic parallelization. The integration of the GRAPHITE framework with automatic parallelization phase of the GCC compiler was discussed thoroughly in Chapter 4.

5.5 Summary

We have shown how a value live range analysis can be used to enable a *lazy* memory expansion scheme – memory is expanded only if absolutely necessary for the correctness of the transformation.

Polyhedral compilation traditionally takes as an input a dependence graph with all dependences, constructs a legal transformation and generates code.

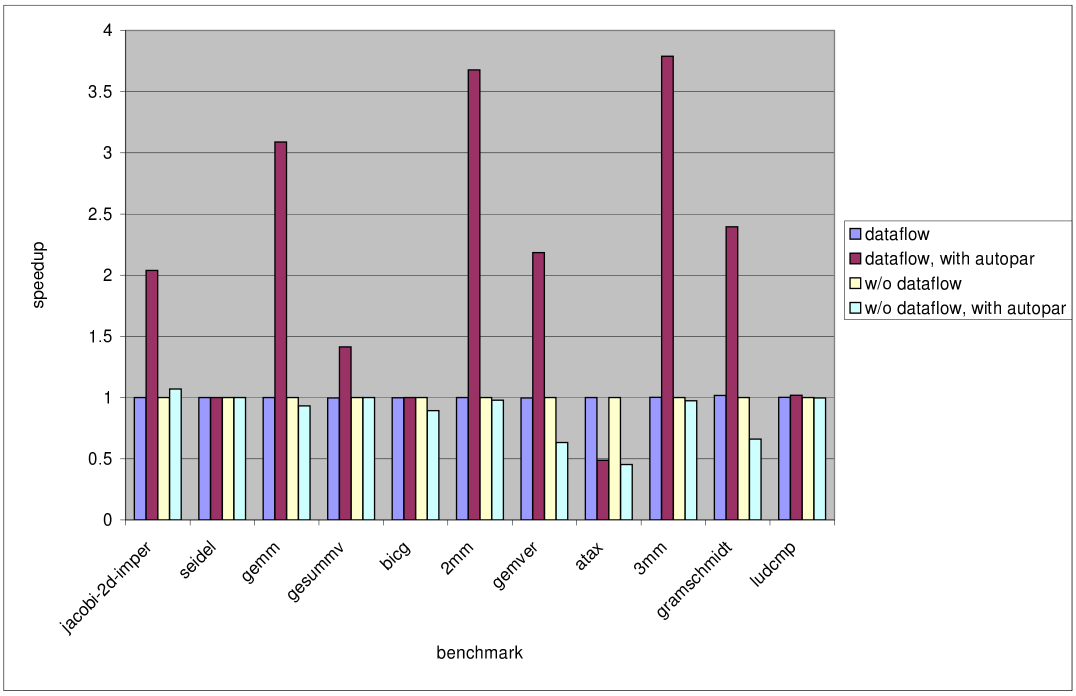


Figure 5.11 – Speedups on Power platform.

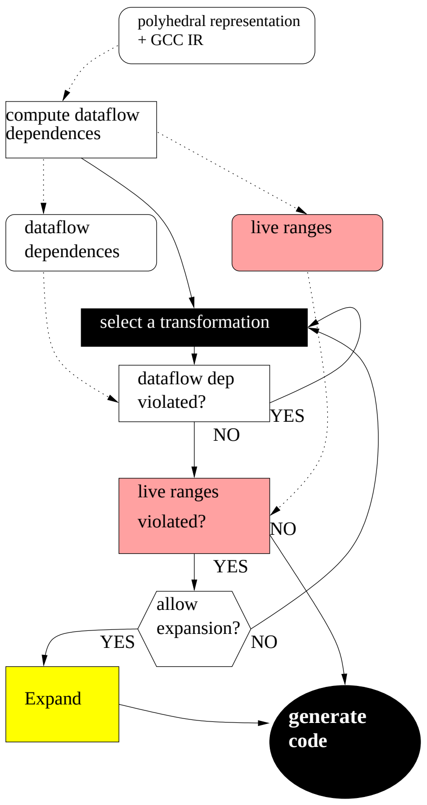


Figure 5.12 – A flow of polyhedral compilation based on violation analysis

The other class of so called *violation analysis* based compilation flow [157] takes as an input a dependence graph with all dependences as well, it constructs a transformation (without legality check), and only then it checks whether the transformation is legal. If it is, then it proceeds with code generation. If it is not, then it reiterates and proposes a next transformation until it finds a legal one.

We take the violation analysis approach a **step further**: we do not take into the account all the dependences. We split the dependence graph into those dependences that are true *data-flow* dependences and those that are *memory based*. We do not check memory-based dependences for scheduling violation violation.

By not taking the memory-based dependences into an account for scheduling violation analysis we cannot guarantee that original values written into specific memory location are not destroyed before they could be read by an appropriate instruction. In order to maintain this property, we first define *live ranges* for all values generated in an original program. Then we check whether a transformation would respect those value live ranges by performing the *live range violation* analysis. If we prove that all value live ranges are still respected, then the transformation is legal.

If a transformation destroys value live range set for a memory location, it is still possible to correct this: we could choose to replicate (memory expand) those memory locations. This leads to a controllable memory expansion scheme - we only expand when there is a need for expansion. A complete scheme of our approach is shown in Figure 5.12.

5.6 Conclusions and perspectives

We presented an algorithmic framework to maximize the effectiveness of loop transformations while maintaining per-hardware-thread memory usage invariant. Unlike traditional array expansion and contraction techniques, dependence removal can be implemented with guaranteed memory footprint constraints. This framework is a promising trade-off between array expansion and the the degrees of freedom for affine transformation.

In the context of a low-level three-address-code polyhedral compilation solving this problem is even more urgent, as evidenced in the experimental evaluation section 5.4. The general compilation flow that includes the lazy memory expansion scheme, presented in 5.5, fits well into the compilation flow of GRAPHITE framework (Chapter 4) and a transformation search method shown in Chapter 7.

Part III

Towards a search strategy

Chapter 6

Analytical cost model

The critical aspect of modern architectures is the need to efficiently coordinate the exploitation of multiple forms of parallelism provided by platforms, while carefully coordinating the use of the memory hierarchy efficiently.

Systematic solutions to harness the interplay of multi-level parallelism and locality are emerging, by advances in automatic parallelization and loop nest optimization [34, 128]. These rely on the polyhedral model of compilation to facilitate efficient exploration and application of very complex transformation sequences. However, the analytical cost models employed by those solutions are not able to model the low-level, machine-dependent aspects of the target architectures.

One such an aspect is exploiting short SIMD parallelism by vectorization [118]. Naturally, such a kind of vectorization has not been represented within the polyhedral model due to its low-level, machine-dependent nature. As a result, there remains a gap in providing a combined framework for exploring complex loop transformation sequences together with vectorization.

The problem of providing the analytical cost-models for predicting the performance is deemed difficult [129], since it seems infeasible to provide that *static* analytical model that could predict the performance of the architectures that are ever-growing in their complexity.

An alternative solution, based on an iterative compilation, has demonstrated the superior performance over the static compilation [1, 8]. Nevertheless, the major drawback of the iterative compilation schemes is their running time, precluding them from being incorporated into the general-purpose, single-pass compilers such as Intel ICC, IBM XL or GCC [71].

Our goal is to provide a hybrid approach: we provide a precise and parameterizable analytical performance cost-predictor and we use it within a transformation *search loop* within the polyhedral model framework.

We present the cost-model function that is *sensitive* to the the actual loop transformation expressed as a schedule (Chapter 3). By exploring a finite and restricted space of schedules (as explained in Section 6.3.1), the cost-model function is evaluated for different loop transformations. In this chapter we present a simple scheme for the search space construction, while the much more expressive search space is presented in Chapter 7.

This search based cost-model function evaluation scheme still incurs some overhead in terms of the compilation time, since the search space has to be constructed and traversed. But as we show, this space is reduced to a reasonable size, and an evaluation of the cost-model function for each point in the search space is effective.

We achieve this effectiveness by enumeratively evaluating the non-linear cost function for different loop transformations, without any code generation nor running the result code, contrary to a feedback directed iterative search [129]. Only after the best schedule is determined, the code is generated *only once* and the final, optimized executable is obtained. This scheme is illustrated in Figure 6.3.

The most recent work of Park et al. [121] is aiming at solving the similar problem: constructing a predictive cost-model that drives the selection of the compiler transformations without resorting to the iterative search during the compilation time. But the fundamental difference, compared to our approach, is that they use the machine learning technique to (still, iteratively) *learn* the optimization sequences for the *selected* suite of benchmarks, based on the premise that the learned sequences are good predictors for the rest of programs. Our model is not based on learning, but *analyzing* the behavior of the program in order to predict its performance.

We are not trying to build a cost-model function that models all the possible aspects of the target architecture. We focus ourselves on the aspect of *SIMD vectorization*, which so far has not been considered in the polyhedral model. Yet this is a low-level and highly target specific optimization that has to be modelled precisely in order to yield the best results. We will give an overview of the SIMD vectorization in Subsection 6.1.1.

In a summary: in this chapter we present a precise analytical, performance predicting cost-model that captures the low-level details of the target architecture in the polyhedral model. In particular, we help bridge the aforementioned gap between analytical and feedback directed methods by incorporating the low-level (vectorization) considerations into a polyhedral model.

A note to the reader: the term *cost-model* is used to refer to the analytical function used to predict the total execution cost of a given program with applied transformation. The term *polyhedral model* is used to refer to the complete program analysis framework, as explained in Part I. The actual cost-model function is implemented within the polyhedral model framework.

6.1 Motivation

Fine-grain data level parallelism is one of the most effective ways to achieve scalable performance of numerical computations. This effectiveness translates into area and power savings for processor architectures, and complexity and performance benefits for applications. These are well known advantages of SIMD over MIMD architectures and programming models, and are best illustrated by the computational density of graphical processing units (GPUs). In this chapter we concentrate on improving the exploitation of short-SIMD parallelism available in modern instruction sets and vector-processing extensions (including Altivec, Cell SPU, and SSE). We will hereafter refer to such fine-grain data parallelism as *subword* parallelism.

6.1.1 SIMD Vectorization

Automatic vectorization for modern short-SIMD instruction sets, such as Altivec, Cell SPU and SSE, has been a popular topic, with successful impact on production compilers [164, 29, 30, 117]. Exploiting subword parallelism in modern SIMD architectures, unlike automatic vectorization for traditional vector computers [7], however suffers from several limitations and overheads [118, 116, 115] (involving alignment, redundant loads and stores, support for reductions and more) which complicate the optimization dramatically.

Automatic vectorization was also extended to handle more sophisticated control-flow restructuring including if-conversion [145] and outer-loop vectorization [118]. Classical techniques of loop distribution and loop interchange [7] can dramatically impact the profitability of vectorization. To be successful, it is vital to avoid inapt strategies that incur severe overheads, which are quite common in complex memory access patterns.

```
     for (v = 0; v < N; v++)
       for (h = 0; h < N; h++) {
S1:      s = 0;
         for (i = 0; i < K; i++)
           for (j = 0; j < K; j++)
S2:          s += image[v+i][h+j] * filter[i][j];
S3:      out[v][h] = s >> factor;}
```

Figure 6.1 – Main loop kernel in Convolve

6.1.2 Motivating Example

Loop vectorization can adversely affect performance, mainly due to data management overheads. The first and foremost goal of a vectorization cost-model is to avoid performance degradations while not missing out on improvement opportunities. In addition, a cost model should also drive the selection of a vectorization strategy, assuming there exist a profitable one. The problem is particularly acute on modern subword SIMD architectures and compilers due to the interplay among compilation passes, the interactions among micro-architecture components, and the dependence of run-time information (array alignment, loop trip count).

Given a loop-nest, a compiler needs to choose which loop to vectorize, and at which position, employing one of several strategies (innermost- or outer-loop vectorization, in-place or based on innermosting, as explained below). This in-turn brings to play other loop-transformations, most notably loop-interchange but also loop-peeling and others. Searching among all these alternatives becomes a non-trivial problem. This problem is especially apparent in computations featuring loop nests that are (1) deep, (2) can be vectorized in several ways, (3) are amenable to other loop optimizations, and (4) are sensitive to underlying architectural features (e.g. alignment handling mechanisms, sensitivity to data locality).

Figure 6.1 introduces the *Convolve* kernel – a simple example of a loop nest exhibiting the above features. *Convolve* performs a 2D-convolution on a NxN 16-bit pixel image. Such convolutions are vastly used in DSP programs for various applications, from edge-detection using 3x3 filters (as in the UTDSP suite [101]), to 17x17 polyphase filters. [1]

There are d possible vectorization alternatives for a loop-nest of depth d (in our case $d = 4$), without involving any other loop-transformation: we can vectorize any of the j, i, h or v loops "in-place" – i.e. in their original position (loop-level). For instance, we can vectorize the j-loop in its innermost position (as shown in Figure 6.2a), which is the common practice of vectorizing compilers. Employing loop-interchange to permute one loop (inwards or outwards) into a specific position within the loop nest and vectorizing it there (keeping the other loops intact), increases the search-space to $d \cdot d$ possibilities. Figures 6.2b and 6.2c show examples of vectorizing the h-loop after permuting it to the innermost and next-to-innermost positions, respectively. Using loop-permutation more aggressively to reorder the loops of a nest according to a specific permutation, and then vectorizing one of them, results in a total of $d(d!)$ combinations. If we also employ loop peeling to align memory accesses, the search space grows to $d(d!)VF$ where VF is the Vectorization Factor (number of elements operated upon in parallel in a vector). In our case this amounts to 768 alternatives.

The search space becomes quite large even for modest depths and only few transformations (interchange and peeling), as shown, and can easily reach much higher volumes if deeper loop nests and/or more loop-transformations are considered. Also note that programs often contain many loop-nests, where

1. For this illustration, we use N=128 and K=16, due to current implementation restrictions.

```
for (v = 0; v < N; v++)
  for (h = 0; h < N; h++) {
    s = 0;
    for (i = 0; i < K; i++) {
      vs[0:7] = {0,0,...,0};
      for (vj = 0; vj < K; vj+=8) {
        vs[0:7] +=
          image[v+i][h+vj:h+vj+7]
          ∗ filter[i][vj:vj+7];
      }
      s += sum(vs[0:7]);
    }
    out[v][h] = s >> factor;
  }
```

(a) j-loop vectorized at level 4

```
for (v = 0; v < N; v++)
  for (h = 0; h < N; h++)
    out[v][h] = 0;
for (v = 0; v < N; v++)
  for (i = 0; i < K; i++) {
    for (j = 0; j < K; j++) {
      c = filter[i][j];
      vfilter[0:7] = {c,c,...,c};
      for (vh = 0; vh < N; vh+=8) {
        out[v][vh:vh+7] += vfilter[0:7]
          ∗ image[v+i][vh+j:vh+7+j];
      }
    }
  }
for (v = 0; v < N; v++)
  for (h = 0; h < N; h++)
    out[v][h] = out[v][h] >> factor;
```

(b) h-loop vectorized at level 4

```
for (v = 0; v < N; v++)
  for (h = 0; h < N; h++)
    out[v][h] = 0;
for (v = 0; v < N; v++)
  for (i = 0; i < K; i++) {
    for (vh = 0; vh < K; vh+=8) {
      vs[0:7] = {0,0,...,0};
      for (j = 0; j < K; j++) {
        c = filter[i][j];
        vfilter[0:7] = {c,c,...,c};
        vs[0:7] += vfilter[0:7]
          ∗ image[v+i][vh+j:vh+j+7]
      }
      out[v][vh:vh+7] += vs[0:7];
    }
  }
for (v = 0; v < N; v++)
  for (h = 0; h < N; h++)
    out[v][h] = out[v][h] >> factor;
```

(c) h-loop vectorized at level 3

Figure 6.2 – Convolve Vectorization Examples

each loop-nest should be optimized.

Approaches that generate each alternative and rely on its (possibly simulated) execution or on performance evaluation at a later low-level compilation stage, are competitive in terms of accuracy but are significantly inferior in terms of scalability to analytical approaches that reason about costs and benefits without actually carrying out the different loop transformations beforehand. Operating on the polyhedral representation itself, rather than relying on code generation, is therefore a key ingredient. Hybrid approaches can provide a more practical solution by combining the feedback-based approach with classical analytical models to narrow the search space. The modest compile-time requirements of our purely analytical approach (about 0.01s to build the model and search for the optimal vectorization strategy for *Convolve*) facilitates its integration in a production compiler.

Having potentially very large search-spaces is only one aspect of the vectorization decision-making problem. A complementary challenge to dealing with the very large search-spaces, is how to evaluate the costs and benefits associated with each alternative efficiently and accurately. Some trade-offs are clearly visible in Figure 6.2. For example, variants (b,c) use loop-permutation, which in this case incurs an overhead of extra memory traffic to/from the *out* array. On the other hand variant (a) incurs a reduction epilogue overhead (see *sum* operation) in each iteration of the *i*-loop. Outer-loop vectorization (vectorizing a loop other that innermost-loop) is used in (c), implying that more code is vectorized. The innermost j-loop in this case continues to advance sequentially, operating simultaneously on values from $VF = 8$ consecutive *h*-loop iterations. On the other hand (b) has better temporal locality ($filter[i][j]$ is invariant in the innermost loop) and the misalignment is fixed (this is explained in more detail later). Overall the speedup factors obtained by transformations a, b, c on PPC970 (relative to the original sequential version shown in Figure 6.1) are 2.99, 3.94, 3.08 respectively. On the Cell SPU the respective speedups are 2.59, 1.44, 3.62.

The following sections describe our approach and demonstrate how our cost model computes its predictions within the analytical polyhedral-based model, considering different loop transformations and metrics. Final cost-model predictions for *Convolve* and analysis of the speedups are given in Section 6.3.3, where we show that the cost model is able to correctly predict the best vectorization option for both PPC and SPU.

6.2 Polyhedral Modelling of Vectorization Metrics

Several key costs impact the expected performance of vectorized code, including: strides of accesses to memory, memory access alignment, loop trip counts, reduction operations across loop iterations and more. These factors depend on the modified scheduling $\theta^{S'}$ and on the modified iteration domain $\mathcal{D}'^{S}$ of each statement.

The underlying assumption of vectorization is that the kernel of a loop usually executes faster if vectorized than if not, but that associated overheads may hinder the vectorized version, diminishing its speedup compared to the original scalar version, and more so for loops that iterate a small number of times. Indeed, if the number N of iterations of a loop is smaller than its *vectorization factor* VF, there is no potential for speeding it up using vectorization; on the contrary, vectorizing such a loop may only slow down its execution due to additional preparatory actions and checks. Furthermore, even if N is larger than VF, the number of iterations of the vectorized loop, although positive, may not suffice to out-weigh the overheads incurred by vectorization.

6.2.1 Modelling the Access Patterns

Recall from Section 2.3.3 that in the classical polyhedral framework memory access functions for array references are represented as a vector of affine expressions:

$$f(\mathbf{i}) = \mathrm{F} \times (\mathbf{i}, \mathbf{g}, 1)^T$$

but there is no notion of the data layout of an array.

One may combine this access function with the data layout of the array. For each array reference, one may form a *linearized memory access function* ℓ, capturing the stream of memory access addresses as a function of the iteration vector:

$$\ell(\mathbf{i}) = b + (\mathrm{L^i}|\mathrm{L^g}|\omega) \times (\mathbf{i}, \mathbf{g}, 1) = b + \mathrm{L^i}\mathbf{i} + \mathrm{L^g}\mathbf{g} + \omega \tag{6.1}$$

where b is the base address[2] of the array and $(\mathrm{L^i}|\mathrm{L^g}|\omega)$ is the row vector of coefficients that encodes the layout information (assuming row-major data layout). This vector is composed of three parts: $\mathrm{L^i}$ is scheduling-dependent, $\mathrm{L^g}$ depends on global parameters, and ω is the constant offset part.

Assuming that matrix `M` is defined as $\mathtt{M}[r_1][r_2]\dots[r_m]$, we can construct the vector R encoding the strides along each subscript:

$$\mathrm{R} = \left(\prod_{i=1}^{m-1} r_i, \prod_{i=2}^{m-1} r_i, \dots, \prod_{i=m-1}^{m-1} r_i, r_{m-1}, 1 \right)$$

Then the following equation holds:

$$(\mathrm{L^i}|\mathrm{L^g}|\omega) = \mathrm{R} \times \mathrm{F} \tag{6.2}$$

where the matrix F defines the access function f. For example, taking the kernel in Figure 6.1 and assuming array `image` is defined as $\mathtt{image}[144][144]$, the linearized access for the array `image` in the statement S_2 of Figure 6.1 can be represented as:

$$(144,1) \times \left[\begin{array}{cccc|cc|c} 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array} \right] = (144,1,144,1,0,0,0)$$

meaning that linearized access function is:

$$\ell(v,h,i,j) = b + 144v + h + 144i + j.$$

The linearized access function is crucial for computing the cost of vectorized data load/store instructions, which constitutes the majority of the vectorizer overhead.

6.2.2 Access Pattern Sensitivity to Scheduling

Based on the canonical scheduling matrix representation defined in Section 3.1 the rescheduled timestamp vector $\mathbf{t}$ is expressed as follows (for modelling purposes we can ignore β):

$$\mathbf{t} = (\mathrm{A}|\Gamma) \times (\mathbf{i}, \mathbf{g}) = \mathrm{A}\mathbf{i} + \Gamma\mathbf{g} \tag{6.3}$$

thus the original iteration vector is $\mathbf{i} = \mathrm{A}^{-1}(\mathbf{t} - \Gamma\mathbf{g})$ which together with Equation (6.1) gives us the new, transformed linearized access function:

$$\ell'(\mathbf{t}) = b + \mathrm{L^i}\mathrm{A}^{-1}\mathbf{t} + (\mathrm{L^g} - \mathrm{L^i}\mathrm{A}^{-1}\Gamma)\mathbf{g} + \omega. \tag{6.4}$$

2. b is typically not known at compilation time; nevertheless, we are only interested in its alignment modulo the VF, which is generally available.

with a new vector of coefficients:

$$L' = (\mathrm{L^i A^{-1}} | \mathrm{L^g} - \mathrm{L^i A^{-1}} \Gamma | \omega).$$

Taking as example the kernel in Figure 6.1, the linearized access functions for arrays `image` and `filter` are as follows:

$$\begin{aligned}\ell_{\texttt{image}}(\mathbf{i}) &= b + [\ 144 \quad 1 \quad 144 \quad 1 \mid 0 \quad 0 \mid \omega\ ] \times (\mathbf{i}, \mathbf{g}, 1)^T \\ \ell_{\texttt{filter}}(\mathbf{i}) &= b + [\ 0 \quad 0 \quad 144 \quad 1 \mid 0 \quad 0 \mid \omega\ ] \times (\mathbf{i}, \mathbf{g}, 1)^T\end{aligned}$$

After applying loop interchange, by swapping the columns 3 and 4, we obtain the following transformation matrix (only A part is shown):

$$\mathrm{A}'^{S_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

the new access functions become as follows:

$$\ell_{\texttt{image}}(\mathbf{t} = (v, h, j, i)) = b + 144v + h + i + 144j$$

$$\begin{aligned}\ell'_{\texttt{image}}(\mathbf{t}) &= b + (\ 144 \quad 1 \quad 1 \quad 144 \mid 0 \quad 0 \mid \omega\ ) \times (\mathbf{i}, \mathbf{g}, 1)^T \\ \ell'_{\texttt{filter}}(\mathbf{t}) &= b + (\ 0 \quad 0 \quad 1 \quad 144 \mid 0 \quad 0 \mid \omega\ ) \times (\mathbf{i}, \mathbf{g}, 1)^T\end{aligned}$$

Notice that the memory access strides with respect to the new scheduling dimensions have changed. This has a dramatic impact on the performance of the vectorized code. Indeed, if we chose to vectorize the innermost level, the vectorized code will suffer from a very costly memory access operations: the stride is 144, so the elements are not consecutive and a vector cannot be loaded within one vector-load instruction.

This shows that linearized access functions, on which the total vectorization cost depends, is transformed automatically with the scheduling transformations. Thus, we do not need to generate code in order to compute the vectorization cost after applying a set of loop transformations – the vectorization cost is the direct function of scheduling matrix. For the rest of presentation we focus on $\mathrm{L^i}$ part of the linearized access function coefficient vector.

6.2.3 Cost Model Function

Our cost model is based on modelling the total execution time of all statement instances, given the modified iteration domain $\mathcal{D}'^S$ and the modified schedule $\theta^{S'}$ of each statement S. We compute the cost function for statement S as follows:

$$\begin{aligned} c(\mathcal{D}'^S, \theta'^S) \quad = \quad & \frac{|\mathcal{D}'^S|}{\mathrm{VF}} (\sum c_{\text{vect\_instr}}) + \\ & \sum_{m \in (\mathcal{W}_S)} (c_a + \frac{|\mathcal{D}'^S|}{\mathrm{VF}} (c_{\text{vect\_store}} + f_m)) + \\ & \sum_{m \in (\mathcal{R}_S)} (c_a + \frac{|\mathcal{D}'^S|}{\mathrm{VF}} (c_{\text{vect\_load}} + c_s + f_m)) \end{aligned}$$

where $|\mathcal{D}'^S|$ denotes the integer cardinality of the iteration space (total number of dynamic instances of S) and VF is the vectorization factor.

We currently support the simplest case, when loop bounds form a rectangular polyhedral iteration space: $|\mathcal{D}^S| = \prod_{i=1}^{dim(S)}(\mathrm{UB}_i - \mathrm{LB}_i)$. In a more general case, algorithms to computing the number of integer points inside polyhedra could be used [19, 43, 135].

Given the desired scheduling dimension d, we can compute a finite difference Δ_d of the linear memory address with respect to the time dimension:

$$\Delta_d = \ell(i_1, \ldots, i_d + 1, \ldots, i_{d^S}) - \ell(i_1, \ldots, i_d, \ldots, i_{d^S}) = \mathrm{L}^{\mathrm{i}}_d$$

Δ_d is a memory access stride w.r.t. a schedule dimension; we will simply call it *the stride*. It can be determined directly from linearized access vector L^{i} by looking at its d-th component.

Factor c_s considers the penalty of load [3] instructions accessing memory addresses with a *stride* across the loop being vectorized. Accesses to non-unit strided addresses require additional data unpack or pack operations, following and/or preceding vector load or store instructions, respectively [116].

For example, VF scalar accesses to memory addresses with stride Δ_{d_v} across the loop being vectorized may require Δ_{d_v} vector loads (each with cost c_1), followed by $\Delta_{d_\mathrm{v}} - 1$ vector extract odd or extract even instructions (each with cost c_2), to produce one vector holding the desired VF elements. On the other hand, if several accesses to the same address are vectorized together (i.e. $\Delta_{d_\mathrm{v}} = 0$), a vector “splat” instruction is often required to propagate the loaded value across all elements of a vector (with cost c_0). Factor f_s is computed as a function of the stride Δ_{d_v}:

$$c_s = \left\{ \begin{array}{lc} \Delta_{d_\mathrm{v}} = 0 : & c_0 \\ \Delta_{d_\mathrm{v}} = 1 : & 0 \\ \Delta_{d_\mathrm{v}} > 1 : & \Delta_{d_\mathrm{v}} \cdot c_1 + (\Delta_{d_\mathrm{v}} - 1) \cdot c_2 \end{array} \right\} \tag{6.5}$$

Factor f_a considers the *alignment* of loads and stores. Typically, accesses to memory addresses that are aligned on VF-element-boundaries are supported very efficiently, whereas other accesses may require loading two aligned vectors from which the desired unaligned VF elements are extracted (for loading) or inserted (for storing). This alignment overhead may be reduced considerably if the stride Δ of memory addresses accessed across loop levels $d_\mathrm{v} + 1..d^S$ is a multiple of VF, since the misalignment remains constant inside the vectorized loop. If this is the case there is the opportunity to reuse loaded vectors and use invariant extraction masks. By having the transformed linearized access function:

$$\ell(\mathbf{i})^T = b + \mathrm{L}^{\mathrm{i}}_1 i_1 + \ldots + \mathrm{L}^{\mathrm{i}}_{d_\mathrm{v}} i_{d_\mathrm{v}} + \cdots + \mathrm{L}^{\mathrm{i}}_{dim(S)} i_{dim(S)} + \mathrm{L}^{\mathrm{g}}\mathbf{g} + \omega$$

it is easy to check if misalignment inside the vectorized loop remains constant – the coefficients from $\mathrm{L}^{\mathrm{i}}_{d_\mathrm{v}+1}$ to $\mathrm{L}^{\mathrm{i}}_{dim(S)}$ (corresponding to strides of all inner loops of the vectorized loop) have to be a multiple of VF.

If the misalignment is constant inside the vectorized loop we also check if the base address which is accessed on each first iteration of the vectorized loop (d_v) is known to be aligned on VF-element-boundary; if so then there is no need for re-aligning any data: $f_a = 0$. This is done by considering strides across outer-loops (enclosing the vectorized loop, if exist), and initial alignment properties such as array alignment. In order to check the alignment in outer loops, we need to check if coefficients from $\mathrm{L}^{\mathrm{i}}_1$ to $\mathrm{L}^{\mathrm{i}}_{d_\mathrm{v}-1}$ are multiple of VF.

By putting together all considerations for alignment, cost can be modelled as:

$$f_a = \left\{ \begin{array}{lcc} \text{aligned} & : & 0 \\ \text{var. misalign.} & : & |\mathcal{D}^S|(c_1 + c_3 + c_4) \\ \text{fixed misalign.} & : & |\mathcal{D}^S_{1..d_\mathrm{v}-1}|(c_1 + c_3) + \\ & & |\mathcal{D}^S|(c_1 + c_4) \end{array} \right\} \tag{6.6}$$

3. Vector store operations with strided access is not yet implemented in GCC.

where c_3 represents the cost of building a mask based on the misalignment amount, c_4 is representing the cost of extraction or insertion and c_1 is the vector load cost. $|\mathcal{D}^S_{1..d_v-1}|$ denotes the number of iterations around the vectorizer loop level.

The vectorization factor VF of a loop is determined according to the size of the underlying vector registers and the smallest data-type size operated on inside the loop. Each individual vector register will thus hold VF values of this small size. However, if there are variables in the loop of larger size, storing VF copies of them will require multiple vector registers, which in turn implies that the associated instructions need to be replicated. Factor f_m records the extra overhead that is associated with this replication. Additional factors that depend on the specific machine resources available may also impact the performance of vectorization, such as the size of register files, available ILP, and complex vector instructions.

By applying different loop interchange transformations and choosing different loops to vectorize, the performance of the resulting vectorization varies considerably. Our model was able to predict the best possible combination of loop interchange and outer/inner vectorization strategy.

6.2.4 Vectorization Profitability Metrics

To summarize, the above vectorization profitability metrics can be classified into the following three classes:

Scheduling invariant metrics are not affected by changing the execution order of the statement instances. Thus, they are invariant with respect to a *scheduling* function θ^S. Vector to scalar reduction cost and multiple type support costs fall into this category.

Scheduling sensitive metrics are affected by changing the execution order of the statement instances. Those metrics have the greatest impact on the resulting cost, since they vary with the change of the scheduling order and thus are affected by loop transformations such as interchange. Changing of the scheduling function θ^S directly affects the cost of strided memory accesses, non-aligned vector loads and spatial locality.

Code generation dependent metrics depend on the actual code-generation strategy implemented in a compiler[4]. These metrics go against the design of our cost model, as they need to regenerate the compiler's internal structures from the modified polyhedral representation. Yet no significant performance factor falls into this category. There is a practical difficulty however: idiom recognition (e.g., saturated arithmetic, absolute differences) are typically a source for scheduling-sensitive metrics, as the identification of algorithmic idioms can be performed on the array data flow [139]. Yet vectorizers currently rely on syntactically fragile pattern-matching techniques and do depend on the code generation.

6.3 Evaluation

We present a systematic method to evaluate the cost function, within a selected transformation search space. Later, we will show the experimental evaluation of the predicted speedups, compared to the actual execution on two different architectures for a selected benchmark, altogether with a detailed discussion.

6.3.1 Driving the search process

To select an optimal vectorization strategy one needs to construct and traverse the relevant search space. We want to do so without generating different syntactic versions of the code and then vectorizing each of them, which is inefficient and sometimes infeasible. Our proposal uses an analytical cost model,

4. in our case it is GCC compiler

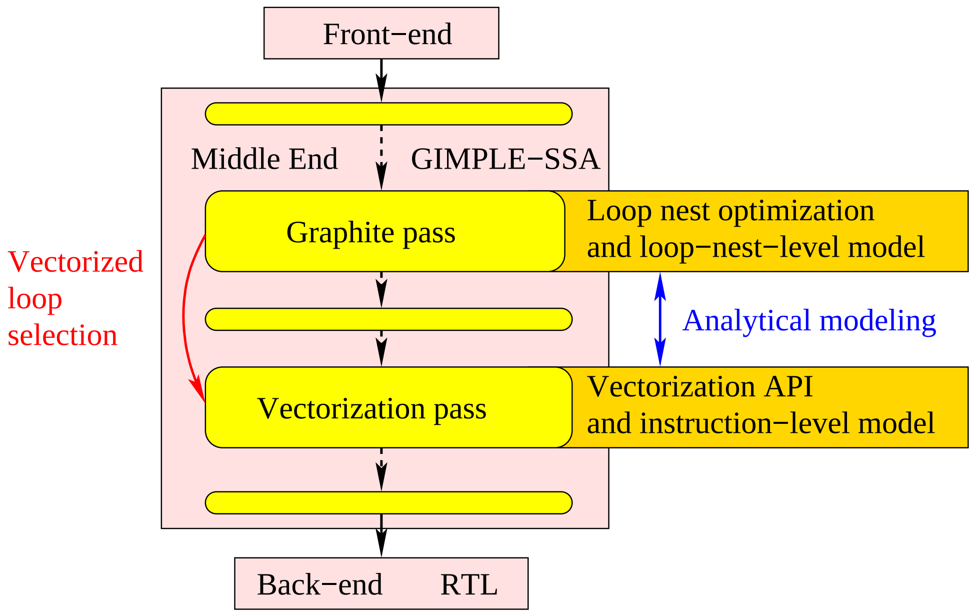

Front−end
Middle End
GIMPLE−SSA
Graphite pass
Loop nest optimization and loop−nest−level model
Vectorized loop selection
Analytical modeling
Vectorization pass
Vectorization API and instruction−level model
Back−end
RTL

Figure 6.3 – compilation flow

and constructs the *finite* search space of chosen loop transformations expressed in terms of modified affine schedules $\theta^{S'}$ and modified iteration domains $\mathcal{D}'^{S}$ of Statements. For each point in the search space we compute an associated cost using a cost function $\phi(\mathbf{x})$.

We model the scheduling of each individual Statement independently. Each point in the search space corresponds to a vector $\mathbf{x} = [\theta'^{S_1}, \ldots, \theta'^{S_n}, \mathcal{D}'^{S_1}, \ldots, \mathcal{D}'^{S_n}]$ of modified schedules and domains for each of the n Statements of the SCoP. The total cost for a given point $\mathbf{x}$ in the space is the sum of costs of executing dynamic instances of all SCoP Statements according to a new schedule and domain:

$$\phi(\mathbf{x}) = \sum_{i=1}^{n} c(\mathcal{D}^{S_i}, \theta^{S_i}). \tag{6.7}$$

The parameters for the cost function for the single Statement S_i are its iteration domain $\mathcal{D}'^{S_i}$ (number of dynamic instances of a Statement depends on its iteration domain) and its scheduling θ'^{S_i} (cost of accessing memory by a Statement instance depends on execution order of other instances). Section 6.2.3 describes this cost function in detail.

The optimization goal is to search for a vector of transformations $\mathbf{x}_{min}$ that minimizes the cost function $\phi(\mathbf{x})$:

$$\mathbf{x}_{min} = \min_{\mathbf{x} \in X} \phi(\mathbf{x}) \tag{6.8}$$

Vector $\mathbf{x}_{min}$ represents an optimal program version in the polyhedral model.

After extracting the SCoPs and building the polyhedral representation of all statements by the GRAPHITE framework, we perform the optimization of each SCoP according to Algorithm 1: first we compute the base cost for the unmodified (input program) representation, by computing the cost of executing all dynamic instances of all statements S_i in the original scheduling order. The current optimal cost is stored in $cost_{min}$ and is updated incrementally by applying different transformations (skipping over infeasible ones) on the polyhedral model (stored in vector $\mathbf{x}$) and computing the new costs using cost function $\phi(\mathbf{x})$. Besides the schedule transformation, performed by permuting (PERMUTE) the columns of the component A of the schedule, each possible level v is strip-mined, which is the way to model the vectorization at level v. At the end of this process the optimal scheduling representation is available in $\mathbf{x}_{min}$.

Note that Algorithm 1 shows only one possible way of constructing a search space. We chose to consider all combinations of loop interchanges due to their impact on vectorization. This small (yet expressive) search space makes it compatible with the constraints of a production compiler. Much more expressive search space will be explained in Chapter 7.

Algorithm 1 Driver for search space exploration

```
$d \leftarrow$ level of a deepest Statement S in a SCoP
$n \leftarrow$ number of Statements in a SCoP
$\mathbf{x}_{min} \leftarrow [\theta^{S_1}, \ldots, \theta^{S_n}, \mathcal{D}^{S_1}, \ldots, \mathcal{D}^{S_n}]$    ▷ Start with the original schedules and iteration domains
$cost_{min} \leftarrow \phi(\mathbf{x}_{min})$
for all $\sigma \in$ (set of d-element permutations) do
    for $i = 1$ to $n$ do
        $\theta'^{S_i} \leftarrow$ PERMUTE$(\sigma, \theta^{S_i})$
        $d^{S_i} \leftarrow$ level of loop nesting for Statement $S_i$
        for $v = 1$ to $d^{S_i}$ do
            $\mathcal{D}'^{S_i} \leftarrow$ STRIPMINE$(v, \mathcal{D}^{S_i})$
            $\mathbf{x} \leftarrow [\theta'^{S_1}, \ldots, \theta'^{S_n}, \mathcal{D}'^{S_1}, \ldots, \mathcal{D}'^{S_n}]$
            if $\phi(\mathbf{x}) < cost_{min}$ then
                $cost_{min} \leftarrow \phi(\mathbf{x})$; $\mathbf{x}_{min} \leftarrow \mathbf{x}$
            end if
        end for
    end for
end for
```

6.3.2 Experimental setup

	N_1	N_2	N_3	N_4	Δ_1	Δ_2	Δ_3	Δ_4
interp_fp	512	16			1,2	1,0,2		
interp	512	16			1,2	1,0,2		
bkfir	512	32			1,0	1,1		
dct	8,8	8,8	8,8		8,0	0,1	1,8	
convolve	128	128	16	16	128, 0, 128	1, 0, 1	128, 16, 0	1, 1, 0
H264	12,7	12,7			1	1		
dissolve	128	128			1	128		
alvinn	512,32	32,32			1,1	512,512		
MMM	16	16	16		16,0	0,1	1,16	
MMM^T	16	16	16		16,0	0,1	1,16	

Table 6.1 – Benchmarks

We evaluate our approach by introducing our model into the polyhedral framework of GCC production compiler and comparing its performance estimates for different loop interchanges and vectorization alternatives against actual execution runs of a set of benchmarks. Table 6.1 summarizes the main relevant features of the kernels used in our experiments: a rate 2 interpolation (*interp*), block finite impulse response filter (*bkfir*), an 8×8 discrete cosine transform (*dct* [101]), 2D-convolution (*convolve*), a kernel from H.264 (*H264*), video image dissolve (*dissolve*), weight-update for neural-nets training (*alvinn*) and a 16×16 matrix-matrix multiply (*MMM*) (including a transposed version MMM^T).

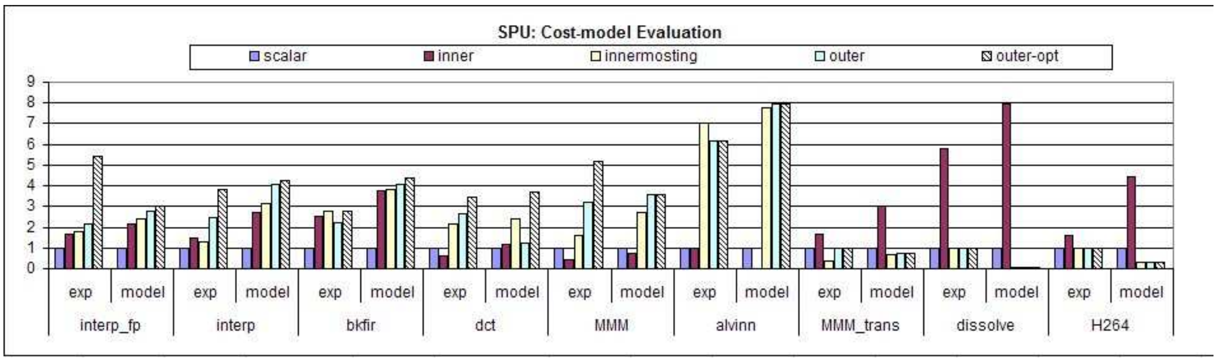


Figure 6.4 – Cost model evaluation: comparison of predicted and actual impact of vectorization alternatives on the Cell SPU

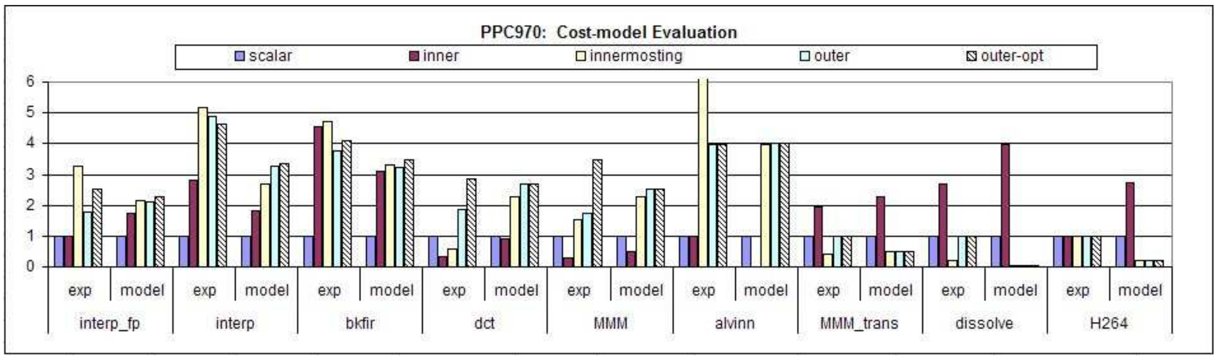


Figure 6.5 – Cost model evaluation: comparison of predicted and actual impact of vectorization alternatives on PPC970

The first four columns of Table 6.1 show the number of iterations N_i of loops nested within the main loop-nest of each benchmark, starting with N_1 for the outermost loop and moving inwards. Loop nests with less that 4 nested loops have empty entries (e.g., N_2 refers to the innermost loop in the doubly-nested *bkfir*). For example, *convolve* has a 4-nest loop whereas *bkfir* has only a doubly-nested loop and thus N_2 refers to its inner-most loop. Multiple values in an entry represent multiple distinct loop nests.

Similarly, the next four columns of Table 6.1 show the strides Δ_i of the memory references across each of the nested loops, with multiple values in an entry representing the strides of different memory references. For example, strides of 8, 512 and 16 are found in the innermost loops of *dct*, *alvinn* and *MMM* respectively, where columns of 2D arrays are scanned resulting in strides at the length of the rows. Lastly, zero strides imply that duplication of a single value across a vector is required.

We first evaluate the cost-model qualitatively, demonstrating that the scores it computes are consistent using one detailed example (subsection 6.3.3). The following subsection (subsection 6.3.4) evaluates the model relative to actual experiments on a set of kernels, analyzing the mispredictions and showing that overall the relative performance ordering of the variants is largely preserved.

6.3.3 Qualitative Evaluation

We use the *convolve* kernel qualitatively (see Figure 6.1, Section 6.1.2). We study only a small subset of the search space described in Chapter 7, restricting our attention to the $d \times d$ combinations of moving each loop inwards/outwards and vectorizing it there, plus the option to vectorize each of the d loops without any interchange. Note however that our technique opens up a much larger transformation space, as shown in the search methodology described in Chapter 7.

loop-level	1(v)	2(h)	3(i)	4(j)
1	0.26	4.00	0.24	4.00
2	0.26	4.06	0.24	4.21
3	0.26	**4.34**	0.23	4.56
4	0.27	**3.76**	0.24	**3.72**

Table 6.2 – Convolve: SPU estimated speedup factors

loop-level	1(v)	2(h)	3(i)	4(j)
1	0.21	3.21	0.19	3.21
2	0.21	3.21	0.19	3.38
3	0.21	**3.18**	0.19	3.70
4	0.21	**3.37**	0.20	**2.99**

Table 6.3 – Convolve: PPC estimated speedup factors

The results of running our model against the $d \times d = 16$ combinations, estimating the performance of each combination for a Cell/SPU and a PowerPC system are shown in Table 6.2 and Table 6.3 respectively. The loops are numbered in the tables from 1 (outer-most) to 4 (inner-most). Entry (i, j) shows the estimated speedup over the sequential version, obtained by moving loop j to position i followed by vectorizing loop j at new position i. Thus entries along the diagonal refer to vectorization with no interchange. Entries (4,4), (4,2), (3,2) (in bold) correspond to the vectorization alternatives shown in Figures 6.2a, 6.2b, 6.2c respectively.

While the estimated speedups for these versions are a little too optimistic compared to actual measured speedups (see Section 6.1.2), the relative ordering of the speedups for both platforms is accurate[5] and the cost model is able to identify the best choice among the three.

The convolve entry in Table 6.1 reveals the key factor for the performance degradations predicted for loops v, i (columns 1 and 3) — there are very large strides along these loops (Δ_1, $\Delta_3 = 128$). The overhead involved in vectorizing these loops and strides is described in Section 6.2.3. The remaining candidate loops for vectorization are therefore loops 2 and 4 (h and j). The best speedup is predicted for entry (3, 4) which corresponds to using outer-loop vectorization to vectorize the j-loop after moving it to level 3. The original i-nest is a perfect nest (there are no operations outside the innermost loop within that nest) and so there are no overheads incurred by this interchange (as opposed to interchanging an imperfect-nest like the h-loop, e.g. as in cases (4,2),(3,2)/Figures 6.2b,6.2c, which involve scalar expansion and loop-distribution costs). In addition, outer-loop vectorization avoids reduction-epilogue costs and also increases the portion of the code that is being vectorized compared to vectorizing the j-loop in its original innermost location. Note that this choice is different from the traditional approach: compilers usually either apply inner-most loop vectorization (entry $(4, 4)$ in the tables) or apply innermosting (entries $(4, *)$).

Partial experimental evaluation of *convolve* confirms these predictions. In Figure 6.6 we show the obtained speedups relatively to the cost model estimations (denoted *exp,model* respectively) for PPC970 and Cell SPU for entries $(3, 2)$, $(4, 2)$, $(3, 4)$ and $(4, 4)$ in the tables.

5. The low 1.59x measured speedup for alternative 6.2b on the Cell SPU is due to an aliasing bug in GCC that results in bad scheduling. The out-of-order wide-issue (5 slots) PowerPC970 is less sensitive to this, but on the in-order 2-width-issue SPU performance drastically suffers as a result. The cost model obviously cannot (and should not) predict compiler bugs, however it can, as in this case, help reveal them.

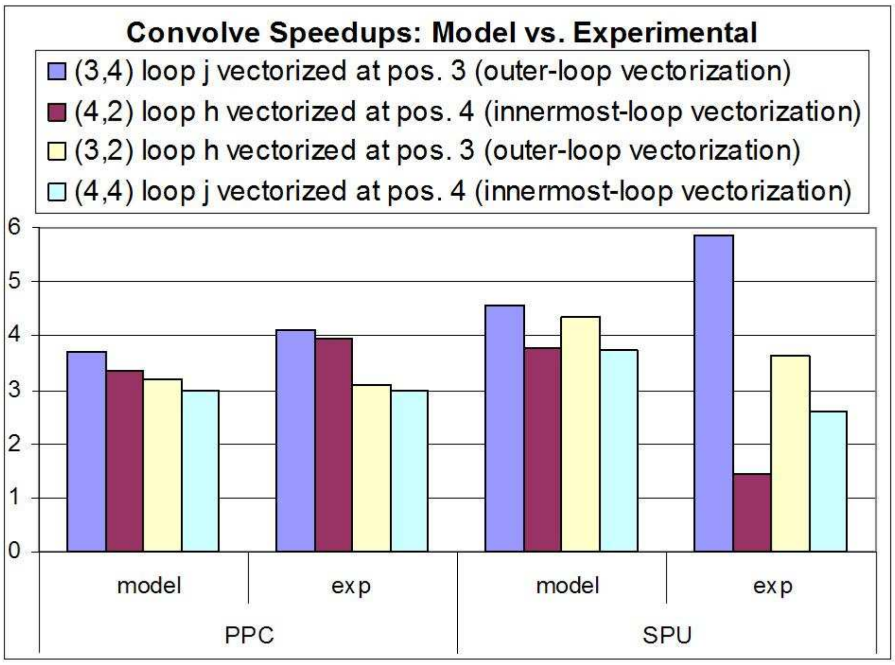


Figure 6.6 – Cost model evaluation: comparison of predicted and actual impact for convolve kernel on PPC970 and Cell SPU

6.3.4 Evaluation on a benchmark suite

We now validate quantitatively the estimates produced by the cost model. For each benchmark we report two sets of results: one showing the experimentally observed speedups, and the other showing estimated speedups computed by the cost model (denoted *exp,model* respectively in Figures 6.4, 6.5). When a given vectorization technique cannot be applied due to limitations of our current implementation of vectorization in the GCC compiler, the scalar performance is reported. This happens in some cases of strided accesses that are not yet fully supported (and that would certainly degrade performance).

We evaluate the relative speedup of four different vectorization alternatives: innermost-loop vectorization (*inner*), interchange followed by innermost-loop vectorization (*innermosting*), and in-place outer-loop vectorization, with and without optimized realignment using unrolling (*outer* and *outer-opt*).

The experiments were generated automatically using an enhanced version of GCC. Speedup are measured over the sequential version of the benchmark, compiled with the same optimization flags. Interchange, when used, was applied manually. Time is measured using the `getrusage` routine on powerpc970, and the decrementer utility on the SPU. Experiments were performed on the IBM PowerPC PPC970 processor with Altivec, and an SPU of the Cell Broadband Engine. Both architectures have 128 bit wide vector registers, and similar explicit alignment constraints.

The first set of kernels (interp, bkfir, dct and MMM) is expected to gain most from in-place outer-loop vectorization with realignment optimization, as they consist of imperfect loop-nests (and therefore get penalized for interchange), and exhibit high data-reuse opportunities across the (vectorized) inner-loop that can be exploited by the unrolling optimization. They also have inner-loop reductions (which are generally done more efficiently using outer-loop vectorization), and two of the benchmarks in this set (dct and MMM) also have large strides in the innermost loop (as the access is column-wise). Alvinn has a perfect nest and no reuse opportunities, and therefore in-place outer-loop vectorization should not gain over traditional interchange, but innermost loop vectorization should be avoided due to the large stride. The last group of benchmarks (MMM^T, dissolve and H264) have consecutive access in the innermost loop, but strided access in the outer-loop, and so for these we expect inner-loop vectorization to be the best technique.

This behavior can be clearly observed in the SPU speedups in Figure 6.4, where overall the *exp* and *model* graphs are largely consistent, with the preservation of the relative performance ordering of the variants. Exceptions are due to low-level target-specific factors that our model does not take into account. Most notable is the misprediction in the first set of benchmarks, where *bkfir* and *dct* are the only benchmarks for which outer-loop vectorization is inferior to innermost loop vectorization due to an SPU specific issue (unhinted branch).

Target-specific issues come to play also on the PPC970 (Figure 6.5). The most significant one appears in the fixed-point bkfir and interp where inner-loop vectorization employs a specialized Altivec instruction to compute a dot-product pattern. We have not yet incorporated idioms into the cost-model and so it does not anticipate this behavior. The model also does not try to estimate register pressure, and therefore does not predict the degradation in performance incurred by the unrolling optimization on interp due to register spilling (this problem does not occur for SPU having 128 vector registers, compared to the 32 Altivec registers of PowerPC970). Inaccurate modelling of spatial locality in our current implementation is the reason why the cost model misses the improved data cache locality when interchanging loops in alvinn (this problem does not occur on the Cell SPU as it doesn't have a cache). Lastly, in some cases interchange can be done with smarter scalar-expansion (hoisting), whereas the model estimates the associated overhead of a naive scheme. This sometimes pessimizes the predicted speedup of interchanged versions both on PPC and the SPU.

6.4 Related Work

Vectorization Cost-Model Related Work. Leading optimizing compilers recognize the importance of devising a cost model for vectorization, but have so far provided only partial solutions. Wu et al. conclude [164] regarding the XL compiler that *"Many further issues need to be investigated before we can enjoy the performance benefit of simdization ... The more important features among them are ... the ability to decide when simdization is profitable. Equally important is a better understanding of the interaction between simdization and other optimizations in a compiler framework"*. Likewise, Bik stresses the importance of user hints in the ICC vectorizer's profitability estimation [29], to avoid vectorization slowdowns due to *"the performance penalties of data rearrangement instructions, misaligned memory references, failure of store-to-load forwarding, or additional overhead of run-time optimizations to enable vectorization."*; on the other hand opportunities may be missed due to overly conservative heuristics.

These state-of-the-art vectorizing compilers incorporate a cost model to decide whether vectorization is expected to be profitable. These models however typically apply to a single loop or basic-block, and do not consider alternatives combined with other transformations at the loop-nest level. This work is the first to incorporate a polyhedral model to consider the overall cost of different vectorization alternatives in a loop-nest, as well as the interplay with other loop transformations.

Loop-nest auto-vectorization in conjunction with loop-interchange has been addressed in prior art [6, 7, 163]. This however was typically in the context of traditional vector machines (such as Cray), and interchange was employed as a preprocessing enabling transformation. Overheads related to short-SIMD architectures (such as alignment and fine-grained reuse) were not considered.

Costs of specific aspects of short-SIMD vectorization were addressed in more recent works. Realignment and data-reuse were considered together with loop-unrolling [144], but in the context of straight-line code vectorization, and not for the purpose of driving loop vectorization. A cost model for vectorization of strided-accesses was proposed in [116], but it does not consider other overheads or loop transformations.

Polyhedral-Model Related Work. Bondhugula et al. [34] integrate inner-loop vectorization as a post-pass of their tiling heuristic, and leverage the interchangeability of inner loops to select one that is vectorizable. Their method does not take into consideration the respective vectorization overheads, nor does it model reductions. Nevertheless, their tiling hyperplane and fusion algorithm can serve as a complementary first pass for our technique, favoring the extraction of interchangeable inner loops.

Pouchet et al. demonstrate how one can systematically study the interplay of loop transformations with backend optimizations (including vectorization) and complex microarchitectures by constructing huge search spaces of unique, valid transformation sequences [128]. These search spaces are tractable using carefully crafted heuristics that exploit the structure of affine schedules. An analytical performance model capable of characterizing the effect of such complex transformations (beyond loop interchange, and accommodating for large-scale locality effects) does not currently exist. Known analytical cache models for loop transformations are quite mature in some domains, loop tiling in particular [36], yet remain sensitive to syntactic patterns and miss key semantical features such as loop fusion effects [39, 69].

6.5 Conclusions and future work

This contribution narrows the gap between analytical cost-model based static compilation techniques [39] and fully iterative search based compilation flow [1, 129, 128], while still maintaining the running time of the compilation acceptable. We have focused on the single aspect of the machine specific cost modelling - that of automatic vectorization on SIMD architectures.

We have presented the first-of-a-kind approach to low-level machine-specific analytical cost model, used for prediction of the performance of vectorization within the polyhedral model framework. Our contributions of this work are the following:

– **Polyhedral modelling of subword parallelism.** We demonstrate how to leverage the polyhedral compilation framework naturally and efficiently to assess opportunities for subword parallelism in combination with complex loop transformation sequences. Our integrated approach relies on abstract matrix manipulations instead of code generation, thereby shortening the evaluation time of the cost model dramatically, compared to iterative optimization approaches [129]
– **Evaluation in a production compiler.** Our model is fully automated and implemented based on GCC [71] 4.5 compiler.
– **Studying the interplay between loop transformations.** We provide a thorough empirical investigation of the interplay between loop interchange with array expansion and loop nest vectorization of both inner and outer loops on modern short-SIMD architectures.

6.5.1 Future work

The experimental results are promising, and the running time of the search space exploration is acceptable for incorporation into the general purpose compiler like GCC. Nevertheless, the cost model is somewhat simplistic and does not incorporate the modelling of cache locality issues [69], interplay with thread-level parallelism nor the incorporation of memory traffic optimizing loop transformations [34] like loop interchange. We see the future directions that could fully exploit the potential of this approach:

Interplay with other transformations. The method could easily be extended and employed in the future, to consider the effects of additional transformations within the polyhedral framework. The search space that is proposed in Chapter 7, which is in turn based on [34], could be used to combine the search for thread-level parallelism, memory locality optimization and optimal vectorization strategy in one, combined manner.

Precise data locality model. Our cost model includes a very rough estimation of the data locality - that of computing the data access strides. While it can capture the *spatial locality* of the given memory access pattern, it cannot precisely model the *temporal locality*. The model could be extended with cache miss equations, an approach proposed by Fraguela [69]. The preliminary integration efforts are ongoing, at the time of writing this dissertation.

Machine learning of target specific factors. The analytical cost function shown in subsection 6.2.3 relies on the *machine specific* atomic instruction costs. While we have obtained those costs for the architectures that we were interested in, mainly through microbenchmarking and knowledge of the instruction latencies, obtaining those coefficients for the new architectures might be. In order to facilitate the porting of the cost-model to the new architectures, we would like to investigate the possibility of employing machine learning, like in [40], to obtain the coefficients automatically for the new architectures. Please note that this does not change our approach: a *learning phase* is done only once, to obtain the coefficients. Later, the ready cost-model function is used in a single-pass compiler directly, without any extra penalty.

Chapter 7

Transformation search strategy

In the previous chapters we have discussed the basic components needed in the polyhedral model based optimization compiler such as static analysis, data dependence analysis, cost model, and code generation. But the core part of each polyhedral model based compilation flow is an algorithm for obtaining the actual transformation. The aim of this chapter is to provide our contribution to this key part of the compilation flow.

While numerous works on program transformations - and polyhedral transformations in particular - have been published, there is no general consensus on whether they are practical and efficient for use in a general purpose optimizing compilers. In this chapter we propose a transformation search methodology, instead of proposing a single specialized transformation algorithm.

The chapter starts with introducing the related work in Section 7.1. A necessary notation, based on the state of the art approaches, is introduced in Section 7.5. The core of the method is explained in Sections 7.6 through 7.10.

7.1 Related work

Optimizing for various architectural features requires complex program transformations to exploit thread-level parallelism, vectorization opportunities and memory hierarchy in a combined manner.

Pioneering works [6, 99, 15, 16, 161] on loop transformations look for the best transformation according to some ad-hoc criteria. Furthermore, they are restricted to a very limited subset of program loops.

Loop optimizations expressed in the polyhedral model subsume all previous works on loop transformations. Optimizations in the polyhedral model are expressed as scheduling problems. There are as many scheduling algorithms as there are different optimization criteria, but they all share the common property: the computed schedule has to provide a *legal* transformation.
The state of the art polyhedral transformation frameworks generally fall in two categories:

– *best effort*, cost model based scheduling approaches
– *iterative* search based approaches

The works of Feautrier [67, 68], Lim and Lam [107, 106], Griebl [76] and Bondhugula [34] fall into the first category, while the works of Pouchet [129, 128] and Vasilache [157] fall into the second category.

The basic principle underlying the best effort scheduling approaches is a well defined cost model function. The problem is cast as a linear programming problem where the data dependences provide the set of constraints, and an objective is to find a solution that minimizes the given cost model function, while preserving the legality of the problem.

Obviously, different cost functions provide different solutions. An objective function in the approach of Feautrier is the *latency*. The goal of the scheduling is to provide minimum latency schedules. Lim and Lam provide an objective function that minimizes the order of required synchronizations and maximize the number of parallel loop dimensions.

The state of the art cost model based approach of Bondhugula [34] provides a cost function that minimizes the *dependence distance*. It turns out that this simple cost function is very powerful, since it maximizes the parallelism in the outermost loop levels and minimizes the volume of memory communication.

While the best effort scheduling approaches provide only one solution - the best solution according to the cost function - the other class of the scheduling approaches is based on the *iterative* search.

Iterative search based scheduling approaches do not provide a single solution. Indeed, they provide a space of the legal and distinct schedules. Each distinct schedule corresponds to one program transformation. The transformations within a space are assessed according to their 'quality' - mostly often the speedup provided.

Iterative approach in the polyhedral model was pioneered by Pouchet [129, 128]. Pouchet shows the systematic way to generate the space of legal and distinct schedules expressed in the polyhedral model and the way to traverse those schedules. For each distinct schedule a code is generated and the transformed program is run on the target hardware. The schedule corresponding to program version with the best runtime is selected.

Since the search space of all legal schedules might be huge, Pouchet proposes to use machine learning techniques to speedup the search [128].

7.2 Limitations of the current approaches

The best effort cost model based techniques are limited in their ability to find the best transformation adapted to a particular architecture.

The approach of Feautrier [67, 68] finds the minimal latency schedule. But this cost model is based on an imaginary machine model where synchronization costs and memory locality issues are ignored. The algorithm is optimal on an idealized PRAM machine, but it is unlikely to be optimal on any real-world multiprocessor machine. This fact was empirically proven by Bondhugula [35].

The approach of Lim and Lam [107, 106] has a similar limitation. While it takes into an account the cost of synchronization, it does not model the memory traffic costs. The optimization process is based on basic linear algebra rather than on linear programming. This has an implications on the quality of obtained solution, since the method can reduce the *degree* of synchronization and maximize the degree of parallelism, but it can miss an exact optimal solution.

The state of the art approach of Bondhugula [34] has a simple, but very powerful cost model that matches the key performance factors relevant to modern architectures. It takes into an account both parallelism and memory traffic optimization.

While being well adapted to extracting parallelism and optimizing for memory locality through loop tiling, the approach Bondhugula [34] does not take into account some lower level architectural features, such as SIMD vectorization and possible cache interferences.

As evidenced in [131], the mentioned problem of Bondhugula's approach could be alleviated if combined with iterative search strategy. The method shown in [131] does not really into any of the categories discussed so far, since it is a combination of both.

The iterative search methods try to overcome the lack of the precision of the cost model based approaches but with an extra cost - that of generating multiple versions of program transformations.

The iterative search method of Pouchet [129, 128] builds a search space of all legal and distinct program transformations expressed as one-dimensional [129] or multi-dimensional [129, 128] affine schedules.

The main problem of iterative search methods is the problem of bounding the search space of legal program versions. In their full generality, the space of affine schedules is unbounded. One has to bound the possible values of the scheduling coefficients to some restricted range in order to obtain the finite-size search space.

Pouchet [129] proposes some practical solutions to the problem of bounding the search spaces to make the iterative search methods practical. We might add a note that the cost model based approaches do not face the problem of bounding the coefficient values, since those values are computed directly from the analytical cost model.

The feedback directed iterative search methods based on empirical program evaluation are obviously not suitable for integration into the general purpose optimizing compiler because of their unpredictable running time - even for simple program kernels, a millions of legal transformations might be evaluated [129].

Since the search space explored by the iterative search method has to be bounded upfront, some possible solutions might be missed, even if an exhaustive search is performed.

Obviously, neither class of the methods is better than the other. While the best effort, cost model based approaches provide the desired solution in the single step, they might not precisely model the target architecture and they might provide suboptimal results. On the other hand, the feedback directed, iterative search based methods might provide the optimal transformations - missed by cost model based approaches - but at the cost of an exhaustive search of the transformation space.

The common limitation of all the methods discussed so far is the lack of control over the complexity of the generated code. To the best of our knowledge, there are no works addressing the problem of finding the good program schedule that also generates the least complex code, though Darte [54] has pointed out this problem. This problem is very relevant for enabling the integration of the scheduling algorithm in a general purpose compiler. We will elaborate on this problem Section 7.3.

All the mentioned methods rely on the representation of the data dependence graph and satisfaction of all the data dependences. As we have shown in Chapter 5, some of the data dependence constraints could be relaxed and some dependence constraints could be completely removed with an extra cost of memory expansions. Removing the dependence constraints enables more optimization choices.

While the techniques for removing data dependence constrains have been extensively studied [64, 152, 37, 17, 97], they were treated separately from the scheduling problem itself. The only work that addresses those two problems in a combined manner is [45]. The related problems were discussed extensively in Chapter 5.

We summarize the limitations of the current best effort, cost model based approaches:

- Provide only a rough, linear analytical model based on an imaginary target execution platform
- Focus on a particular performance issue, while not taking into an account the holistic effect of multiple program transformations
- The optimization process takes a *greedy* optimization approach that can stick to local minimum and miss a global optimal solution
- No control over the complexity of the generated code
- No way to integrate the relaxation of the scheduling constraints through dependence removal

The limitations of the iterative search based approaches are summarized:

- Exhaustively searching the space of all transformations is expensive in terms of time
- The complete search space is unbounded and a bounding solution has to be provided
- Even if an exhaustive search is performed, the optimal solution might be missed
- Does not help in understanding the program transformations
- No control over the complexity of the generated code

7.3 Motivation

Having on mind the limitations of the current scheduling approaches, our goal is to propose the transformation search methodology that could provide the best trade-off between cost model based scheduling methods and search based techniques.

The core idea is to combine the enumeration search method with an evaluation of the specialized analytical cost model function. In this way, we combine the best of both worlds: the *precision* of iterative enumeration and *effectiveness* of the cost model based approaches.

The specialized cost function is tailored to the specific architecture and models the selected performance phenomena. Such a cost model function was shown in Chapter 6.

Previous cost model based approaches could not handle the complex non-linear objective functions. The reason is simple: they use (integer) linear programming techniques that only can operate on linear functions.

We cannot use the techniques proposed by Feautrier and Bondhugula to evaluate and choose the best transformation according to the non-linear cost function. Instead, we resort to enumerative evaluation of the non-linear cost function, in a similar way as the iterative approach.

Feedback directed iterative optimization [129, 128] relies on the feedback, for each generated program version, obtained from measuring the performance of the transformed program. But instead of relying on the empirical feedback, we plug in the non-linear objective cost function to provide the assessment of each transformation version.

Provided that we have a precise cost function tailored to a target architecture and that we can construct a bounded but expressive search space, we can perform an *enumerative search* of the cost function evaluations and pick the best point in the search space.

The success of our approach relies on providing the following key properties:

1. construction of an expressive search space of affine transformations
2. ability to control the size of the search space
3. ability to efficiently evaluate the cost function for each point in the search space

Based on our new search methodology, we want to address the problems that were not addressed in an integrated manner in the previous approaches. Mostly importantly, we aim at providing the transformation engine that could be integrated as an optimization pass of a general purpose compiler.

Two properties of our method enable its easy integration into the general purpose compiler: 1) embedding of the precise cost model tailored to a given architecture 2) not relying on the empirical feedback as the current iterative optimization approaches [129, 128]

An enumeration based nature of our approach enables us to control the complexity of the generated code by ignoring some transformations (loop skewing for example) that might generate the transformed code that is too costly [54].

Also, an enumerative approach for the schedule construction goes in tandem with a lazy memory expansion scheme discussed in Chapter 5. There is a trade-off between the cost of the memory expansion and the expressiveness of the transformation. To the best of our knowledge, there is no work that shows how to capture this trade-off in a purely best effort based methods [68, 106, 34]. An enumerative based approach can capture this trade-off in its search space.

7.4 Problem statement

The goal of the transformation search strategy is the enumeration of the search space of a class of legal multidimensional schedules. The space of the legal solutions $\mathcal{L}$ is obtained. Each legal solution $T \in \mathcal{L}$ have the form:

$$T = \{\theta^{S_1}, \ldots, \theta^{S_p}\}$$

Each legal solution describes the complete scheduling matrices for all statements $\mathcal{S} = \{S_1, \ldots, S_p\}$. For each scheduling dimension of the legal solution a set of *scheduling properties* (parallel, sequential, permutable) is determined.

7.5 Notation and definitions

Here we give the necessary definitions and notations used throughout the rest of the chapter.

Given a scheduling matrix Θ^S for a statement S, we will denote the k-th row of the scheduling matrix Θ^S as Θ^S_k. This row represents an *affine scheduling hyperplane* θ^S_k. The similar notation is used in works relating to program transformations and scheduling in the polyhedral model, bearing different names or meanings - schedules, mappings or partitions. [34, 67, 92, 105].

Recalling the defined canonical form of the scheduling matrix in section 2.3.4, the scheduling matrix Θ^S could be split in three components: A, Γ and β. Thus, a k-th row of the matrix that represents an affine function has the following form:

$\theta^{S_i}_k = a^{S_i}_{k,1} i_1 + a^{S_i}_{k,2} i_2 + \ldots + a^{S_i}_{k,n} i_n + \gamma^{S_i}_{k,1} g_1 + \ldots + \gamma^{S_i}_{k,n_g} g_{n_g} + \beta^{S_i}_k$

For the purpose of this chapter, we will ignore the Γ part and consider it to be set to all zeros. [1] Taking this into an account, we can represent a single row of the matrix as:

$$\theta^{S_i}_k = a^{S_i}_{k,1} i_1 + a^{S_i}_{k,2} i_2 + \ldots + a^{S_i}_{k,n} i_n + \beta^{S_i}_k \tag{7.1}$$

Definition 7.5.1 (Affine form of Farkas Lemma). Given a nonempty polyhedron $\mathcal{P} = \{\mathbf{x} \in \mathbb{R}^n | A\mathbf{x} + \mathbf{b} \geq \mathbf{0}\}$ then an affine form $f : \mathbb{R}^n \to \mathbb{R}$ is non-negative at each point in a polyhedron $\mathcal{P}$ iff it is a positive affine combination of the polyhedron faces:

$$f(\mathbf{x}) = \lambda_0 + \sum_{k=1}^{m} \lambda_k (A_{k,\bullet}\mathbf{x} + b_k), \lambda_0, \lambda_1, \ldots, \lambda_m \geq 0 \tag{7.2}$$

7.5.1 Dependence satisfaction

We will briefly recall the definition given in Section 3.3. Given a dependence graph $G = (V, E)$, a valid schedule has to satisfy all the dependence edges $e \in E$. Each dependence edge $e \in E$ is labelled by a dependence polyhedron $\mathcal{P}_e$. A given dependence edge e from S_i to S_j is satisfied iff:

$$\forall (\mathbf{i}, \mathbf{j})^T \in \mathcal{P}_e : \theta^{S_i}(\mathbf{i}) \ll \theta^{S_j}(\mathbf{j}) \tag{7.3}$$

where $\ll$ is the lexicographical comparison operator. Since we are assuming multidimensional schedules, a given dependence edge e has to be *satisfied* at some *level*. We formally define this condition:

Definition 7.5.2 (Dependence satisfaction at level). A dependence edge e from statement S_i to statement S_j, labelled by a dependence polyhedron $\mathcal{P}_e$, is *satisfied* at level l if it is the first level to satisfy the following:

$$\forall (\mathbf{i}, \mathbf{j}) \in \mathcal{P}_e : \left\{ \forall_{1 \leq k < l} : \left(\theta^{S_j}_k(\mathbf{j}) - \theta^{S_i}_k(\mathbf{i}) = 0 \right) \wedge \theta^{S_j}_l(\mathbf{j}) - \theta^{S_i}_l(\mathbf{i}) \geq 1 \right\}$$

1. The Γ part corresponds to *parametric shift*, allowing a scheduling function to be dependent on global parameters (usually the problem size).

We say that the given dependence is *strongly* satisfied at level l, if the condition $\theta_l^{S_j}(\mathbf{j}) - \theta_l^{S_i}(\mathbf{i}) \geq 1$ holds for all statement instance pairs that are in dependence. If only the weaker condition $\theta_l^{S_j}(\mathbf{j}) - \theta_l^{S_i}(\mathbf{i}) \geq 0$ is satisfied, we say that a given dependence is *weakly* satisfied at a given level. Both definitions assume that the dependence is at least weakly satisfied at all previous levels $1 \leq k < l$.

Definition 7.5.3 (Outer parallelism enabling hyperplanes)**.** Given the set of statements $\mathcal{S} = \{S_1, \ldots, S_p\}$ we say that the set of hyperplanes $\{\theta_1^{S_1}, \ldots, \theta_1^{S_p}\}$ enable parallelism at the outermost level ($l = 1$) if the following condition is satisfied:

$$\forall(\mathbf{i},\mathbf{j}) \in \mathcal{P}_e : \left(\theta_1^{S_j}(\mathbf{j}) - \theta_1^{S_i}(\mathbf{i}) = 0\right), \forall e \in E$$

In other words, for all dependence edges $\forall e \in E$ and for all pairs of statement iterations in a dependence relation $\mathcal{P}_e$, the source and the sink of the dependence are mapped to the same scheduling hyperplane. If we treat the scheduling hyperplane as a *partition*, as in [107], it means that all the dependent iterations are executed within the same partition, while the partitions themselves could be executed in parallel. This kind of parallel execution is known as the DOALL or synchronization free parallelism.

Definition 7.5.4 (Permutable scheduling band)**.** The set of scheduling hyperplanes $\{\theta^{S_1}, \ldots, \theta^{S_p}\}$ at levels $l, l+1, \ldots, l+s-1$ form a permutable scheduling band iff:

$$\forall e \in E_l, \forall k(l \leq k \leq l+s-1) : \forall(\mathbf{i},\mathbf{j}) \in \mathcal{P}_e, \theta_k^{S_j}(\mathbf{j}) - \theta_k^{S_i}(\mathbf{i}) \geq 0$$

where E_l is the set of dependence edges not satisfied up to level $l-1$. The loops that correspond to scheduling hyperplanes within the permutable scheduling band could be permuted freely, without violating any dependence. The similar definition is used in several works [106, 60, 86], where the goal is to find the maximal bands of fully permutable loops.

We have given the basic definitions that form the building blocks of our search methodology. Please note that some definitions are *local* – they apply for one dependence edge e at the time. In our transformation search method we will combine those constraints, step by step, into the global solution that satisfies the full dependence graph $G = (V, E)$.

7.6 The space of legal affine transformations

As stated before, the goal of any program transformation based on the polyhedral model is to find the coefficients of the scheduling matrices: $T = \{\theta^{S_1}, \ldots, \theta^{S_p}\}$.

Taking the full dependence graph $G = (V, E)$, each dependence edge $e \in E$ is described by a dependence polyhedron $\mathcal{P}_e$. A dependence edge e from S_i to S_j mandates that a condition shown in Equation 7.3 must be *strongly* satisfied. If we now consider the set of *one-dimensional* schedules, this condition becomes:

$$\forall(\mathbf{i},\mathbf{j})^T \in \mathcal{P}_e : \theta^{S_j}(\mathbf{j}) - \theta^{S_i}(\mathbf{i}) > 1 \tag{7.4}$$

But the scheduling matrices that encode the scheduling functions θ^{S_i} and θ^{S_j} have unknown coefficients. If we use the representation of the single row of the matrix as an affine function, as shown in Equation 7.1, we will get the following form of the constraint:

$\forall(\mathbf{i},\mathbf{j})^T \in \mathcal{P}_e : a_{1,1}^{S_j} j_1 + \ldots + a_{1,n}^{S_j} j_n + \omega_1^{S_j} - (a_{1,1}^{S_i} i_1 + \ldots + a_{1,n}^{S_i} i_n + \omega_1^{S_i}) > 1$

But this constraint is nonlinear, since both loop induction variables and coefficients are unknown. This obstacle can be avoided by linearizing the constraints by using the Farkas lemma. All the affine scheduling algorithms [2] rely on the Farkas lemma [67, 68, 128, 104, 132, 91] to perform the linearization.

2. An alternative approach is to use an equivalent *vertex method* as in [80, 159]

To perform the linearization, one has to represent the dependence polyhedron $\mathcal{P}_e$ as an intersection of affine inequalities: $\mathcal{P} = \{\mathbf{x} \in \mathbb{R}^n | A\mathbf{x} + \mathbf{b} \geq \mathbf{0}\}$. By applying Equation 7.2 and substituting the dependence satisfaction constraint on the left side of the equation and the affine combination of the faces of the dependence polyhedron $\mathcal{P}_e$ on the right side, one will get the *Farkas multipliers* - the unknowns $\lambda_0, \lambda_1, \ldots, \lambda_m \geq 0$. Afterwards, one can project out those multipliers and obtain the system of affine inequalities describing the set of transformation coefficients that provide a legal schedule w.r.t. the dependence polyhedron $\mathcal{P}_e$ - this set itself is a polyhedron $\mathcal{L}_{\mathcal{P}_e}$.

As an example, let us take the code shown in Figure 7.4. In this case we have only one statement and one dependence edge $e : S_1 \to S_1$. The dependence is the true (read-after-write) dependence described by the polyhedron:

$\mathcal{P}_e = \{(i, j, i', j') | i' = i + 1 \wedge j' = j + 1 \wedge 0 \leq i \leq N - 2 \wedge 0 \leq j \leq N - 2\}$

Our goal is to find the unknown coefficients $a^{S_1}_{1,1}, a^{S_1}_{1,2}, \omega^{S_1}_1$ of the one-dimensional affine scheduling function $\theta^{S_1}_1$ satisfying the dependence constraints:

$$\forall (i, j, i', j') \in \mathcal{P}_e : a^{S_1}_{1,1} i' + a^{S_1}_{1,2} j' + \omega^{S_1}_1 - (a^{S_1}_{1,1} i + a^{S_1}_{1,2} j + \omega^{S_1}_1) > 1 \tag{7.5}$$

In the case of uniform dependences, application of the Farkas lemma could be simplified.[3] The distance between source and sink iteration is constant - $i' = i + 1$ and $j' = j + 1$. If those equalities are put into the previous constraint system, the loop induction variables are canceled out. After simplifying we get the constraint: $a^{S_1}_{1,1} + a^{S_1}_{1,2} \geq 1$.

This inequality defines an unbounded polyhedron $\mathcal{L}_{\mathcal{P}_e}$ containing the possible values for scheduling coefficients. This set is an unbounded set – there is no upper nor lower bound on the coefficient values. Please note that coefficient $\omega^{S_1}_1$ is completely unconstrained. The legal solution polyhedron is described as follows:

$$\mathcal{L}_{\mathcal{P}_e} = \{(a^{S_1}_{1,1}, a^{S_1}_{1,2}, \omega^{S_1}_1) | a^{S_1}_{1,1} + a^{S_1}_{1,2} \geq 1\} \tag{7.6}$$

Figure 7.1 shows an iteration domain, and uniform dependences for the program in Figure 7.4. One of the possible choices for the coefficients is $a^{S_1}_{1,1} = 1, a^{S_1}_{1,2} = 1, \omega^{S_1}_1 = 0$, which gives the skewed scheduling hyperplane $\theta^{S_1}_1 = i + j$ shown in Figure 7.1. The skewed scheduling hyperplanes are executed sequentially, while the iterations that belong to the same hyperplane might be executed in parallel. But the *simpler* solution is to take $a^{S_1}_{1,1} = 1, a^{S_1}_{1,2} = 0, \omega^{S_1}_1 = 0$, which gives an orthogonal scheduling hyperplane - the one that corresponds to the original loop iterator `i`.

The goal of all the affine scheduling algorithms [67, 68, 128, 104, 132, 91] is to choose *the best* coefficient values among those contained inside the legal solution polyhedron $\mathcal{L}_{\mathcal{P}_e}$. The best coefficients are chosen according to some well defined criteria - maximal parallelism [67, 68], minimal synchronization [104] or minimal memory traffic [34] for an instance.

Another approach is to exhaustively search for all the possible schedules, given some bounds on the coefficient values. That is the approach that *iterative compilation* takes [130, 128]. Pouchet shows that it is feasible to exhaustively search the space of all the legal and distinct schedules, provided that the scheduling coefficients are bounded. As shown empirically by Pouchet, bounding the coefficients to integer values in the interval $[-1, 1]$ gives the schedules that are usually expressive enough [130].

In the example we have shown, there is only one dependence edge. One dependence edge gives one set of the constraints on the coefficients expressed as a polyhedron $\mathcal{L}_{\mathcal{P}_e}$. But in the general case, the dependence graph contains several dependence edges. To obtain the global solution, one has to take the

3. For a full example of applying a Farkas lemma, a reader is could refer to [130].

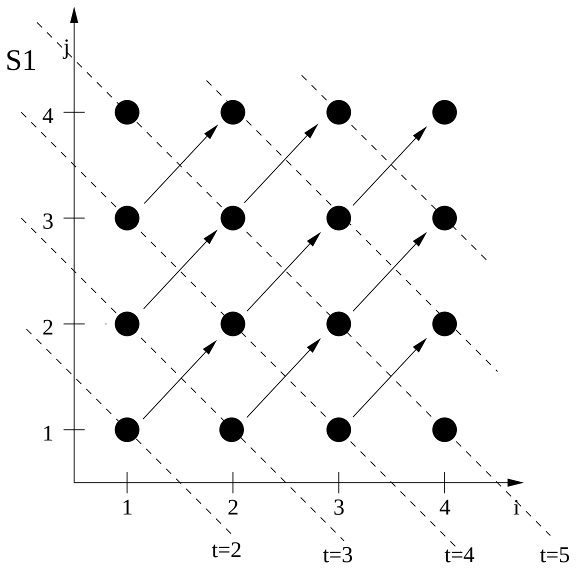


Figure 7.1 – Uniform dependences and skewed hyperplane

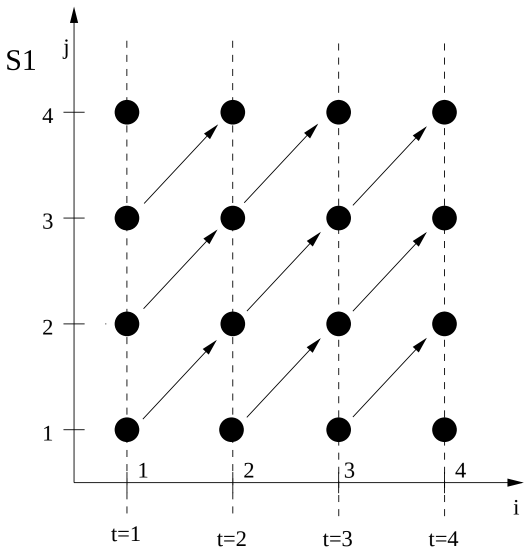


Figure 7.2 – Uniform dependences and orthogonal hyperplane

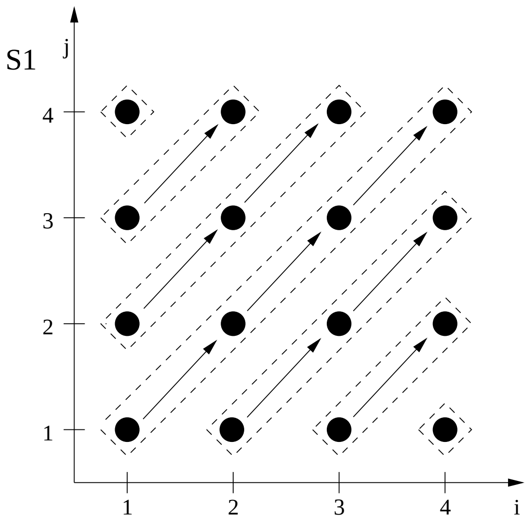


Figure 7.3 – Uniform dependences and outermost parallel hyperplane

```
for (i = 1; i < N; i++)
   for (j = 1; j < N; j++)
S1: A[i][j] = A[i-1][j-1] + X;
```

Figure 7.4 – Simple uniform dependence

```
for (i = 1; i < N; i++)
  for (j = 1; j < N; j++)
S1: A[i][j] = A[i][j-1] + A[i-1][j];
```

Figure 7.5 – Two uniform dependences

intersection of all the coefficient constraints, for all the dependence edges of the graph $G = (V, E)$:

$$\mathcal{L} = \bigcap_{\forall e \in E} \mathcal{L}_{\mathcal{P}_e} \tag{7.7}$$

7.6.1 Multidimensional schedules

One-dimensional schedules are not always obtainable. In practice, there are many codes for which the set of legal coefficients for the one-dimensional schedules is empty. As shown by Feautrier [68], one needs to extend the notion of one-dimensional schedules with the multidimensional schedules. It has been proven [68] that there exists a multidimensional schedule for each dependence graph induced by the loop nests coming from imperative programs. Indeed, the multidimensional schedule corresponding to the original loop nest is always legal, and it is one of the possible solutions.

Having the one-dimensional schedule, it is always mandatory to strongly satisfy all the dependences at the first (and only) dimension. Having the multidimensional schedule, there is a combinatorial choice of the depth of the dimension that strongly satisfies the dependence.

Feautrier decouples the problem of selecting the dependence satisfaction depth from forming the set of feasible schedule coefficient values. By introducing the $0-1$ variable $x_k^e \in \{0, 1\}$, one can model whether the given dependence edge $e \in E$ is satisfied weakly ($x_k^e = 0$) or strongly ($x_k^e = 1$):

$$\forall (\mathbf{i}, \mathbf{j}) \in \mathcal{P}_e : \theta_k^{S_j}(\mathbf{j}) - \theta_k^{S_i}(\mathbf{i}) \geq x_k^e$$

Each dependence edge $e \in E$ has to be satisfied strongly at some depth l. Let us consider the set of scheduling hyperplanes $\theta_k^{S_1}, \ldots, \theta_k^{S_p}$ at different dimensions, $k \in 1..m$. If the dependence edge e is satisfied strongly ($x_l^e = 1$) at depth l, this edge could be removed from the system of constraints when considering deeper levels $k > l$. Putting it more formally, for a given edge $e : S_i \to S_j$, the following is a sufficient condition for the legality of the multidimensional schedule w.r.t. the dependence edge e:

$$\begin{aligned} &\exists 1 \leq l \leq m, x_l^e = 1 \\ &\wedge \forall k < l, x_k^e = 0 \\ &\wedge \forall k \leq l, \forall (\mathbf{i}, \mathbf{j}) \in \mathcal{P}_e, \theta_k^{S_j}(\mathbf{j}) - \theta_k^{S_i}(\mathbf{i}) \geq x_k^e \end{aligned} \tag{7.8}$$

This reasoning comes from the simple fact on lexicopositivity of the difference of the timestamp vectors [68].[4]

The fact that there is a combinatorial choice of the weak/strong satisfaction for each edge $e \in E$ and for each scheduling level $1 \leq k \leq m$ has led to the heuristics that try to structuralize this decision problem. Feautrier [68] has proposed to maximize the number of strongly satisfied dependences at each level:

$$max \sum_{e \in E} x_k^e$$

4. when comparing the two vectors **a** and **b** lexicographically, it is sufficient to find the first position l where $a_0 = b_0, \ldots, a_{l-1} = b_{l-1}, a_l < b_l$, while the ordering of the remaining positions is irrelevant.

. This is the greedy approach that tries to satisfy as much dependences as possible in the outer scheduling dimensions, and to expose the parallelism in the inner loops. Also, it aims at multidimensional schedules that have the minimal dimensionality.

Bondhugula [34, 32] aims at different objective: that of exposing the outer loop parallelism. Thus, the strong dependence satisfaction is moved to the higher dimensions (inner loops), and the parallelism and tiling opportunities are exposed in the outermost loop levels.

As an example of the impact of the choice of dependence satisfaction (strong/weak), let us again take an example in Figure 7.4. If we consider the first dimension of the multidimensional schedule, after applying the Farkas lemma and simplifying, the constraint is:

$$a^{S_1}_{1,1} + a^{S_1}_{1,2} \geq x^e_1, x^e_1 \in \{0,1\}$$

There is a choice between satisfying the dependence strongly $x^e_1 = 1$ or weakly $x^e_1 = 0$. According to Feautrier's heuristic, one maximizes the number of the strongly satisfied dependences at each level, thus $x^e_1 = 1$. This leads to the constraint $a^{S_1}_{1,1} + a^{S_1}_{1,2} \geq 1$, which resolves the schedule within the first dimension. This enables the innermost loop parallelism, since the second scheduling dimensions is completely unconstrained (after strong satisfaction at the first dimension, the dependence edge is removed from consideration at deeper levels).

Figure 7.2 shows the minimum latency orthogonal hyperplanes $\theta^{S_1}_1 = i$ providing the first scheduling dimension. The iterations belonging to the same scheduling hyperplane could be executed in any order - in parallel in particular.

With Bondhugula's approach [34] the goal is to minimize the *dependence distance* starting from the outermost scheduling dimensions. This translates to the heuristic which prefers to weakly satisfy the dependence at the outer levels and to strongly satisfy the dependence at the inner levels. This leads to the constraint $a^{S_1}_{1,1} + a^{S_1}_{1,2} \geq 0$. In particular, it is preferable to obtain the outer parallel enabling scheduling hyperplane: $a^{S_1}_{1,1} + a^{S_1}_{1,2} = 0$. Since the first scheduling dimension is not strongly satisfying the edge, the satisfaction has to occur at deeper dimension. This forces the choice of the strong satisfaction at the second scheduling level $a^{S_1}_{2,1} + a^{S_1}_{2,2} \geq 1$.

By assigning the values to the scheduling coefficients for the first level, one gets a possible parallel enabling scheduling hyperplane $\theta^{S_1}_1 = i - j$ ($a^{S_1}_{1,1} = 1, a^{S_1}_{1,2} = -1$). For the second dimension, the scheduling hyperplane is $\theta^{S_1}_2 = i$ ($a^{S_1}_{2,1} = 1, a^{S_1}_{2,2} = 0$). This multidimensional schedule enables the *outermost parallelism* - the outermost loop is a DOALL loop. This is depicted in Figure 7.3.

Depending on the underlying architecture, either the inner parallelism (Figure 7.2) or outer parallelism (Figure 7.3) might be more beneficial.

All the mentioned approaches stick to the one scheme of selecting the dimensions that are to be strongly satisfied. A new approach of Pouchet et al. [132] tries to express the convex space of all the possible choices for the x^e_k variables and to enumerate those choices. While this seems a promising approach, the amount of the possible solutions is prohibitively huge.

In the later sections of this chapter we will show the approach of enumerating the possible decisions in a specially designed binary decision tree. This enables us to enumerate the different decisions, while still maintaining the reasonable problem sizes.

7.7 Discrete sets of one-dimensional legal solutions

While the previously mentioned affine scheduling algorithms consider a very expressive search space of all legal affine transformations, it is not always necessary nor desirable to consider the full space of affine transformations. What is more, Feautrier's affine scheduler [67, 68] or Bondhugula's [34]

Transformation name	Matrices involved
Interchange, skewing (unimodular), reversal	A
Strip-mining, tiling	D, θ
Pipelining (multidimensional)	β
Parametric pipelining (multidimensional)	Γ
Reversal (unimodular)	A, Γ
Motion, fusion, fission (distribution)	β

Table 7.1 – Classical loop transformations and their polyhedral matrix representation

combined parallelism and memory locality heuristic are *best effort* heuristics - they provide one and only one solution chosen by some cost function.

As evidenced by Pouchet [131], the best effort heuristics are not well adapted to different architectural features - the optimal solution according to heuristics might be optimal for one architecture and suboptimal for other architecture. The other problem is

7.7.1 State of the art

We take the idea of Darte [54] stating that arbitrary affine transformations are not always desirable. Narrowing down the scope of affine transformations might result in the better control over the generated code.

Our goal is to enable the exhaustive search of the possible affine transformations while still narrowing down the search space to the reasonable size. We have already shown in Section 7.6 that the space of all the legal coefficients for affine transformation might be unbounded.

Bounding the coefficients to the $[-1,1]$ interval was proposed by Pouchet [129, 128] in his work on iterative compilation within the polyhedral model.

Pouchet also shows [129] that some coefficients of the affine transformation are less critical than others. This fact was investigated empirically by assessing the impact of different transformation coefficients on the performance of the transformed output program. That fact has led to the *decoupled* heuristic [129] - first the coefficients corresponding to the loop induction variables are found, and later the rest of the coefficients are completed.

Girbal has shown [74] that the different parts of the canonical scheduling matrix format correspond to different *classical* loop optimizations [7]. This is summarized in Table 7.1.

7.7.2 Narrowing the search space

If one wants to enumerate all the possible transformations within an affine transformation space, one has to bound the values of the transformation coefficients.

Instead of bounding the values of the coefficients of affine transformation, as done by Pouchet [129], we restrict the possible space of transformations right away at the beginning of our search. We restrict our search space to the following transformations:

– loop permutation
– non-parametric shift
– motion, fusion, distribution

Strip-mining and tiling transformations are treated specially and they are applied afterwards. Restricting the search space to loop permutations only, we do not look for full unimodular transformation space (skewing, loop reversal). This enables us to restrict the search space considerably, but the resulting search space is non-convex.

```
for (i = 1; i < N; i++)
  for (j = 0; j < N; j++)
S1: A[i][j] = B[i-1][j];
S2: B[i][j] = A[i-1][j];
```

Figure 7.6 – Two statements, uniform dependences

The motivation would be clear after following the subsequent sections. Loop permutation defines only the *orthogonal* hyperplanes. While this decision might restrict some possible transformations (like loop skewing for extracting fine grained parallelism) it is powerful enough for the purpose of finding outer and inner loop parallelism in many cases. The problem is similar to Kennedy's [5] *loop selection problem* but it is expressed in the much more general framework of the polyhedral model: we support the non-perfectly loop nests, non-uniform dependences, shifting for transformation correction and integrate this into the loop distribution/fusion decisions.

7.7.3 Building non-convex sets of one-dimensional legal schedules

We have to take into an account the restrictions on the possible schedules that we have imposed up-front when considering the affine scheduling functions. Considering the A part of the scheduling matrix for each statement S_i, for each row k we constrain the scheduling coefficients to the following form:

$$\sum_{s=1}^{m} a_{k,s}^{S_i} = 1, a_{k,s}^{S_i} \in \{0,1\} \tag{7.9}$$

In other words, each row is composed of all zeros except one position - say s - that contains an integer constant 1. This corresponds to selecting the loop s [5] at the scheduling level k.

Let us consider an example in Figure 7.4. The general form of the affine scheduling function for the statement S_1 is:

$$\theta_1^{S_1} = a_{1,1}^{S_1} i + a_{1,2}^{S_1} j + \omega_1^{S_1}$$

A respective affine scheduling legality constraint (shown in Section 7.6) is $a_{1,1}^{S_1} + a_{1,2}^{S_1} \geq 1$. Taking into an account the constraints we have imposed in Equation 7.9, we get the following table of the possible solutions:

solution no.	$a_{1,1}^{S_1}$	$a_{1,2}^{S_1}$	$\omega_1^{S_1}$
1.	0	1	unconstrained
2.	1	0	unconstrained

The solution is trivial: there are only two possible loop interchanges and all of them are legal. The shifting factor $\omega_1^{S_1}$ is unconstrained [5]. The solution set is non-convex. The actual solution set is the enumeration of legal loop permutations alongside with legal shifting factors for each selection of loop permutations.

Let us take a more complex example, containing two statements, S_1 and S_2, and two dependence edges $e_1 : S_1 \rightarrow S_2$ and $e_2 : S_2 \rightarrow S_1$. The dependence polyhedra describing those two dependence edges are the following:

$\mathcal{P}_{e_1} = \{(i,j,i',j') | i' = i+1 \wedge j' = j \wedge 1 \leq i \leq N-2 \wedge 0 \leq j \leq N-1\}$
$\mathcal{P}_{e_2} = \{(i,j,i',j') | i' = i+1 \wedge j' = j \wedge 1 \leq i \leq N-2 \wedge 0 \leq j \leq N-1\}$

5. The shifting transformation expresses the relative shift of two statements. It makes sense only if we have more than one statement. In the case of a single statement we will assume $\omega_1^{S_1} = 0$.

$a^{S_1}_{1,1}$	$a^{S_1}_{1,2}$	$a^{S_2}_{1,1}$	$a^{S_2}_{1,2}$	convex part
1	0	1	0	$\omega^{S_2}_1 - \omega^{S_1}_1 \geq 0$
0	1	0	1	$\omega^{S_2}_1 - \omega^{S_1}_1 \geq 1$

Table 7.2 – Solution set $\mathcal{L}^{e_1}$

$a^{S_1}_{1,1}$	$a^{S_1}_{1,2}$	$a^{S_2}_{1,1}$	$a^{S_2}_{1,2}$	convex part
1	0	1	0	$\omega^{S_1}_1 - \omega^{S_2}_1 \geq 0$
0	1	0	1	$\omega^{S_1}_1 - \omega^{S_2}_1 \geq 1$

Table 7.3 – Solution set $\mathcal{L}^{e_2}$

The prototype schedules from both statements are:

$$\theta^{S_1}_1 = a^{S_1}_{1,1} i + a^{S_1}_{1,2} j + \omega^{S_1}_1 \tag{7.10}$$

$$\theta^{S_2}_1 = a^{S_2}_{1,1} i + a^{S_2}_{1,2} j + \omega^{S_2}_1 \tag{7.11}$$

The legality condition is expressed as:
$\theta^{S_2}_1(i',j') - \theta^{S_1}_1(i,j) \geq 1$

Since we have the uniform dependences, we can substitute $i' = i+1$ and $j' = j+1$ in the legality condition expression. After simplification we get:
$(a^{S_2}_{1,1} - a^{S_1}_{1,1})i + (a^{S_2}_{1,2} - a^{S_1}_{1,2})j + \omega^{S_2}_1 - \omega^{S_1}_1 + a^{S_2}_{1,1} \geq 0$

After applying Farkas lemma (the details could be found in Appendix A), we get the following system of constraints on the scheduling coefficients:

$$\begin{cases} a^{S_2}_{1,1} - a^{S_1}_{1,1} \geq 0 \\ a^{S_2}_{1,2} - a^{S_1}_{1,2} \geq 0 \\ \omega^{S_2}_1 - \omega^{S_1}_1 + 2a^{S_2}_{1,1} - a^{S_1}_{1,1} - 1 \geq 0 \end{cases}$$

By enumerating the possible values of the coefficients that in addition satisfy the Equation 7.9, we get the set of solutions shown in Figure 7.2.

The solution set is now split into the non-convex part and convex part. The non-convex part corresponds to the enumeration of the legal loop selections - encoded within A part of the scheduling matrix. For each legal assignment of the $a^{S_i}_{1,k}$ coefficients, the remaining constraints on the *shifting* part $\omega^{S_i}_1$ are expressed in the convex polyhedron form.

The similar solution set is obtained for the dependence edge $e_2 : S_2 \to S_1$. The solution set is shown in Table 7.3.

While the solution sets represent the space of legal coefficients for single dependence edges, the aim is to obtain the global solution, as in Equation 7.7. For that purpose, the intersection of the solution sets has to be provided. After combining the solution sets $\mathcal{L}^{e_1}$ and $\mathcal{L}^{e_2}$, we obtain the global solution $\mathcal{L}$ shown in Table 7.4.

$a^{S_1}_{1,1}$	$a^{S_1}_{1,2}$	$a^{S_2}_{1,1}$	$a^{S_2}_{1,2}$	convex part
1	0	1	0	$\omega^{S_2}_1 - \omega^{S_1}_1 + 1 \geq 0 \wedge \omega^{S_1}_1 - \omega^{S_2}_1 + 1 \geq 0$
0	1	0	1	$\omega^{S_2}_1 - \omega^{S_1}_1 \geq 0 \wedge \omega^{S_1}_1 - \omega^{S_2}_1 \geq 0$

Table 7.4 – Solution set $\mathcal{L} = \mathcal{L}^{e_1} \bigcap \mathcal{L}^{e_2}$

$A^{S_1}_{1,\bullet}$	$A^{S_2}_{1,\bullet}$	convex part
1	1	$\omega_1^{S_2} - \omega_1^{S_1} \geq 0$
2	2	$\omega_1^{S_2} - \omega_1^{S_1} \geq 1$

Table 7.5 – Compressed encoding of the set $\mathcal{L}^{e_1}$

$A^{S_1}_{1,\bullet}$	$A^{S_2}_{1,\bullet}$	convex part
1	1	$\omega_1^{S_2} - \omega_1^{S_1} \geq 0$
2	2	$\omega_1^{S_1} - \omega_1^{S_2} \geq 1$

Table 7.6 – Compressed encoding of the set $\mathcal{L}^{e_2}$

Representing convex rational and integer solution sets is a well studied problem in convex optimization theory [42, 143].

Omega [133] library uses Fourier-Motzkin elimination extended to integer linear programming and integer set operations. Several efficient implementations based on *dual representation* [143] have been proposed. The most popular one was implemented in Polylib, the work of Le Verge and Wilde [160]. It is based on Chernikova's algorithm for transforming between constraint representation (the set of affine constraints) and generator representation(set generated by a set of vertices, rays and lines) [42, 100].

The most recent developments are PPL [12] and ISL [158] libraries for manipulating the convex sets of rational and integer solutions respectively.

7.8 Non-convex solution sets

As we have shown in the previous section, we split the solution set into the non-convex and convex part. For representing convex sets, we use techniques that are already well developed.

While the representation of the convex sets has sound mathematical and algorithmic background, the representation of the non-convex sets poses more complexity problems. Each non-convex problem is particular and thus requires dedicated techniques with heuristics well adapted to the problem.

Following is the work that describes our approach to representing the non-convex set of all the legal loop selections for a single dimension. In a Section 7.9 we will extend this to multidimensional solutions.

Representing non-convex solutions

We propose to use *decision trees* as a representation of non-convex solution sets. We have imposed a constraint on the possible set of coefficient values for the row Θ_k^S of A part of the scheduling matrix of statement S in Equation 7.9. A row is composed of all zeros except single integer constant 1 at some position s. Instead of representing the full row, we can only represent the position s, $1 \leq s \leq dim(S)$ of the integer constant 1 in a row.

Taking the solution set shown in Table 7.2, we encode the legal solution set as shown in Table 7.5. The same is done for the solution set $\mathcal{L}^{e_2}$, as shown in Table 7.6.

After encoding the full matrix row as single integer value, we construct the decision tree representing the solution sets. Figures 7.7 and 7.8 show the respective n-ary decision trees.

Even though one might use a simple representation of the sets, such as linked lists or arrays, we propose to use the n-ary decision trees for that purpose. The reason is the fact that we will require the set operations to be performed efficiently. A tree structure enables us to achieve this goal.

The leaf node $\perp$ represents a 'no solution' - a given selection of the decision variables gives an illegal schedule. The rest of the leaf nodes represent a legal solutions and they are labelled by the polyhedron

$$\begin{tikzcd}
 & A^{S_1}_{1,\bullet} \arrow[dl, "1"'] \arrow[dr, "2"] & \\
A^{S_2}_{1,\bullet} \arrow[d, "1"'] \arrow[dr, "2"] & & A^{S_2}_{1,\bullet} \arrow[dl, "1"'] \arrow[d, "2"] \\
\omega^{S_2}_1 - \omega^{S_1}_1 \geq 0 & \perp & \omega^{S_2}_1 - \omega^{S_1}_1 \geq 1
\end{tikzcd}$$

Figure 7.7 – n-ary decision tree representing the solution set $\mathcal{L}^{e_1}$

$$\begin{tikzcd}
 & A^{S_1}_{1,\bullet} \arrow[dl, "1"'] \arrow[dr, "2"] & \\
A^{S_2}_{1,\bullet} \arrow[d, "1"'] \arrow[dr, "2"] & & A^{S_2}_{1,\bullet} \arrow[dl, "1"'] \arrow[d, "2"] \\
\omega^{S_1}_1 - \omega^{S_2}_1 \geq 0 & \perp & \omega^{S_1}_1 - \omega^{S_2}_1 \geq 1
\end{tikzcd}$$

Figure 7.8 – n-ary decision tree representing the solution set $\mathcal{L}^{e_2}$

representing the legal shifting factors - a convex part of the solution set.

At each depth k of the tree, we consider the nodes of the form $A^{S_k}_{1,\bullet}$ that represent the decision variables that encode the possible rows of the A part of the scheduling matrix for each statement S_k. A given node $A^{S_k}_{1,\bullet}$ might have up to $dim(S_k)$ *legal* child nodes. The legal child node is the node that does not lead to the $\perp$ terminating node.

Initially, we represent the sets for the pairs of statements only, so each tree would have a depth of two. When combining the trees into the global solution, we will obtain a tree that encodes the global schedule with a depth p, where p is the number of the statements whose schedules we are constructing. We will now show how to perform the basic set operations on n-ary decision treesthat we construct.

7.8.1 Efficient operations on solution sets

Once the solution sets are represented as n-ary decision trees, there is a need to provide the basic operators on those solution sets. The basic operators that we are interested in are:

1. EMBED$(\mathcal{L}, X)$
2. INTERSECT$(\mathcal{L}', \mathcal{L}'')$
3. PROJECT$(\mathcal{L}, X)$
4. ORTHOGONAL$(E', \mathcal{L}, k)$

We will discuss the first three of them in this subsection. The last operator - ORTHOGONAL- will be discussed in Section 7.9.

Embed operator

Each decision tree have a set of decision variables $\{X1, \ldots, Xn\}$. For example, the decision diagram in Figure 7.7 has two decision variables: $A^{S_1}_{1,\bullet}$ and $A^{S_2}_{1,\bullet}$. In order to perform an intersection operation

$$\begin{tikzcd}
 & A^{S_0}_{1,\bullet} \arrow[d, bend right, "1"'] \arrow[d, bend left, "2"] & \\
 & A^{S_1}_{1,\bullet} \arrow[dl, "1"'] \arrow[dr, "2"] & \\
A^{S_2}_{1,\bullet} \arrow[d, "1"'] \arrow[dr, "2"] & & A^{S_2}_{1,\bullet} \arrow[dl, "1"'] \arrow[d, "2"] \\
\omega^{S_2}_1 - \omega^{S_1}_1 \geq 0 & \perp & \omega^{S_2}_1 - \omega^{S_1}_1 \geq 1
\end{tikzcd}$$

Figure 7.9 – The result of the operator EMBED$(\mathcal{L}^{e_1}, \{A^{S_0}_{1,\bullet}\})$

on two sets represented as n-ary decision trees it is mandatory that they have the same sets of decision variables.

Given a decision tree with a set of variables $\{X1, \ldots, Xn\}$, an embed operation EMBED extends this set with one decision variable, say X_{n+1}, while not changing the elements contained in the original set.

As an example, let us take the original set in Figure 7.7. Figure 7.9 shows the result of the embed operation on the set $\mathcal{L}^{e_1}$ with a (hypothetical) decision variable $A^{S_0}_{1,\bullet}$. Now the decision tree is extended with one more decision variable - $A^{S_0}_{1,\bullet}$. The set of the solutions remains the same - the value of the $A^{S_0}_{1,\bullet}$ variable does not impact the set of valid results represented in a set.

The ordering of the decision variables has an impact on the size of the decision tree. Embedding the variable at the deeper levels increases the number of the nodes, since it requires the replication of that decision node.

Intersection operator

The intersection operator performs the set intersection. The result is the intersection of two solution sets, say $\mathcal{L}'$ and $\mathcal{L}''$, so that the resulting set $\mathcal{L} = \mathcal{L}' \cap \mathcal{L}''$ contains the solutions that are both legal in $\mathcal{L}'$ and in $\mathcal{L}''$.

The operator requires that both arguments contain the same set of decision variables and that their ordering is the same. If this is not the case, then the EMBED is used to introduce the common decision variables in both sets.

The intersection operation $\mathcal{L}' \cap \mathcal{L}'' = \text{INTERSECT}(\mathcal{L}', \mathcal{L}'')$ proceeds by recursively following the decision variable nodes from top to bottom for both arguments and finding the set of common terminating nodes. For those terminating nodes that have legal solutions, the intersection of convex sets is found.

An example is shown in Figure 7.12 which is the result of applying the intersection operator on trees shown in Figures 7.10 and 7.11. A common set of legal terminating nodes for sets $\mathcal{L}'$ and $\mathcal{L}''$ is the terminating node for $X_1 = 2, X_2 = 2$. Subsequently, the common terminating node is labelled by the intersection of convex sets $\mathcal{P}_1$ and $\mathcal{P}_3$.

Projection operator

The projection operator PROJECT$(\mathcal{L}, X)$ takes the decision tree $\mathcal{L}$ containing the decision variable X and removes that decision variable from the representation while keeping the set of legal solutions

$$\begin{tikzcd}
& X_1 \arrow[dl, "1"'] \arrow[dr, "2"] & \\
X_2 \arrow[d, "1"'] \arrow[dr, "2"] & & X_2 \arrow[dl, "1"'] \arrow[d, "2"] \\
\mathcal{P}_1 & \perp & \mathcal{P}_2
\end{tikzcd}$$

Figure 7.10 – set $\mathcal{L}'$

$$\begin{tikzcd}
& X_1 \arrow[dl, "1"'] \arrow[dr, "2"] & \\
X_2 \arrow[d, "1"'] \arrow[dr, "2"] & & X_2 \arrow[dl, "1"'] \arrow[d, "2"] \\
\perp & \perp & \mathcal{P}_3
\end{tikzcd}$$

Figure 7.11 – set $\mathcal{L}''$

$$\begin{tikzcd}
& X_1 \arrow[dl, "1"'] \arrow[dr, "2"] & \\
X_2 \arrow[d, "1"'] \arrow[dr, "2"] & & X_2 \arrow[dl, "1"'] \arrow[d, "2"] \\
\perp & \perp & \mathcal{P}_3 \cap \mathcal{P}_2
\end{tikzcd}$$

Figure 7.12 – INTERSECT$(\mathcal{L}', \mathcal{L}'')$

$$\begin{tikzcd}
& X_1 \arrow[dl, "1"'] \arrow[dr, "2"] & \\
X_2 \arrow[d, "1"'] \arrow[dr, "2"] & & X_2 \arrow[dl, "1"'] \arrow[d, "2"] \\
\mathcal{P}_1 & \perp & \mathcal{P}_2
\end{tikzcd}$$

Figure 7.13 – set $\mathcal{L}$

$$\begin{tikzcd}
& X_2 \arrow[dl, "1"'] \arrow[dr, "2"] & & & X_2 \arrow[dl, "1"'] \arrow[dr, "2"] & \\
\mathcal{P}_1 & & \perp & \perp & & \mathcal{P}_2
\end{tikzcd}$$

Figure 7.14 – Split tree after removing X_1 node

selected by remaining decision variables [6].

While removing the decision variable nodes corresponding to the variable X that is projected out, the decision tree is split into several subsets. To obtain the final solution, the union of those subsets is taken and represented with the remaining decision variables.

As an example, let us take the set in Figure 7.13. The goal is to project out the X_1 variable. After removing the node corresponding to decision variable X_1, the tree is split in two parts shown in Figure 7.14. Those two parts are combined into the final solution by taking their union.

6. This operation is the 'non-convex' counterpart of the projection operation performed on convex sets.

$$\begin{tikzcd}
& X_2 \arrow[dl, "1"'] \arrow[dr, "2"] & \\
\mathcal{P}_1 & & \mathcal{P}_2
\end{tikzcd}$$

Figure 7.15 – The final result of PROJECT$(\mathcal{L}, X_1)$

```
Procedure CombineInitialSolutions
INPUT: $E$
OUTPUT: $\mathcal{L}$

$E$ - a subset of the edges required to be satisfied either weakly or strongly at scheduling level one

1. $\mathcal{L} \leftarrow \emptyset$
2. for all $e \in E$
     – compute $\mathcal{L}^e$
     – $\mathcal{L} \leftarrow$ Intersect$(\mathcal{L}', \mathcal{L}'')$
```

Figure 7.16 – Combining the solutions

A note on complexity

By splitting the solution set representation into the non-convex part represented by n-ary decision treesand convex part, operated on by using the regular convex optimization theory, we have a better control over the complexity of operations on those sets.

Initially we build $|E|$ decision trees representing the sets $\mathcal{L}^e, \forall e \in E$. Each such a decision tree represents the legal solutions w.r.t to one dependence edge. Those trees have the depth of two - one level for each statement. Given an edge $e: S_i \rightarrow S_j$, the tree that represents the legal solutions for this dependence might have at most $dim(S_i) \cdot dim(S_j)$ nodes. Each node might be labelled by one convex polyhedron. The worst case is rarely attainable, even in the case of simple uniform dependences.

Having the full dependence graph, one needs to build $|E|$ decision trees. Let d be the maximal depth of the statement in a program. The worst case complexity of building the constraints for all the dependence edges is $O(|E| \cdot d^2)$.

In order to build the global solution, one needs to combine all those pairwise constraints into the global system. This is achieved by performing the sequence of intersect operations, which in turn might require an embedding operation. This simple scheme is shown in the form of pseudo-code in Figure 7.16.

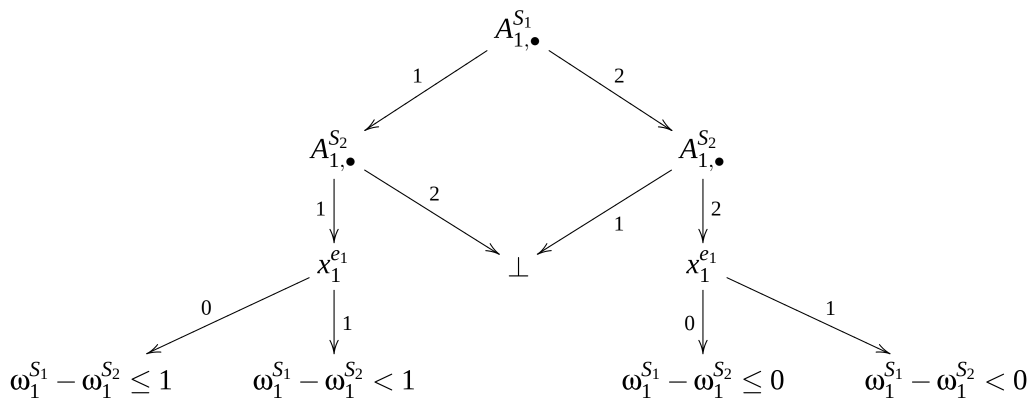
The complexity of the operations is sensitive to the order in which the intersection operations are performed. The most desirable is to intersect two sets, say $\mathcal{L}', \mathcal{L}''$, that have exactly the same set of decision variables - no embedding operation is needed in this case. The complexity of such an intersection is $O(n_1 n_2)$, where n_1 and n_2 are the number of tree nodes.

7.9 Sets of multi-dimensional legal solutions

So far we have discussed the representation of one-dimensional solution sets. While the one-dimensional solution sets are the starting point for the exploration of transformations, the ultimate goal is to provide the set of legal multidimensional solutions, since one-dimensional schedules are not possible for some class of the programs.

7.9.1 Representing weak/strong satisfaction

When we were considering the sets of one-dimensional schedules only, the strong satisfaction is required in the first dimension - all the dependences have to be satisfied within the first and only dimension. If we extend the set of solutions to multidimensional schedules, the combinatorial problem becomes more complex. There is a combinatorial choice of whether to strongly or weakly satisfy the dependence edges $e \in E$ at each scheduling level.



$$\begin{tikzcd}
 & & & A^{S_1}_{1,\bullet} \arrow[dll, "1"'] \arrow[drr, "2"] & & & \\
 & A^{S_2}_{1,\bullet} \arrow[d, "1"'] \arrow[drr, "2"] & & & & A^{S_2}_{1,\bullet} \arrow[d, "2"] \arrow[dll, "1"'] & \\
 & x^{e_1}_1 \arrow[dl, "0"'] \arrow[d, "1"] & & \perp & & x^{e_1}_1 \arrow[d, "0"'] \arrow[dr, "1"] & \\
\omega^{S_1}_1 - \omega^{S_2}_1 \leq 1 & \omega^{S_1}_1 - \omega^{S_2}_1 < 1 & & & & \omega^{S_1}_1 - \omega^{S_2}_1 \leq 0 & \omega^{S_1}_1 - \omega^{S_2}_1 < 0
\end{tikzcd}$$

Figure 7.17 – the solution set $\mathcal{L}^{e_1}$ with decision variable $x^{e_1}_1$

We have shown in Subsection 7.6.1 the different heuristics considering this problem. Those heuristics try to find the best weak/strong satisfaction choice for each scheduling level at once. We want to take the different approach: we implicitly enumerate all those decisions in the decision tree of legal solutions.

For that purpose we introduce a decision variable x^e_d for each dependence edge $e \in E$ at each scheduling level d. The meaning of the decision variable is the same as used in Feautrier's multidimensional scheduling [68]: the decision is made whether a given edge is *weakly satisfied*($x^e_d = 0$) or *strongly satisfied*($x^e_d = 1$)

Taking those definitions into an account, we build a decision tree for the first scheduling dimension in the same way as we did in 7.8 while adding a decision variable x^e_d that selects the convex part of the solution to correspond to strict weak/strong satisfaction.

The original decision tree that unconditionally enforces strong satisfaction on the edge e_1 was shown in Figure 7.7. If we now split the dependence satisfaction decision into weak and strong satisfaction (variable $x^{e_1}_k$), we get the decision tree as shown in Figure 7.17. The leaf nodes are labelled by convex sets of the legal shifting factors giving the desired edge satisfaction condition (weak/strong). The same splitting is done for edge e_2, as shown in Figure 7.18.

Combining solutions

In order to obtain the global solution for the first scheduling dimension, one has to intersect the two solution sets for edges e_1 and e_2. But the decision trees for those two sets have two different decision variables: $x^{e_1}_k$ and $x^{e_1}_k$ respectively. Thus, the result of the intersection operation combines both of them (using embedding operator) into the final solution shown in Figure 7.19.

The global solution set $\mathcal{L}_1$ is the exhaustive enumeration of all the possible one-dimensional schedules within the space defined in Subsection 7.7.2. The decision variables $x^{e_1}_1$ and $x^{e_2}_1$ partition the space of convex sets of legal shifting factors giving different edge satisfaction. Some interesting properties of the solutions contained in this set are summarized in Table 7.7.

7.9.2 Orthogonal completion

As shown in Table 7.7, some one-dimensional solution sets do not satisfy all the dependence edges at the first scheduling dimensions. What is more, in order to provide *permutable loop band* we would like to enforce the satisfaction of all edges at deeper levels, regardless of whether they have been satisfied at earlier levels or not.

$$\begin{tikzcd}
 & & A^{S_1}_{1,\bullet} \arrow[dl, "1"'] \arrow[dr, "2"] & & \\
 & A^{S_2}_{1,\bullet} \arrow[d, "1"'] \arrow[dr, "2"] & & A^{S_2}_{1,\bullet} \arrow[dl, "1"'] \arrow[d, "2"] & \\
 & x^{e_2}_1 \arrow[dl, "0"'] \arrow[d, "1"] & \perp & x^{e_2}_1 \arrow[d, "0"'] \arrow[dr, "1"] & \\
\omega^{S_1}_1 - \omega^{S_2}_1 \geq -1 & \omega^{S_1}_1 - \omega^{S_2}_1 > -1 & & \omega^{S_1}_1 - \omega^{S_2}_1 \geq 0 & \omega^{S_1}_1 - \omega^{S_2}_1 > 0
\end{tikzcd}$$

Figure 7.18 – the solution set $\mathcal{L}^{e_2}$ with decision variable $x^{e_2}_1$

$$\begin{tikzcd}
 & & & A^{S_1}_{1,\bullet} \arrow[dl, "1"'] \arrow[dr, "2"] & & \\
 & & A^{S_2}_{1,\bullet} \arrow[d, "1"'] \arrow[dr, "2"] & & A^{S_2}_{1,\bullet} \arrow[dl, "1"'] \arrow[d, "2"] & \\
 & & x^{e_1}_1 \arrow[dl, "0"'] \arrow[d, "1"] & \perp & x^{e_1}_1 \arrow[d, "0"'] \arrow[ddr, "1"] & \\
 & x^{e_2}_1 \arrow[dl, "0"'] \arrow[d, "1"] & x^{e_2}_1 \arrow[d, "0"'] \arrow[dr, "1"] & & x^{e_2}_1 \arrow[d, "0"'] \arrow[dr, "1"] & \\
-1 \leq \omega^{S_1}_1 - \omega^{S_2}_1 \leq 1 & -1 < \omega^{S_1}_1 - \omega^{S_2}_1 \leq 1 & -1 \leq \omega^{S_1}_1 - \omega^{S_2}_1 < 1 & -1 < \omega^{S_1}_1 - \omega^{S_2}_1 < 1 & \omega^{S_1}_1 - \omega^{S_2}_1 = 0 & \perp
\end{tikzcd}$$

Figure 7.19 – The global solution set $\mathcal{L}_1$ with decision variables $x^{e_1}_1$ and $x^{e_2}_1$

$x^{e_1}_1$	$x^{e_2}_1$	non-convex	convex part	comment
$= 0$	$= 0$	$A^{S_1}_{1,\bullet} = (01)$ $A^{S_2}_{1,\bullet} = (01)$	$\omega^{S_1}_1 - \omega^{S_2}_1 = 0$	Both edges strictly weakly satisfied. Enables outermost DOALL loop.
> 0	> 0	$A^{S_1}_{1,\bullet} = (10)$ $A^{S_2}_{1,\bullet} = (10)$	$-1 < \omega^{S_2}_1 - \omega^{S_1}_1 < 1$	Both edges strongly satisfied at level 1. Enables innermost DOALL loop. Does not require further scheduling dimension.
$= 0$	> 0	$A^{S_1}_{1,\bullet} = (10)$ $A^{S_2}_{1,\bullet} = (10)$	$\omega^{S_1}_1 - \omega^{S_2}_1 = 1$	Edge e_1 strictly weakly satisfied, edge e_2 strongly satisfied. Schedule is not complete at level 1, requires completion at second level.
$=>$	$= 0$	$A^{S_1}_{1,\bullet} = (10)$ $A^{S_2}_{1,\bullet} = (10)$	$\omega^{S_2}_1 - \omega^{S_1}_1 = 1$	Edge e_2 strictly weakly satisfied, edge e_1 strongly satisfied. Schedule is not complete at level 1, requires completion at second level.

Table 7.7 – Summary of the set of one-dimensional schedules

$$A^{S_1} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Figure 7.20 – Full ranked A matrix

$$A^{S_1} = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$$

Figure 7.21 – Non-full ranked A matrix

$$A^{S_1} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad A^{S_2} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\omega_1^{S_1} - \omega_1^{S_2} = 0$$
$$-1 < \omega_2^{S_2} - \omega_2^{S_1} < 1$$

Figure 7.22 – Complete solution obtained

In order to get the complete solution, one needs to complete the schedules at scheduling dimensions $1 < k \leq dim(S_i)$. The complete schedule has the full-ranked A part of the scheduling matrix [7]. In order to get the complete, full-ranked scheduling for each statement S_i, each statement needs to be provided with a full ranked A^{S_i} matrix, with the dimensionality of $dim(S_i) \times dim(S_i)$.

Bondhugula [34] is using the concept of orthogonal sub-spaces so that the solutions found at the following rows of the scheduling matrix are linearly independent w.r.t. rows that have already been found. This guarantees the full-ranked matrix.

In our search space, the problem of finding the independent solutions is easier. We restrict the rows of A matrix to satisfy Equation 7.9. If we have selected the position s of the integer constant 1 at the first row of the scheduling matrix, then this position could not be used in subsequent rows.

In general, if we have selected independent positions $s_1, \ldots, s_k$ for levels $1 \leq l \leq k$, those positions cannot be used when completing the subsequent rows $k+1 \leq l \leq dim(S_i)$. This is equivalent with stating that the A represents a loop permutation matrix. An example of the full ranked permutation matrix is shown in Figure 7.20. Obviously, the solution shown in Figure 7.21 is not a full-ranked multidimensional schedule.

Given the complete solution set for the first scheduling dimension represented as a decision tree, each valid leaf node could be completed with the next scheduling level that gives the independent solution. While it is possible to provide a completion for each leaf node of the decision tree, this procedure might lead to combinatorial explosion, since each subsequent level is a also a decision tree. In Section 7.10 we will discuss the systematic search methodology that enables us to control the complexity of completing the schedule at deeper scheduling levels.

If we want to complete the second scheduling dimension, we have to find a solution set that satisfies some subset of the dependence edges $E' \subset E$, while providing a row $A_{2,\bullet}^{S_i}$, for each statement S_i, that is linearly independent from a row $A_{1,\bullet}^{S_i}$. We will denote this operation as ORTHOGONAL$(E', \mathcal{L}, k)$, where E' is the subset of the edges that have to be satisfied at a new scheduling level k and $\mathcal{L}$ is the solution set of the solutions found at levels $1 \leq l \leq k-1$.

Let us take a decision tree shown in Figure 7.19 that represents the solution set $\mathcal{L}_1$. We want to complete the scheduling matrices by providing the second scheduling dimension. Let us take one of the solutions in the set $\mathcal{L}_1$ and provide an orthogonal completion for it. Figure 7.23 shows one chosen solution from the first dimension (emphasized with double arrows) and the orthogonal completion in the second dimension. After orthogonal completion in the second dimension, we have one complete solution:

If we now assign some concrete values for shifting factors: $\omega_1^{S_1} = 0, \omega_1^{S_2} = 0, \omega_2^{S_2} = 0, \omega_2^{S_1} = 0$, the complete solution is:

$\theta_1^{S_1} = j$, $\theta_2^{S_1} = i$

7. This is also necessary for correct code generation from the polyhedral model

$$\begin{tikzcd}
 & & A^{S_1}_{1,\bullet} \arrow[dl, "1"'] \arrow[dr, Rightarrow, "2"] & \\
 & A^{S_2}_{1,\bullet} \arrow[d, "1"'] \arrow[dr, "2"] & & A^{S_2}_{1,\bullet} \arrow[dl, "1"'] \arrow[d, Rightarrow, "2"] \\
 & x^{e_1}_1 \arrow[dl, "=0"'] \arrow[d, ">0"] & \perp & x^{e_1}_1 \arrow[d, Rightarrow, "=0"'] \\
x^{e_2}_1 \arrow[d, ">0"] & x^{e_2}_1 \arrow[d, ">0"'] \arrow[dr, "=0"] & & x^{e_2}_1 \arrow[d, Rightarrow, "=0"'] \\
\omega^{S_1}_1 - \omega^{S_2}_1 = 1 & -1 < \omega^{S_2}_1 - \omega^{S_1}_1 < 1 & \omega^{S_2}_1 - \omega^{S_1}_1 = 1 & \omega^{S_1}_1 - \omega^{S_2}_1 = 0 \arrow[d, Rightarrow, "\textsc{Orthogonal}(E=\{e_1,e_2\},\mathcal{L}_1,k=2)"'] \\
 & & & A^{S_1}_{2,\bullet} \arrow[dl, "1"'] \\
 & & A^{S_2}_{2,\bullet} \arrow[d, "1"'] & \\
 & & x^{e_1}_2 \arrow[d, ">0"] & \\
 & & x^{e_2}_2 \arrow[d, ">0"'] & \\
 & & -1 < \omega^{S_2}_2 - \omega^{S_1}_2 < 1 &
\end{tikzcd}$$

Figure 7.23 – The global solution set $\mathcal{L}_1$ with decision variables $x^{e_1}_1$ and $x^{e_2}_1$

$\theta^{S_2}_1 = j,\ \theta^{S_2}_2 = i$

This particular solution enables the outermost loop level parallelism (DOALL loop). This is only one of the possible solutions. The search method presented in the next Section enables the exploration of the full search space of multidimensional schedules.

7.10 Towards search methodology

In this section we will propose a search methodology based on decision tree set representation. The goal is to exhaustively explore the space of multidimensional affine transformations within the constraints defined in Subsection 7.7.2.

For that purpose, we will define some more operators on the decision trees and then we will propose the recursive procedure for building the sets of complete multidimensional schedules.

We will also explore the possibility of splitting the dependence graph $G = (V, E)$ into strongly connected components and performing the search independently. This direction is strictly connected with loop fusion/distribution [96, 141, 53, 59].

7.10.1 Finding global solutions

In previous sections we have discussed the way to build the pairwise (per dependence edge e) solutions sets, the way to combine then in the global solution for one scheduling level, and a technique to complete the scheduling with independent solutions at further dimensions. Also, we have shown how to encode the decision on whether to strictly/weakly satisfy a given dependence edge.

On top of those operators and data structures, we build a tunable search methodology that enumerates the possible multidimensional schedules. It might be exhaustive - it will enumerate all the possible schedules and edge satisfaction conditions, or it might be *goal driven* - it will filter out only those schedules that have a certain property (to be defined later), giving an empty solution set if there is no such a schedule.

In addition to enumerating the multidimensional schedules, the method reports the different *properties* for each multidimensional schedule. For our purposes, the properties are:

1. A given dimension (loop) level could be executed in parallel
2. A given dimension (loop) must be executed sequentially
3. A given band of dimensions forms a permutable loop band

Some notation

Some extra notation is introduced for the sake of the search algorithm:

The solution set of legal transformations is denoted by $\mathcal{L}$. The solution set satisfying dependence edge e_1 is denoted by $\mathcal{L}^{e_1}$. The solution set satisfying the set of dependence edges E is denoted by $\mathcal{L}^E$. Each single solution $T \in \mathcal{L}$ of the solution set $\mathcal{L}$ consists of part that encodes the A part of the rows A_k and satisfaction decision variables $x_k^{e \in E}$. We denote those the A part of the solution as T^A.

If a particular T^A solution is selected in a solution tree, the remaining part T_X encodes the strong/weak satisfaction of the edges $e \in E$. Each assignment of the decision variable $x_k^{e \in E}$ that leads to a non-empty solution selects some subset $E^S \in E$ of the edges that could be strongly satisfied. Indeed, the full decision tree of the decision variables $x_k^{e \in E}$ determines the *family* $\mathcal{X}$ of the possible strong satisfy subsets $E^S \in \mathcal{X}$.

As an example of the solution set, let us take a solution set $\mathcal{L}^E$ shown in Figure 7.19. Here an edge set is $E = \{e_1, e_2\}$. One particular solution is $T = \{A_{1,\bullet}^{S_1} = 1, A_{1,\bullet}^{S_2} = 1, x_1^{e_1} = 0, x_1^{e_2} = 0\}$. The A part of the solution is $T^A = \{A_{1,\bullet}^{S_1} = 1, A_{1,\bullet}^{S_2} = 1\}$. If the A part T^A of the tree is selected, one gets the the sub-tree of decision variables. In this example, if one selects $T^A = \{A_{1,\bullet}^{S_1} = 1, A_{1,\bullet}^{S_2} = 1\}$ one will obtain the tree rooted at $x_1^{e_1}$ that represents the family $\mathcal{X}$ of possible edge satisfaction sets E^S. In this example, the family is equivalent to the powerset 2^E of all edges.

Recursive procedure

Here we show a complete recursive procedure for computing the multidimensional schedules. For simplicity, we only show the procedure for obtaining one solution, but it could be easily adapted to explore the space of more complete schedules - it is a matter of changing the termination condition of the main loop and implementing a way of controlling the desired number of candidate solutions at each scheduling depth.

The overall recursive procedure for enumerating the multidimensional schedules is shown in Figure 7.24. An initial set of solutions for the first scheduling dimension is obtained:

$$\mathcal{L} \leftarrow \textsc{CombineInitialSolutions}(E)$$

The procedure for computing the initial solution set was shown in Figure 7.16. The recursive procedure is called with the following arguments:

COMPLETESCHEDULE ($\mathcal{L}, E, d = 1$)

The initial solution set is $\mathcal{L}$, the full set of the dependence edges E is to be satisfied, and the building process is starting with the first scheduling dimension $d = 1$.

In the case the procedure is invoked with $d = dim_{max}(\mathcal{S})$, $dim_{max}(\mathcal{S})$ being the maximum domain dimensionality of the statements, [8] one complete multidimensional schedule is obtained. This is the terminating condition for the recursion. At step (1.1) the schedule properties are provided and the complete schedule is saved at step (1.2). By returning Trueat the step (1.3), it is signalled to the calling procedure that the complete schedule is obtained.

A loop starting at step (3) is executed until at least one solution is obtained. As mentioned earlier, this loop could be changed so that more than one solution is obtained if desired.

At each iteration of the loop the next solution is selected at step (3.1) and its A part is stored as T^A. At step (3.2) an orthogonal subspace for the transformation T^A is obtained.

A given transformation T^A determines an orthogonal subspace at the deeper levels. The core part of the search algorithm is to determine the solution set $\mathcal{L}'$ for the next scheduling level such that $H \cap \mathcal{L}' \neq \emptyset$. If no such a set could be obtained, then the given transformation T^A at the level d could not be completed with independent solutions at deeper levels.

There is a combinatorial number of ways to provide the set $\mathcal{L}'$. Indeed the set $\mathcal{L}' = \mathcal{L}E - E^S$. There is a combinatorial number of ways to select the strong satisfaction edge set E^S. In a special case when $E^S = \emptyset$, the $\mathcal{L}' = \mathcal{L}^E = \mathcal{L}$. This is the case when we can have a permutable loop band.

In order to avoid useless backtracking steps, a quick test is performed to determine whether a given T^A could have a non-empty completion at deeper levels. First, a set of edges E^{must} that must be removed at the current level d is determined at step (3.3). We will show the way to perform this check in Subsection 7.10.2. Later, at steps (3.4) and (3.5), it is determined whether this set could be strongly satisfied by a selected transformation T^A. If it cannot be strongly satisfied, then the set T^A is not taken into the consideration anymore.

At step (3.6) the family of strong satisfaction sets is restricted to family $\mathcal{X}'$ that contains only the satisfaction sets E^S that are supersets of those edges that must be satisfied: $E^S \supset E^{must}$. This steps narrows down the search space of strong edge satisfactions that is traversed at further steps.

The loop at step (3.9) performs the traversal of the strong satisfaction sets E^S. A naive approach is to traverse all the $E^S \in \mathcal{X}'$, in any order. The order in which those sets are traversed has an impact on the convergence of the method. We will show the different strategies of the traversal order. A predicate EXHAUSTED controls the number of the solutions that one wants to investigate. In any case, $strategy < N$, where $N = \mathcal{X}'$.

Once a given strong satisfaction set E^S is selected, the remaining set E' of those edges that must be satisfied at deeper levels is obtained at step (3.9.2). A check is made at step (3.9.3) whether removing those edges gives a solution $\mathcal{L}^{E'}$ that could be completed at deeper $d+1$. If this is not the case, the next satisfaction set E is checked according to some new strategy (a strategy counter is increased). Otherwise, a current solution T^A is completed with a recursive call to the schedule completion procedure for deeper levels made at the step (3.9.5).

7.10.2 Dependence satisfaction strategies

As mentioned in Subsection 7.6.1, the problem of selecting the dependence edges to be strongly satisfied at a given loop depth d has so far been solved by proposing heuristics [68, 34, 32]. Darte [55] has formally shown that this problem is NP-complete. Given a dependence graph $G = (V, E)$ having $|E|$ edges, there are $2^{|E|}$ possible subsets of the edges to be strongly satisfied.

8. For the implementation convenience, the schedule matrices of those statements whose depth is less than $dim_{max}(\mathcal{S})$ are padded with zeros

Procedure COMPLETESCHEDULE
INPUT: $\mathcal{L}$, E, d

$\mathcal{L}$ - a solution set found so far (for levels $1 \leq l < d$), satisfying the set of edges E

E - a subset of edges satisfied at the given or deeper levels

d - a scheduling dimension to be completed

```
(1) IF $d = dim_{max}(\mathcal{S})$
        { A complete multidimensional solution is obtained }
    (1.1) provide schedule properties
    (1.2) save the complete solution $\mathcal{L}'$
    (1.3) RETURN True
(2) $found \leftarrow False$
(3) WHILE $(\neg found)$
    (3.1) IF empty ($T^A \leftarrow$ next solution from $\mathcal{L}$)
              BREAK
        {compute orthogonal subspace of transformation }
    (3.2) $H \leftarrow \textsc{Orthogonal}(T^A)$
        {Compute the set of edges that must be strongly satisfied in order
            to give the non-empty solution: $H \cap \mathcal{L}' \neq \emptyset$}
    (3.3) $E^{must} \leftarrow \textsc{MustSatisfy}(H)$
        { Select the family $\mathcal{X}$ of possible edge satisfaction sets $E^S \in \mathcal{X}$ for transformation $T^A$ }
    (3.4) $\mathcal{X} \leftarrow \textsc{Select}(\mathcal{L}, T^A)$
        {If the set $E^{must}$ does not belong to $\mathcal{X}$ then break early }
    (3.5) IF$E^{must} \notin \mathcal{X}$
              CONTINUE
        {Select the family $\mathcal{X}' \subset \mathcal{X}$ such that $\mathcal{X}' = \{E^S \in \mathcal{X} : E^S \supset E^{must}\}$ }
    (3.6) $\mathcal{X}' \leftarrow \textsc{Select}(\mathcal{X}, E^{must})$
    (3.7) $N \leftarrow |\mathcal{X}'|$
    (3.8) $strategy \leftarrow 0$
    (3.9) WHILE $(\neg found \wedge \textsc{Exhausted}(strategy, N))$
        (3.9.1) $E^S \leftarrow \textsc{SelectSatSet}(\mathcal{X}', strategy)$
        (3.9.2) $E' \leftarrow E - E^S$
        (3.9.3) IF $H \cap \mathcal{L}^{E'} = \emptyset$
              $strategy \leftarrow strategy + 1$
              CONTINUE
        (3.9.5) $found \leftarrow found \vee \textsc{CompleteSchedule}(\mathcal{L}^{E'}, E', d+1)$
(4) RETURN $found$
```

Figure 7.24 – Schedule completion strategy

One can exhaustively try all the possible subsets - a naive enumeration method. We rather propose to rank those subsets according to some preferable properties. This oracle, named SELECTSATSET, is called from the enumerative search algorithm shown in Figure 7.24.

The first and most preferable strategy is the case when $E^S = \emptyset$. This choice forces the next scheduling level $d+1$ to satisfy all the dependence edges E satisfied at previous level d. If this is possible, we will obtain a permutable loop band $d \dots d+1$ as in [34].

If this is not a feasible solution, then at least one edge $e \in E$ has to be strongly satisfied. The joint work of Pouchet and Bondhugula [132] shows the simple heuristic that decides to satisfy the maximal number of edges at a given depth if the E^S can not be empty. We also believe that following this scheme mostly likely increases the chance of getting the non-empty completion of the schedule at the next level. Thus, the next preferable strategy is the one that selects $\{E^S \in \mathcal{X} : max|E^S|\}$

As Kennedy has shown in his approach to loop selection problem [7], the greedy heuristic of selecting the solution that covers the maximal number of the edges at once might fail in the case when this maximal subset does not cover those edges that prevent obtaining the solution.

We improve the heuristic of Pouchet and Bondhugula by a shortcutting step since we determine upfront the set of those edges that *must* be satisfied E^{must}. This set is not determinable in the case of general affine schedules. But in our case we have a restricted search space and we also have an enumeration of the possible solutions built upfront.

The goal is to obtain $H \cap \mathcal{L}' \neq \emptyset$

The goal of the procedure MUSTSATISFY(H) called at step (3.3) of the algorithm in Figure 7.24 is to obtain the set $E^{must} \subset E^S$ of those edges that must be satisfied in order to $H \cap \mathcal{L}^{E-E^S} \neq \emptyset$. Obviously if $\mathcal{L}^{e_i} \cap H = \emptyset$ then the dependence edge e_i has to belong to the set E^{must}.

Although the pruning step we have shown can cut down the search space of possible satisfaction sets, it only gives the necessary condition, but not the sufficient condition for $H \cap \mathcal{L}^{E-E^S} \neq \emptyset$.

7.10.3 Distribution/fusion scheme

The search method discussed so far does find a complete solution in terms of complete A part of the scheduling matrices. But it does not represent the loop distribution(fusion) schemes since the β part of the scheduling matrices is not constructed. Indeed, the scheme presented implicitly assumes that all the statements are fused (β part of all the schedule matrices being the zero vector).

Forcing all the statements to be fused within one loop might prevent obtaining any legal solution. Indeed, if an original program contains non-perfectly nested loops, some statements do not belong to the same loop - they are distributed. It is not always legal to transform such a program to a completely fused version.

A classical literature on loop fusion and distribution [95, 111, 53, 54] shows the benefits of both: distribution enables more transformations, including parallelism, while loop fusion is beneficial for optimizing the memory access locality by reducing the distance between producer and consumer pairs. Darte [53] has shown that the problem of determining the optimal loop fusion structure is NP-complete.

The polyhedral model naturally incorporates the loop fusion/distribution decisions, through β part of the scheduling matrix. A systematic way to approach the fusion/distribution problem in the polyhedral model was shown in [33, 131]. While expressing the set of legal fusion/distribution structures in the polyhedral model is not difficult problem [131], selecting the desired structure out of the combinatorial space is hard problem.

We do not incorporate the search for fusion/distribution structures in our search space. The reason is that we want to decouple this particular search problem from the problem of searching the complete multidimensional schedules.

Our original combinatorial problem is the problem of searching for sets of edges to be satisfied strongly at particular scheduling depth . This problem is modelled as a decision tree that represents the possible subsets of the edges that could be satisfied strongly at the same time.

The combinatorial problem of enumerating the fusion structures is the problem of enumerating the legal partitions of the set of statements $\mathcal{S} = \{S_1, \ldots, S_p\}$. If p is the number of statements, the total number of partitions is the Bell number B_p, which gives the worst case complexity of the traversal.

We perform the maximal distribution of the statement sets, according to the strongly connected components of the dependence graph $G = (V, E)$. This corresponds to one of the heuristics used by Bondhugula [35] in his scheduling approach.

The reason for choosing the maximal distribution strategy is the following: one can always maximally distribute the statements into partitions according to the SCCs of a dependence graph. In addition, this choice gives the most freedom in choosing the subsequent multidimensional schedules. This guarantees us that we will get the most expressive space of legal multidimensional schedules within our search space. Once the full space of fully distributed multidimensional schedules is obtained, one might use one of the fusion strategies already developed in [33, 131] to get the desired fusion structure, if it is possible to merge the solution sets into tighter constraints given by fusion.

In order to incorporate the max distribution strategy into the search procedure, we slightly modify the search procedure. At each depth of the search procedure shown in Figure 7.24, after selecting the set of the edges E^S to be removed at the next scheduling level, reform the dependence graph $G' = (V, E - E^S)$ and compute the set of SCCs for this new graph $G' \subset G$. Topologically sort SCCs, and form an ordered sequence $(SCC_1, \ldots, SCC_m)$. For each statement $S_i \in SCC_j$ assign its static scheduling part (β part) to j, where j is the position of the SCCin a topological order. After this step, recursively call the procedure COMPLETESCHEDULE for each SCCindependently.

7.11 Conclusions

We have presented the general search strategy used for traversing the legal transformation search space. The current approaches to automatic transformation in the polyhedral model are based either on linear programming (one-shot, best-effort heuristics of Feautrier [67] and Bondhugula [34]) or on the exhaustive search of the space of legal affine transformations (iterative compilation of Pouchet [129]).

We propose to represent the set of legal solutions as a discrete set of selected and legal loop permutations, together with enabling *shifting* factors that are represented as the convex sets. We use the decision diagram data structure to represent those solution sets. Our search space is a subset of the full space of affine transformations.

Contrary to the iterative approach of Pouchet, which traverses the full space of legal and distinct affine transformations, our search space is restricted, but this enables the control over the complexity of the solution. On one side, the solution set is restricted to the *discrete set* of legal loop permutations, but on the other side, each legal loop permutation is augmented with a convex set of shifting factors that decides whether this permutation satisfies the dependence *strongly* or *weakly*.

The fact that we have an explicit control over the weak/strong satisfaction of the dependence edges, enables us to traverse the space of *multidimensional* schedules in an exhaustive way (if needed). This is not possible in the current multidimensional affine schedule space traversal approach of Pouchet [128]. Pouchet employs a greedy heuristic that forces the *unique choice* of dependence edge satisfaction at each scheduling level, to keep the search space tractable [9] . Our contribution achieves a degree of flexibility and controllability of the search space that is not possible with the current iterative approach.

9. Though the latest contribution [132] shows (only) a theoretical way to achieve this flexibility

Nevertheless, our search space is not as exhaustive as the one of Pouchet. Because we restrict the scheduling coefficients of the dynamic scheduling component (Section 3.1) to the loop permutation only, we might miss some important transformations such as loop skewing, loop reversal or loop slowing. But this is a necessary price to pay if we cannot accept the running time of the exhaustive iterative search used in the feedback-directed iterative compilation [128].

The future work

The search strategy might be improved in several directions. First, the algorithmic improvements are necessary in order to avoid the overhead of recomputing the same solution sets during the construction of multidimensional schedules. This could be achieved by *caching* the already computed sets to avoid the redundant recomputations.

The representation of the convex part of the solution relies on the operations on the polyhedral sets, that in turn require the Chernikova's algorithm [42, 100]. But it could be observed that our convex sets have a special form, that of the two-variable per constraint [49]. Those sets are the subsets of general polyhedra, and they could be operated on with less costly techniques than the general techniques used for polyhedral sets [154].

Our search strategy enables the branch-and-bound metaheuristic. But in order to make it effective, we have to provide a systematic way of *pruning* the search space, cutting the non-profitable branches of the search tree. This could be achieved by having a *partially evaluated* cost functions that could provide the estimate of the cost, even without having a completed schedule. Our current cost function (Chapter 6) does not have this property.

Chapter 8

Conclusions and perspectives

In this thesis we have presented the theory, design and implementation of the search based program transformation strategy based on top of the three-address code polyhedral compiler. We have shown that in order to obtain the precise execution cost-model function, it is necessary to integrate the polyhedral model framework directly into the three-address code compiler. This is in contrast with the traditional approach for polyhedral compilation based on source-to-source compilers.

Integrating the polyhedral compilation framework directly into the three-address code compiler posed several challenges that were not investigated in the known literature. We have provided some efficient solutions for those problems, concerning the efficient representation of the scalar and memory-based dependences. We have also presented an unifying approach for representing all the transformations, including tiling purely through the scheduling functions.

We have shown a precise execution cost-model function that precisely captures the low-level details, such as SIMD vectorization, of the target architecture. Contrary to simplistic linear cost-model functions, our function is a complex, non-linear characterization. The evaluation of this function requires a search based strategy, contrary to the approaches that use linear programming to directly optimize the simple linear cost-functions. Our cost-model function is evaluated for each point of the constructed search space.

Providing an efficient search strategy requires the construction of the expressive, but size-limited search space of legal program transformations. We took a novel approach: contrary to the current iterative optimizers that explore the huge space of affine schedules, we build a search space of *discrete set* of transformations. Our search space is a subset of the full affine search space. For an efficient representation of the sets we propose to use the decision diagrams and we provide the basic operators on those sets.

For assessing the feasibility of the approach, we have provided the practical implementation of the techniques in GRAPHITE polyhedral framework that is a part of the production quality compiler GCC. We will summarize the detailed contributions of this thesis:

Three-address code polyhedral compilation framework

Traditional polyhedral program transformation frameworks are implemented as the source-to-source program translators. This design is natural and practical for research, but it is somewhat simplistic when it comes to precise modelling of the target machine execution model. We provide the design and implementation of the sophisticated polyhedral transformation framework operating directly on the three-address code in SSA form. We have provided an in-depth investigation of the issues related to the direct polyhedral compilation of the three-address code. To the best of our knowledge, it is one of the first widely available and published polyhedral compiler that operates on the three-address code.

Lazy approach for relaxing scheduling constraints

The problem of effectively handling memory-based data dependences is well-known. This problem is particularly exposed in the three-address code polyhedral compiler, since the low-level code contains many temporary variables for storing intermediate results. Those temporary variables induce many memory-based dependences. The traditional approach to eliminating memory-based dependences is the *memory expansion* - expanding scalars into arrays, or arrays into higher-dimensional arrays. This approach eliminates all memory-based dependences, but it might have a prohibitive cost. We propose the *lazy* approach for handling memory expansion. The memory-based data dependences are firstly ignored, and the live range *violation analysis* is used to compute the minimal set of dependences that have been violated. Only the violated memory-based dependences are removed through expansion. This approach works well with search based strategy, since the cost of the memory expansion might be evaluated for multiple transformation candidates.

Semantical transparency

The traditional source-to-source compilers are *sensitive* to syntactical details of the input program. The same computational kernel that is written in a slightly different style (by introducing the scalar temporaries for example) might be ignored by an optimizer, simply because it does not conform to the syntactical constraints of the source-to-source compiler. This limitation is one of the most important reasons why the current polyhedral compilers are not able to transparently compile *legacy codes* or industry-standard benchmarks such as SPECint or SPECfp. Providing the polyhedral framework that is part of the production quality compiler, such as GCC, and operating on the low-level intermediate representation provides the *semantical transparency* for the end user - an user might still write the programs in the sequential style, while the compiler is responsible for extracting the intrinsic semantics of the written program. This form of the semantical transparency is possible in GRAPHITE due to the fact that the polyhedral framework operates at the stage where the code is transformed into SSA form on which the essential scalar optimizations are performed. A code in such a form captures the essential semantics of the input program and is much less sensitive to the peculiarities of the syntactic form of the program. In addition, the lazy approach to memory-expansion can transparently remove the *spurious data dependences* that might be inadvertently introduced by the programmer.

Precise performance predicting cost-model

The one-shot, best-effort scheduling heuristics are based on using (integer) linear programming to optimize simple linear objective cost functions. The linear cost-models are based on an abstract computational metrics, and they are not adequate for modelling the complex, low-level aspects of the target machine execution models. One such an aspect is SIMD vectorization, which is highly target specific. We propose a novel approach to this problem by proposing to use *machine specific* accurate cost-model functions. We demonstrate the feasibility of the approach, by constructing the precise cost-model function for SIMD architectures. Those precise machine-specific cost-models cannot be expressed as linear functions and they could not be handled by linear programming machinery. Instead, they have to be evaluated at each point of the search space. The critical aspect for the effectiveness of our cost function evaluation is the construction of the feasible search space. This relies on our next contribution, which is a construction of effective transformation search space.

Efficient transformation search space

We propose the novel method for constructing the search space of legal polyhedral transformations. Our method is based on providing the *discrete sets* of legal transformations. Those sets are represented as decision diagrams and they are subset of the full search space of affine transformations. Discrete sets limit the possible affine transformations, but they enable us to control the complexity of the search space. This is crucial contribution in enabling the search-based methodology that is *efficient*. In the case of multidimensional scheduling, we provide more flexibility: we enable the enumeration of the *dependence satisfaction* strategies. The current iterative transformation approaches that construct the space of multidimensional schedules take the greedy heuristic for selecting the data dependence satisfaction at different scheduling levels, to keep the search spaces tractable. They could miss some points in the search space that our enumeration strategy enumerates. We show that we can afford this level of flexibility, since we restrict the legal solution sets early in the search process.

8.1 Perspectives

We have provided a novel approach for the polyhedral program transformations based on the search strategies that optimize the complex, machine dependent cost-model functions. Still, there are numerous possible improvements and future directions that are worth investigating. We will give brief ideas for the possible improvements and directions for the prospective research topics:

Even more precise cost-model

The precise cost model function that we have proposed includes a very rough estimation of the data locality. While it can capture the *spatial locality* of the given memory access pattern, it cannot precisely model the *temporal locality*. The model could be extended with cache miss equations [69] that fit very well with the current model. The architectural interplay between SIMD parallelism and thread-level coarse grained parallelism [62] could be modelled in the cost function as well.

Parametric cost function

The cost-function that we have presented works on a premise that the loop iteration domains are known at the compile time. This is necessary because the total cost depends on the exact number of iterations of each loop. This also simplifies the comparison of the costs, since the costs are expressed as simple integer numbers. But the polyhedral model allows the *parametric* loop bounds - bounds that are expressed as symbolic constants that are not known at compile time. We would like to investigate the technique of computing the number of iterations [43] for parametric iteration domains. This would require representing the costs as symbolic expressions, rather than integer constants. This could however fundamentally change the approach: the best solution will depend on the parameter values, and it could not be obtained automatically. Some assistance from the user, who understands the domain of the problem and can estimate the problem size, would be required.

Machine learning assisted portability of the cost-model

The analytical cost-model function that we have presented relies on the *machine specific* instruction costs. We have obtained those costs for several architectures, through microbenchmarking and knowledge of the exact instruction latencies. Obtaining the instruction costs for the new architectures is a time-consuming process which hinders the *portability* of the cost-model based approach. In order to facilitate the porting of the cost-model to the new architectures, it interesting to investigate the possibility of employing machine learning, like in [40], to automatically obtain the instruction costs and other platform-specific coefficients that are used in the cost-function.

Scalability

We propose the search space construction method that can control the explosion of the transformation search space by limiting the number of *non-convex* solutions. Still, we leave some parametric part (the shifting factor) that is represented as convex polyhedra. This part is currently operated on by the standard polyhedral techniques. But the special form of this parametric part is amenable to representation as the two-variable per constraint system [49], which could be operated on with less costly techniques than the general techniques used for polyhedral sets [154]. An interesting research direction is the investigation of the *sub-polyhedral* domains. Sub-polyhedral domains are less powerful than the traditional polyhedral techniques, but the complexity of operations on those sets is guaranteed to be polynomial or linear.

Extending the scope of the analyzable programs

The traditional polyhedral model representation is restricted to static control programs (SCoP). This is the fundamental restriction of the polyhedral model. Programs with irregular control, non-affine memory accesses and loop bounds are not amenable to analysis within the polyhedral model. The methods [26, 18, 78] to overcome those limitations have been proposed, but they are still not widely used, due to the complications in code generation for non-affine programs. An interesting direction is to investigate the static polyhedral techniques, coupled with dynamic techniques [142, 112] for program parallelization.

Just In Time Compilation

We have provided the polyhedral framework for the direct compilation of the three-address codes. Our technique was investigated in the context of the static compiler, translating the source code to binary machine code.

The fact that the polyhedral technique is implemented directly on the three-address-code representation could be extended and used in the JIT compilers that directly operate on the bytecode. Indeed, Polly project, based on LLVM compiler framework [79] goes in this direction. The crucial problem to be solved though is the scalability of the polyhedral techniques, especially the transformation part.

Going beyond the affine transformations

The affine transformations expressed in the polyhedral model cover a wide range of the classical loop transformations and enable coarse-grained parallelization techniques. Nevertheless, there are programs that could be expressed in the polyhedral model, but for which the coarse-grained parallelism could not be expressed through affine transformations [25]. There are solutions that go beyond the affine transformations, and they are based on computing the transitive closure [28, 24] of affine relations. Even though the dependence relations are affine, their transitive closure might be non-affine [28]. Unfortunately, the problem of code generation for non-linear forms is particularly hard and not-so-well studied problem [78] for which solutions are still awaiting.

Appendix A

Application of Farkas lemma

$\mathcal{P}_{e_1} = \{(i, j, i', j') | i' = i + 1 \wedge j' = j \wedge 1 \leq i \leq N - 2 \wedge 0 \leq j \leq N - 1\}$
The faces of the dependence polyhedron are:

$$\begin{cases} i - 1 \geq 0 \\ j \geq 0 \\ N - 2 - i \geq 0 \\ N - 1 - j \geq 0 \end{cases}$$

$(a^{S_2}_{1,1} - a^{S_1}_{1,1})i + (a^{S_2}_{1,2} - a^{S_1}_{1,2})j + \omega^{S_2}_1 - \omega^{S_1}_1 + a^{S_2}_{1,1} - 1 \geq 0$
Combining them with Farkas multipliers gives:

$$\lambda_0 + \lambda_1(i - 1) + \lambda_2(j) + \lambda_3(N - 2 - i) + \lambda_4(N - 1 - j)$$

$\lambda_0 \geq 0, \lambda_1 \geq 0, \lambda_2 \geq 0, \lambda_3 \geq 0, \lambda_4 \geq 0$

Reorganizing the terms so that the induction variables and parameters could be equated to the legality constraint:

$$i(\lambda_1 - \lambda_3) + j(\lambda_2 - \lambda_4) + N(\lambda_3 + \lambda_4) + \lambda_0 - \lambda_1 - 2\lambda_3 - \lambda_4$$

After equating the left hand and right hand sides of the Farkas lemma one obtains:

$$\begin{cases} a^{S_2}_{1,1} - a^{S_1}_{1,1} = (\lambda_1 - \lambda_3) \\ a^{S_2}_{1,2} - a^{S_1}_{1,2} = (\lambda_2 - \lambda_4) \\ 0 = (\lambda_3 + \lambda_4) \\ \omega^{S_2}_1 - \omega^{S_1}_1 + a^{S_2}_{1,1} - 1 = \lambda_0 - \lambda_1 - 2\lambda_3 - \lambda_4 \\ \lambda_0 \geq 0, \lambda_1 \geq 0, \lambda_2 \geq 0, \lambda_3 \geq 0, \lambda_4 \geq 0 \end{cases}$$

One can immediately notice: $\lambda_3 = 0 \wedge \lambda_4 = 0$

$$\begin{cases} a^{S_2}_{1,1} - a^{S_1}_{1,1} = (\lambda_1) \geq 0 \\ a^{S_2}_{1,2} - a^{S_1}_{1,2} = (\lambda_2) \geq 0 \\ \omega^{S_2}_1 - \omega^{S_1}_1 + a^{S_2}_{1,1} - 1 = \lambda_0 - \lambda_1 \\ \lambda_0 \geq 0, \lambda_1 \geq 0, \lambda_2 \geq 0, \lambda_3 \geq 0, \lambda_4 \geq 0 \end{cases}$$

$$\begin{cases} a^{S_2}_{1,1} - a^{S_1}_{1,1} \geq 0 \\ a^{S_2}_{1,2} - a^{S_1}_{1,2} \geq 0 \\ \omega^{S_2}_1 - \omega^{S_1}_1 + 2a^{S_2}_{1,1} - a^{S_1}_{1,1} - 1 \geq 0 \end{cases}$$

List of Figures

Personal Bibliography

- Konrad Trifunovic, Albert Cohen, Razya Ladelsky and Feng Li. Elimination of Memory-Based Dependences for Loop-Nest Optimization and Parallelization: Evaluation of a Revised Violated Dependence Analysis Method on a Three-Address Code Polyhedral Compiler. In *3rd International Workshop on GCC Research Opportunities*, Chamonix, France, April 2011.
- Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjodin and Ramakrishna Upadrasta. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *2nd International Workshop on GCC Research Opportunities*, Pisa, Italy, February 2010.
- Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks and Ira Rosen. Polyhedral-Model Guided Loop-Nest Auto-Vectorization In *International Conference on Parallel Architectures and Compilation Techniques*, Raleigh, North Carolina, September 2009. IEEE Computer Society.
- H. Munk, E. Ayguadé, C. Bastoul, P. Carpenter, Z. Chamski, A. Cohen, M. Cornero, M. Duranton, M. Fellahi, R. Ferrer, R. Ladelsky, M. Lindwer, X. Martorell, C. Miranda, D. Nuzman, A. Ornstein, A. Pop, S. Pop, L.-N. Pouchet, A. Ramírez, D. Ródenas, E. Rohou, I. Rosen, U. Shvadron, K. Trifunovic and A. Zaks. ACOTES Project: Advanced Compiler Technologies for Embedded Streaming. In *International Journal of Parallel Programming*, 2010. Springer Verlag.
- Wlodzimierz Bielecki, Konrad Trifunovic, Tomasz Klimek. Calculating Exact Transitive Closure for a Normalized Affine Integer Tuple Relation. In *Electronic Notes in Discrete Mathematics*, 2009. Elsevier.

Other publications:

- Piotr Dziurzanski, Wlodzimierz Bielecki, Konrad Trifunovic and Michal Kleszczonek. A System for Transforming an ANSI C Code with OpenMP Directives into a SystemC Description. In *Proceedings of the 9th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2006)*, Prague, Czech Republic, April 2006. IEEE Computer Society.

Bibliography

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

[2] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] C. Alias, F. Baray, and A. Darte. Bee+cl@k: an implementation of lattice-based array contraction in the source-to-source translator rose. *SIGPLAN Not.*, 42:73–82, June 2007.

[5] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, SIGPLAN '84, pages 233–246, New York, NY, USA, 1984. ACM.

[6] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9:491–542, October 1987.

[7] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[8] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '04, pages 231–239, New York, NY, USA, 2004. ACM.

[9] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '91, pages 39–50, New York, NY, USA, 1991. ACM.

[10] L. O. Andersen. Program analysis and specialization for the c programming language. Technical report, DIKU, 1994.

[11] A. W. Appel. Ssa is functional programming. *SIGPLAN Not.*, 33:17–20, April 1998.

[12] R. Bagnara, P. M. Hill, and E. 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.*, 72:3–21, June 2008.

[13] U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Nov. 1976.

[14] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.

[15] U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*, pages 192–219, Irvine, Aug. 1990.

[16] U. Banerjee. *Loop Transformations for Restructuring Compilers*. Kluwer Academic, 1993.

[17] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. *Int. J. Parallel Program.*, 28:213–243, June 2000.

[18] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *J. Parallel Distrib. Comput.*, 40:210–226, February 1997.

[19] A. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19(4):769–779, november 1994.

[20] C. Bastoul. Cloog: The chunky loop generator. `http://www.cloog.org`.

[21] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *Proceedings of the Second international conference on Parallel and distributed computing*, ISPDC'03, pages 23–30, Washington, DC, USA, 2003. IEEE Computer Society.

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

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

[24] A. Beletska, D. Barthou, W. Bielecki, and A. Cohen. Computing the transitive closure of a union of affine integer tuple relations. In *Proceedings of the 3rd International Conference on Combinatorial Optimization and Applications*, COCOA '09, pages 98–109, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] A. Beletska, W. Bielecki, A. Cohen, M. Palkowski, and K. Siedlecki. Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. In *Proceedings of the 2009 Eighth International Symposium on Parallel and Distributed Computing*, ISPDC '09, pages 73–80, Washington, DC, USA, 2009. IEEE Computer Society.

[26] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In R. Gupta, editor, *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 283–303. Springer, 2010.

[27] A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, october 1966.

[28] W. Bielecki, T. Klimek, and K. Trifunovic. Calculating exact transitive closure for a normalized affine integer tuple relation. *Electronic Notes in Discrete Mathematics*, 33:7–14, 2009.

[29] A. J. C. Bik. *The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[30] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic intra-register vectorization for the Intel architecture. *IJPP*, 30(2):65–98, 2002.

[31] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[32] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proceedings of the Joint European Conferences on Theory and*

Practice of Software 17th international conference on Compiler construction, CC'08/ETAPS'08, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag.

[33] U. Bondhugula, O. Gunluk, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 343–352, New York, NY, USA, 2010. ACM.

[34] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[35] U. K. R. Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, Columbus, OH, USA, 2008. Adviser-Sadayappan, P.

[36] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? In *IEEE Scalable High-Performance Computing Conf.*, May 1994.

[37] P.-Y. Calland, A. Darte, Y. Robert, and F. Vivien. On the removal of anti- and output-dependences. *Int. J. Parallel Program.*, 26:285–312, June 1998.

[38] L. Carter, J. Ferrante, and C. Thomborson. Folklore confirmed: reducible flow graphs are exponentially larger. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 106–114, New York, NY, USA, 2003. ACM.

[39] C. Cascaval, L. Derose, D. A. Padua, and D. A. Reed. Compile-time based performance prediction. In *LCPC*, 1999.

[40] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. F. P. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 24–34, New York, NY, USA, 2006. ACM.

[41] C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, U. of Southern California, 2008.

[42] N. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *USSR Computational Mathematics and Mathematical Physics*, 1965.

[43] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: applications to analyze and transform scientific programs. In *Intl. Conf. on Supercomputing*, pages 278–285, Philadelphia, may 1996.

[44] C. Coarfa, F. Zhao, N. Tallent, J. Mellor-Crummey, and Y. Dotsenko. Open-source compiler technology for source-to-source optimization.

[45] A. Cohen. Parallelization via constrained storage mapping optimization. In *Proceedings of the Second International Symposium on High Performance Computing*, ISHPC '99, pages 83–94, London, UK, 1999. Springer-Verlag.

[46] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *Intl. Conf. on Supercomputing (ICS'05)*, pages 151–160, Boston, Massachusetts, June 2005.

[47] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 292–303. Springer, 2004.

[48] A. Cohen and V. Lefebvre. Storage mapping optimization for parallel programs. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, Euro-Par ’99, pages 375–382, London, UK, UK, 1999. Springer-Verlag.

[49] E. Cohen and N. Megiddo. Improved algorithms for linear inequalities with two variables per inequality. *SIAM J. Comput.*, 23:1313–1350, December 1994.

[50] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, 1993.

[51] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Fifth ACM Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Jan. 1978.

[52] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’89, pages 25–35, New York, NY, USA, 1989. ACM.

[53] A. Darte. On the complexity of loop fusion. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT ’99, pages 149–, Washington, DC, USA, 1999. IEEE Computer Society.

[54] A. Darte and G. Huard. Loop shifting for loop compaction. *Int. J. Parallel Program.*, 28:499–534, October 2000.

[55] A. Darte and G. Huard. Complexity of multi-dimensional loop alignment. In *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science*, STACS ’02, pages 179–191, London, UK, UK, 2002. Springer-Verlag.

[56] A. Darte and G. Huard. New complexity results on array contraction and related problems. *J. VLSI Signal Process. Syst.*, 40:35–55, May 2005.

[57] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser, 2000.

[58] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Trans. Comput.*, 54:1242–1257, October 2005.

[59] A. Darte and G.-A. Silber. Temporary arrays for distribution of loops with control dependences. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par ’00, pages 357–367, London, UK, 2000. Springer-Verlag.

[60] A. Darte, G.-A. Silber, and F. Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[61] A. Darte and F. Vivien. On the optimality of allen and kennedy’s algorithm for parallel extraction in nested loops. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume I*, Euro-Par ’96, pages 379–388, London, UK, 1996. Springer-Verlag.

[62] P. K. Dubey, G. B. Adams, III, and M. J. Flynn. Evaluating performance tradeoffs between fine-grained and coarse-grained alternatives. *IEEE Trans. Parallel Distrib. Syst.*, 6:17–27, January 1995.

[63] R. v. Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the 10th International Conference on Compiler Construction*, CC ’01, pages 118–132, London, UK, 2001. Springer-Verlag.

[64] P. Feautrier. Array expansion. In *Proceedings of the 2nd international conference on Supercomputing*, ICS ’88, pages 429–441, New York, NY, USA, 1988. ACM.

[65] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[66] P. Feautrier. Dataflow analysis of scalar and array references. *Intl. Journal of Parallel Programming*, 20(1):23–53, february 1991.

[67] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I: one dimensional time. *Intl. Journal of Parallel Programming*, 21(5):313–348, october 1992.

[68] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Intl. Journal of Parallel Programming*, 21(6):389–420, december 1992.

[69] B. B. Fraguela, R. Doallo, and E. L. Zapata. Probabilistic miss equations: Evaluating memory hierarchy performance. *IEEE Trans. Comput.*, 52(3):321–336, 2003.

[70] FSF. *GNU Compiler Collection (GCC) Internals Manual*, 2010. http://gcc.gnu.org/onlinedocs/gccint.

[71] FSF. Gnu compiler collection (gcc), 2011. http://gcc.gnu.org.

[72] G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 281–295, London, UK, 1993. Springer-Verlag.

[73] S. Girbal. *Optimisation d'applications - Composition de transformations de programme: modèle et outils*. PhD thesis, University Paris-Sud 11, 2005.

[74] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34:261–317, June 2006.

[75] O. Golovanevsky, A. Dayan, A. Zaks, and D. Edelsohn. Trace-based data layout optimizations for multi-core processors. In Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, editors, *High Performance Embedded Architectures and Compilers*, volume 5952 of *Lecture Notes in Computer Science*, pages 81–95. Springer Berlin / Heidelberg, 2010.

[76] M. Griebl. *Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. Facultät für Mathematik und Informatik, Universität Passau*. PhD thesis, Facultät für Mathematik und Informatik, Universität Passau, 2004.

[77] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 106–, Washington, DC, USA, 1998. IEEE Computer Society.

[78] A. Groesslinger. *The Challenges of Non-linear Parameters and Variables in Automatic Loop Parallelisation*. Lulu Enterprises, UK Ltd, 2010.

[79] T. Grosser, H. Zheng, R. Aloor, A. Simburger, A. Groesslinger, and L.-N. Pouchet. Polly - polyhedral optimization in llvm. In *IMPACT 2011 First International Workshop on Polyhedral Compilation Techniques*, Chamonix, France, 2011.

[80] G. Gupta, D. Kim, and S. V. Rajopadhye. Scheduling in the z-polyhedral model. In *IPDPS*, pages 1–10, 2007.

[81] M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[82] M. Hall, D. Padua, and K. Pingali. Compiler research: the next 50 years. *Commun. ACM*, 52:60–67, February 2009.

[83] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 147–157, New York, NY, USA, 2009. ACM.

[84] INRIA and The Ohio State University. Polybench, the polyhedral benchmark suite. `http://www-rocq.inria.fr/~pouchet/software/polybench`.

[85] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *Intl. Conf. on Supercomputing (ICS'91)*, Cologne, Germany, June 1991.

[86] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM.

[87] KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX.

[88] R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, july 1967.

[89] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, Department of Computer Science, University of Maryland at College Park, 1996.

[90] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega library interface guide. Technical report, University of Maryland at College Park, College Park, MD, USA, 1995.

[91] W. Kelly and W. Pugh. Finding legal reordering transformations using mappings. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '94, pages 107–124, London, UK, 1995. Springer-Verlag.

[92] W. Kelly and W. Pugh. Minimizing communication while preserving parallelism. Technical report, College Park, MD, USA, 1995.

[93] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. Technical report, University of Maryland at College Park, College Park, MD, USA, 1995.

[94] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)*, FRONTIERS '95, pages 332–, Washington, DC, USA, 1995. IEEE Computer Society.

[95] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, pages 301–320, Portland, 1993.

[96] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 407–416, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[97] K. Knobe and V. Sarkar. Array ssa form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 107–120, New York, NY, USA, 1998. ACM.

[98] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM.

[99] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, 17:83–93, February 1974.

[100] H. Le Verge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.

[101] C. G. Lee. Utdsp benchmarks. http://www.eecg.toronto.edu/˜corinna/DSP/infrastructure/UTDSP.html, 1998.

[102] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Comput.*, 24:649–671, May 1998.

[103] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, editors, *LCPC*, volume 757 of *Lecture Notes in Computer Science*, pages 391–405. Springer, 1992.

[104] A. Lim. *Improving Parallelism and Data Locality with Affine Partitioning*. PhD thesis, Stanford University, 2001.

[105] A. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '94, pages 92–106, London, UK, 1995. Springer-Verlag.

[106] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 228–237, New York, NY, USA, 1999. ACM.

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

[108] V. Loechner and D. K. Wilde. Parameterized polyhedra and their vertices. *Int. J. Parallel Program.*, 25:525–549, December 1997.

[109] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T.-f. Ngai. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 348–357, Washington, DC, USA, 2009. IEEE Computer Society.

[110] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 2–15, New York, NY, USA, 1993. ACM.

[111] N. Megiddo and V. Sarkar. Optimal weighted loop fusion for parallel programs. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 282–291, New York, NY, USA, 1997. ACM.

[112] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 166–176, New York, NY, USA, 2009. ACM.

[113] B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. 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.

[114] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[115] D. Naishlos. Autovectorization in gcc. In *the GCC Developer's summit*, pages 105–118, June 2004.

[116] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *PLDI*, 2006.

[117] D. Nuzman and A. Zaks. Autovectorization in GCC – two years later. In *the GCC Developer's summit*, June 2006.

[118] D. Nuzman and A. Zaks. Outer-loop vectorization - revisited for short SIMD architectures. In *PACT*, October 2008.

[119] M. O'Boyle. MARS: a distributed memory approach to shared memory compilation. In *Proc. Language, Compilers and Runtime Systems for Scalable Computing*, Pittsburgh, May 1998. Springer-Verlag.

[120] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29:1184–1201, December 1986.

[121] E. Park, L.-N. Pouchet, J. Cavazos, A. Cohen, and P. Saddayapan. Predictive Modeling in a Polyhedral Optimization Space. In *International Symposium on Code Generation and Optimization (CGO'11)*, Chamonix France, 2011.

[122] D. J. Pearce, P. H. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1):4, 2007.

[123] PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.

[124] PoCC: the Polyhedral Compiler Collection. http://www-rocq.inria.fr/ pouchet/software/pocc/.

[125] S. Pop, A. Cohen, C. Bastoul, S. Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4þ GCC Developper's Summit*, Ottawa, Canada, June 2006. To appear.

[126] S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. In T. M. Conte, N. Navarro, W. mei W. Hwu, M. Valero, and T. Ungerer, editors, *HiPEAC*, volume 3793 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2005.

[127] B. Pottenger and R. Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th international conference on Supercomputing*, ICS '95, pages 444–448, New York, NY, USA, 1995. ACM.

[128] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 90–100, New York, NY, USA, 2008. ACM.

[129] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 144–156, Washington, DC, USA, 2007. IEEE Computer Society.

[130] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *IEEE/ACM Intl. Conf. on Code Generation and Optimization (CGO'07)*, pages 144–156, San Jose, California, Mar. 2007.

[131] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[132] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: convexity, pruning and optimization. In *Proceedings of the 38th*

annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '11, pages 549–562, New York, NY, USA, 2011. ACM.

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

[134] W. Pugh. Uniform techniques for loop optimization. In *Proceedings of the 5th international conference on Supercomputing*, ICS '91, pages 341–352, New York, NY, USA, 1991. ACM.

[135] W. Pugh. Counting solutions to presburger formulas: how and why. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 121–134, New York, NY, USA, 1994. ACM.

[136] W. Pugh and D. Wonnacott. Eliminating false data dependences using the omega test. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 140–151, New York, NY, USA, 1992. ACM.

[137] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 546–566, London, UK, 1994. Springer-Verlag.

[138] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing*, 9(4):365–389, 1995.

[139] X. Redon and P. Feautrier. Scheduling reductions. In *Proceedings of the 8th international conference on Supercomputing*, ICS '94, pages 117–125, New York, NY, USA, 1994. ACM.

[140] L. Renganarayana, U. Bondhugula, S. Derisavi, A. E. Eichenberger, and K. O'Brien. Compact multi-dimensional kernel extraction for register tiling. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 45:1–45:12, New York, NY, USA, 2009. ACM.

[141] G. Roth and K. Kennedy. Loop fusion in high performance fortran. In *Proceedings of the 12th international conference on Supercomputing*, ICS '98, pages 125–132, New York, NY, USA, 1998. ACM.

[142] S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 263–273, New York, NY, USA, 2007. ACM.

[143] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.

[144] J. Shin, J. Chame, and M. W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *PACT*, September 2002.

[145] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *CGO*, March 2005.

[146] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A unified framework for schedule and storage optimization. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 232–242, New York, NY, USA, 2001. ACM.

[147] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable autotuning framework for computer optimization. In *IPDPS'09*, Rome, May 2009.

[148] K. Trifunovic and A. Cohen. Enabling more optimizations in graphite: ignoring memory based dependences. In *GCC Summit*, Ottawa, Canada, October 2010.

[149] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjodin, and R. Upadrasta. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *2nd International Workshop on GCC Research Opportunities*, October 2010.

[150] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *(PACT'09)*, Raleigh, North Carolina, Sept. 2009.

[151] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, SIGPLAN '86, pages 176–185, New York, NY, USA, 1986. ACM.

[152] P. Tu and D. Padua. Array privatization for shared and distributed memory machines (extended abstract). *SIGPLAN Not.*, 28:64–67, January 1993.

[153] P. Tu and D. A. Padua. Automatic array privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 500–521, London, UK, 1994. Springer-Verlag.

[154] R. Upadrasta and A. Cohen. Potential and challenges of two-variable-per-inequality sub-polyhedral compilation. In *IMPACT 2011 First International Workshop on Polyhedral Compilation Techniques*, Chamonix, France, 2011.

[155] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In A. Mycroft and A. Zeller, editors, *CC*, volume 3923 of *Lecture Notes in Computer Science*, pages 185–201. Springer, 2006.

[156] N. Vasilache, C. Bastoul, A. Cohen, and S. Girbal. Violated dependence analysis. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 335–344, New York, NY, USA, 2006. ACM.

[157] N. Vasilache, A. Cohen, and L.-N. Pouchet. Automatic correction of loop transformations. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 292–304, Washington, DC, USA, 2007. IEEE Computer Society.

[158] S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer Berlin / Heidelberg, 2010.

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

[160] D. K. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, Rennes, France, 1993.

[161] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2:452–471, October 1991.

[162] M. Wolfe. Iteration space tiling for memory hierarchies. In *3rd SIAM Conf. on Parallel Processing for Scientific Computing*, pages 357–361, december 1987.

[163] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.

[164] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao. An integrated Simdization framework using virtual vectors. In *ICS*, 2005.

[165] J. Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Comput.*, 20:711–728, May 1994.

[166] J. Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Comput.*, 22:1621–1645, February 1997.

[167] Y.-Q. Yang, C. Ancourt, and F. Irigoin. Minimal data dependence abstractions for loop transformations. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC ’94, pages 201–216, London, UK, 1995. Springer-Verlag.