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

In Partial Fulfillment of the Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Specialty: Computer Science

Louis-Noël POUCHET

Subject:

ITERATIVE OPTIMIZATION IN THE POLYHEDRAL MODEL

Thesis committee:

Dr. Cédric Bastoul	Associate Professor at University of Paris-Sud
Dr. Albert Cohen	Senior Research Scientist at INRIA Saclay Ile-de-France
Pr. Paul Feautrier	Emeritus Professor at Ecole Normale Superieure de Lyon
Dr. Richard Lethin	Chief Executive Officer at Reservoir Labs Inc.
Pr. P. Sadayappan	Professor at the Ohio State University
Dr. Marc Schoenauer	Senior Research Scientist at INRIA Saclay Ile-de-France

Acknowledgments

Writing the acknowledgment chapter of your thesis is one of the most anticipated moment of the production of the manuscript. This is going to be your final pass on probably one of the most complex document you will ever have to write on your own. And then, time flies, as usual. You focus on the technical content, the polishing, the integration of your reviewers' suggested revisions. And then, you start a new job, a new life, thousands of kilometers away. And time goes on... Two years have gone by, leaving "To Be Done after graduation" as the only acknowledgment chapter of your dissertation...

Two years after. Yes. Let me look at the benefits: writing an acknowledgment chapter two years after graduation gives perspective to your thesis, its writing work effort (really, not that much in comparison of the rest), and more importantly at the end gives you the opportunity to thank those who really have influenced yourself, your work, and as a consequence the life that comes after that.

My first, and foremost, acknowledgment is dictated by both my brain and my heart: it goes to Albert, it has to go to Albert. He proved to be an amazing Ph.D advisor: the one that lets you do what you want, even if he thinks you are wrong (yes, mistakes are part of the learning process). The one that believes in you, even when you think your work is at best below average (yes, this happens all the time). The one that first reacts to your idea(s) by thinking "why is it right?" (yes, so many people instead think "why is it wrong?"). The one that shows you the way, one of many, to conduct ethically correct and interesting research without any self-interest or political consideration. Albert, fifteen good minutes in a week can have more impact than one useless hour every day, trust me. From this advisor-advisee relationship was born a fruitful work and personal relationship, that will last way beyond the current collaborations we have. I owe you a lot.

My second acknowledgment is dedicated to Cédric. I was immensely lucky to work with him, someone with whom I share so many points of interests in computer science, so many similar objectives about what a fruitful research should be about. We both truly believe that delivering good software is required to help disseminate any research effort, and helps other researcher focus on new problems. We both truly believe a single good paper is better than five average ones. We both truly believe living our passion and sharing our ideas is more important than the money we can make out of it. My only hope is that one day my software will have as much impact as CLooG had for our community.

I would now like to depart from "advisors" acknowledgment and particularly mention two people, well I should say two friends, Nicolas Vasilache and Uday Bondhugula. Both have set a bar that has set my expectation about a sucessful Ph.D. Uday has truly made ground-breaking contributions, providing both theoretical and practical results that made our community able to deliver high-performance optimizations on a variety of programs. Yet we quickly started to work together to improve upon his results, and he proved to be the definition of a true scientist: always interested in making things work better, no matter who finds the solution, and always interested in disseminating the results to the masses. Our collaboration led to a few papers I am particularly proud of, his spirit was always there to motivate me to reach better and more complete understanding of the problems at hand. From this work relationship emerged a personal relationship that grew in Paris and then in NYC, that will also last way beyond our current collaborations. Nicolas is another example of a true scientist. He was completing his Ph.D while

I was starting mine. While some could have focused on their own work, he spent so many hours discussing with me his research and mine, always making the required abstraction so that I can understand and participate. His ability to formulate problems, and his integrity in the conduct of research were the starting points of my thesis work. Our numerous "coffee breaks" (I still remember the explanation of wavefront tile scheduling with you literally making steps on the tile floor in front of our building!) are cheerful memories. Our joint experience at Reservoir was another proof of your scientific talent.

Many thanks have to be given also to my Thesis defense committee, where a total of 30,000 miles have been traveled just to attend the actual defense in Paris (well, including myself!). I was truly honored to have Paul Feautrier and P. Sadayappan as reviewers of my thesis. Paul proved, if any need to, to be an extremely open-minded reviewer, providing insightful and constructive criticism about my work. His seminal works have deeply inspired my research, to the point of my tribute title for my first two papers. I am also very grateful to Saday for his hard work and important feedback on my thesis. But, two years after, I am even more grateful for his supervision of my post-doc work. He is really an amazing character, who is committed to his work to an extent that I have never seen (Saday, have you heard, some people take breaks and go in "vacation"? Yes, it is a strange concept where people actually don't work for entire days in a row!). We have worked together on such a variety of topics that I can sincerely call him responsible for my learning of high-performance computing as a whole, and I still feel I have only scratched the surface of the research he is conducting. So many projects have emerged from our collaboration that the next ten years may not be enough to just complete what we have started. He is a mentor in the true definition of the word. I also would like to give a special thanks to Richard Lethin, who has accepted to travel from NYC for my defense. His company has developed an amazing compiler, that I was lucky enough to discover during a 4-month visit at Reservoir Labs. Richard is the living proof that one can live from polyhedral compilation research, and writing proposals on polyhedral compilation has been made so much easier thanks to his contributions and commitment. I would finally like to thank Marc Schoenauer to have accepted to lead my defense committee. I was especially happy to disseminate our work in adaptive compilation to the community of machine learning research.

I want to conclude this acknowledgment chapter by thanking all the people that I am proud to consider as my friends. They have been a constant support, they accepted my absence, my moods, my failures, and they all believed in me to an extent that is beyond common sense. I am who I am because of them. Caro, Greg, Vincent, David, Seb, Jerome, Steph, Charlotte, Aurel, Cecile, and all others: you owe a piece of this work. And I am saving the best for the end: this work is literally the product of the product of my Mother Martine and my Father André, without their thorough education and advice I would simply not be writing those words. They deserve so much tribute that words are unable to even start to express how grateful I am to them. Thank You.

This thesis is dedicated to *Julie ma filleule, Paul, Camille, Côme mon filleul, Lou-Anne, Titouan, Manon ma doudou (et filleule !), Louis and Zoé*. May this little piece of work inspire the young generation to commit to the Research effort.

Contents

1 Introduction **11**

I Optimization Framework **17**

2 Polyhedral Program Optimization **19**
2.1 Thinking in Polyhedra 19
2.2 Polyhedral Program Representation 20
2.2.1 Mathematical Background Snapshot 20
2.2.2 Static Control Parts 21
2.2.3 Statements 23
2.2.4 Iteration Domains 24
2.2.5 Access Functions 25
2.2.6 Schedules 25
2.3 Polyhedral Program Transformations 27
2.3.1 One-Dimensional Schedules 27
2.3.2 Multidimensional Schedules 28
2.4 Program Semantics Extraction 29
2.4.1 Data Dependence Representation 30
2.4.2 Building Legal Transformations 31
2.5 Code Generation 31
2.6 Summary 33

3 Semantics-Preserving Full Optimization Spaces **35**
3.1 Semantics-Preserving Affine Transformations 35
3.1.1 Program Dependences 36
3.1.2 Convex Set of One-Dimensional Schedules 36
3.1.3 Generalization to Multidimensional Schedules 38
3.1.4 Related Work 41
3.2 Semantics-Preserving Statement Interleavings 41
3.2.1 Encoding Statement Interleaving 42
3.2.2 Affine Encoding of Total Preorders 47
3.2.3 Pruning for Semantics Preservation 50
3.2.4 Related Work 52

4 Computing on Large Polyhedral Sets **53**
4.1 Computing on High-Dimensionality Polyhedra 53
4.2 Existing Techniques for Polyhedral Manipulation 54
4.2.1 Manipulation of Presburger Formulas 54
4.2.2 Dual and Double-description of Polyhedra 54
4.2.3 Number Decision Diagrams 55
4.3 Polyhedral Operations on the Implicit Representation 56
4.3.1 Required Polyhedral Operations 56
4.3.2 Redundancy Elimination 57
4.4 Efficient Dynamic Polytope Scanning 58
4.4.1 The Dynamic Scanning Problem 59
4.4.2 Scanning Points Using Projections 59
4.5 Fourier-Motzkin Projection Algorithm 61
4.5.1 The Genuine Algorithm 61
4.5.2 Known Issues 63
4.5.3 Redundancy-Aware Fourier-Motzkin Elimination 63
4.5.4 Some Related Works 65

II Constructing Program Optimizations **67**

5 Building Practical Search Spaces **69**
5.1 Introduction 69
5.1.1 The Trade-Off Between Expressiveness and Practicality 70
5.1.2 Different Goals, Different Approaches 72
5.2 Search Space Construction 73
5.2.1 One-Dimensional Schedules 73
5.2.2 Generalization to Multidimensional Schedules 76
5.2.3 Scanning the Search Space Polytopes 81
5.3 Related Work 82

6 Performance Distribution of Affine Schedules **85**
6.1 Scanning the Optimization Search Space 85
6.1.1 Experimental Setup 85
6.1.2 Exhaustive Space Scanning 86
6.1.3 Intricacy of the Best Program Version 86
6.1.4 The Compiler as an Element of the Target Platform 86
6.1.5 On the Influence of Compiler Options 89
6.1.6 Performance Distribution 89
6.2 Extensive Study of Performance Distribution 91
6.2.1 Experimental Protocol 91
6.2.2 Study of the `dct` Benchmark 91
6.2.3 Evaluation of Highly Constrained Benchmarks 95
6.2.4 Addressing the Generalization Issue 96

7 Efficient Dynamic Scanning of the Search Space **99**
7.1 Introduction . . . 99
7.2 Heuristic Search for One-Dimensional Schedules . . . 99
7.2.1 Decoupling Heuristic . . . 99
7.2.2 Discussion . . . 100
7.3 Heuristic Search for Multidimensional Schedules . . . 101
7.3.1 Schedule Completion Algorithm . . . 101
7.3.2 A Multidimensional Decoupling Heuristic . . . 102
7.3.3 Experiments on AMD Athlon . . . 104
7.3.4 Extensive Experimental Results . . . 105
7.4 Evolutionary Traversal of the Polytope . . . 107
7.4.1 Genetic Operators Design . . . 107
7.4.2 Experimental Results . . . 109

8 Iterative Selection of Multidimensional Interleavings **113**
8.1 Introduction . . . 113
8.2 Problem Statement . . . 114
8.2.1 Motivating Example . . . 114
8.2.2 Challenges and Overview of the Technique . . . 115
8.3 Optimizing for Locality and Parallelism . . . 115
8.3.1 Additional Constraints on the Schedules . . . 116
8.3.2 Computation of the Set of Interleavings . . . 118
8.3.3 Optimization Algorithm . . . 120
8.3.4 Search Space Statistics . . . 123
8.4 Experimental Results . . . 124
8.4.1 Experimental Setup . . . 124
8.4.2 Performance Improvement . . . 125
8.4.3 Performance Portability . . . 127
8.5 Related Work . . . 128

9 Future Work on Machine Learning Assisted Optimization **129**
9.1 Introduction . . . 129
9.2 Computing Polyhedral Program Features . . . 130
9.2.1 A Grain for Each Goal . . . 130
9.2.2 Some Standard Syntactic Features . . . 130
9.2.3 Polyhedral Features . . . 131
9.3 Knowledge Representation . . . 132
9.3.1 Dominant Transformation Extraction . . . 133
9.3.2 Optimized Program Abstract Characteristics . . . 134
9.4 Training Set and Performance Metrics . . . 134
9.4.1 Starting Benchmarks . . . 135
9.4.2 A Potentially infinite Set of Training Benchmarks . . . 135
9.4.3 Performance Metrics . . . 135
9.5 Putting It All Together . . . 136
9.5.1 Training Phase . . . 136
9.5.2 Compiling a New Program . . . 137

10 Conclusions **139**

A Correctness Proof for $\mathcal{O}$ **143**

Personal Bibliography **147**

Bibliography **149**

Chapter 1

Introduction

Compilation Issues on Modern Architectures

The task of compiling a program refers to the process of translating an input source code, which is usually written in a human-readable form, into a target language, which is usually a machine-specific form. The most standard application of compilation is to translate a source code into a binary code, that is a program that can be executed by the machine. Compiling a program has become an increasingly difficult challenge with the evolution of computers and computing needs as well. In particular, the programmer expects the compiler to produce *effective* codes on a wide variety of machines, making the most of their theoretical performance.

For decades it was thought that Moore's law on transistor density would also translate into increasing the processor operating speed. Hence more transistors was often associated with a faster execution time for the same program, with little or no additional effort to be done in the compiler for the upgraded chip. This era has come to an end, mostly because of dramatic thermal dissipation issues. As a consequence, the increase in the number of transistors now translates into more and more functional units operating at the same speed as five years ago.

With the emergence of specific and duplicated functional units started the trend of parallelism in modern machine. Instruction-Level Parallelism is exploited in most processors and increasingly complex mechanisms to manage it on the hardware has been a source for additional performance. On the software side, the compiler is asked to automatically adapt, or *transform* the input program to best fit the target architecture features, in particular for Single-Instruction-Multiple-Data programs. Moreover, nowadays a full processor core is replicated on the same chip, and the widespread adoption of multi-core processors and massively parallel hardware accelerators (GPUs) now urge production compilers to provide (automatic) coarse-grain parallelization capabilities as well.

Considering that most of the computational power is spent in loops over the same set of instructions, it is of high priority to efficiently map such loops on the target machines. High-level loop optimizations are necessary to achieve good performance over a wide variety of processors. Their performance impact can be significant because they involve in-depth program transformations that aim to sustain a balanced workload over the computational, storage, and communication resources of the target architecture. Therefore, it is mandatory that the compiler accurately models the target architecture as well as the effects of complex code restructuring.

However, most modern optimizing compilers use simplistic performance models that abstract away many of the complexities of modern architectures. They also rely on inaccurate dependence analysis, and lack a framework to model complex interactions of transformation sequences. Therefore they typically uncover only a fraction of the peak performance available on many applications. An ideal compiler is asked to achieve *performance portable* over an increasingly larger and heterogeneous set of architectures. For a given (multi-)processor architecture, the compiler attempts to map the proper grain of independent computation and the proper data locality to a complex hierarchy of memory, computing and interconnection resources. Despite five decades of intense research and development, it remains a challenging task for compiler designers and a frustrating experience for the programmers of high-performance applications. The performance gap between expert-written code and automatic optimization (including parallelization) of simple numerical kernels is *widening* with every hardware generation.

Iterative Compilation

In recent years, feedback-directed iterative optimization has become a promising direction to harness the full potential of future and emerging processors with modern compilers. Iterative compilation consists in testing different optimization possibilities for a given input program, usually by performing the transformation and evaluating its impact by executing the optimized program on the target machine. Building on operation research, statistical analysis and artificial intelligence, iterative optimization generalizes profile-directed approach to integrate precise feedback from the runtime behavior of the program into optimization algorithms. Through the many encouraging results that have been published in this area, it has become apparent that achieving better performance with iterative techniques depends on two major challenges.

1. *Search space expressiveness.* To achieve good performance with iterative techniques that are portable over a variety of architectures, it is essential for the transformation search space to be expressive enough to let the optimizations target a good usage of all important architecture components and address all dominant performance anomalies.
2. *Search space traversal.* It is also important to construct search algorithms (analytical, statistical, empirical) and acceleration heuristics (performance models, machine learning) that effectively traverse the search space by exploiting its static and dynamic characteristics.

Complex compositions of loop transformations are needed to make an effective use of modern hardware. Using the polyhedral compilation framework, it is possible to express those complex transformations into an algebraic formalism, exposing a rich structure and leveraging strong results in linear algebra. Unlike most other transformation frameworks, the polyhedral model allows focusing on the properties of the *result* of an arbitrarily complex sequence of transformations, without the burden of composing the properties of individual loop transformations to build the sequence. Moreover, with polyhedral compilation it is possible to design the loop nest optimization flow as a series of *optimization-space pruning* steps, rather than as a series of incremental optimization steps. Building on this idea, our work is the first to simultaneously address the two aforementioned challenges.

Part of the reason for the failure of optimizing compilers to match the performance of hand-written implementations can be found in the way loop nest optimizers attempt to break the global optimization problem down to simpler sub-problems. Classical optimizers walk a single program transformation path,

each step resulting from the resolution of a complex decision problem. We propose to model *sets of candidate program transformations as convex polyhedra*, incrementally pruning those polyhedra and deferring irrevocable decisions until it is guaranteed that we will not miss an important exploration branch leading to a much more profitable program transformation. Candidate program transformations represent arbitrarily complex sequences of loop transformations in a *fixed length* fashion — that is, as a point in a polyhedron. Polyhedron pruning involves the insertion of additional affine inequalities, embedding semantics-preservation, transformation uniqueness, target-specific performance modeling and heuristic simplifications. The symbolic pruning approach may at some point become intractable, for algorithmic complexity or non-linearity reasons. Only at such a point, an irrevocable decision has to be taken: it typically takes the form of the instantiation of several coefficients of the matrices defining the composition of program transformations being constructed.

We present in this thesis all required blocks to achieve the design and evaluation of a scalable, automatic and portable process for program optimization, based on the polyhedral program representation. We now summarize our main contributions.

Contributions

Convex characterization of semantics-preserving transformations

A program transformation must preserve the semantics of the program. The polyhedral program representation defines it with the finest granularity: each executed instance of a syntactic statement is considered apart. One of the most powerful feature of this representation is the ability to express a unique convex set of all semantics-preserving transformations, as shown by Vasilache. As a starting point for the optimization space construction, we present in Chapter 3 a *convex, polyhedral characterization of all legal multidimensional affine schedules for a program* with bounded scheduling coefficients. Building on Feautrier and Vasilache results, we provide the optimization framework with the most tractable expression of this set known to date.

Focusing on a subset of transformations such as loop fusion and loop distribution is of critical interest for the purpose of parallelization and memory behavior improvement. Yet extracting the convex subset of all legal compositions of these two transformations cannot be done in a straightforward fashion from the set of legal multidimensional schedules. We propose, for the first time, an *affine encoding of all legal multidimensional statement interleavings* for a program in Chapter 3. To achieve the modeling of this space, we first show the equivalence of the one-dimensional problem with the modeling of total preorders. We then present the first affine characterization of the set of all, distinct total preorders before generalizing to multi-dimensional interleavings. We finally present an algorithm to efficiently prune this set of all the interleavings that do not preserve the semantics.

Iterative compilation in the space of affine schedules

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

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

This thesis takes a more offensive position in this fight. To avoid diminishing returns in tuning individual phases or combinations of those, we collapse multiple optimization phases into a single, unconventional, iterative search algorithm. By construction, the search spaces we explore encompasses *all legal program transformations* in a particular class. Technically, we consider (1) the whole class of loop nest transformations that can be modeled as *one-dimensional schedules*; (2) a relevant subset of the class of transformations that can be modeled as *multidimensional schedules*; and (3) the subset of all *multidimensional statement interleavings*, the broadest class of loop fusion and distribution combinations.

This results in a significant leap in model and search space complexity compared to state-of-the-art applications of iterative optimization. Our approach is experimentally validated in a software platform we have built especially for this purpose. *LetSee*, the Legal Transformation Space Explorator, is the first complete platform for iterative compilation in the polyhedral model. It is integrated into the full-flavored iterative and model-driven compiler we have built during this thesis *PoCC*, the Polyhedral Compiler Collection. We present in Part II of this thesis the optimization search space construction and traversal algorithms, as well as the associated experimental results.

Scalable techniques for polytope projection and dynamic scanning

One cornerstone of iterative program optimization in the polyhedral model is the ability to efficiently build and traverse search spaces represented as polyhedra. In order to reach scalability on large program parts, it is required to move forward in two directions: provide scalable optimization algorithms, and provide scalable techniques to traverse polyhedral sets. The Fourier-Motzkin elimination algorithm is often thought of as not suitable for large problem solving, as its major drawback is to generate a high number of redundant constraints during the projection steps. This problem has particularly been observed for the task of generating a code scanning a polyhedron. We contradict this misconception by proposing in Chapter 4 a slightly modified version of this algorithm, which scales up to the hundreds of dimensions of the problems considered in this thesis. We leverage Le Fur's results to present a redundancy-less implementation, allowing to reshape the largest convex sets in a form suitable for dynamic scanning. Eventually we present a linear-time scanning technique which operates on the resulting polyhedron. All these results are implemented in the free software *FM*, a library for polyhedral operations based on the Fourier-Motzkin projection algorithm.

Performance Distribution of Affine Schedules

Although numerous previous work used the polyhedral framework to compute a program optimization, none provided quantitative experiments about the performance distribution of the different possibilities to transform a program. In Chapter 6 we propose to study and characterize the performance distribution of a search space of affine schedules by means of statistical analysis of the results, over an extensive set of program versions

We report the static and dynamic characteristics of a vast quantity of program versions, attempting to capture the largest amount of information about the performance distribution of affine schedules. As

a consequence of this study we experimentally validate a *subspace partitioning* based on the relative impact on performance of classes of coefficients. To the best of our knowledge, this is the first time such a characterization is performed and experimentally validated.

Heuristic Traversal

We build upon the results of Chapter 6 to motivate the design of several heuristic mechanisms to traverse the search spaces, these are presented in Chapter 7. First we present a decoupling heuristic tailored to the exploration of one-dimensional schedules. This heuristic is able to discover the wall-clock optimal schedule in our experiments, and is able to systematically outperform the native compiler. For instance on MatrixMultiply the process outperforms ICC by 6× on an `AMD Athlon 3700+` in less than 20 runs. We then extend this result to the case of multidimensional schedules, and validate our approach on three architectures, including the VLIW STMicroelectronics `ST231` processor and the `AMD Au1500` processor. In average, our iterative process is able to outperform the native compiler by 13% for the `ST231` processor, over the highly optimized ST200cc compiler. Performance improvements of up to 37% in average are obtained for the other evaluated single-core processors. To further improve the speed of the traversal and reduce the number of candidates to test for, we propose a Genetic Algorithm approach. We design the first generic operators tailored to preserve the semantics of the program while exploring a rich set of loop transformations. To the best of our knowledge, this is the first time that genetic operators closed under affine constraints are developed. We experimentally observed that for larger benchmarks, the GA performs 2.46× better in average than the decoupling heuristic and up to 16× better.

Iterative selection of multidimensional statement interleavings

The selection of a profitable combination of loop transformations is a hard combinatorial problem. We propose in Chapter 8 to explore an iterative approach that is based on the selection of multidimensional statement interleaving, modeling generalized forms of loop fusion and distribution up to enabling affine transformations. This subspace focuses the search on the most difficult part of the problem: it models transformations that have a significant impact on the overall performance, isolated from enabling transformations for which effective heuristics exist. We propose a practical optimization algorithm to explore the pruned search space polyhedron, while heuristically building a profitable, semantics-preserving enabling transformation.

Compared to the state-of-the-art in loop fusion, we consider arbitrarily complex sequences of enabling transformations, in a multidimensional setting. This generalization of loop fusion is called *fusability* and results in a dramatic broadening of the expressiveness of the optimizer. We model candidate statement interleavings for fusability as total preorders, and we reduce the problem of deciding the fusability of statements to the existence of compatible pairwise loop permutations. Our algorithms are applied to relevant benchmarks, demonstrating good scalability and strong performance improvements over state-of-the-art multi-core architectures and compilers. We experimented on three high-end machines ranging from 4 to 24 cores. Our approach systematically outperforms the best auto-parallelizing compilers Intel ICC and IBM XL, by a factor up to 15× in our experiments. Compared to the other iterative search techniques presented in this thesis, this approach does not focus on single-threaded programs: automatic and aggressive coarse-grain parallelization is achieved, thanks to a generalized and improved version of Bondhugula's algorithm for tiling hyperplane computation.

Openings on machine learning assisted compilation

For the past decade, compiler designers have looked for automated techniques to improve the quality and portability of the optimization heuristics implemented in compilers. They naturally looked towards *machine learning* processes, mainly to improve the search speed for iterative compilation processes, and to improve the performance of a dedicated optimization heuristic. For such cases, the main idea is to build automated processes to help computing a good optimization based on the result of previous compilations. We present in Chapter 9 some critical remarks about how polyhedral compilation can be harnessed by a machine learning oriented compiler. We describe the most critical steps and give key observations on how to reshape the process according to the polyhedral constraints.

Part I

Optimization Framework

Chapter 2

Polyhedral Program Optimization

"All parts should go together without forcing. You must remember that the parts you are reassembling were disassembled by you. Therefore, if you can't get them together again, there must be a reason. By all means, do not use a hammer."

– IBM Manual, 1925

2.1 Thinking in Polyhedra

Most compiler internal representations match the inductive semantics of imperative programs (syntax tree, call tree, control-flow graph, SSA). In such reduced representations of the dynamic execution trace, a 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 optimizations which often need to consider a representation granularity at the level of dynamic *statement instances*. For example, complex transformations like loop interchange, fusion or tiling operate on the execution order of statement instances [126]. Due to compilation-time constraints and to the lack of an adequate algebraic representation of the semantics of loop nests, traditional (non-iterative) compilers are unable to adapt the schedule of statement instances of a program to best exploit the architecture resources. For example, compilers can typically not apply any transformation if data dependences are non-uniform (unimodular transformations, tiling), if the loop trip counts differ (fusion) or simply because profitability is too unpredictable. As a simple illustration, consider the Ring-Roberts edge detection filter shown in Figure 2.1. While it is straightforward to detect a high level of data reuse between the two loop nests, none of the compilers we considered — Open64 4.0, ICC 10.0, PathScale 3.0, GCC 4.2.0 — were able to apply loop fusion for a potentially 50% cache miss reduction when arrays do not fit in the data cache (plus additional scalar promotion and instruction-level-parallelism improvements). Indeed, this apparently simple transformation actually requires a non-trivial composition of (two-dimensional) loop shifting, fusion and peeling.

To build complex loop transformations, an alternative is to represent programs in the *polyhedral model*. It is a flexible and expressive representation for loop nests with statically predictable control flow. The polyhedral model captures control-flow and data-flow with three linear algebraic structures, described in the following sections. Such loop nests amenable to algebraic representation are called

static control parts (SCoP) [39, 51].

Polyhedral program optimization is a three stage process. First, the program is analyzed to extract its polyhedral representation, including dependence information and access pattern. This is the subject of Section 2.2. Following Section 2.3 and Section 2.4 present the second step of polyhedral program optimization, which is to pick a transformation for the program. Such a transformation captures in a single step what may typically correspond to a sequence of several tens of textbook loop transformations [51]. It takes the form of a carefully crafted affine schedule, together with (optional) iteration domain or array subscript transformations. Finally, syntactic code is generated back from the polyhedral representation on which the optimization has been applied, as discussed in Section 2.5.

```
  /* Ring blur filter */
  for (i=1;i<lg-1;i++)
    for (j=1;j<wt-1;j++)
R     Ring[i][j]=(Img[i-1][j-1]+Img[i-1][j]+Img[i-1][j+1]+
                  Img[i][j+1]  +            Img[i][j-1]  +
                  Img[i+1][j-1]+Img[i+1][j]+Img[i+1][j+1])/8;

  /* Roberts edge detection filter */
  for (i=1;i<lg-2;i++)
    for (j=2;j<wt-1;j++)
P     Img[i][j]=abs(Ring[i][j]-Ring[i+1][j-1])+
                abs(Ring[i+1][j]-Ring[i][j-1]);
```

Figure 2.1: Ring-Roberts edge detection for noisy images

2.2 Polyhedral Program Representation

The polyhedral model takes its origin in the work of Karp, Miller and Winograd for the automatic mapping of systems of uniform recurrence equations [62]. Later work motivated by systolic arrays generalized to systems of linear and affine recurrence equations [97, 129], along with the connection to standard imperative programs [39, 41]. Although more recent work tries to unleash the power of polyhedral optimization by broadening the applicability of the techniques [56, 17], we stick in this section to the presentation of a more "standard" polyhedral framework operating on static control parts. In the following we first recall some of the key concepts of polyhedral theory, before defining the elements used to represent a program into the polyhedral framework.

2.2.1 Mathematical Background Snapshot

Polyhedral optimization is a vast research topic, and providing an extensive background on the subject is clearly out of the scope of this thesis. Instead, we will recall on a need-to-know basis the key definitions and results of polyhedral theory, and refer the reader to a more extensive description from other works.

As a starting point of polyhedral optimization, we define the concept of affine functions and polyhedron, the two fundamental bricks of program representation in the polyhedral model.

Definition 2.1 (Affine function) *A function $f : \mathbb{K}^m \to \mathbb{K}^n$ is affine if there exists a vector $\vec{b} \in \mathbb{K}^n$ and a matrix $A \in \mathbb{K}^{m\times n}$ such that:*

$$\forall \vec{x} \in \mathbb{K}^m, \;\; f(\vec{x}) = A\vec{x} + \vec{b}$$

Definition 2.2 (Affine hyperplane) *An affine hyperplane is an $m-1$ dimensional affine sub-space of an m dimensional space.*

Definition 2.3 (Polyhedron) *A set $\mathcal{P} \in \mathbb{K}^m$ is a polyhedron if there exists a system of a finite number of inequalities $A\vec{x} \leq \vec{b}$ such that:*

$$\mathcal{P} = \{\vec{x} \in \mathbb{K}^m \mid A\vec{x} \leq \vec{b}\}$$

Definition 2.4 (Parametric polyhedron) *Given $\vec{n}$ the vector of symbolic parameters, $\mathcal{P}$ is a parametric polyhedron if it is defined by:*

$$\mathcal{P} = \{\vec{x} \in \mathbb{K}^m \mid A\vec{x} \leq B\vec{n} + \vec{b}\}$$

Definition 2.5 (Polytope) *A polytope is a bounded polyhedron.*

The reader may refer to Schrijver's work [101] for an extensive description of polyhedral theory, and is encouraged to navigate through Feautrier's pioneering work about efficient solutions for automatic parallelization of static-control programs [41, 42].

2.2.2 Static Control Parts

The polyhedral representation models a widely used subclass of programs called *Static Control Parts* (SCoP). A SCoP is a maximal set of consecutive instructions such that:

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

It is worth noting that many scientific codes respect the SCoP and static reference conditions, at least on hot spots of the code. A survey of Girbal et al. highlights the high proportion of SCoP in these codes [51]. An empirical well-known observation is that 80% of the processor time is spent on less than 20% of the code, yielding the need to aggressively optimize these code segments. These segments are most of the time in loop nests, and we refer to them as *kernels*. The polyhedral model was first aimed at modeling these kernels (under the SCoP and static reference conditions), and to be able to perform transformations (meaning changing the execution order but keep the output order) on these kernels.

At first glance the definition of a SCoP may seem restrictive, but many programs which does not respect those conditions directly can thus be expressed as SCoPs. A pre-processing stage (typically inside a compiler architecture) can ease the automatic raising of SCoPs.

Pre-processing for SCoPs *Constant propagation* is the process of substituting a symbolic constant by its value. As the affine condition forbids to multiply a parameter with for instance an iterator, one can resort to substituting the constant parameter by its value as shown in Figure 2.2. We assume that constant folding (the process of statically compute the value of a constant from its arithmetic expression) is systematically performed. Let us mention that constant propagation should not be performed in a systematic fashion, only when it enables SCoP formation. This observation takes place in the mathematical complexity of the algorithms used in polyhedral compilation: scalar loop bounds may translate into very

large coefficients during system resolution, significantly exceeding the standard Integer representation. One has then to resort to large number arithmetic libraries such as GMP, that may significantly increase the compilation time.

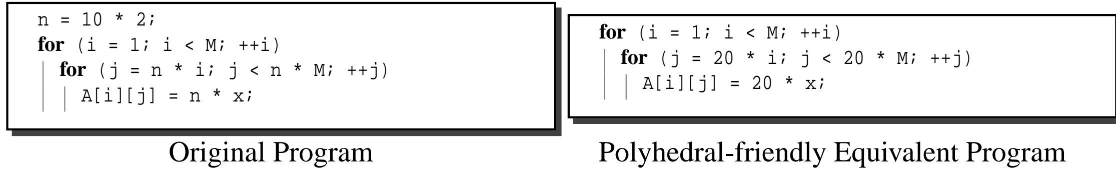


Figure 2.2: Constant Propagation

Loop normalization is also very useful when considering non-unit loop stride. Although other frameworks such as the $\mathcal{Z}$-polyhedral model directly supports such loop strides [57] by relying on Integer Lattices to represent programs, a simple loop normalization step can enable the representation of such programs as shown in Figure 2.3. We also perform another loop normalization step, to make loops 0-*normalized* as non-negative iteration spaces simplify the design of optimization algorithms.

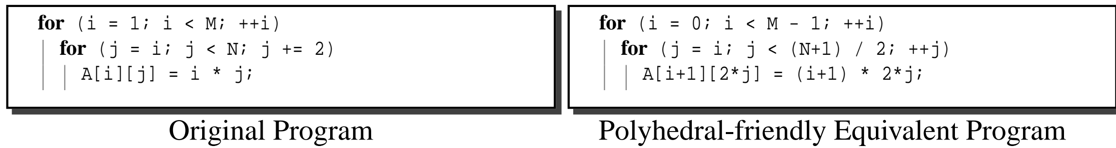


Figure 2.3: Loop Normalization

Another pre-processing stage is *WHILE-loop DO-loop conversion*, which can be associated with induction variable detection [4]. Figure 2.4 gives an example of such a conversion, which is implemented in the GRAPHITE framework.

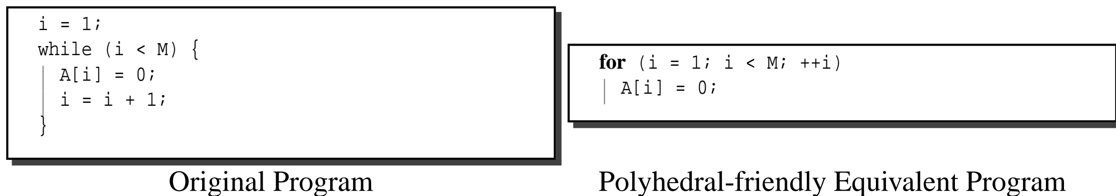
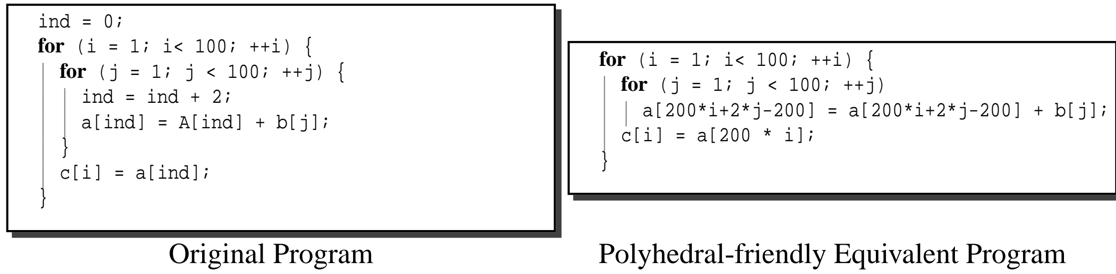


Figure 2.4: WHILE-loop DO-loop conversion

Finally, let us mention *induction variable substitution* [4] which has proved to be very useful to normalize access patterns in an affine form. An example is shown in Figure 2.5.

Several other pre-processing stages can enable the formation of SCoPs. Let us mention *inlining*, *goto/break removal*, *pointer substitution* among many others. The reader may refer to Allen and Kennedy's work for details on them [4].

Static representation of non-static control flow Recent work led by Mohamed-Walid Benabderrahmane have shown that the limitation to static control-flow is mainly artificial [17]. One can resort to



```
ind = 0;
for (i = 1; i< 100; ++i) {
  for (j = 1; j < 100; ++j) {
    ind = ind + 2;
    a[ind] = A[ind] + b[j];
  }
  c[i] = a[ind];
}
```

Original Program

```
for (i = 1; i< 100; ++i) {
  for (j = 1; j < 100; ++j)
    a[200*i+2*j-200] = a[200*i+2*j-200] + b[j];
  c[i] = a[200 * i];
}
```

Polyhedral-friendly Equivalent Program

Figure 2.5: Induction Variable Substitution

converting control-flow dependences into data dependences, together with conservative affine approximations of the iteration spaces to model almost any program with a polyhedral representation. While we do not explicitly leverage this extended representation in this thesis, *all techniques described in this manuscript are fully compatible with it*. In other words, a polyhedral representation can be extracted from arbitrary programs and the optimization techniques presented in this thesis applied seamlessly on those programs.

A complete description of static control parts was given by Xue [128] and their applicability to compute intensive, scientific or embedded applications have been extensively discussed by Girbal et al. and Palkovič [51, 89]. Frameworks to highlight SCoPs in general programs and to extract both iteration domains and subscript functions already exist or are in active development in compiler platforms like WRAP-IT/URUK for Open64 [51], Graphite for GCC [90], IBM XL/C compiler, the ROSE Compiler, R-Stream from Reservoir Labs, and in the prototype compilers PoCC and Loopo.

2.2.3 Statements

A *polyhedral statement* is the atomic block for polyhedral representation. To each statement is associated an iteration domain, a set of access functions and a schedule, as detailed in the following. Note that a disconnection may exist between a statement in the input source code and a statement in the polyhedral representation. Several passes of the compiler that occur before polyhedral extraction, such as for instance inlining and SSA conversion, may change the internal program representation.

We formally define a polyhedral statement as follows.

Definition 2.6 (Polyhedral statement) *A polyhedral statement is a program instruction that is:*

- *not an* `if` *conditional statement with an affine condition*
- *not a* `for` *loop statement with affine loop bounds*
- *having only affine subscript expressions for array accesses*
- *not generating control-flow side effects*

As a concrete example, consider the program of Figure 2.6. Statement T contains a data-dependent conditional, but can still be modeled in the representation by considering it as a single statement: no side-effect do exist with other statements. Similarly for statement U: the function call `sqrt` does not have any side-effect on data and control flows.

```
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++) {
R       A[i][j] = A[i][j] + u1[i]*v1[j]
        if (N - i > 2)
S          A[i][j] -= 2;
      }
T   res = A[0][0] == 0 ? u1[i] : v1[j];
U   dres = sqrt(res);
```

Figure 2.6: A Complex Program Example

Several techniques can be used to increase or decrease the number of polyhedral statements in the representation. Single assignment and three-address code conversion typically increase the freedom to schedule each instruction, by assigning to each of them a different schedule. Nevertheless, due to the complexity of the mathematical operations, one typically wishes to reduce the number of polyhedral statements. Macro-block formation is a simple process increasing the scalability of the optimization algorithms, in particular when starting from a three-address code. Yet the design of an efficient and systematic block formation heuristic, that correctly balances the schedule freedom (e.g., parallelism opportunities) versus the reduction of the number of statements, has still to be devised and is left as a future work of this thesis.

2.2.4 Iteration Domains

Iteration domains capture the dynamic instances of all statements — all possible values of surrounding loop iterators — through a set of affine inequalities. For example, statement R in Figure 2.1 is executed for every value of the pair of surrounding loop counters, called the *iteration vector*: the iteration vector of statement R is $\vec{x}_R = (i, j)$. Hence, the iteration domain of R is defined by its enclosing loop bounds:

$$\mathcal{D}_R = \{i, j \mid 1 \le i \le \mathrm{lg} - 1 \wedge 1 \le j \le \mathrm{wt} - 1\}$$

which forms a polyhedron (a space bounded by inequalities, a.k.a. *hyperplanes* or *faces*). To model iteration domains whose size are known only symbolically at compile-time, we resort to *parametric* polyhedra. Program constants which are unknown at compile-time are named *global parameters* and the parameters vector is noted $\vec{n}$. For instance for the Ring-Roberts example lg and wt are global parameters and $\mathcal{D}_R$ is a parametric form of $\vec{n} = \begin{pmatrix}\mathrm{lg} & \mathrm{wt}\end{pmatrix}$.

Each integral point inside this polyhedron corresponds to exactly one execution of statement R, and its coordinates in $\mathcal{D}_R$ matches the values of the loop iterators at the execution of this instance. This model let the compiler manipulate statement execution and iteration ordering at the most precise level.

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

$$\mathcal{D}_R : \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ \mathrm{lg}-1 \\ -1 \\ \mathrm{wt}-1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & -1 & 0 & 1 & -1 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ \mathrm{lg} \\ \mathrm{wt} \\ 1 \end{pmatrix} \ge \vec{0}$$

2.2.5 Access Functions

Access functions capture the data locations on which a statement operates. In static control parts, memory accesses are performed through array references (a variable being a particular case of an array). We restrict ourselves to subscripts of the form of affine expressions which may depend on surrounding loop counters (e.g., i and j for statement R) and global parameters (e.g., lg and wt in Figure 2.1). Each subscript function is linked to an array that represents a read or a write access. For instance, the subscript function for the read reference `Img[i-1][j]` of statement R is simply `Img[`$f(\vec{x}_R)$`]` with:

$$f(i,j) = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} = (i-1, j)$$

Other kinds of array references (that is, non-affine ones) have to be modeled conservatively by their affine hull. Pointer arithmetic is forbidden — except when translated by a former restructuring pass to array-based references [45] — and function calls have to be either inlined or checked for the lack of side-effect.

2.2.6 Schedules

Iteration domains define exactly the set of dynamic instances for each statement. However, this algebraic structure does not describe the order in which each instance has to be executed with respect to other instances. Of course, we do not want to rely on the inductive semantics of the sequence and loop iteration for this purpose, as it would break the algebraic reasoning about loop nests.

A convenient way to express the execution order is to give each instance an execution date. It is obviously impractical to define all of them one by one since the number of instances may be either very large or unknown at compile time. An appropriate solution is to define, for each statement, a *scheduling* function that specifies the execution date for each instance of a corresponding statement. For tractability reasons, we restrict these functions to be affine (relaxation of this constraint may exist [8], but challenges the code generation step [10]).

A schedule is a function which associates a logical execution date (a timestamp) to each execution of a given statement. In the target program, statement instances will be executed according to the increasing order of these execution dates. Two instances (possibly associated with distinct statements) with the same timestamp can be run in parallel. This date can be either a scalar (we will talk about one-dimensional schedules), or a vector (multidimensional schedules).

Definition 2.7 (Affine schedule) *Given a statement S, a p-dimensional affine schedule Θ^R is an affine form on the outer loop iterators $\vec{x}_S$ and the global parameters $\vec{n}$. It is written:*

$$\Theta^S(\vec{x}_S) = \mathbf{T}_S \begin{pmatrix} \vec{x}_S \\ \vec{n} \\ 1 \end{pmatrix}, \quad \mathbf{T}_S \in \mathbb{K}^{p \times dim(\vec{x}_S) + dim(\vec{n}) + 1}$$

Early work often required the affine schedule to be unimodular or at least invertible [6, 78, 98], mainly due to code generation limitations [10]. Other work such as Feautrier's have used non-negative scheduling coefficients in order to design minimization functions on the schedule latency [41]. Others

let the schedule be rational, for instance to model resource constraints. In this thesis we do not impose any constraint on the form of the schedule, as we use the CLooG code generator which does not require special properties on the schedule [10]. In general, we use in this thesis $\mathbb{K} = \mathbb{Z}$ unless explicitly specified otherwise.

Multidimensional dates can be seen as clocks: the first dimension corresponds to days (most significant), next one is hours (less significant), the third to minutes, and so on. As said, a schedule associates a timestamp to each executed instance. As an illustration, let us consider the following schedules for the Ring-Roberts example:

$$\Theta^R(\vec{x}_R) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} . \begin{pmatrix} i & j & \text{lg} & \text{wt} & 1 \end{pmatrix}^T = (i, j)$$
$$\Theta^S(\vec{x}_S) = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} . \begin{pmatrix} i & j & \text{lg} & \text{wt} & 1 \end{pmatrix}^T = (i + lg, j)$$

As Θ functions are 2-dimensional, timestamps are vectors of dimension 2. One can now compute the timestamp associated with each point of the iteration domain as specified by $\Theta^R : \mathcal{D}_R \to \mathbb{Z}^2$

$$\begin{array}{rcl} (1,1)_R & \to & (1,1) \\ (1,2)_R & \to & (1,2) \\ \vdots & & \vdots \\ (\text{lg}-2, \text{wt}-2)_R & \to & (\text{lg}-2, \text{wt}-2) \end{array}$$

Similarly for S and $\Theta^S : \mathcal{D}_S \to \mathbb{Z}^2$

$$\begin{array}{rcl} (1,1)_S & \to & (1+\text{lg}, 1) \\ (1,2)_S & \to & (1+\text{lg}, 2) \\ \vdots & & \vdots \\ (\text{lg}-3, \text{wt}-2)_S & \to & (2\times\text{lg}-3, \text{wt}-2) \end{array}$$

The schedule of statement R orders its instances according to i first and then j. This matches the structure of the loops of Figure 2.1. This is similar for statement P, except for the offset on the first time-dimension which states that the first nest runs before the second one: while the largest value of the first time-dimension for R is $\text{lg}-2$, the smallest value of the first dimension of P is $\text{lg}-1$. Hence the loop surrounding P "starts" after the loop surrounding R.

More formally, in the target program the execution order of the instances is the given by the *lexicographic ordering* on the set of associated timestamps. That is, for a given pair of instances $\langle \vec{x}_R, \vec{x}_S \rangle$, $\vec{x}_R$ is executed before $\vec{x}_S$ if and only if:

$$\Theta^R(\vec{x}_R) \prec \Theta^S(\vec{x}_S)$$

where $\prec$ denotes the lexicographic ordering. We recall that $(a_1, \ldots, a_n) \prec (b_1, \ldots, b_m)$ iff there exists an integer $1 \le i \le min(n,m)$ s.t. $(a_1, \ldots, a_{i-1}) = (b_1, \ldots, b_{i-1})$ and $a_i < b_i$.

Returning to the example, as we have:

$$\forall \vec{x}_R \in \mathcal{D}_R,\ \forall \vec{x}_S \in \mathcal{D}_S, \quad \Theta^R(\vec{x}_R) \prec \Theta^R(\vec{x}_R)$$

then all instances of R are executed before any instance of S, as in the original code.

2.3 Polyhedral Program Transformations

A transformation in the polyhedral model is represented as a set of affine schedules, one for each polyhedral statements, together with optional modification of the polyhedral representation. We can distinguish two families of transformations. (1) *Schedule-only* transformations operates only on the schedule of instructions, without changing the iteration domains and the number of statements in the polyhedral representation. These are described in this section, and the operation research algorithms presented in this thesis operates explicitly only on such transformations. (2) *Schedule and representation* transformations require to alter the polyhedral representation to be performed. Two of most well-known transformations falling into that category are loop tiling and loop unrolling [51]. For such transformations, we rely on *enabling* their application by computing a schedule with some dedicated properties (e.g., permutability), while delegating to an external process their actual application, typically right before code generation.

2.3.1 One-Dimensional Schedules

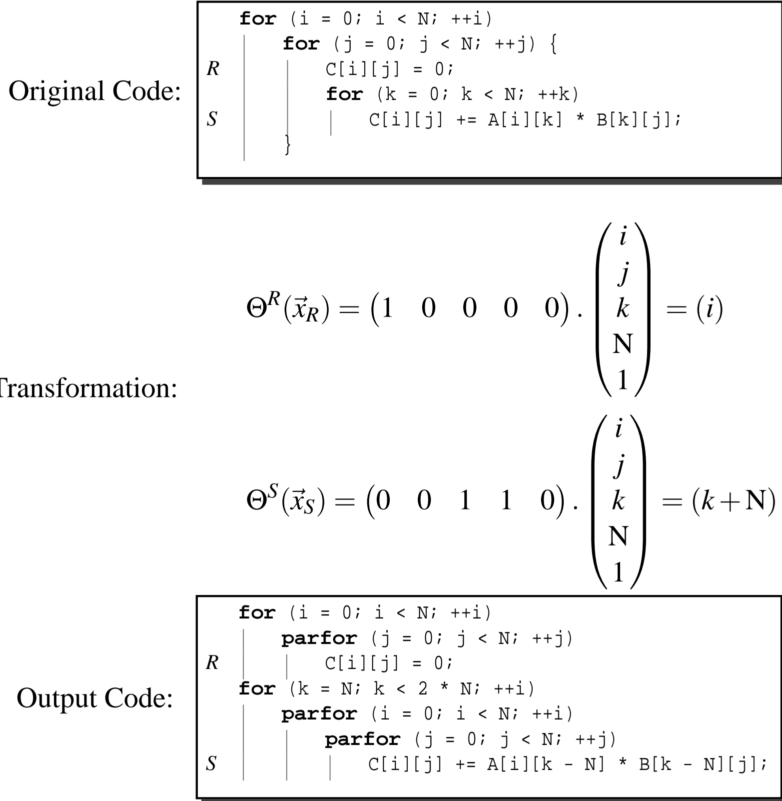
A *one-dimensional* schedule expresses the program as a single *sequential* loop, possibly enclosing one or more *parallel* loops. Affine schedules have been extensively used to design systolic arrays [97] and in automatic parallelization programs [41, 35, 54], then have seen many other applications.

Given a statement S, a one-dimensional affine schedule is an affine form on the outer loop iterators $\vec{x}_S$ and the global parameters $\vec{n}$ where $\mathbf{T}_S$ is a constant **row** matrix. Such a representation is much more expressive than sequences of primitive transformations, since a single one-dimensional schedule may represent a potentially intricate and long sequence of any of the transformations shown in Figure 2.7. All these transformations can be represented as a partial order in the space of all instances for all statements, and such orderings may be expressed with one-dimensional scheduling functions [123].

Transformation	Description
`reversal`	Changes the direction in which a loop traverses its iteration range
`skewing`	Makes the bounds of a given loop depend on an outer loop counter
`interchange`	Exchanges two loops in a perfectly nested loop, a.k.a. `permutation`
`peeling`	Extracts one iteration of a given loop
`shifting`	Reorder loops
`fusion`	Fuses two loops, a.k.a. `jamming`
`distribution`	Splits a single loop nest into many, a.k.a. `fission` or `splitting`

Figure 2.7: Possible Transformations Embedded in a One-Dimensional Schedule

For a concrete intuition of one-dimensional schedule, consider the matMult example in Figure 2.8. The schedules $\Theta^R(\vec{x}_R) = (i)$ and $\Theta^S(\vec{x}_S) = (k + \mathrm{N})$ consist in a composition of (1) distribution of the two statements, and (2) interchange of loops k and i for the second statements. Note that as we use one-dimensional schedules, only the execution order of the outer-most time dimension is specified: the remaining dimensions that are required to scan the original domain can be executed in any order, including in parallel. As many programs do not contain such an amount of parallelism, for those programs a one-dimensional schedule cannot describe a correct execution order.



$$\Theta^R(\vec{x}_R) = \begin{pmatrix}1 & 0 & 0 & 0 & 0\end{pmatrix} . \begin{pmatrix} i \\ j \\ k \\ \mathrm{N} \\ 1 \end{pmatrix} = (i)$$

$$\Theta^S(\vec{x}_S) = \begin{pmatrix}0 & 0 & 1 & 1 & 0\end{pmatrix} . \begin{pmatrix} i \\ j \\ k \\ \mathrm{N} \\ 1 \end{pmatrix} = (k+\mathrm{N})$$

Figure 2.8: matMult kernel

2.3.2 Multidimensional Schedules

A multidimensional schedule expresses the program as one or more nested sequential loops, possibly enclosing one or more parallel loops. Given a statement S, a multidimensional schedule is also an affine form on the outer loop iterators $\vec{x}_S$ and the global parameters $\vec{n}$ with the notable difference that $\mathbf{T}_S$ is a **matrix** of constants.

Existence and decidability results on multidimensional schedules have been proved mostly by Feautrier [41, 42]. We recall that, unlike with one-dimensional affine schedules, *every static control program has a multidimensional affine schedule* [42]. Hence the application domain extends to *all* static control parts in general programs.

Multidimensional affine schedules support arbitrary complex compositions of a wide range of program transformations. Moreover, the expressiveness is significantly increased compared to one-dimensional schedule. Considering the transformations reported in Figure 2.7, multidimensional schedules represent *any* composition of those. As an example, one can specify loop interchange on the whole loop depth, enabling the representation of the order (i,k,j) for the loops of the MatMult program, as shown in Figure 2.9.

This increased expressiveness translates into the fact that *any loop transformation can be represented in the polyhedral representation* [126].

Several frameworks have been designed to facilitate the expression of such transformations [42, 63], or to enable their composition and semi-automatic construction [51, 114, 77]. As illustration, a trivial

```
   for (i = 0; i < N; ++i)
      for (j = 0; j < N; ++j)
R        C[i][j] = 0;
   for (i = 0; i < N; ++i)
      for (k = 0; k < N; ++k)
         for (j = 0; j < N; ++j)
S           C[i][j] += A[i][k] * B[k][j];
```

Figure 2.9: MatMult kernel with $\Theta^R = (0, i, j)$ and $\Theta^S = (1, i, k, j)$

loop fusion is not possible to improve data locality on the Ring-Roberts kernel in Figure 2.1. Because of both data dependences and non-matching loop bounds, only a partial loop fusion is possible, which translates into a sequence of, e.g., *fusion*, *shifting* and *index-set splitting* [126]. Using multidimensional schedules, a correct transformation (found using *chunking* [13]) is simply: $\theta_R(i, j) = (i, j)$ and $\theta_P(i, j) = (i+2, j)$. The corresponding target code is the result of a quite complex composition of syntactic transformations, as shown in Figure 2.10.

```
   if (wt == 2) {
     for (i=1; i < lg-1; i++) {
R      Ring[i][1]=(Img[i-1][0]+Img[i-1][1]+Img[i-1][2]+
                   Img[i][2]  +           Img[i][0]  +
                   Img[i+1][0]+Img[i+1][1]+Img[i+1][2])/8;
     }
   }
   if (wt >= 3) {
     for (i=1; i < min(lg-1,2); i++) {
       for (j=1; j < wt-1; j++) {
R        Ring[i][j]=(Img[i-1][j-1]+Img[i-1][j]+Img[i-1][j+1]+
                     Img[i][j+1]  +           Img[i][j-1]  +
                     Img[i+1][j-1]+Img[i+1][j]+Img[i+1][j+1])/8;
       }
     }
     for (i=1; i < lg-1; i++) {
R      Ring[i][1]=(Img[i-1][0]+Img[i-1][1]+Img[i-1][2]+
                   Img[i][2]  +           Img[i][0]  +
                   Img[i+1][0]+Img[i+1][1]+Img[i+1][2])/8;
       for (j=1; j < wt-1; j++) {
P        Img[i-2][j]=abs(Ring[i-2][j]-Ring[i-1][j-1])+
                     abs(Ring[i-1][j]-Ring[i-2][j-1]);
R        Ring[i][j]=(Img[i-1][j-1]+Img[i-1][j]+Img[i-1][j+1]+
                     Img[i][j+1]  +           Img[i][j-1]  +
                     Img[i+1][j-1]+Img[i+1][j]+Img[i+1][j+1])/8;
       }
     }
   if ((wt >= 3) && (lg >= 3)) {
     for (j=1; j < wt-1; j++) {
P      Img[lg-2][j]=abs(Ring[lg-2][j]-Ring[lg-1][j-1])+
                    abs(Ring[lg-1][j]-Ring[lg-2][j-1]);
     }
   }
```

Figure 2.10: Optimized Version of Ring-Roberts

2.4 Program Semantics Extraction

A central concept of program optimization is to preserve the semantics of the original program through the optimization steps. Obviously not all transformations, and hence not all affine schedules, do system-

atically preserve the semantics, for all programs. To compute a *legal* transformation we resort to first extracting the *data dependences* shaped in a polyhedral representation before constraining the schedules to respect the computed dependences.

2.4.1 Data Dependence Representation

Two statements instances are in *dependence relation* if they access the same memory cell and at least one of these accesses is a write operation. For a program transformation to be correct, it is necessary to preserve the original execution order of such statement instances and thus to know precisely the instance pairs in dependence relation. In the algebraic program representation depicted earlier, it is possible to characterize exactly the set of instances in dependence relation in a very synthetic way.

Three conditions have to be satisfied to state that a statement instance $\vec{x}_R$ depends on a statement instance $\vec{x}_S$. (1) They must refer the same memory cell, which can be expressed by equating the subscript functions of a pair of references to the same array. (2) They must be actually executed, i.e. $\vec{x}_S$ and $\vec{x}_R$ have to belong to their corresponding iteration domains. (3) $\vec{x}_S$ is executed before $\vec{x}_R$ in the original program.

Each of these three conditions may be expressed using affine inequalities. The consequence is that exact sets of instances in dependence relation can be represented using affine inequality systems. The exact matrix construction of the affine constraints of the dependence polyhedron used in this thesis was formalized by Feautrier [39], and more specifically we use its description as given by Bastoul [11, 14].

```
  for (i = 0; i <= n; i++) {
R |  s[i] = 0;
  |  for (j = 0; j <= n; j++)
S |  |  s[i] = s[i] + a[i][j] * x[j];
  }
```

Figure 2.11: MatVect kernel

For instance, if we consider the `matvect` kernel in Figure 2.11, dependence analysis gives two dependence relations: instances of statement S depending on instances of statement R (e.g., R produces values used by S), $R \rightarrow S$, and similarly, $S \rightarrow S$.

Dependence relation $R \rightarrow S$ does not mean that all instances of R and S are in dependence (for all values of $\vec{x}_R$ and $\vec{x}_S$); in fact, there is only a dependence if $i_R = i_S$. We can then define a *dependence polyhedron*, being a subset of the Cartesian product of the iteration domains, containing all the values of i_R, i_S and j_S for which the dependence exists. We can write this polyhedron in matrix representation (the first line represents the equality $i_R = i_S$, the two next ones the constraint that (i_R) have to belong to the iteration domain of R and similarly, the four last lines states that (i_S, j_S) belongs to the iteration domain of S):

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

To capture all program dependences we build a set of dependence polyhedra, one for each pair of

array references accessing the same array cell (scalars being a particular case of array), thus possibly building several dependence polyhedra per pair of statements. The *polyhedral dependence graph* is a multi-graph with one node per statement, and an edge $e^{R \to S}$ is labeled with a dependence polyhedron $\mathcal{D}_{R,S}$, for all dependence polyhedra.

2.4.2 Building Legal Transformations

For a program transformation to respect the program semantics, it has to ensure that the execution order of instances will respect the precedence condition, for each pairs of instances in dependence. So for a program schedule to be legal, the following must hold.

Definition 2.8 (Precedence condition) *Given two statements R and S, and dependence polyhedra $\mathcal{D}_{R,S}$. Θ^R and Θ^S preserve the program semantics if:*

$$\forall \mathcal{D}_{R,S}, \forall \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S},$$
$$\Theta^R(\vec{x}_R) \prec \Theta^S(\vec{x}_S)$$

Checking if a transformation is legal can be done very efficiently. One simply has to check that the precedence condition is respected, for all dependences [11]. In Chapter 3 we discuss a technique to linearize the constraints imposed by the dependence polyhedra into a convex form. Using this technique, one can check for each dimension of the schedule (that is, for each row of Θ) if the precedence condition is enforced or not, thus determining the legality of the schedule.

Building legal schedules only While it is easy to check *a posteriori* if a transformation preserve the program semantics, we aim at providing techniques and tools to encompass directly into the space of candidate solutions the legality criterion. Chapter 3 presents a convex characterization of all (bounded) affine schedules preserving the program semantics. Chapter 5 and following go further and propose tractable solutions to efficiently build search spaces of affine schedules that preserve the semantics.

A posteriori schedule corrections Another approach to build legal schedules, beyond encompassing the legality criterion directly into the space, is to fix a posteriori a schedule to make it legal. Vasilache proposed an efficient framework for automatic correction of schedules by shifting and index-set splitting [114], by means of the analysis of violated dependences [113]. We also propose such mechanisms in Chapter 7 and significantly improve the correction applicability, by offering a *schedule completion heuristic* that can fix any schedule to lie in the constructed search space.

2.5 Code Generation

Code generation is the last step of polyhedral program optimization. It consists of regenerating a syntactic code from the polyhedral representation.

The code generation stage generates a *scanning code* of the iteration domains of each statement with the lexicographic order imposed by the schedule. Statement instances that share the same timestamp are typically executed under the same loop, resulting in loop fusion. Scanning code is typically an

intermediate, AST-based representation that can be translated in an imperative language such as C or FORTRAN. Logically, one can translate back this syntactic code to a polyhedral representation enabling *re-entrance* [111]. In doing so, several advantages such as code generation optimizations based on the generated output quality can be performed. One can for instance successively refine the schedule until some criteria on the generated code quality are met, e.g. code size or control complexity.

Modern code generation For many years this stage was considered as one of the major bottleneck of polyhedral optimization, due to the lack of scalability of the code generation algorithms. Eventually the lock was removed, thanks to the work of Bastoul [10, 11] who proposed an extended version of Quilleré's algorithm [96] that significantly outperformed previously implemented techniques such as Kelly's et al. in the Omega framework [65] or Griebl's in the Loopo framework [55]. Efficient algorithms and tools now exist to generate target code from a polyhedral representation with multidimensional affine schedules. Recent work by Vasilache et al. [112, 111] and by Reservoir Labs [77] improved these algorithms to scale up to thousands of statements. All along this thesis, we use the state-of-the art code generator CLOOG [10] to perform the code generation task.

The only constraints imposed by the code generator are (1) to represent iteration domains with a union of polyhedra, and (2) to represent scheduling functions as affine functions of the iteration domain dimensions. This general setting removes previous limitations such as schedule invertibility [6]. Still, more advanced code generation techniques are the subject of active research in particular in the context of the iteration space slicing approach for automatic parallelization [93, 15, 16]. While these techniques enable non-affine scheduling functions, we limit in this thesis to multidimensional affine scheduling functions.

Relation between the schedule and the syntactic code An interesting property of code generation is that *different scheduling functions will generate different syntactic codes*. This is counter-intuitive as some schedules may express the same relative ordering of instances, hence the scanning code should be the same. However, the genuine code generation algorithm of CLooG translates, without modification, the schedule coefficients into controls in the syntactic code. Considering for instance a shifting of 1 on all loops, a $+1$ coefficient will appear in each loop bound computation, despite expressing the exact same relative ordering of instance. Numerous previous works have tried to remove this property — considered more as an unwanted side effect — mostly through schedule normalization techniques [112]. Their objective was to provide the code generator with an equivalent but simpler schedule generating the simplest controls. On the other hand, in the context of iterative search of a good scheduling function, we want to keep having different syntactic codes for different schedules. The reason is twofold.

- As we will show in Chapter 6, each schedule coefficient may have an impact on performance. We observed that current implementations of some compiler optimizations are fragile, and may be triggered by almost unpredictable syntactic changes in the input code.
- As the uniqueness of a candidate is defined as generating a different syntactic code, it is equivalently defined as generating scheduling functions with different coefficients. This simplifies the encoding of uniqueness in the search space.

We discuss in Chapter 6 the tight coupling between the code produced by the code generator, and the back-end compiler in the context of our source-to-source framework. In several situations the generated codes have a weird shape, and would seem inaccurate at first glance to a compiler specialist. Yet, solid

performance improvement are achieved. This comes from the conjunction of the schedule, the code generator and the back-end compiler. The code generator applies a first optimization phase (mainly removing useless controls), which is embedded in the code generation algorithm [10, 112]. Then, the back-end compiler applies its own optimization phases, yielding a complex and multiple transformation process for the schedule. This led us to observe that potentially ineffective code (that is, with too much complex controls) produced by the code generator are able to trigger optimizations in the compiler the original code would not.

2.6 Summary

Program restructuring traditionally is broken into sequences of primitive transformations. In the case of loops, typical primitives are for instance loop *fusion*, loop *tiling*, or loop *interchange*. This approach has severe drawbacks. First, it is difficult to decide the completeness of a set of directives and to understand their interactions. Many different sequences lead to the same target code and it is typically impossible to build an exhaustive set of candidate transformed programs in this way. Next, each basic transformation comes with its own application criteria such as legality check or pattern-matching rules. Finally, long sequences of transformations contribute to code size explosion, polluting instruction cache and potentially forbidding further compiler optimizations.

Instead of reasoning on transformation sequences, we look for a representation where composition laws have a simple structure, with at least the same expressiveness as classical transformations, but without conversions to or from transformation descriptions based on sequences of primitives. To achieve this goal, we use an algebraic representation of both programs and transformations. This is the *polyhedral representation*. We use the least constrained framework for polyhedral optimization, with non-unimodular, non-invertible scheduling functions to transform non-perfectly nested programs.

Reasoning about programs in such a polyhedral representation has many advantages, for both program analysis and transformation:

1. instancewise dependence analysis is possible [40, 92];
2. there exists efficient algorithms and tools to regenerate imperative code [96, 10];
3. loop transformation sequences of arbitrary complexity can be constructed and transparently applied in one single step;
4. properties on the transformations, such as semantics preservation, can be modeled as constraints on affine functions [42].

Chapter 3

Semantics-Preserving Full Optimization Spaces

"Mathematics is concerned only with the enumeration and comparison of relations"

– Carl Friedrich Gauss

High-level loop transformations are the main instrument to map a computational kernel to efficiently exploit resources of modern processor architectures. Nevertheless, selecting compositions of loop transformations remains a challenging task. We propose in this thesis to address this fundamental challenge in its most general setting, relying on affine loop transformations in the polyhedral model.

As a first step we present the optimizer with a convex characterization of all distinct, legal affine transformations. We first introduce in Section 3.1 a convex characterization of all, distinct semantics-preserving affine multidimensional schedules with bounded coefficients. All combination of transformations — which does not require to modify the polyhedral representation — is contained in a single, convex space. This is to date the most expressive modeling of composition of transformations into one convex space.

We then present in Section 3.2 an affine characterization of all, distinct and semantics-preserving *statement interleavings*, modeling arbitrary compositions of loop fusion and distribution.

3.1 Semantics-Preserving Affine Transformations

We consider in this thesis program transformations that alter the ordering of *statement instances*. A program transformation must preserve the semantics of the program. As a starting point for optimization space pruning, we build a *convex, polyhedral characterization of all legal multidimensional affine schedules* for a static control program, with bounded schedule coefficients. Such a characterization is essential to devise optimization problems in the form of (integer) linear programs, which can be efficiently solved.

3.1.1 Program Dependences

We recall that two statements instances are in *dependence relation* if they access the same memory cell and at least one of these accesses is a write. Given two statements R and S, a *dependence polyhedron* $\mathcal{D}_{R,S}$ is a subset of the Cartesian product of $\mathcal{D}_R$ and $\mathcal{D}_S$: $\mathcal{D}_{R,S}$ contains all pairs of instances $\langle\vec{x}_R,\vec{x}_S\rangle$ such that $\vec{x}_S$ depends on $\vec{x}_R$, for a given array reference. Hence, for an optimization to preserve the program semantics, it must ensure that

$$\Theta^R(\vec{x}_R) \prec \Theta^S(\vec{x}_S),$$

where $\prec$ denotes the lexicographic ordering.

Let us consider again the `matvect` kernel in Figure 3.1, with two dependences $R \to S^1$ and $S \to S^2$.

```
  for (i = 0; i <= n; i++) {
R |  s[i] = 0;
  |  for (j = 0; j <= n; j++)
S |  |  s[i] = s[i] + a[i][j] * x[j];
  }
```

Figure 3.1: MatVect kernel

For instance, dependence relation $R \to S^1$ is written:

$$\mathcal{D}^1_{R,S} = \left[\begin{array}{ccccc} 1 & -1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 \end{array}\right] \cdot \begin{pmatrix} i_R \\ i_S \\ j_S \\ n \\ 1 \end{pmatrix} \begin{array}{c} = 0 \\ \hline \geq \vec{0} \end{array}$$

This dependence polyhedron will serve as the running example for the next section.

3.1.2 Convex Set of One-Dimensional Schedules

Considering Θ_R and Θ_S two one-dimensional scheduling functions, in order to respect the dependence $\mathcal{D}_{R,S}$ the schedules have to satisfy the precedence condition

$$\theta_R(\vec{x}_R) < \theta_S(\vec{x}_S)$$

for each point of $\mathcal{D}_{R,S}$. So one can state that

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

must be non-negative everywhere in $\mathcal{D}_{R,S}$.

The schedule constraints imposed by the precedence constraint can be expressed as finding all non-negative functions over the dependence polyhedra [41]. It is possible to express the set of affine, non-negative functions over $\mathcal{D}_{R,S}$ in an affine way using the affine form of the Farkas lemma [101].

Lemma 3.1 (Affine form of Farkas Lemma) *Let $\mathcal{D}$ be a nonempty polyhedron defined by the inequalities $A\,\vec{x} + \vec{b} \geq \vec{0}$. Then any affine function $f(\vec{x})$ is non-negative everywhere in $\mathcal{D}$ iff it is a positive affine combination:*

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

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

Since we can express the set of affine non-negative functions over $\mathcal{D}_{R,S}$, the set of legal schedules satisfying the dependence $R \to S$ is given by the relation

$$\Delta_{R,S} = \lambda_0 + \vec{\lambda}^T \left(D_{R,S} \begin{pmatrix} \vec{x}_R \\ \vec{x}_S \end{pmatrix} + \vec{d}_{R,S} \right) \geq 0$$

where $D_{R,S}$ is the constraint matrix representing the polyhedron $\mathcal{D}_{R,S}$ over $\vec{x}_R$ and $\vec{x}_S$, and $\vec{d}_{R,S}$ is the scalar part of these constraints.

Let us go back to the `matvect` example in Figure 3.1. The two prototype affine one-dimensional schedules for R and S are:

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

Using the previously defined dependence representation, we can split the system into as many inequalities as there are independent variables, and equate the coefficients in both sides of the equation. For the dependence $\mathcal{D}^1_{R,S}$ we have

$$\left\{ \begin{array}{rcrcl} i_R & : & -t_{1_R} & = & \lambda_1 + \lambda_2 - \lambda_3 \\ i_S & : & t_{1_S} & = & -\lambda_1 + \lambda_4 - \lambda_5 \\ j_S & : & t_{2_S} & = & \lambda_6 - \lambda_7 \\ n & : & t_{3_S} - t_{2_R} & = & \lambda_3 + \lambda_5 + \lambda_7 \\ 1 & : & t_{4_S} - t_{3_R} - 1 & = & \lambda_0 \end{array} \right.$$

where λ_x is the Farkas multiplier attached to the xth line of $D_{R,S}$.

This system expresses all the constraints a schedule has to respect according to the dependence $\mathcal{D}^1_{R,S}$. In order to get a tractable set of constraints on the schedule coefficients, we need to eliminate the Farkas multipliers and project their constraints on the schedule coefficients [41], with for example the Fourier-Motzkin projection algorithm [43]. If there is no solution, then no affine one-dimensional schedule is possible for this dependence, and we have to resort to multidimensional schedules to fully characterize the program execution [42].

If we build and solve the system for the dependence $\mathcal{D}_{R,S}$, we obtain a polyhedron $\mathcal{T}_{\mathcal{D}_{R,S}}$, by projecting the λ dimensions on the t ones (the corresponding schedule variables of R and S). This polyhedron represents the set of legal values for the schedule coefficients, in order to satisfy the dependence. To build the set of legal schedule coefficients for the whole program, we have to build the intersection of each polyhedron obtained for each dependence. The result is a global polyhedron $\mathcal{T}$ — with as many dimensions as there are schedule coefficients for the SCoP — which is the intersection of the constraints obtained for each dependence. Transitively dependent statements are correctly handled in the global solution: by intersecting the set of legal schedules obtained for each dependence, we end up with a set of

schedules satisfying all dependences. The intersection operation implicitly extends the dimensionality of polyhedra to the dimensionality of $\mathcal{T}$, and sets the missing dimensions as unconstrained. So we have for all dependence polyhedra:

$$\mathcal{T} = \bigcap_{\forall \mathcal{D}_{R,S}} \mathcal{T}_{\mathcal{D}_{R,S}}$$

As all systems are built and solved one dependence at a time before being intersected, the computation of the legal space can be done simultaneously with the dependence analysis.

Intuitively, to each (integral) point of $\mathcal{T}$ corresponds a different schedule for the original program, i.e., a different program version (or also a valid, distinct transformation sequence). Nevertheless, several transformation sequences are in fact expressing the exact same relative ordering of instances, and would be equal under a simple schedule normalization step as proposed by Vasilache [111]. For the purpose of iteratively selecting an efficient schedule, diversity serves the interest of iterative optimization. We rely on the fact that with our code generator implementation, distinct transformation sequences lead to different syntactic programs to increase this diversity.

3.1.3 Generalization to Multidimensional Schedules

Addressing the generalization to the case of multi-dimensional schedules, respecting the precedence constraint is a necessary and sufficient characterization. First let us distinguish between strong and weak precedence satisfaction. A dependence $\mathcal{D}_{R,S}$ is *strongly satisfied* when for all pairs of instances in dependence relation the strict precedence condition is met.

Definition 3.1 (Strong dependence satisfaction) *Given $\mathcal{D}_{R,S}$, the dependence is strongly satisfied at schedule level k if*

$$\forall \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S}, \quad \Theta^S_k(\vec{x}_S) - \Theta^R_k(\vec{x}_R) \geq 1$$

But a weaker situation may occur. A dependence $\mathcal{D}_{R,S}$ can be *weakly satisfied* when the precedence condition is not enforced on all instances, and for some permitting $\Theta^S_1(\vec{x}_S) = \Theta^R_1(\vec{x}_R)$.

Definition 3.2 (Weak dependence satisfaction) *Given $\mathcal{D}_{R,S}$, the dependence is weakly satisfied at dimension k if*

$$\begin{aligned} \forall \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S}, &\qquad \Theta^S_k(\vec{x}_S) - \Theta^R_k(\vec{x}_R) \geq 0 \\ \exists \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S}, &\qquad \Theta^S_k(\vec{x}_S) = \Theta^R_k(\vec{x}_R) \end{aligned}$$

For instance when no one-dimensional schedule exists at a given dimension and for a given dependence, it means that at this dimension the dependence can only be weakly satisfied.

We introduce variable $\delta_1^{\mathcal{D}_{R,S}}$ to model the dependence satisfaction. Considering the first time dimension (the first row of the scheduling matrices), to preserve the precedence relation we have:

$$\begin{aligned} \forall \mathcal{D}_{R,S},\ \forall \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S}, \quad \Theta^S_1(\vec{x}_S) - \Theta^R_1(\vec{x}_R) &\geq \delta_1^{\mathcal{D}_{R,S}} \\ \delta_1^{\mathcal{D}_{R,S}} &\in \{0,1\} \end{aligned} \tag{3.1}$$

The Farkas Lemma offers a loss-less linearization of the constraints from the dependence polyhedron into direct constraints on the schedule coefficients. This model extends to multidimensional schedules, observing that once a dependence has been strongly satisfied, it does not contribute to the semantics preservation constraints for the subsequent time dimensions. This comes from the lexicopositivity on the execution order of instances. Once the dependence is strongly satisfied at a given level, instances in dependence are guaranteed to be executed such that the precedence condition is strictly enforced. Furthermore, for a schedule to preserve semantics, it is sufficient for every dependence to be strongly satisfied at least once. Following these observations, one may state a sufficient condition for semantics preservation (adapted from Feautrier's formalization [42]).

Lemma 3.2 (Semantics-preserving affine schedules) *Given a set of affine schedules $\Theta^R, \Theta^S \ldots$ of dimension m, the program semantics is preserved if:*

$$\begin{aligned} & \forall \mathcal{D}_{R,S},\ \exists p \in \{1,\ldots,m\},\ \delta_p^{\mathcal{D}_{R,S}} = 1 \\ \wedge\ & \forall j < p,\ \delta_j^{\mathcal{D}_{R,S}} = 0 \\ \wedge\ & \forall j \le p, \forall \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S},\ \Theta_p^S(\vec{x}_S) - \Theta_p^R(\vec{x}_R) \ge \delta_j^{\mathcal{D}_{R,S}} \end{aligned}$$

The proof directly derives from the lexicopositivity of dependence satisfaction [42].

Regarding the schedule dimensionality m, it is sufficient to pick $m = d$ to guarantee the existence of a legal schedule (the maximum program loop depth is d). Feautrier proved it was always possible to build a schedule corresponding to the original program execution order, demonstrating the existence of a solution to Lemma 3.2 for any static control program part [42].

This formalization involves an "oracle" to select, for each dependence, the dimension p at which it should be strongly satisfied. To avoid the combinatorial selection of this dimension, we conditionally nullify constraint (3.1) on the schedules *when the dependence was strongly satisfied at a previous dimension*. To nullify the constraint, a solution is to pick a lower bound lb such that (3.2) expresses constraints which do not intersect with the polyhedron of legal affine multidimensional schedules.

$$\Theta_k^S(\vec{x}_S) - \Theta_k^R(\vec{x}_R) > lb \tag{3.2}$$

When considering arbitrary values for the coefficients of Θ_k, the only valid lower bound is $-\infty$. Consider again the the matvect example, at the first dimension. If we select $t_{1_R} = K$, $t_{1_S} = 1$ and all other coefficients to be 0, then we have:

$$lb = \min_{\vec{x}_R \in \mathcal{D}_R,\ \vec{x}_S \in \mathcal{D}_S} \left(\Theta_k^S(\vec{x}_S) - \Theta_k^R(\vec{x}_R) \right) = -K.N$$

Considering $N > 0$, since $K \in \mathbb{Z}$ we have:

$$\lim_{K \to \infty} \left(\min \left(\Theta_k^S(\vec{x}_S) - \Theta_k^R(\vec{x}_R) \right) \right) = -\infty$$

In order to compute a finite value of lb, one has to resort to bounding the values of the θ coefficients, as proposed by Vasilache [111]. Without any loss of generality, we assume (parametric) loop bounds are non-negative. Returning again to the matvect example, considering arbitrary schedules for the first dimension we have:

$$\left| \min \left(\Theta_k^S(\vec{x}_S) - \Theta_k^R(\vec{x}_R) \right) \right| = \left(|t_{1_S}| + |t_{2_S}| + |t_{3_S}| + |t_{1_R}| + |t_{2_R}| \right).N + |t_{4_S}| + |t_{3_R}|$$

One can then select $K = (|t_{1_S}| + |t_{2_S}| + |t_{3_S}| + |t_{1_R}| + |t_{2_R}| + |t_{4_S}| + |t_{3_R}|) + 1$ to ensure

$$\min\left(\Theta_k^S(\vec{x}_S) - \Theta_k^R(\vec{x}_R)\right) > -K.N - K$$

The existence of K is stated in generality in the following Lemma.

Lemma 3.3 (Schedule lower bound) *Given Θ_k^R, Θ_k^S such that each coefficient value is bounded in $[x,y]$. Then there exists $K \in \mathbb{Z}$ such that:*

$$\min\left(\Theta_k^S(\vec{x}_S) - \Theta_k^R(\vec{x}_R)\right) > -K.\vec{n} - K$$

Proof. If the iteration domains are bounded then the highest value of the iteration domain in any of its dimension is $\vec{a}.\vec{n} + c$. By definition of static control parts the iteration domain cannot be modified during execution, hence $\vec{a}$ and c are constant values that can be computed statically. As the schedule coefficients are bounded, there exists a finite value bounding the timestamp difference.

If at least one of the iteration domains is not bounded, then if a dependence exists between the two statements the only possibility to express a non-negative function over the dependence polyhedron is with a (parametric) constant function. As the schedule coefficients are bounded, there exists a finite value bounding the timestamp difference. ■

Vasilache discussed another method to nullify a constraint, based on multiplying the whole constraint by a term which equals 0 when the constraint should be nullified. This technique has two drawbacks: first and foremost, the generated solution set is not convex; second it involves extending the range of the δ_k^S variables to $[-1,1]$ instead of $[0,1]$, which makes the problem more complex.

Note that this lower bound $-K\vec{n} - K$ can also be linearized into constraints on Θ^R, Θ^S using the Farkas Lemma. To obtain the schedule constraints we reinsert this lower bound in the previous formulation, such that either the dependence has not been previously strongly satisfied and then the lower bound is $\delta_k^{\mathcal{D}_{R,S}}$, or it has been and the lower bound is $-K\vec{n} - K$. We thus derive the convex form of semantics-preserving affine schedules of dimension m for a program with bounded schedule coefficients, as a corollary of Lemma 3.2.

Lemma 3.4 (Convex form of semantics-preserving affine schedules) *Given a set of affine schedules $\Theta^R, \Theta^S \ldots$ of dimension m, the program semantics is preserved if the three following conditions hold:*

$$\textit{(i)} \quad \forall \mathcal{D}_{R,S},\ \delta_p^{\mathcal{D}_{R,S}} \in \{0,1\}$$

$$\textit{(ii)} \quad \forall \mathcal{D}_{R,S},\ \sum_{p=1}^{m} \delta_p^{\mathcal{D}_{R,S}} = 1 \tag{3.3}$$

$$\textit{(iii)} \quad \forall \mathcal{D}_{R,S},\ \forall p \in \{1,\ldots,m\},\ \forall \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S}, \tag{3.4}$$

$$\Theta_p^S(\vec{x}_S) - \Theta_p^R(\vec{x}_R) \geq -\sum_{k=1}^{p-1} \delta_k^{\mathcal{D}_{R,S}}.(K.\vec{n} + K) + \delta_p^{\mathcal{D}_{R,S}}$$

Given the conditions from Lemma 3.4, it is possible to build a convex set $\mathcal{L}$ of semantics-preserving schedules, with one variable per coefficient in the scheduling matrix (which has m rows) and m Boolean variables per dependence. For an efficient construction of $\mathcal{L}$, one should proceed dependence by dependence. Building the constraints of the form of (3.4) is done for each dependence, then the Farkas multipliers are eliminated for instance with the Fourier-Motzkin projection technique as presented in

Chapter 4. The set of constraints obtained for each dependences are then intersected, and the process is replicated for all dimensions.

A convex set of affine multidimensional schedules is the perfect tool to model all kinds of optimization problems. It opens the door to well understood operation research algorithms. It also facilitates search-space pruning strategies, to focus the optimization problem towards the most profitable transformations. Nevertheless, given its high dimension, our convex formalization may induce a tractability challenge on the larger benchmarks. Pruning is also a powerful method to reduce the dimension of the search space. In Chapter 8, we illustrate the power of joining convex modeling of multidimensional schedules and pruning strategies when applied to loop fusion. Section 3.2 introduces the background concepts supporting this pruning strategy.

3.1.4 Related Work

Feautrier was the first to propose a convex encoding of all affine non-negative one-dimensional schedules [41]. He then extended to multi-dimensional schedules, focusing on providing an algorithm to compute an optimal schedule for fine-grain parallelism [42]. In its original form his formulation of the space of legal schedule is a combinatorial combination of subspaces, one for each weakly / strongly dependence satisfaction scenario. We build on his approach, and by adding the constraint of bounding the schedule coefficient we model directly into a single space the semantics-preserving multidimensional schedules.

Vasilache proposed the first convex characterization of all bounded affine multidimensional schedules [111]. His approach, closely related to the one presented in this section, leads to the construction of an affine set with only integer vertices and the same number of variables as in Feautrier's formulation [42]. Yet the drawbacks of such an approach are twofold. First, Vasilache resorts to decision variables δ_k^S which are integers in $[-1,1]$. In contrast, our approach uses only *Boolean* decision variables, significantly simplifying the complexity of linear programming. Second, Vasilache's form involves coefficients in the power of 2 to encode lexicopositivity of dependence satisfaction, with a maximum value of 2^{m-1} where m is the schedule depth. Practical experiments indicate that large coefficient values may decrease the efficiency of parametric integer programming solvers. Finally, let us note also that Feautrier proposed, in an unpublished communication from 2007, the same simplifications of Vasilache's characterization as the one we have described earlier.

3.2 Semantics-Preserving Statement Interleavings

After providing a formulation to model in a single convex space the set of all possible affine transformations, we now focus on two highly performance-impacting transformations: loop fusion and loop distribution. For that purpose we extract an affine subspace of all distinct, legal *statement interleavings*, modeling compositions of loop fusion and distribution. This subspace focuses the search on the most difficult part of the optimization problem: it models transformations that have a significant impact on the overall performance, isolated from enabling transformations for which effective heuristics exist.

We are interested in building an affine representation for the set of all legal and distinct multi-level loop fusion and distribution possibilities for a program. Moreover, this set should be built in isolation of the possibly required complementary transformations to make a given fusion / distribution legal. Obviously, all these possibilities are included in the set $\mathcal{L}$ of all semantics-preserving schedules. However, using directly the formulation for semantics-preservation has two main disadvantages. First, in the per-

spective of performing iterative search in this set, uniqueness of the solution is a critical concern. By using a representation of fusion / distribution based on the handling of some specific schedule coefficients, it is not possible to present a subset of $\mathcal{L}$ in which each solution represents a distinct combination of fusion / distribution, as discussed in Section 3.2.2. Second, scalability is greatly challenged. To extract the subspace of coefficients which models explicitly loop fusion and distribution, it is expected a projection step is required. This operation can quickly become intractable on systems as large as $\mathcal{L}$, especially for programs with numerous dependences. On the contrary, we present a decoupled approach in Chapter 8 where a schedule according to a given fusion / distribution scheme is recomputed, in a tractable fashion.

Compared to the state-of-the-art in loop fusion, we now consider arbitrarily complex sequences of enabling transformations, in a multidimensional setting. This generalization of loop fusion is called *fusability* and results in a dramatic broadening of the expressiveness of the optimizer. As we model candidate statement interleavings for fusability as total preorders, we must first provide the first convex characterization of the space of all distinct total preorders, before presenting a technique to prune this space from the redundant and non semantics-preserving transformations. This encoding and the resulting search space is used as a basis of Chapter 8, where we propose a complete iterative and model-driven technique for the efficient selection of multidimensional statement interleavings.

3.2.1 Encoding Statement Interleaving

Fusion and fusability of statements In the polyhedral model, loop fusion is characterized by the fine-grain interleaving of statement instances [20]. Two statements are fully distributed if the range of the timestamps associated to their instances never overlap. Syntactically, this results in distinct loops to traverse the domains. One may define fusion as the negation of the distribution criterion. For such case we say that two statements R,S are fused under at least one common loop if there exists at least one pair of iterations for which R is scheduled before S, and another pair of iterations for which S is scheduled before R. This is stated in Definition 3.3.

Definition 3.3 (Fusion of two statements) *Given two statements R,S. They are fused at level p if, $\forall k \in \{1 \dots p\}$, there exists at least two pairs of executed instances $\vec{x_R}, \vec{x_S}$ and $\vec{x_R}', \vec{x_S}'$ such that:*

$$\Theta_k^R(\vec{x_R}) \leq \Theta_k^S(\vec{x_S}) \wedge \Theta_k^S(\vec{x_S}') \leq \Theta_k^R(\vec{x_R}')$$

But for fusion to have a performance impact, a stronger criterion is preferred to guarantee that at most x instances are *not* finely interleaved. In general, computing the exact set of interleaved instances requires complex techniques. A typical example is schedules generating a non-unit stride in the supporting lattice of the transformed iteration domain. For such cases, computing the exact number of non-fused instance could be achieved using the Ehrhart quasi-polynomial of the intersection of the image of iteration domains by the schedules [28]. However, this refined precision is not required to determine if a schedule represents a potentially interesting fusion. We allow for a lack of precision to present an easily computable test for fusability based on an *estimate* of the number of instances that are not finely interleaved.

We first propose to define an estimator of the number of unfused instances, considering two statements and their associated schedules. Definition 3.4 introduces an Integer Program which admits a solution only if the difference between the lowest timestamp for the instances of R and the lowest timestamp

for the instances of S is lower than a given constant c, for all schedules dimensions $1..p$. The constant c is the timestamp difference between the first scheduled instance of R and the first scheduled instance of S. By tuning the allowed size for this interval, one can range from full, aligned fusion ($c = 0$) to full distribution (with c greater than the schedule latency of R or S). Note c is an integer constant *estimating* the number of instances which are not fused, this definition does not strictly state that at most c instances are not fused.

Definition 3.4 (Estimator for the fusion of statements) *Given two statements R and S and their associated schedules Θ_R and Θ_S. Given the parametric integer program $FUSE(R,S,p)$ defined by:*

$$FUSE(R,S,p): \quad \forall k \in \{1,\dots,p\}, \qquad -c < \min\left(\Theta^R_k(\vec{x}_R)\right) - \min\left(\Theta^S_k(\vec{x}_S)\right) < c$$

If $FUSE(R,S,p)$ has a solution $\forall \vec{x}_R \in \mathcal{D}_R$, $\forall \vec{x}_S \in \mathcal{D}_S$, and if Θ^R_k and Θ^S_k are not constant schedules, then c is an estimator of the the number of fused instances of R and S.

To compute a value of c for which a solution to $FUSE(R,S,p)$ implies that R and S are fused at level p, one can take a small integer corresponding to the shifting one wishes to allow (e.g., $c = 10$). As an upper bound for c, one can enforce c to be lower than the smallest loop trip count to avoid the possibility of finding a solution where the two loops are indeed distributed. Note that for the case of parametric loop bounds, there is no restriction to define c as a function of some parameters which value is not known at compile time, e.g. $c = min(M/3,N/3)$. We can now propose a definition for the *fusability* of statements, based on the program $FUSE(R,S,p)$.

Definition 3.5 (Fusability) *Given two statements R,S. They are fusable at level p if there exists semantics-preserving schedules Θ^R,Θ^S such that $\forall k \in \{1\dots p\}$,*

$$\begin{array}{ll} (i) & FUSE(R,S,p) \text{ has a solution} \\ (ii) & \text{If and only if } \Theta^R_k \text{ and } \Theta^S_k \text{ are constant schedules, then } \Theta^R_k = \Theta^S_k \\ (iii) & c < \min\left(\max\left(\Theta^R_k(\vec{x}_R)\right), \max\left(\Theta^S_k(\vec{x}_S)\right)\right) \end{array}$$

The last step is to propose an encoding of the problem such that we can determine from the dependence graph if a schedule leading to fuse the statements exists, without having to instantiate the schedules. Let us first study the simpler problem when schedule coefficients are non-negative (that is, $\theta_{i,j} \in \mathbb{N}$). For the sake of simplicity, we assume that loop bounds have been normalized such that $\vec{0}$ is the first iteration of the domain. When considering non-negative coefficients, the lowest timestamp assigned by a schedule Θ^R_k is simply $\Theta^R_k(\vec{0})$. One can recompute the corresponding timestamp by looking at the values of the coefficients attached to the parameters and the constant. Hence, the timestamp interval c between the two first scheduled instances by Θ^R_k and Θ^S_k is simply the difference of the parametric constant parts of the schedules. In addition, to avoid the case where Θ^R_k and/or Θ^S_k are constant schedules, we force the linear part of the schedule to be non-null. This is formalized in Definition 3.6.

Definition 3.6 (Fusability restricted to non-negative schedule coefficients) *Given two statements R,S such that R is surrounded by d^R loops, and S by d^S loops. They are fusable at level p if, $\forall k \in \{1\dots p\}$, there exist two semantics-preserving schedules Θ^R_k and Θ^S_k such that:*

$$\begin{array}{ll} (i) & \forall k \in \{1,\dots,p\}, \qquad -c < \Theta^R_k(\vec{0}) - \Theta^S_k(\vec{0}) < c \\ (ii) & \sum_{i=1}^{d^R} \theta^R_{k,i} > 0, \ \sum_{i=1}^{d^S} \theta^S_{k,i} > 0 \end{array}$$

This very tractable definition is heavily used in Chapter 8, where we propose a practical implementation of the selection of multidimensional interleavings.

Addressing the general case where $\theta_{i,j} \in \mathbb{Z}$ is far more complex, as there is no easy way to retrieve the instance scheduled first by Θ_k^R. To achieve a reasonable expression of a test for fusability in the general case, we will actually limit ourselves to finding an encoding for Definition 3.3. This definition holds by exhibiting two instances for which R is executed before S, and R is executing after S. For that purpose, we select $\vec{0}$, and $\vec{m}_R$ the lexicographically largest instance of $\mathcal{D}_R$. Then, to check for the fusability of statements, one can check the possible orderings $\Theta_k^R(\vec{0}) < \Theta_k^S(\vec{0})$, $\Theta_k^R(\vec{0}) < \Theta_k^S(\vec{m}_S)$, etc. As we check for a pair of distinct instances, it is not required to ensure the schedules are not constant: all checks will fail if it is the case.

Definition 3.7 (Generalized fusability check) *Given two statements R,S. They are fusable at level p if, $\forall k \in \{1 \ldots p\}$, there exist two semantics-preserving schedules Θ_k^R and Θ_k^S such that either one of the four following problems has a solution:*

$$
\begin{array}{ll}
(i) & \Theta_k^R(\vec{0}) < \Theta_k^S(\vec{m}_S) \wedge \Theta_k^S(\vec{0}) < \Theta_k^R(\vec{m}_R) \\
(ii) & \Theta_k^R(\vec{m}_R) < \Theta_k^S(\vec{m}_S) \wedge \Theta_k^S(\vec{0}) < \Theta_k^R(\vec{0}) \\
(iii) & \Theta_k^R(\vec{0}) < \Theta_k^S(\vec{m}_S) \wedge \Theta_k^S(\vec{m}_S) < \Theta_k^R(\vec{m}_R) \\
(iv) & \Theta_k^R(\vec{m}_R) < \Theta_k^S(\vec{0}) \wedge \Theta_k^S(\vec{m}_R) < \Theta_k^R(\vec{0})
\end{array}
$$

For completeness, one should in addition test for different values for $\vec{m}_R, \vec{m}_S$, in particular all vertices of the iteration domains. As soon as there exists a value for $\vec{m}_R, \vec{m}_S$ for which one of the above test do not fail, then the statements are fusable. There exist pathological cases where the above tests all fail while it is indeed possible to express a schedule leading to fusion. However, such schedules will correspond to fusing only a small portion of the iteration domains. Hence, had we used a stricter fusability test (that is, using a small integer value for c) the statements would have been detected as non-fusable with those schedules.

Statement interleaving For our optimization problems, we are interested in building a space representing all ways to interleave the program statements. This relates to loop fusion and loop distribution: possibles ways to interleave statements include them sharing the same outer loops, and them being distributed and placed at different positions in the program. Consider as an example a series of three matrix-products, ThreeMatMat, shown in Figure 3.2.

```
  for (i1 = 0; i1 < N; ++i1)
    for (j1 = 0; j1 < N; ++j1)
      for (k1 = 0; k1 < N; ++k1)
R       C[i1][j1] += A[i1][k1] * B[k1][j1];
  for (i2 = 0; i2 < N; ++i2)
    for (j2 = 0; j2 < N; ++j2)
      for (k2 = 0; k2 < N; ++k2)
S       F[i2][j2] += D[i2][k2] * E[k2][j2];
  for (i3 = 0; i3 < N; ++i3)
    for (j3 = 0; j3 < N; ++j3)
      for (k3 = 0; k3 < N; ++k3)
T       G[i3][j3] += C[i3][k3] * F[k3][j3];
```

Figure 3.2: ThreeMatMat: $C = AB$, $F = DE$, $G = CF$

To reason about statement interleaving, one may associate a vector β^S of dimension d to each statement S such that their lexicographic ordering encodes exactly the ordering and fusion information of each loop level. If some statement S is surrounded by less than d loops, β^S is post-padded with zeroes. As an introductory example of statement interleavings Figure 3.3 shows four possible transformations for the illustrating example, as defined by different configurations of the β vectors. We also show, for each version, $\mathbf{L}$ the sub-part of the scheduling matrix associated to the iterator dimensions enabling the transformation. Note that we do not represent the schedule coefficients γ associated to the global parameters $\vec{n}$ and the coefficient c associated to the constant, they are set to 0 for these examples.

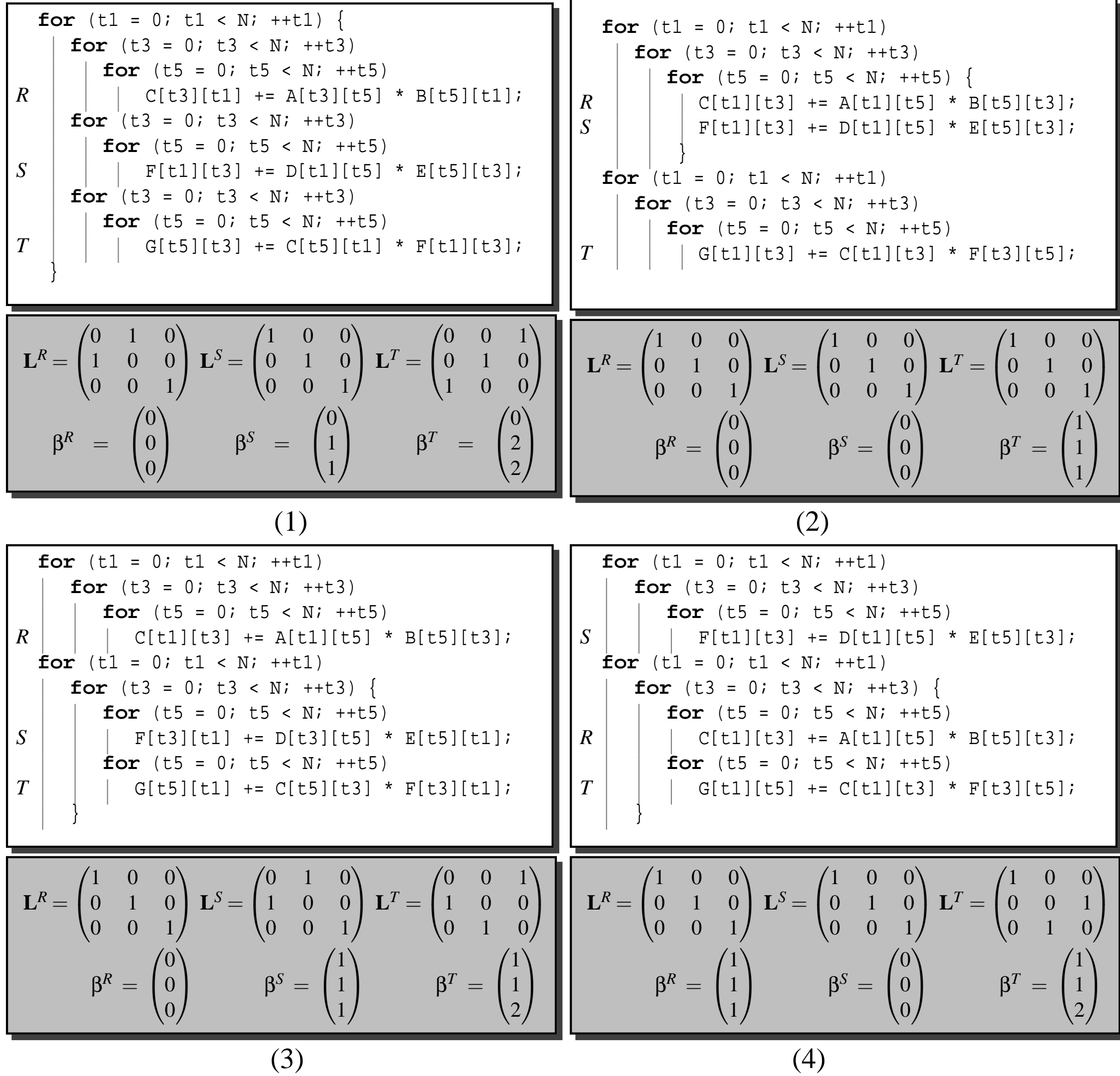


Figure 3.3: Four possible legal transformations for $C = AB$, $F = DE$, $G = CF$

Multidimensional affine schedules can be restricted to a form where statement interleaving vectors are explicit: for each statement S, one may constrain the rows of Θ^S to alternate between constant forms of the β^S vector and affine forms of the iteration and global parameter vectors. This $2d+1$-dimensional encoding does not incur any loss of expressiveness [42, 64, 30, 51, 12]. For instance to build the full scheduling matrix Θ^R from $\mathbf{L}^R$ and β^R, one may proceed by interleaving $\mathbf{L}^R$ and β^R such that ($\vec{\gamma}^R = \vec{0}$ and

$\vec{c}^R = \vec{0}$ here):

$$\Theta^R = \begin{pmatrix} 0 & 0 & 0 & 0 & \beta_1^R \\ L_{1,1}^R & L_{1,2}^R & L_{1,3}^R & \gamma_1^R & c_1^R \\ 0 & 0 & 0 & 0 & \beta_2^R \\ L_{2,1}^R & L_{2,2}^R & L_{2,3}^R & \gamma_2^R & c_2^R \\ 0 & 0 & 0 & 0 & \beta_3^R \\ L_{3,1}^R & L_{3,2}^R & L_{3,3}^R & \gamma_3^R & c_3^R \end{pmatrix} \cdot \begin{pmatrix} i_R \\ j_R \\ k_R \\ N \\ 1 \end{pmatrix}$$

Unlike standard $2d+1$ encodings, we explicitly **force** the β vectors to exhibit important structural properties of the transformed loop nest:

1. if $\beta_k^R = \beta_k^S,\ \forall k \in \{1,\ldots,p\}$ then the statements share (at least) p common loops;
2. if $\beta_p^R \neq \beta_p^S \wedge \forall k \in \{1,\ldots,p-1\}, \beta_k^R \neq \beta_k^S$ then the statements do not share any common loop at depth p (or more).

However, intuitively, several choices of β vectors represent the same multidimensional statement interleaving: for example, the transformed code is invariant to translation of all coefficients at a given dimension, or by multiplication of all coefficients by a non-negative constant. Consider the following example:

$$\beta_1^R = (\ 0\ ),\ \beta_1^S = (\ 2\ ),\ \beta_1^T = (\ 2\ )$$

This ordering defines that S and T are fused together, and that R is not and is executed before S and T. An equivalent description is:

$$\beta_1'^R = (\ 0\ ),\ \beta_1'^S = (\ 1\ ),\ \beta_1'^T = (\ 1\ )$$

To abstract away these equivalences, let us now formally define the concept of multidimensional statement interleaving.

Definition 3.8 (Multidimensional statement interleaving) *Consider a set of statements $\mathcal{S}$ enclosed within at most d loops and their associated vectors $\mathcal{B} = \{\beta^S\}_{S\in\mathcal{S}}$. For a given $p \in \{1,\ldots,d\}$, the one-dimensional statement interleaving of $\mathcal{S}$ at dimension p defined by $\mathcal{B}$ is the partition of $\mathcal{S}$ according to the coefficients β_p^S. The multidimensional statement interleaving of $\mathcal{S}$ at dimension p defined by $\mathcal{B}$ is the list of d partitions at dimension p.*

The structural properties of statement interleaving indicate that equivalence classes at dimension p correspond to distinct loops at depth p in the transformed loop nest.

Definition 3.9 (Total preorder) *A total preorder on a set $\mathcal{S}$ is a relation $\lhd$ which is reflexive, transitive, and such that for any pair of elements $(S_1, S_2) \in \mathcal{S}$, either $S_1 \lhd S_2$ or $S_2 \lhd S_1$ or both.*

An important result is that any preorder of a set $\mathcal{S}$ is isomorphic to a partial order of some equivalence classes of $\mathcal{S}$. Applying this result to the structural properties of statement interleavings yields the following lemma.

Lemma 3.5 (Structure of statement interleavings) *Each distinct one-dimensional statement interleaving corresponds to a unique canonical total preorder of the statements and reciprocally.*

We now propose an affine, complete characterization of multi-dimensional statement interleavings.

3.2.2 Affine Encoding of Total Preorders

One-dimensional case For a given set of n elements, we define $\mathcal{O}$ as the set of all and distinct total preorders of its n elements. The key problem is to model $\mathcal{O}$ as a polyhedron. This problem is not a straight adaptation of standard order theory: we look for the *set* of *all distinct* total preorders of n elements, in contrast to classical work defining counting functions of this set [104].

We recall that uniqueness implies only distinct total preorders are represented in the set. To the best of our knowledge, uniqueness cannot be modeled in a convex fashion on the set of β vectors. The problem lies in the ability to express affine constraints to prune all and only redundant β vectors: there is no affine description possible to remove all β vectors expressing the same preorder. To overcome this problem we propose to model the ordering of two elements $i,\ j$ with three *binary* decision variables, defined as follows. $p_{i,j} = 1$ iff i precedes j, $e_{i,j} = 1$ iff i equals j and $s_{i,j} = 1$ iff i succeeds j. To model the entire set, we introduce three binary variables for each ordered pair of elements, i.e., all pairs (i, j) such that $1 \le i < j \le n$. This models $\mathcal{O}$ with $3 \times n(n-1)/2$ variables.

$$\mathcal{O} = \left\{ \begin{array}{c} 0 \le p_{i,j} \le 1 \\ 0 \le e_{i,j} \le 1 \\ 0 \le s_{i,j} \le 1 \end{array} \right\}$$

For instance, the interleaving $\beta_1^R = 0,\ \beta_1^S = 0,\ \beta_1^T = 1$ of Figure 3.3(2) is represented by:

$$\begin{array}{c} e_{R,S} = 1,\ e_{R,T} = 0,\ e_{S,T} = 0 \\ p_{R,S} = 0,\ p_{R,T} = 1,\ p_{S,T} = 1 \\ s_{R,S} = 0,\ s_{R,T} = 0,\ s_{S,T} = 0 \end{array}$$

From there, one may easily recompute the corresponding total preorder $\{\beta_1^R = 0, \beta_1^S = 0, \beta_1^T = 1\}$, for instance by computing the lexicographic minimum of a system with 3 non-negative variables ($\{\beta_1^R, \beta_1^S, \beta_1^T\}$) embedding the ordering constraints defined by all $p_{i,j}$, $e_{i,j}$ and $s_{i,j}$:

$$\begin{array}{c} e_{R,S} = 1 \Rightarrow \beta_1^R = \beta_1^S \\ p_{R,T} = 1 \Rightarrow \beta_1^R < \beta_1^T \\ p_{S,T} = 1 \Rightarrow \beta_1^S < \beta_1^T \end{array}$$

The first issue is the consistency of the model, e.g. setting $e_{1,2} = 1$ and $p_{1,2} = 1$ would make impossible to recompute a valid total preorder. The second issue is the totality of the relation. These two conditions can be merged into the the following equality, capturing both mutual exclusion and totality:

$$p_{i,j} + e_{i,j} + s_{i,j} = 1 \tag{3.5}$$

To simplify the system, we immediately get rid of the $s_{i,j}$ variables, using (3.5). We also relax (3.5) to get:

$$p_{i,j} + e_{i,j} \le 1$$

Mutually exclusive decision variables capture the consistency of the model for a single pair of elements. However, one needs to insert additional preordering constraints to capture transitivity.

Basic transitivity of e coefficients To enforce transitivity, the following rule must hold true for all triples of statements (i, j, k):

$$e_{i,j} = 1 \wedge e_{i,k} = 1 \Rightarrow e_{j,k} = 1 \tag{3.6}$$

We will omit the $= 1$ in the rest of the section. Then, the following equation is equivalent to Eq (3.6):

$$e_{i,j} \wedge e_{i,k} \Rightarrow e_{j,k}$$

Similarly, we have:

$$e_{i,j} \wedge e_{j,k} \Rightarrow e_{i,k}$$

These two rules set the basic transitivity of e variables. Since we are dealing with binary variables, the implications can be easily modeled as affine constraints:

$$\left\{ \begin{array}{ll} \forall k \in]j,n], & e_{i,j} + e_{i,k} \leq 1 + e_{j,k} \\ & e_{i,j} + e_{j,k} \leq 1 + e_{i,k} \end{array} \right\}$$

Generalizing this reasoning, we collect all constraints to enforce the transitivity of the total preorder relation.

Basic transitivity of p coefficients We apply a similar reasoning for the p coefficients. We have:

$$p_{i,k} \wedge p_{k,j} \Rightarrow p_{i,j}$$

This translates into:

$$\left\{ \begin{array}{ll} \forall k \in]i,j[, & p_{i,k} + p_{k,j} \leq 1 + p_{i,j} \end{array} \right\} \tag{3.7}$$

Complex transitivity on p and t coefficients We also have transitivity conditions imposed by a connection between the value for some e coefficients and some p ones. For instance, $R < S$ and $S = T$ implies $R < T$. The general equations for those cases are:

$$e_{i,j} \wedge p_{i,k} \Rightarrow p_{j,k}$$

$$e_{i,j} \wedge p_{j,k} \Rightarrow p_{i,k}$$

$$e_{k,j} \wedge p_{i,k} \Rightarrow p_{i,j}$$

These translate to the following affine constraints:

$$\left\{ \begin{array}{ll} \forall k \in]j,n] & e_{i,j} + p_{i,k} \leq 1 + p_{j,k} \\ & e_{i,j} + p_{j,k} \leq 1 + p_{i,k} \\ \forall k \in]i,j[& e_{k,j} + p_{i,k} \leq 1 + p_{i,j} \end{array} \right\} \tag{3.8}$$

Complex transitivity on s and p coefficients Lastly, we have to take into account the transitivity on the fictitious s variables (those modeling $R > S$). The transitivity condition is:

$$s_{i,k} \wedge p_{j,k} \Rightarrow s_{i,j}$$

Since the reduction equation gives:

$$s_{i,j} = 1 - p_{i,j} - e_{i,j}$$

with $p_{i,j}$ and $e_{i,j}$ being mutually exclusive, the rule translates to the following affine constraints:

$$\left\{ \begin{array}{ll} \forall k \in]j,n] & e_{i,j} + p_{i,j} + p_{j,k} \leq 1 + p_{i,k} + e_{i,k} \end{array} \right\} \tag{3.9}$$

General formulation of $\mathcal{O}$ All the previous constraints are gathered in the following expression of $\mathcal{O}$, the convex set of all, distinct total preorders of n elements. For $1 \leq i < n,\ i < j \leq n,\ \mathcal{O}$ is:

$$\mathcal{O} = \begin{cases}
\begin{array}{lr}
 & 0 \leq p_{i,j} \leq 1 \\
 & 0 \leq e_{i,j} \leq 1
\end{array} \Big\} & \text{Variables are binary} \\
\quad p_{i,j} + e_{i,j} \leq 1 \Big\} & \text{Relaxed mutual exclusion} \\
\begin{array}{lr}
\forall k \in]j,n] & e_{i,j} + e_{i,k} \leq 1 + e_{j,k} \\
 & e_{i,j} + e_{j,k} \leq 1 + e_{i,k}
\end{array} \Big\} & \text{Basic transitivity on } e \\
\forall k \in]i,j[\quad p_{i,k} + p_{k,j} \leq 1 + p_{i,j} \Big\} & \text{Basic transitivity on } p \\
\begin{array}{lr}
\forall k \in]j,n] & e_{i,j} + p_{i,k} \leq 1 + p_{j,k} \\
 & e_{i,j} + p_{j,k} \leq 1 + p_{i,k} \\
\forall k \in]i,j[& e_{k,j} + p_{i,k} \leq 1 + p_{i,j}
\end{array} \Bigg\} & \text{Complex transitivity on } p \text{ and } e \\
\forall k \in]j,n] \quad e_{i,j} + p_{i,j} + p_{j,k} \leq 1 + p_{i,k} + e_{i,k} \Big\} & \text{Complex transitivity on } s \text{ and } p
\end{cases}$$

Lemma 3.6 (Completeness and correctness of $\mathcal{O}$) *The set $\mathcal{O}$ contains one and only one point per distinct total preorder of n elements.*

Proof. The full proof (shown in Appendix A) proceeds by showing that the transitivity of the total preorder relation is preserved for all points in the set, i.e., all possible cases of transitivity have been enforced. Totality of the preorder comes from the mutual exclusion condition, and reflexivity is trivially satisfied. ■

Multidimensional case To encode all possible and distinct values for the multi-level statement interleaving for a program, we need to replicate the set $\mathcal{O}$ for each row of the β vectors. Each row model a given total preorder, but further constraints are needed to achieve consistency and uniqueness of the characterization across dimensions. Intuitively, if a statement is distributed at dimension k, then for all remaining dimensions it will also be distributed. This is modeled with the following equations:

$$\forall l > k,\ \left(p^k_{i,j} = 1 \Rightarrow p^l_{i,j} = 1\right) \ \wedge\ \left(s^k_{i,j} = 1 \Rightarrow s^l_{i,j} = 1\right)$$

The final expression of the set $\mathcal{I}$ of all, distinct d-dimensional statement interleavings is:

$$\mathcal{I} = \begin{cases}
\forall k \in \{1,\ldots,d\}, \quad \text{constraints on } \mathcal{O}^k \Big\} & \text{Total preorders at level } k \\
\begin{array}{r}
p^k_{i,j} \leq p^{k+1}_{i,j} \\
e^{k+1}_{i,j} + p^{k+1}_{i,j} \leq p^k_{i,j} + e^k_{i,j}
\end{array} \Big\} & \text{Statement interleaving uniqueness}
\end{cases}$$

It is worth noting that each $\mathcal{O}^k$ contains numerous variables and constraints; but when it is possible to determine — with the dependence graph for instance — that a given ordering of two elements i and j is impossible, some variables and constraints are eliminated. Our experiments indicate that these simplifications are quite effective, improving the scalability of the approach significantly.

3.2.3 Pruning for Semantics Preservation

The set $\mathcal{I}$ contains all and only distinct multi-level statement interleavings. Another level of pruning is needed to remove all interleavings that does not preserve the semantics. The algorithm proceeds level-by-level, from the outermost to the innermost. Such a decoupling is possible because we have encoded multi-dimensional interleaving constraints in $\mathcal{I}$, and that fusion at level k implies fusion at all preceding levels. In addition, leveraging Lemma 3.4 and Definition 3.7 we can determine the *fusability* of a set of statements at a given level by exposing sufficient conditions for fusion on the schedules.

The general principle is to detect all the smallest possible sets of p unfusable statements at level k, and for each of them, to update $\mathcal{I}^k$ by adding an affine constraint of the form:

$$e^k_{R,S} + e^k_{S,T} + \ldots + e^k_{V,W} < p - 1 \tag{3.10}$$

thus preventing them (and any super-set of them) to be fused all together. We note $\mathcal{F}$ the final set with all pruning constraints for legality, $\mathcal{F} \subseteq \mathcal{I}$. A naive approach could be to enumerate all unordered subsets of the n statements of the program at level k, and check for fusability, while avoiding to enumerate a super-set of an unfusable set.

Instead, we leverage the polyhedral dependence graph to reduce the test to a much smaller set of structures. The first step of our algorithm is to build a graph G to facilitate the enumeration of sets of statements to test for, with one node per statement. Sets of statements to test for fusability will be represented as nodes connected by a path in that graph. Intuitively, if G is a complete graph then we can enumerate all unordered subsets of the n statements: enumerating all paths of length 1 gives all pairs of statements by retrieving the nodes connected by a given path, all paths of length 2 gives all triplets, etc. We aim at building a graph with less edges, so that we lower the number of sets of statements to test for fusability.

We first check the fusability of all possible pairs of statements, and add an edge between two nodes only if (1) there is a dependence between them, and (2) either they must be fused together or they can be fused and distributed at that level. When two statements must be fused, they are merged to ensure all schedule constraints are considered when checking for fusability.

The second step is to enumerate all paths of length ≥ 2 in the graph. Given a path p, the nodes in p represent a set of statements that has to be tested for fusability. Each time they are detected to be not fusable, all paths with p as a sub-path are discarded from enumeration, and $\mathcal{F}^k$ is updated with an equation in the form of (3.10). The complete algorithm is shown in Figure 3.4.

Procedure `buildLegalSchedules` computes the space of legal schedules $\mathcal{L}$ according to Lemma 3.4, for the set of statements given in argument.

Procedure `mustDistribute` tests for the emptiness of $\mathcal{L}$ when augmented with fusion conditions from Definition 3.7 up to level d. If there is no solution in the augmented set of constraints, then the statements cannot be fused at that level and hence must be distributed.

Procedure `mustFuse` checks if it is legal to distribute the statements R and S. The check is performed by inserting a *splitter* at level d. This splitter is a constant one-dimensional schedule at level d, to force the full distribution of statement instances at that level. If there is no solution in this set of constraints, then the statements cannot be distributed at that level and hence must be fused.

Procedure `canDistributeAndSwap` tests if it is legal to distribute R and S at level d and to execute S before R. The latter is required to compute the legal values of the $s_{R,S}$ variables at that level. The check

```
PruneIllegalInterleavings: Compute 𝓕
Input:
  pdg: polyhedral dependence graph
  n: number of statements
  maxDepth: maximum loop depth
Output:
  𝓕 : the space of semantics-preserving distinct interleavings

1   𝓕 ← 𝓘
2   unfusable ← ∅
3   for d ← 1 to maxDepth do
4      G ← newGraph(n)
5      forall pairs of dependent statements R,S do
6         𝓛_{R,S} ← buildLegalSchedules({R,S}, pdg)
7         if mustFuse(𝓛_{R,S}, d) then
8            𝓕^d ← 𝓕^d ∩ {e^d_{R,S} = 1}
9         elseif mustDistribute(𝓛_{R,S}, d) then
10           𝓕^d ← 𝓕^d ∩ {e^d_{R,S} = 0}
11        else
12           if ¬ canDistributeAndSwap(𝓛_{R,S}, d) then
13              𝓕^d ← 𝓕^d ∩ {e^d_{R,S} + p^d_{R,S} = 1}
14           end if
15           addEdge(G, R, S)
16        end if
17     end for
18     forall pairs of statements R,S such that e^d_{R,S} = 1 do
19        mergeNodes(G, R, S)
20     end for
21     for l ← 2 to n−1 do
22        forall paths p in G of length l such that
                there is no prefix of p in unfusable do
23           𝓛_{nodes(p)} ← buildLegalSchedules({nodes(p)}, pdg)
24           if mustDistribute(𝓛_{nodes(p)}, d) then
25              𝓕^d ← 𝓕^d ∩ {∑_p e^d_{pairs in p} < l−1}
26              unfusable ← unfusable ∪ p
27           end if
28        end for
29     end for
30  end for
```

Figure 3.4: Pruning algorithm

is performed in a similar fashion as with `mustFuse`, except the splitter is made to make R execute after S.

Applications The first motivation of building a separate search space of multidimensional statement interleavings is to decouple the selection of the interleaving from the selection of the transformation that enables this interleaving. One can then focus on building search heuristics for the fusion / distribution of statements only, and through the framework presented in this chapter compute a schedule that respects this interleaving. Additional schedule properties such as parallelism and tilability can then be exploited without disturbing the outer level fusion scheme. We present in Chapter 8 a complete technique to optimize programs based on iterative interleaving selection, leading to parallelized and tiled transformed programs. This technique is able to automatically adapt to the target framework, and successfully discovers the optimal fusion structure, whatever the specifics of the program, compiler and architecture.

Another motivation of building $\mathcal{I}$ is to enable the design of objective functions on fusion with the widest degree of applicability. For instance one can maximize fusion at outer level, by maximizing $\sum_{i,j} e^1_{i,j}$ or similarly distribution by minimizing the same sum. One can also assign a weight to the

coefficients $e_{i,j}$ to favor fusion for statements carrying more reuse, for instance, etc. This formulation allows devising further pruning algorithms, offering to the optimization the widest choice of legal only interleavings.

3.2.4 Related Work

To the best of our knowledge, no previous work addressed the problem of modeling the set of possible loop fusion in a single convex set, especially with the range of enabling transformations for fusion presented above. Traditional approaches to loop fusion [66, 84, 85, 103] are restricted in their ability to reason about compositions of loop transformations. This is mainly due to the lack of a powerful representation for dependences and transformations. Hence, non-polyhedral approaches typically study fusion in isolation from other transformations. Megiddo and Sarkar [85] proposed a way to perform fusion for an existing parallel program by grouping components in a way that parallelism is not disturbed. Decoupling parallelization and fusion clearly misses several interesting solutions that would have been captured if the legal fusion choices were itself cast into their framework. Darte et al. [38, 36] studied fusion for data-parallelization, but only in combination with shifting. In contrast to all of these works, our search space can enable fusion in the presence of all polyhedral transformations.

Bondhugula et al. proposed the first integrated fusion and tiling heuristic based on the polyhedral model [20, 21], and subsuming a large space of additional loop transformations (interchange, skewing, shifting). It inherits the flexibility of the tiling hyperplane method [60, 54] to build complex sequences of enabling and communication-minimizing transformations. The tiling hyperplane method proposes an efficient model-driven technique to unify locality and parallelism together in a single cost function. Tiled parallel code is generated, resulting from a complex composition of transformations. Nevertheless, in several situations more efficient code could be generated. An important fact is that the cost function is geared towards maximal fusion as it will tend to maximally fuse statements under a common tiling hyperplane, if it exists, to increase locality and reduce communication [20]. Although it is possible to treat strongly-connected components of the dependence graph separately and not fuse across them, the method does not permit the modeling of the legal fusion choices in a single space.

Chapter 4

Computing on Large Polyhedral Sets

"Pessimists, we're told, look at a glass containing 50% air and 50% water and see it as half empty. Optimists, in contrast, see it as half full. Engineers, of course, understand the glass is twice as big as it needs to be."

– Bob Lewis

4.1 Computing on High-Dimensionality Polyhedra

One of the challenges of iterative optimization in the polyhedral model is the capability to manipulate rich and complex search spaces of transformation candidates. To maximize expressiveness one has to build a very complex search space, both in terms of the number of variables, and in terms of the space cardinality. In this thesis we propose to build spaces of candidate scheduling coefficient values. To get a clinch at the complexity of the sets we wish to manipulate, consider that the set of candidate transformations has:

- possibly *hundreds* of dimensions, one for each scheduling coefficient of the full program;
- possibly *thousands* of constraints, coming for instance from the semantics-preserving constraints that are embedded into the space;
- possibly *infinite* number of points, as the number of semantics-preserving transformation for a program is often infinite.

Previous approaches for iterative search of scheduling coefficients did not consider building a single space of transformation. Neither Long and Fursin [82] nor Nisbet [87] were able to build candidate optimizations as a single set of legal transformations. This comes from two major challenges. First, the task of building a set of semantics-preserving affine schedules is a very complex problem, and space construction heuristics are needed as shown in Chapter 5. Second, no algorithm was available to enable the traversal of such sets, given their complexity constraints as shown above.

We propose in this chapter to solve the problem of efficiently constructing and traversing very large polyhedral sets, which can represent the set of semantics-preserving schedules for a program. We revisit the problem of manipulating very large polyhedra, pinpointing the scalability bottleneck of existing

algorithms in Section 4.2. We then discuss in Section 4.3 an appropriate representation for these sets, such that standard operations are efficient. To address the problem of dynamically scanning a large polyhedron, we recall the key properties required on the constraints such that a linear-time scanning procedure can be developed, and link these properties to the Fourier-Motzkin elimination algorithm in Section 4.4. We present in detail the Fourier-Motzkin algorithm in Section 4.5 before introducing a small variation of this algorithm which enables the required scalability on our large-sized problem instances.

4.2 Existing Techniques for Polyhedral Manipulation

4.2.1 Manipulation of Presburger Formulas

Presburger arithmetic is a convenient mathematical abstraction to manipulate integer sets. The language of Presburger formulas contains affine equalities and inequalities on integer variables, conjunction and disjunction of those, negation, and first-order quantifiers $\forall$ and $\exists$. Integer linear programming relates to the task of checking the satisfiability of a set of Presburger formulas, an NP-complete problem, and is the core of many optimization algorithms in particular for polyhedral program optimization. Although numerous previous work addressed the problem of checking the emptiness of a polyhedron by eliminating quantified variables, such as Ancourt and Irigoin [6], Irigoin et al. with the PIPS system [61] or Lassez [59], the Omega Test developed by Pugh [92] is a powerful technique based on Fourier-Motzkin elimination to check the emptiness of an integer set. This technique is implemented in the Omega library. It is a complete system for simplifying and verifying Presburger formulas. Of course, the Omega test cannot simplify all Presburger formulas efficiently (there is a non-deterministic lower bound and a deterministic upper bound on the time required to verify Presburger formulas). However, in practice the Omega test is efficient for small-sized problems.

All these tools have mostly been developed for instancewise dependence analysis in the polyhedral model, computing a single optimization for the program, and generate the resulting transformed program. They have to be seen as research efforts focused on the constraints imposed by their application: working on reasonably small sets (a few tens of variables at most), with the emphasis on *deciding* and simplifying Presburger formulas. In contrast, our objective is to manipulate sets with orders of magnitude more variables, and to be able to efficiently perform a dynamic scan of these sets. It is little surprise that our experiments on simplifying and scanning sets based on such tools totally fail to achieve the scalability required.

Another recent development is the Integer Set Library by Verdoolaege [116], which roughly offers the same functionality as the Omega library but using different algorithms. Although this work provides very strong improvements in terms of scalability and efficiency of the mathematical algorithms implemented, the focus still differs from our objectives. The library is dedicated to optimizing the common case for parametric integer sets, and is not able to reach the scalability requirements for the dynamic scanning of very large polyhedral sets. We recall once more that our problem is highly context specific, and differs from standard problems faced in polyhedral program optimization.

4.2.2 Dual and Double-description of Polyhedra

Probably the most standard representation nowadays of (parametric) polyhedra is based on the implicit and dual description of polyhedra. The implicit representation defines a polyhedron as the intersection of

finitely many hyperplanes, each of which in the form of an affine constraint. Given $\mathbb{Q}^n$ the set of rational vectors of dimension n, a polyhedron $\mathcal{P} \in \mathbb{Q}^n$ is defined as follows.

Definition 4.1 (Implicit representation)

$$\mathcal{P} : \{\vec{x} \in \mathbb{Q}^n \mid A\vec{x} = \vec{b},\ C\vec{x} \geq \vec{d}\}$$

The matrix A sets the dimensions which are affine combinations of others, if any. The matrix C sets the affine hyperplanes used to bound the space, if any. One may note that neither A nor C are imposed to be canonical. Implicit equalities can be contained in C: from two or more constraints (that is, rows of C) their combination may exhibit a linear combination of some dimensions. Redundant constraints can be contained too: some constraints may be equivalent or non contributing to bounding the polyhedron $\mathcal{P}$.

If $\mathcal{P}$ is a polyhedron, then it can be decomposed as a polytope plus a polyhedral cone, this is the fundamental theorem of decomposition for polyhedra [101]. The *dual* representation models a polyhedron as a combination of lines L and rays R (forming the polyhedral cone) and vertices V (forming the polytope). The dual representation of a polyhedron $\mathcal{P} \in \mathbb{Q}^n$ is defined as follows.

Definition 4.2 (Dual representation)

$$\mathcal{P} : \{\vec{x} \in \mathbb{Q}^n \mid \vec{x} = L\vec{\lambda} + R\vec{\mu} + V\vec{\nu},\ \vec{\mu} \geq 0,\ \vec{\nu} \geq 0,\ \sum_i \nu_i = 1\}$$

Every polyhedron has both an implicit and dual representation [101]. Furthermore, it is possible to compute a dual representation from the implicit one by using Chernikova's algorithm [26, 74]. Although the complexity of this operation is $\Theta(k^{\lfloor n/2 \rfloor})$, efficient implementations of this algorithm do exist. The implementation by Le Verge and Wilde in PolyLib [74, 122] greatly contributed to popularizing this technique.

Building on these representations, polyhedral operations such as intersection, image under an affine mapping and elimination of redundant constraints can be achieved efficiently. The PolyLib is the result of many years of theoretical developments in the manipulation of parametric polyhedral sets, and is widely used in polyhedral compilation research. It has found numerous applications, in particular in code generation [96, 10] which is a problem strongly related to ours.

But the pitfall of the dual description lies in the explicit representation of the polyhedron's vertices. When again considering very large polytopes, our experiments here shown that the thousands of constraints of the implicit representation can translate into billions of vertices. Simply asking the PolyLib to compute the dual, redundancy-less representation of our solution sets could take hours, making the use of dual representation-based algorithms unsuitable for our problem.

Let us note also the Parma Polyhedra Library (PPL) [2] and Jolylib from Reservoir Labs Inc., two other libraries to manipulate (parametric) polyhedra based on the dual and double-description principles. Although we did not perform experiments with these libraries, our observation on the explosion of the number of vertices still holds.

4.2.3 Number Decision Diagrams

Any Presburger-definable set can be represented with an automaton encoding Number Decision Diagrams [29]. In a nutshell, one can build an automaton that recognizes a language L such that each word

in L is a point in the integer set encoded. Number Decision Diagrams are the automata-based symbolic representation for manipulating sets of integer vectors encoded as strings of digit vectors.

Standard operations such as union, intersection or projection can be computed directly on the automaton representation [127, 72]. An implementation of such techniques is available in the LASH tool [1]. This is a promising direction which is able to offer polynomial-time algorithms to compute heavily expensive algorithms such as the integer hull [73] or other good algorithms to compute the (partial) projection of polyhedral sets [19]. Yet these techniques are not mature enough to be adapted to our problems. Based on the computation time reported to build the automaton representation of a polyhedral set, which gathers the core of the complexity, modeling sets of more than a few tens of variables is simply out of reach in a reasonable time. Still it is a promising direction and we left as a future work of this thesis the deep investigation of number decision diagrams for polyhedral program optimization.

4.3 Polyhedral Operations on the Implicit Representation

From the observations made in the previous section, we conclude that the dual representation of polyhedra is not well suited to address the scalability challenges. Hence we decide to select the implicit representation of polyhedra as a basis for manipulating the solution sets. We first recall the main operations to be used on large polyhedral sets, before discussing in detail the problem of polyhedron simplification for the case of the implicit representation.

4.3.1 Required Polyhedral Operations

Intersection Intersection is the most commonly used operation. During the computation of the solution set, starting from the universe polyhedron we successively reduce it with additional constraints, to remove points from the set. Given $\mathcal{P} = \mathcal{P}_1 \cap \mathcal{P}_2$. The intersection of polyhedra is defined as the conjunction of the constraints defining $\mathcal{P}_1$ and the constraints defining $\mathcal{P}_2$. We choose to not perform any more operation than simply adding all constraints to the polyhedron: no emptiness test is explicitly performed for instance to detect if the resulting intersection is empty.

Note that in the case of the intersection of polyhedra with different dimensionality, we choose to define that the operation implicitly extends the dimensionality of the smallest polyhedron to the dimensionality of the largest one, and leaves the missing dimensions as unconstrained.

Emptiness test To determine if a polyhedron contains at least one integer point, several techniques are available. The Omega test is based on an extension of Fourier-Motzkin variable elimination to integer programming, and has worst-case exponential time complexity [92]. However, for many situations in which other (polynomial) methods are accurate, the Omega test has low order polynomial time complexity. Another technique developed by Feautrier is Parametric Integer Programming (PIP) [39], which also handles the decision problem of the existence of an integer solution in a parametric polyhedron. It is a specific extension of the simplex algorithm to handle parameters, preceded by a parameterized Gomory cuts algorithm.

Although we do not aim at manipulating parametric polyhedra for the solution sets, we use the PIP algorithm as implemented in the PIPLib. The motivation is mainly a convenience for interfacing the tools, but PIPLib is a highly efficient implementation for our concerns.

Projection and dynamic scanning Dynamic polytope scanning is the cornerstone of the decisions taken for the design of algorithms operating on high-dimensionality polyhedra. In order to be efficient, a traversal algorithm must be exhibited to operate with a low complexity. Other operations, such as arbitrarily picking a point in the set, must also be performed efficiently. In Section 4.4 and later we address this problem, by providing an efficient scanning method based on a previous projection of the full set.

Simplification We recall that a constraint is redundant in the description if, once removed, the obtained polyhedron is identical. The benefits of having a redundancy-less representation are twofold. First, no space in memory is wasted to store useless constraints. Second, as the complexity of many polyhedral operations is a function of the number of constraints, one can expect these operations to perform faster on a simplified polyhedron.

Hence we offer the possibility of simplifying a polyhedron by removing the redundant constraints in its implicit description. In particular, we provide a mechanism to efficiently reduce redundancy by removing at almost no computational cost all parallel hyperplanes used to define the polyhedron. We also provide a mechanism to remove *all redundant* constraints. These are detailed below in Section 4.3.2.

4.3.2 Redundancy Elimination

A mandatory operation when working on the implicit representation is the ability to remove redundant constraints from the description. When considering the dual description and the Chernikova algorithm as implemented in the PolyLib, the backward conversion procedure from the dual representation generates a redundancy-less implicit representation. Here, additional effort is required to exhibit a similar capability for polyhedron simplification.

Basic Redundancy Elimination We propose to distinguish two kinds of redundancy. We differentiate the *local redundancy*, where the redundancy can be observed between two constraints, and the *global redundancy* where a constraint has to be checked against more than one constraint.

Definition 4.3 (Local redundancy) *Given two constraints* $\alpha : \sum_{i=1}^{m} c_i x_i \leq q$ *and* $\beta : \sum_{i=1}^{m} d_i x_i \leq q'$. α *is said to be locally redundant with respect to* β *if one of the following holds:*

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

(ii) $\exists k,\ c_k \neq 0,\ \alpha/|c_k| = \beta/|d_k|$ *(same hyperplane)*

The three following examples highlight local redundancy. $x_1 + x_2 \geq 2$ is redundant with respect to $x_1 + x_2 \geq 1$; $2x_1 + 2x_2 \geq 2$ is redundant with $x_1 + x_2 \geq 1$; and $2x_1 + 2x_2 \geq 2$ is redundant with $x_1 + x_2 \geq 0$.

Testing for local redundancy is one of the motivation for our choice to relax the problem over rationals instead of integers. Testing for local redundancy is trivial if the constraints are *normalized*. A constraint with coefficients in $\mathbb{Q}$ can be normalized by selecting selecting i as the highest value for which $c_i \neq 0$, and then set $c_k\prime = c_k/c_i$. The constant part q also must be normalized: $q\prime = q/c_i$. Then, testing if two constraints are parallel (or equal) simply consists to check the equality of all c_k, d_k coefficients, and in that event only look at q and q'. On the other hand, normalizing a constraint which has coefficients

in $\mathbb{Z}$ requires dividing the coefficients by the *gcd* of the full constraint (including q). This prevents from determining local redundancy by simply looking at the equality of coefficient values, and actually requires a much heavier computation.

To efficiently eliminate local redundancy, we apply the following principle. Given a new constraint, first it is normalized and a hash key is associated to it. This key is a function of the c_k coefficients only. Second, the constraint is tested for local redundancy against all other constraints, by first checking the equality of the hash key, then if needed the equality of the c_k coefficients, then if needed the constant parts. Local redundancy reduction is used at each step of all the algorithms operating on polyhedra. Similar techniques are implemented in the Omega library and are standard implementation techniques to control redundant constraint construction.

Although this principle may seem relatively trivial at first glance, we experimentally observed it is a cornerstone of redundancy removal when used with the Fourier-Motzkin elimination algorithm.

Global redundancy elimination A constraint is said to be globally redundant if it is not locally redundant with any other constraint of the system, and the system defines the same polyhedron if we remove this constraint. To detect this kind of redundancy we resort to performing an emptiness test on the global polyhedron according to the following test.

Definition 4.4 (Global redundancy test) *Given $\mathcal{P} \in \mathbb{Q}^m$ and $\alpha : \sum_{i=1}^{m} c_i x_i \leq q$ a hyperplane. α is redundant in the definition of $\mathcal{P}$ if:*

$$\{\mathcal{P} \setminus \alpha\} \cap \left\{ \sum_{i=1}^{m} c_i x_i > q \right\} = \emptyset$$

Local redundancy elimination is detected through a set of lightweight tests that are embedded directly into the polyhedral operations. Global redundancy requires much heavier computation and is a costly process. It implies to perform at worst one polyhedron emptiness test per constraint.

One may note that removing a redundant constraint can make the other redundant constraints non redundant, so it is not possible in general to remove them all at the same time. The order in which constraints are checked for global redundancy can significantly modify the efficiency of the test. Le Fur experimented with several traversal orders to perform the redundancy check [46]. We experimentally evaluated these traversal orders on the system we manipulate, in particular the *ascending* and *descending* orders. The descending order checks first the constraints which are of highest dimensionality, and then progressively checks constraints of lower and lower dimensionality. Conversely, the ascending order checks first the constraints of lowest dimensionality. Our experiments conclude also that the descending method is the most efficient for our problem instances, in particular when used during the elimination of variables with the Fourier-Motzkin algorithm. Local redundancy tests usually perform well on low-dimensionality constraints, motivating the good behavior of the descending order.

4.4 Efficient Dynamic Polytope Scanning

As we aim to represent solution sets as polyhedra, it is required to provide efficient mechanisms to traverse these sets. To design iterative and feed-back driven techniques for the selection of an effective transformation, operations such as partial, exhaustive or random scan must be available. Moreover, for the iterative search to perform efficiently the time to instantiate a point in the space must be negligible.

In the context of polyhedral program optimization, this scanning problem has been largely addressed: the code generation phase does exactly a scanning of the domain polytopes. Still, in the context of iterative compilation, we face solution polytopes of a dimensionality that is orders of magnitude higher than domain polyhedra. Currently, the performance of the best known algorithm for static code generation does not permit to reuse these techniques on the polytopes we consider. This motivates the need for a *dynamic* and scalable approach for scanning large polyhedra.

4.4.1 The Dynamic Scanning Problem

The problem of statically generating a code to scan a polyhedron, called code generation, strongly relates to our problem. These techniques usually rely on recursively projecting the polyhedron to end up with an expression of the projection bounds on the inner-most dimension. Consider for example the following polyhedron $\mathcal{P}$:

$$\mathcal{P} \; : \; \begin{cases} i \geq 0 \\ j \geq 0 \\ i + j \leq 2 \end{cases}$$

To construct all integer points $p \in \mathcal{P}$, one must enumerate the different values for its two coordinates p_1 and p_2, corresponding to the i and j dimensions. Here, $p = [0\ 0]$, $p = [0\ 1]$, $p = [0\ 2]$, $p = [1\ 0]$, $p = [1\ 1]$ and $p = [2\ 0]$ are the 6 integer points in $\mathcal{P}$. To compute these points, we intuitively computed the bounds for p_1 as the projection of $\mathcal{P}$ along the i dimension, enumerated all possible values for it, and for each of the values for p_1 we also computed the bounds for p_2 as the projection along the j dimension and enumerated all possible values for it. We formalize this procedure with the following algorithm to dynamically scan all points $p \in \mathcal{P}$ of dimension m.

EXPLORE $(p, k, \mathcal{P})$:

1. compute the lower bound lb and the upper bound Ub of p_k in $\mathcal{P}$, provided the coordinate values for $p_1, \ldots, p_{k-1}$;
2. for each $x \in [lb, Ub]$:
   (a) set $p_k = x$,
   (b) if $k < m$ call EXPLORE $(p, k+1, \mathcal{P})$ else output p.

The main difference between static and dynamic polyhedron scanning is the knowledge of $p_1, \ldots, p_{k-1}$ when computing lb and Ub in the dynamic case. Moreover, our problem is much simpler in general: we only want to scan a single polyhedron, which is not parametric, and do not care about simplifying the expressions of lb and Ub.

4.4.2 Scanning Points Using Projections

Our objective is to minimize the complexity of the EXPLORE procedure. To achieve this goal, we must control the complexity of the computation of lb and Ub. This computation corresponds to evaluating the bounds of the projection along a specific dimension, given the coordinate values for some other

dimensions. Without further treatment, the complexity of computing lb and Ub is not linear in the number of constraints. Consider for example the following polyhedron $\mathcal{P}$:

$$\mathcal{P} \ : \ \begin{cases} i \quad \geq 0 \\ i \quad \leq 2 \\ \quad j \geq 0 \\ \quad j \leq 2 \\ i + j \geq 3 \end{cases}$$

Here, to compute the actual bounds of i we cannot limit ourselves to inspect the constraints defining $0 \leq i \leq 2$, but rather we have to resort to computing the actual projection of $\mathcal{P}$ along i, etc. We can allow for a pre-processing pass which can be costly if it enables the process of instantiating a point to be very efficient. This is motivated by the fact that the search space is usually built once and for all, while a huge number points may be instantiated.

In order for the EXPLORE procedure to have a complexity linear in the number of constraints, we rely on a fundamental property of the Fourier-Motzkin algorithm. This elimination algorithm generates an equivalent set of constraints such that it is guaranteed that, provided a value in the projection of $v_1, \ldots, v_{k-1}$, a value exists for v_k, for all k. This is called the *FM-property* by some authors (including ourselves), although it is often referred to as row echelon form. This property has been heavily used in the design of code generation algorithms based on the Fourier-Motzkin projection [6]; we recall it in Definition 4.5.

Definition 4.5 (FM property) *Given a polyhedron $\mathcal{P}$ and its implicit representation A. A has the FM property if, for $p \in \mathcal{P}$, the value of the kth coordinate p_k* only depends *on $p_1, \ldots, p_{k-1}$. In other words, if all the affine inequalities in A needed to compute p_k has non-null coefficients only for $a_1, \ldots, a_k$.*

Consider now the polyhedron $\mathcal{P}\prime$, the result of the application of the Fourier-Motzkin algorithm on $\mathcal{P}$:

$$\mathcal{P}\prime \ : \ \begin{cases} i \quad \geq 0 \\ i \quad \geq 1 \\ i \quad \leq 2 \\ \quad j \geq 0 \\ \quad j \geq 1 \\ \quad j \leq 2 \\ i + j \geq 3 \end{cases}$$

Now by simply inspecting the constraints $1 \leq i \leq 2$, one has the correct bounds for the projection of $\mathcal{P}$ on i. This reasoning generalizes of course for polyhedra of arbitrary dimensions, provided that the sequential order chosen to build coordinates is the reverse order of the Fourier-Motzkin elimination steps.

Furthermore, had we applied the simplification techniques described in the previous section, we ended up with a simplified polyhedron with a minimal set of constraints. Combining the FM-property with the EXPLORE procedure yields a linear time technique to build a point: each constraint is visited at most m times to build a point, for a polyhedron of dimension m.

Note that we have not considered in this reasoning the case of holes in the projection of $\mathbb{Z}$-polyhedra. In practice we have relaxed the manipulation of integer sets to rational sets, and the integer hull is obtained by computing the ceil (or floor) of the lb and Ub values obtained. Projection of integer polyhedra

can lead to non-convex sets. Consider for instance the polyhedron $\mathcal{P}$:

$$\mathcal{P} \ : \ \begin{cases} i \geq 0 \\ j \leq 1 \\ 2j - i \leq 0 \\ 3j - i \geq 0 \end{cases}$$

The set of integer points in $\mathcal{P}$ are $p = [0\ 0]$, $p = [2\ 1]$ and $p = [3\ 1]$. Hence the projection on the i dimension is not convex: while $lb = 0$ and $Ub = 3$, there is no point in $\mathcal{P}$ with $p_1 = 1$. For such cases, the FM-property does not necessarily hold for all points between lb and Ub. When an integer hole is encountered during the scan, it is detected by observing for some later dimension that $lb > Ub$. The recursive branching is simply not performed, and the process continues to iterate to the next value for the previous coordinate. We acknowledge the existence of pathological cases where the integer hole occurs at an early dimension while the hole observation is done at a much later depth of the recursion. The design of an efficient backtracking technique for this case is left as a future work. Note that in practice for the polyhedra we scan in the experiments presented in this thesis, the integer hole issue was rarely encountered, and had a totally negligible impact on the traversal efficiency when encountered.

4.5 Fourier-Motzkin Projection Algorithm

There are many methods to solve a system of linear inequalities, some of them giving *one* solution, or simply deciding if a solution exists (see [101] for a comprehensive survey). Since the elimination method works well to solve a system of linear equalities, it was natural to investigate a similar method to solve linear inequalities. Fourier first designed the method, which was rediscovered and studied several times, including by Motzkin in 1936 and Dantzig [34].

The Fourier-Motzkin projection (or elimination) algorithm successively eliminates variables in a given order. The result is a new set of constraints which defines the same polyhedron, but with the notable property of enabling an easy construction of any point in this polyhedron. We now define precisely the algorithm following Banerjee's description of an efficient implementation for it [7].

4.5.1 The Genuine Algorithm

Given a polyhedron $\mathcal{P} : \{\vec{x} \in \mathbb{Q}^m \mid C\vec{x} \geq \vec{d}\}$, we first reduce its dimensionality by performing a Gaussian elimination using the explicit equalities used to define $\mathcal{P}$. The Fourier-Motzkin projection algorithm solves the system defined by C, with m variables and n inequalities:

$$\sum_{i=1}^{m} c_{ij} x_i \geq d_j \quad (1 \leq j \leq n) \tag{4.1}$$

A solution to solve this system is to eliminate the variables one at a time, in the order $x_m,\ x_{m-1},\ \ldots,\ x_1$. The elimination of x_m consists in a projection of the polyhedron (4.1) on the subspace $\{x_1,\ \ldots,\ m-1\}$. This projected polyhedron is defined by:

$$\sum_{i=1}^{m-1} p_{ij} x_i \geq q_j \quad (1 \leq j \leq n') \tag{4.2}$$

(a) Sorting The first step of the algorithm is to rearrange the lines of C such that inequalities where c_{mj} is positive come first, then those where c_{mj} is negative, and eventually those where c_{mj} is 0. We then find integers n_1 and n_2 such that:

$$c_{mj} = \begin{cases} > 0 & \text{if } 1 \leq j \leq n_1, \\ < 0 & \text{if } n_1 + 1 \leq j \leq n_2, \\ = 0 & \text{if } n_2 + 1 \leq j \leq n, \end{cases} \tag{4.3}$$

Coefficients n_1 and n_2 index the positive and negative inequalities.

(b) Normalization For $1 \leq j \leq n_2$, divide the jth inequality by $|c_{mj}|$, to get:

$$\begin{cases} \sum_{i=1}^{m-1} t_{ij} x_i + x_m \geq q_j & (1 \leq j \leq n1) \\ \sum_{i=1}^{m-1} t_{ij} x_i + x_m \leq q_j & (n_1 + 1 \leq j \leq n2) \end{cases} \tag{4.4}$$

Where

$$\begin{cases} t_{ij} & = c_{ij}/|c_{mj}| \\ q_j & = d_j/|c_{mj}| \end{cases}$$

From (4.4) we derive $-\sum_{i=1}^{m-1} t_{ij} x_i + q_j$, an upper bound for x_m for $1 \leq j \leq n_1$, and a lower bound for x_m for $n_1 + 1 \leq j \leq n_2$. It is so possible to define respectively b_m, the "lower bound ball", and B_m the "upper bound ball" as:

$$\begin{aligned} b_m(x_1, \ldots, x_{m-1}) &= \max_{n+1 \leq j \leq n2} \left(-\sum_{i=1}^{m-1} t_{ij} x_i + q_j \right) \\ B_m(x_1, \ldots, x_{m-1}) &= \min_{1 \leq j \leq n1} \left(-\sum_{i=1}^{m-1} t_{ij} x_i + q_j \right) \end{aligned} \tag{4.5}$$

Note that if x_m has no lower bound (meaning $n_1 = n_2$), we simply define $b_m = -\infty$. Reciprocally if x_m has no upper bound ($n_1 = 0$) we define $B_m = \infty$. We can then express the range of x_m as:

$$b_m(x_1, \ldots, x_{m-1}) \leq x_m \leq B_m(x_1, \ldots, x_{m-1}) \tag{4.6}$$

The equation (4.6) is a description of the solution set for x_m.

(c) Create projection We now have the solution set for x_m, and we need to build the constraints for the x_{m-1} dimensions. In addition to the constraints where $c_{mj} = 0$, we simply linearly add each constraint where $c_{mj} > 0$ to each constraints where $c_{mj} < 0$, and add the obtained constraint to the system. One may note that we add $n_1(n_2 - n_1)$ inequalities, and the new system has $n' = n - n_2 + n_1(n_2 - n_1)$ inequalities for $m - 1$ variables. The original system (4.1) has a solution iff this new system has a solution, and so on from m to 1. If during this step, a contradiction occurs ($0 \leq q_j$ with $q_j < 0$) then the system has no solution.

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

The algorithm From the previous formulation, we derive the algorithm in Figure 4.1.

```
Input: A system of n linear inequalities of the form
  sum_{i=1}^{m} a_ij x_i <= c_j   (1 <= j <= n)
Output: The solution set of the input system

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

Figure 4.1: Fourier-Motzkin elimination algorithm

4.5.2 Known Issues

The major drawback of the Fourier-Motzkin elimination algorithm is the strong possibility of generating redundant constraints during step 3 of the algorithm. A constraint is redundant if it is implied by (an)other constraint(s) of the system. Consider the following example:

$$\begin{cases} x_1 + x_2 \geq 0 \\ 2x_1 - x_2 \geq 0 \\ x_1 \geq 0 \end{cases}$$

Then eliminating x_2 produces the additional constraint $3x_1 \geq 0$, which is redundant with $x_1 \geq 0$. The number of constraints can grow exponentially, and the worst case bound of the total number of polynomials in the output of the Fourier-Motzkin algorithm is, for m constraints and n dimensions [120]:

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

More roughly if $n \geq 2$, it is bounded by:

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

4.5.3 Redundancy-Aware Fourier-Motzkin Elimination

The possible explosion of redundant constraints and thus useless computations makes this algorithm poorly scalable, especially on large systems. The main modification which is required to make it scalable is to remove redundant constraints generated by a combination of two constraints. We defined in Section 4.3 useful tools for the detection of local redundancy and constraint normalization. We re-inject

these ideas into the reformulated algorithm of Figure 4.2. In addition, we also reduce the complexity by using six smaller sets of constraints instead of one (their cumulative size is at most the same as the original set).

```
Input: A system of n linear inequalities of the form
  Σ_{i=1}^{m} a_ij x_i ≤ c_j   (1 ≤ j ≤ n)
Output: The solution set of the input system


forall j do
   Normalize the constraint by |a_mj| (if a_mj ≠ 0)
   if a_mj > 0 STORE the inequality in S+
   if a_mj < 0 STORE the inequality in S−
   if a_mj = 0 STORE the inequality in S0
end do
B_m ← S−
b_m ← S+
for k ← m − 1 to 1 do
  ∀ s+,s− ∈ S+ × S−, C = s+ + s−. If a
  contradiction occurs, stop: no solution
  ∀ s0 ∈ S0 where s0_{k−1} ≠ 0, |C = s0
  if k > 1 then
      ∀ C:
          normalize C on the k − 1^th variable.
          if c_{k−1} > 0 STORE the inequality in S+′
          if c_{k−1} < 0 STORE the inequality in S−′
          if c_{k−1} = 0 STORE the inequality in S0′
   if S+′ = ∅ ∧ S−′ = ∅ ∧ S0′ = ∅ ∧ S0 = ∅ stop: finished
   if k > 1, |B_k = S−, b_k = S+
   if k = 1, |B_k = max(S+), b_k = min(S−)
   S+ = S+′, |S− = S−′, |S0 = S0′
   removeGlobalRedundancy(S+ ∩ S− ∩ S0)
end do
```

Figure 4.2: Modified Fourier-Motzkin elimination

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

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

For global redundancy elimination, we resort to testing and eliminating constraints on the set resulting from a full projection step. For this particular purpose the Le Fur descending order performs very well. The output of our algorithm is thus a simplified polyhedron with a minimal set of constraints for its implicit description.

This modified algorithm does not contain highly complex mathematical optimization. Instead, we simply provide an efficient mechanism to control redundancy generation at each step of the algorithm. The output of this algorithm is a new set of constraints for the polyhedron which respects the FM-property, and containing *no redundant constraints*. Our extensive experiments over the thousands of polyhedra generated for the purpose of iterative schedule selection have demonstrated that redundancy was the only major bottleneck of Fourier-Motzkin elimination on such problem instances. The modified algorithm was implemented in the FM library [91] and, to the best of the author's knowledge, is to date

the most adapted algorithm for the projection and normalization of scheduling coefficients polyhedra as constructed in this thesis.

4.5.4 Some Related Works

The study of eliminating redundant constraints is a long-term issue, and was treated by numerous authors. Many of the most useful references can be found in Schrijver [101]. For a few additional references let us explicitly cite Chernikov's work [25, 83] who produced the *reduced convolution method*, which relies on the concept of linear-dependence of the vectors generating a cone. Let us also recall the work of Pugh [92] on the Omega Test, a practical method to solve integer linear programs; and Sehr et *al.* [105] who worked on redundancy reduction heuristics based on the Chernikov criterion. We may also note the work of Weispfenning [120] who proposed a derivative of the Fourier-Motzkin elimination with an exponential worst case upper bound complexity.

Part II

Constructing Program Optimizations

Chapter 5

Building Practical Search Spaces

5.1 Introduction

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

A limitation of model-based approaches is the accuracy and portability of the optimization objective. A strong motivation for offering a complete methodology to build a set of candidate optimizations is to isolate and postpone the selection of a transformation to a subsequent stage. This result in a decoupling of the expressiveness issue and the selection problem. Furthermore, by letting iterative processes such as adaptive compilation working on a well-formed space we enable the possibility to focus the search on relevant candidates only.

Our solution for the search space construction is based on embedding critical properties directly into the search space, to reduce the complexity of the selection stage. The three following properties are embedded in any search space generated by our algorithms.

1. *Expressiveness.* We aim at building a search space which encompasses arbitrarily complex sequences of transformations.

2. *Legality.* All candidates in the search space preserve the semantics of the original program.

3. *Uniqueness.* There is no duplicate in the space: each point corresponds to a distinct transformation, leading to a distinct candidate version (that is, a distinct syntactic program) after the application of the transformation.

In Chapter 3 we have shown that a space with maximal expressiveness can be built in a convex fashion, hence bearing some of the mandatory tractability properties for its traversal. We address now the problem of building practical search spaces, navigating the trade-off between expressiveness and optimality of the solution versus tractability of the space construction and its traversal.

5.1.1 The Trade-Off Between Expressiveness and Practicality

When considering a search space with maximal expressiveness which contains all transformations accessible for the current framework, scalability and convergence towards the optimal solution are the dominant challenges.

1. *Pros:* Optimality of the solution guaranteed within the scope of the framework.
2. *Cons:* High complexity of polyhedral operations: they operate on high dimensional polyhedra.
3. *Cons:* Very large search spaces, challenging the convergence towards the optimal solution.

Complexity of Linear Programs The method proposed in Chapter 3 requires computing on a space containing a possibly huge set of variables. For an illustration, consider the example of Figure 5.1.

```
/* Determine mean of column vectors of input data matrix */
for (j = 1; j <= m; j++) {
    mean[j] = 0.0;
    for (i = 1; i <= n; i++)
        mean[j] += data[i][j];
    mean[j] /= float_n;
}
/* Determine standard deviations of column vectors of data matrix. */
for (j = 1; j <= m; j++) {
    stddev[j] = 0.0;
    for (i = 1; i <= n; i++)
        stddev[j] += (data[i][j] - mean[j]) * (data[i][j] - mean[j]);
    stddev[j] /= float_n;
    stddev[j] = sqrt(stddev[j]);
    /* The following in an inelegant but usual way to handle
       near-zero std. dev. values, which below would cause a zero-
       divide. */
    stddev[j] = stddev[j] <= eps ? 1.0 : stddev[j];
}
/* Center and reduce the column vectors. */
for (i = 1; i <= n; i++)
    for (j = 1; j <= m; j++) {
        data[i][j] -= mean[j];
        data[i][j] /= sqrt(float_n) * stddev[j];
 }
/* Calculate the m * m correlation matrix. */
for (j1 = 1; j1 <= m-1; j1++) {
    symmat[j1][j1] = 1.0;
    for (j2 = j1+1; j2 <= m; j2++) {
        symmat[j1][j2] = 0.0;
        for (i = 1; i <= n; i++)
           symmat[j1][j2] += (data[i][j1] * data[i][j2]);
        symmat[j2][j1] = symmat[j1][j2];
    }
}
symmat[m][m] = 1.0;
```

Figure 5.1: Correlation program

In this example, constructing the space of all and only legal 3-dimensional affine schedules involves computing on a system with ≈ 500 variables. Hence, any optimization method which would aim at finding a space optimal point would require to solve an ILP on those 500 variables. Furthermore, in the context of iterative search projecting the legality constraints to shape the space in a form suitable for dynamic traversal is required. This may imply to perform thousands of emptiness tests on such sized

polytopes. At the time of writing of this manuscript, no known solving method exists to scale efficiently to the number of variables involved for programs of more than a few tens of statements. Although we have presented in Chapter 4 an efficient and scalable technique for projection, this technique reaches its scalability limit on larger spaces. When considering programs beyond 10 to 15 statements, another strategy must be devised.

One of the practicality concerns when building a search space is to control the number of variables involved at each stage of the process. We present in this chapter different strategies for significantly reducing the space dimensionality.

Performance Distribution Along with more expressiveness comes the increasing chance to have (possibly many) candidates which have a similar or even very bad performance. For instance, considering the matVect example of Figure 5.2.

```
  for (i = 0; i <= n; i++) {
R   s[i] = 0;
    for (j = 0; j <= n; j++)
S     s[i] = s[i] + a[i][j] * x[j];
  }
```

Figure 5.2: matVect kernel

Applying a shift of 1 with the transformation:

$$\Theta^R = \begin{pmatrix} 1 & 0 & 1 \end{pmatrix} . \begin{pmatrix} i_R \\ n \\ 1 \end{pmatrix}, \quad \Theta^S = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} i_S \\ j_S \\ n \\ 1 \end{pmatrix}$$

should not alter the performance. Applying a slowdown of 5 with the transformation:

$$\Theta^R = \begin{pmatrix} 5 & 0 & 0 \end{pmatrix} . \begin{pmatrix} i_R \\ n \\ 1 \end{pmatrix}, \quad \Theta^S = \begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} i_S \\ j_S \\ n \\ 1 \end{pmatrix}$$

will generate complex control including modulo operations, as shown in Figure 5.3.

```
  for (i = 0; i <= 5 * n; i++) {
    if (i % 5 == 0) {
R     s[i / 5] = 0;
      for (j = 0; j <= n; j++)
S       s[i / 5] = s[i / 5] + a[i / 5][j] * x[j];
    }
  }
```

Figure 5.3: Transformed matVect kernel

These intuitive comments point us towards a space pruning approach, where some transformations (that is, some coefficients values) are not explored. We show in Chapter 6 that, mostly due to the complexity of the optimization processes in most modern compilers, achieving the best performance may require tweaking all coefficients. Yet, as unnecessary complex controls should be avoided, we will favor

a coefficient bounding approach. Note that one strategy could use *a posteriori* normalization techniques for the schedule discussed by Vasilache [111], but this would harm the expressiveness and remove the uniqueness property for the space as several distinct schedules may be identical after normalization.

Target Architecture Features Another concern about building a highly expressive search space is the adequacy of the properties of the candidate program versions for the target architecture features. The target architecture must drive the simplification, by considering at least:

- parallelism (multi-core, multiple vector computation units, etc.);
- memory features (existence of cache memory, depth of the hierarchy, etc.);
- dedicated blocks for some operations (hard-wired modulo operations, for instance).

High-level characteristics such as parallelism or schedule latency are not the only concern. It is critical to observe that the complex interplay between all architectural features, emphasized by the growing complexity of modern processors, requires finely tuning the program to exhibit high-performance. Even the tiniest, most of the time unexpected, modification of some of the scheduling coefficients can increase performance in a hardly predictable manner. To assess the need to preserve a significant degree of expressiveness in the search space, we show in Chapter 6 that this tuning has a dramatic impact, for instance even within a set of candidates exhibiting identical vectorizable inner-loops.

Traversal and Convergence We have discussed in Chapter 3 some of the mandatory properties an affine search space should encompass to enable an efficient traversal. Reducing the dimensionality of the space is a key factor for the normalization step (to force the Fourier-Motzkin property on the space), hence the dimensionality must be adapted to the scalability capability of the projection algorithm discussed in Chapter 4.

For an efficient convergence of an iterative technique towards a good solution, it is essential to build dedicated space traversal methods. This is because exhaustive search would not be feasible on spaces larger than a few hundreds of points, and because random or pseudo-random search may be efficient only if the proportion of good solutions is large. Intuitively, larger spaces are harder to explore, unless some properties could be exploited to focus the search. We show in Chapter 7 that such methods, leveraging both static and dynamic characteristics of the performance distribution, can be developed.

5.1.2 Different Goals, Different Approaches

To the best of our knowledge, no previous work studied the performance distribution of affine schedules for computation-intensive programs. We can exhibit two main directions to construct and traverse a search space of affine transformations, the first favoring expressiveness in the search space, the second focusing on aggressive pruning of the space.

Favoring Expressiveness The first approach we develop is to build and enumerate a search space of affine transformations by constructing a large and very expressive search space, where only the smallest input bias is used to prune the space. In a word, this first approach favors the quantity of the possible transformations.

With this method, it is necessary to develop traversal techniques that are adapted to the performance distribution as we delegate to the traversal phase the task of focusing on relevant candidates. The only bias that we choose to embed into the space itself is the form of parallelism: we propose an algorithm for the search space construction which is geared towards inner parallelism when possible, allowing for an efficient vectorization. This approach typically targets architectures with SIMD-capable processors, but is not limited to it.

We show in the following that this approach enables the construction of extremely rich search spaces, both in terms of the variability of the produced output and in performance distribution. In the following chapter, we perform an extensive study of this distribution, exhibiting key properties required to design very aggressive and efficient pruning strategies.

Favoring Space Pruning The second approach that is studied in this work is to perform several levels of pruning directly into the search space. In a word, this approach favors the static characteristics of the possible transformations.

Beyond constructing a search space which models only very few scheduling coefficients — in contrast of the first method — we refine the obtained space by pruning it to focus on several static properties of the schedules: tilability, parallelism, locality and latency. This approach provides a more constrained framework: candidate transformations are already optimized amongst several goals. The task of selecting a good schedule in the search space is extremely simplified, and even for programs of a dozen of statements an exhaustive enumeration is tractable. This technique is developed in Chapter 8.

5.2 Search Space Construction

We first present an algorithm to construct and practically bound the search space of all, distinct one-dimensional schedules in Section 5.2.1. We then generalize this technique for the case of multidimensional schedules in Section 5.2.2, then building a search space of program transformations for any program amenable to polyhedral representation.

5.2.1 One-Dimensional Schedules

We have described in Chapter 3 a technique to linearize the precedence constraints contained in the dependence polyhedra into a single affine set. This results in the construction of the space of legal one-dimensional schedules $\mathcal{T}$ where, for k dependences,

$$\mathcal{T} = \bigcap_k \mathcal{T}_k$$

This formulation was first designed by Feautrier [41], the only notable difference with ours being that we take schedule coefficients in $\mathbb{Z}$ instead of $\mathbb{N}$. This results in removing one linearization step that involved applying the Farkas Lemma on the schedule coefficients also. Had we applied Feautrier's initial method using Farkas multipliers to compute a non-negative schedule, we would have faced the problem that the function giving the schedule coefficients from the schedule Farkas multipliers is not injective (many points of the obtained polyhedron can give the same schedule). This method, to the contrary, is very well suited for an exploration of the different legal schedules since every (integral) point in $\mathcal{T}$ is a different schedule.

Constructing $\mathcal{T}$ is efficient and tractable: it only involves solving small linear programs. The size of the systems to solve can be easily expressed. For each statement S, there are exactly:

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

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

$$S_{\mathcal{D}_{R,S}} = 2.d_R + 2.d_S + p + s + 1$$

where p is the size of the precedence constraints (at most $\min(d_R, d_S) - 1$) and s is the subscript equality. Given Ω the set of statements, the dimension of the computed systems are at most:

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

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

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

This formulation gives a practical method to compute the (possibly infinite) set of legal one-dimensional schedules for a program. We need to bound $\mathcal{T}$ into a polytope in order to make achievable an exhaustive search of the transformation space. The bounds are given by considering a side-effect of the used code generation algorithm [10] for sequential targets: *the higher the transformation coefficients, the more likely the generated code to be ineffective*.

Practical Bounding of the Space

The legal one-dimensional schedule space for a given SCoP as described in Chapter 3 is possibly infinite. For instance it is easy to see that if there is no data dependence at all, every value of the schedule coefficients is possible. It is necessary to bound this space in such a way that an exhaustive scan becomes possible. Bounding the space will remove some possible program transformations. We have to ensure we remove only the less interesting solutions for performance.

We can distinguish two families of coefficients in the schedule expressions, (1) iterator coefficients, (2) parameter and constant coefficients. Each family will provide a specific contribution to the global program transformation [12]. The iterator coefficients will impact on loop structure and bounds (*skewing*-like transformations for instance) while parameters and constant will impact on loop ordering and statement ordering within a loop (*shifting*-like transformations for instance). It follows, while the order of magnitude of coefficients values for parameters and constant do not have any influence on performance, using big iterator coefficients will result in a very high control overhead (like generation of complex loop

bounds and costly modulo operation) that will waste the optimization they are potentially enabling [10]. Hence we should bound the values of the iterator coefficients with small values (we checked empirically that the bounding interval $[-1,1]$ is wide enough most of the time).

Parameter coefficients may have a hidden influence on locality. Let us consider the example locality of Figure 5.4

```
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j) {
R     b[j] = a[j];
S     c[j] = a[j + m];
    }
```

Figure 5.4: locality

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

Had we used Feautrier's genuine method to compute positive affine schedules by applying Farkas lemma also on schedule functions [41], we would have missed the bounds on m, since it is not involved in any loop bound (no Farkas multiplier is attached to it, its schedule coefficient would always have been 0).

Still, the coefficients of the parameters and the constant have also to be bounded to avoid an infinite search space. The difference between the two bounds should be greater than the number of statements to ensure that at least every ordering of the statements within or outside loops is possible. Greater intervals will offer more possibilities, for instance to achieve more peeling transformations but a large flexibility is rarely useful in practice.

We made several tests to compare our approach, taking into account only the legal schedules, and considering all schedules then the filter legal ones using a legality check, as Long et al. suggest [82]. We used different compute-intensive kernel benchmarks coming from various origins and listed in Figure 5.5. `h264` is a fractional sample interpolation of the H.264 standard [121]. `fir` and `fft` are DSP kernels extracted from UTDSP benchmark suite [121]. `lu`, `gauss`, `crout` and `matmult` are well known mathematical kernels corresponding to LU factorization, Gaussian elimination, Crout matrix decomposition and matrix-matrix multiply. `MVT` is a kernel including two matrix-vector multiplications, one matrix being the transposition of the other. `locality` is a hand-written memory access intensive kernel. Notice our motivation is not to evaluate the performance of our schedules with respect to aggressive optimizations performed manually (like the BLAS), or by application-specific active libraries (like ATLAS or SPIRAL): we are evaluating an automatic source-to-source framework, exploring *all but only* one-dimensional schedules, and not considering any domain-specific knowledge.

These kernels are typically small, from 2 to 17 statements. They suit well the present study and allow for a fair comparison with present production compiler: first, they should not challenge present production compiler optimization schemes, and second, they will make it possible to achieve an exhaustive traversal of our search space which is necessary to evaluate the potential of the method and to design heuristic techniques. Dealing with larger benchmarks presents some technical difficulties: first of all, every SCoP does not have a one-dimensional schedule, and the likeliness decreases with the complexity of

the dependence graph; second, although we achieved a breakthrough by giving the possibility for much larger optimization spaces to be characterized and traversed, going beyond 20 to 30 statements challenges the scalability of our constraint simplification method (based on Fourier-Motzkin elimination), due to the hundreds of transformation coefficients to consider simultaneously.

Further scalability may be achieved through algorithmic improvements in the exploitation of regularity properties in the constraint systems, and through heuristics to prioritize the most important dependences and / or to partition the problem into smaller, modular scheduling spaces.

Figure 5.5 summarizes the study of the search space. The first column presents the various kernel benchmarks; the second one labeled #Dependences shows the number of dependence relations for the corresponding kernel; $\vec{\imath}$-Bounds shows the iterator coefficient bounds used for search space bounding; $\vec{p}$-Bounds shows the parameter coefficient bounds; c-Bounds shows the constant coefficient bounds; #Schedules shows the total number of schedules, including illegal ones; #Legal shows the number of actual schedules in our space, i.e. the number of legal schedules; finally, Time shows the search space computation time on a Pentium 4 Xeon, 3.2GHz.

We can represent classical loop transformations like reversal, skewing, and slowing only with the $\vec{\imath}$ coefficients. Their values directly imply the complexity of the control bounds, and may generate modulo operations in the produced code; and bounds between -1 and 1 are accurate most of the time. Yet to enable the discovery of a one-dimensional schedule, in some cases like FFT it is required to increase the bounds, to allow for larger slowing or for the creation of more distinct loop nests with loop distribution.

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

Figure 5.5: Search space computation

The results show the very high benefit of working directly on a space including only legal transformations since it lowers the number of considered transformations by one to many orders of magnitude for a quite acceptable computation time. These results also show that without such a policy, achieving an exhaustive search is not possible even for small kernels. While these results show profitability, it is not a demonstration of scalability. In Chapter 7 we will propose to actually visit the search space exhaustively or heuristically.

5.2.2 Generalization to Multidimensional Schedules

One-dimensional schedules suffer from many limitations, the dominant being that:

1. not all programs accept a one-dimensional schedule,
2. many combinations of transformations are not modeled.

Multidimensional schedules allows working on any program, and the space of multidimensional affine schedules is very expressive. Each point in the space corresponds to potentially very different program versions, exposing a wide spectrum of interactions between architectural components and back-end compiler optimizations. This section presents the construction of a practical *space of legal, distinct affine multi-dimensional schedules*.

Multidimensional Problem

Nisbet [87], then Long and Fursin [81] experimentally observed that choosing a schedule at random is very likely to lead to an illegal program version. Moreover, the probability of finding a *legal* one (which does not alter semantics) decreases exponentially with program size, as shown in the previous section. This challenge can only be tackled when integrating data dependence information into the construction of the search space.

We showed it is possible to build efficiently the legal space for small programs that accept one-dimensional schedules. But dealing with multidimensional schedules leads to a combinatorial explosion.

Using one-dimensional schedules, all dependences have to be satisfied within a single time dimension: the precedence constraint is simply $\theta_R(\vec{x_R}) < \theta_S(\vec{x_S})$ and θ is a row vector. In multidimensional schedules, the legality constraints can also be built time dimension per time dimension, with the difference that a dependence needs to be *weakly satisfied* — $\theta_S(\vec{x_S}) - \theta_R(\vec{x_R}) \geq 0$ — for the first time dimensions until it is *strongly satisfied* — $\theta_S(\vec{x_S}) - \theta_R(\vec{x_R}) > 0$ — at a given time dimension d. Once a dependence has been strongly satisfied, no additional constraint is required for legality at dimensions $d' > d$. Reciprocally, a dependence must be weakly satisfied for all $d'' < d$. There is freedom to *decide* at which time dimension a dependence will be strongly satisfied. Each possible decision leads to a potentially different search space. Furthermore, it is possible to arbitrarily increase the number of time dimensions of the schedule, resulting in an infinite set of scenarios in general.

The output of our algorithm is in the form of a list of polyhedra of legal schedules, one for each time dimension. This scheme significantly favors polyhedral tractability when compared to the technique presented in Chapter 3: the dimension of polyhedra is limited to the number of schedule coefficients *for one schedule dimension*. That is, we have removed all binary variables required for dependence satisfaction, and partitioned the schedule coefficients in rows, with one polyhedron of possible values per row. The downside is that, for each dependence satisfaction scenario, we may end up building a different set of polyhedra for the search space. This makes the problem highly combinatorial.

Building a Practical Search Space

To build the search space, we face two combinatorial problems. First, there are too many scenarios to be considered. Second, one needs to limit the search to bounded polytopes. Yet, even the smallest bound leads to polytopes that are too large to be explored exhaustively for complex loop nests.

Feautrier found a systematic solution to the explosion of the number of polyhedra: he considers a space of legal schedules leading to maximum fine-grain parallelism [42, 118]. To achieve this, a greedy algorithm maximizes the number of dependences solved for a given dimension. While this solution is interesting because it reduces the number of dimensions and may exhibit inner parallelism, it is not practical enough in its original formulation for several reasons.

First, it needs to solve a system of linear inequalities involving every schedule coefficient *plus* a

decision variable per dependence [42]. This makes the problem very complex to solve for kernels with a large set of dependences.

Maximizing the number of dependences satisfied at outer levels relaxes the scheduling constraints on the inner levels. Then, inner parallelism may be exhibited in the schedule and exploited by the back-end compiler. However, reducing the number of schedule dimensions is not wanted for our purpose, as we wish to build a search space where the search process is able to operate on each loop level. To maximize the expressiveness of the search space, and hence its expected efficiency, iteratively selecting coefficients of the inner dimensions is critical. Thus it can have a dramatic performance impact: inner dimensions are associated with the inner loops, and the vectorization process can be highly influenced by the shape of these inner loops.

In addition, minimizing the number of dimensions often translates into big schedule coefficients; these generally lead to algorithmic complexity and both significant loop bounds and control flow overhead after generation of the target imperative code [63]. We can bound the coefficient values of the linear part of the schedule within $\{-1,0,1\}$ to minimize control-flow overhead. Non-unit coefficients for the linear part of the schedule are required to model compositions of slowing and non-unit skewing, however we believe the benefit to search for such compositions of transformations is in average exceeded by the benefit of generating programs with simpler control-flow. Note this bounding would have been very restrictive if we were constrained to one-dimensional schedules. In the multidimensional case, although it eliminates some schedules from the space, these bounds are compatible with the expression of arbitrary compositions of loop fusion, distribution, interchange, code motion; in the worst case, it translates into additional time dimensions.

Algorithm 5.6 sketches our search space construction for a given static control part. It outputs a collection of polytopes $\mathcal{T} = \{\mathcal{T}_d\}$, where $\mathcal{T}_d$ is the polytope of legal scheduling coefficients for dimension d. Procedure `createPolytope` creates a polytope with one variable per coefficient in a row of the program scheduling matrix. We use the first range argument ($[-1,1]$ here) to bound the values of coefficients attached to loop iterators (that is, the linear part of the schedule). We use the second range argument ($[-1,1]$ also here) to bound the values of coefficients attached to the parametric constant part. Procedure `buildWeakLegalSchedules` builds the constraints on the scheduling coefficients such that the dependence is weakly satisfied by any schedule in the computed set. The Farkas Lemma is used to express the conditions on affine functions which are non-negative over the given dependence polyhedron $\mathcal{D}_{R,S}$, for $\Delta_{R,S} = \Theta^S_d(\vec{x}_S) - \Theta^R_d(\vec{x}_R)$. Farkas multipliers are projected so that the set $\mathcal{W}_{\mathcal{D}_{R,S}}$ contains only constraints on the schedule coefficients. Procedure `buildStrongLegalSchedules` builds the constraints on the scheduling coefficients such that the dependence is strongly satisfied by any schedule in the computed set. Each schedule in $\mathcal{S}_{\mathcal{D}_{R,S}}$ enforce $\Delta_{R,S} = \Theta^S_d(\vec{x}_S) - \Theta^R_d(\vec{x}_R) - 1$ to be non-negative for all points of the dependence polyhedron. The reader may refer to Section 3.1.2 for an example of how such a set is computed.

To study the termination of our algorithm, we first observe that arbitrary bounds for the scheduling coefficients may prevent finding at least one schedule for the program. A simple example is given in Figure 5.7, together with the bounds $[0,1]$ for all coefficients. Here a range of 3 possible values for the constant part (e.g., $[0,2]$) is required to exhibit a valid schedule.

In general, bounding the linear part of the schedule to $[0,1]$ does not prevent finding a schedule, however bounding the constant part to $[0,1]$ may. To guarantee the algorithm terminates, we first observe that an interval of size x for the bounding coefficients attached to the constant part, for a program with x statements, is enough to guarantee the existence of a schedule. Second, we rely on the same termination proof as Feautrier's multidimensional scheduling algorithm [118]: at least one dependence can be

```
BuildSearchSpace: Compute 𝒯
Input:
   pdg: polyhedral dependence graph
Output:
   𝒯 : the bounded space of candidate multidimensional schedules

1  d ← 1
2  while pdg ≠ ∅ do
3      𝒯_d ← createPolytope([−1,1],[−1,1])
4      for each dependence 𝒟_{R,S} ∈ pdg do
5          𝒲_{𝒟_{R,S}} ← buildWeakLegalSchedules(𝒟_{R,S})
6          𝒯_d ← 𝒯_d ∩ 𝒲_{𝒟_{R,S}}
7      end for
8      for each dependence 𝒟_{R,S} ∈ pdg do
9          𝒮_{𝒟_{R,S}} ← buildStrongLegalSchedules(𝒟_{R,S})
10         if 𝒯_d ∩ 𝒮_{𝒟_{R,S}} ≠ ∅ then
11             𝒯_d ← 𝒯_d ∩ 𝒮_{𝒟_{R,S}}
12             pdg ← pdg - 𝒟_{R,S}
13         end if
14     end for
15end do
```

Figure 5.6: Algorithm for Search Space Construction

```
  for (i = 0; i <= n; i++)
R |  s += s;
  for (i = 0; i <= n; i++)
S |  s += s;
  for (i = 0; i <= n; i++)
T |  s += s;
```

Figure 5.7: A Program with no schedule in $[0,1]$

strongly solved per time dimension d. Note that in our experiments presented in the next section, we have used the bounding $[-1,1]$ for all coefficients: these bounds did not prevent us from finding a valid solution set for the considered benchmarks.

Our algorithm is a bounded variation of Feautrier's genuine algorithm for maximal fine-grain parallelism. Although it benefits from the same termination property provided an appropriate coefficient bounding, it differs in the properties of the generated set of schedules. It does not guarantee a maximal number of dependences solved per dimension. Therefore, it may not minimize the number of dimensions of the schedule. Interestingly, if we remove the coefficient bounding and initialize $\mathcal{T}_d$ as the universe polyhedron instead, this algorithm is fully equivalent to Feautrier's genuine version. This is because the maximal set of dependences which can be strongly solved for a given dimension is unique [42]. Hence, our algorithm can be used to favor the solving of several smaller problems (one emptiness test per dependence) in place of a single problem involving additionally one binary variable per dependence.

Our algorithm is efficient and only needs one polyhedron emptiness test per dependence (over $\mathcal{T}_d$ which contains exactly one variable per schedule coefficient). The elimination of Farkas multipliers used to enforce the precedence constraint on schedule coefficients is performed dependence per dependence (i.e., on very small systems).

So far, we have not defined the order in which dependences are considered when checking against strong satisfaction. This problem does not arise with Feautrier's genuine algorithm because of the uniqueness of the maximal set of strongly satisfied dependence for a given dimension. But with our bounded

version, this order can have a significant impact on the constructed space.

Considering two dependences d^1 and d^2, such that they can individually be strongly satisfied for the current scheduling level — that is, there exists a schedule Θ^1 strongly satisfying d^1 and weakly satisfying all other program dependences, similarly for Θ^2 and d^2. Without any coefficient bounding, it is possible to strongly solve d^1 and d^2 at the current scheduling level, $\Theta = \Theta^1 + \Theta^2$ is such a solution. But because we have bounded the coefficients, the scheduling constraints imposed by d^1 may prevent to find a solution to strongly satisfy also d^2 at the same time. Intuitively, this arises because we may not be able to form $\Theta = \Theta_1 + \Theta_2$, the composite schedule satisfying both dependences, without going out of the schedule coefficients bounds. As a consequence, the dependence order in which we perform the check of step 10 can lead to having a different number of dependences to be strongly solved for a given schedule dimension.

A long term approach would be to consider this order as part of the search space, but this is not currently practical due to combinatorial explosion. Instead, we use two analytical criteria to order the dependences. First of all, each dependence is assigned a priority, depending on the memory traffic generated by the pair of statements in dependence. We use a simplified version of the model by Bastoul and Feautrier [13]: for each array A and dimension d, we approximate the traffic as $m_d^{r_A}$, where m_d is the size of the dth dimension of the array, and r_A is the rank of the concatenation of the subscript matrices of all references to dimension d of array A in the statement. Thus, the generated traffic evaluation for a given statement is a multivariate polynomial in the parametric sizes of all arrays. We use profiling to instantiate these size parameters. Intuitively, maximizing the depth where a dependence is strongly solved maximizes reuse in inner loops and minimizes the memory traffic in outer loops. Therefore, we start with dependences involved in the statements with the least traffic. Our second criterion is based on *dependence interference*; it is used in case of non-discriminating priorities resulting from the first criterion. Two dependences interfere if it is impossible to build a one-dimensional schedule strongly satisfying these two dependences. We first try to solve dependences interfering with the smaller number of other dependences, maximizing our chance to strongly solve more dependences within the current time dimension.

Search Space Statistics

Benchmark	#Inst.	#Loops	#Dep.	#Dim.	dim 1	dim 2	dim 3	dim 4	Total
compress-dct	6	6	56	3	20	136	10857025	*n/a*	2.9×10^{10}
edge	3	4	30	4	27	54	90534	43046721	5.6×10^{15}
iir	8	2	66	3	18	6984	$> 10^{15}$	*n/a*	$> 10^{19}$
fir	4	2	36	2	18	52953	*n/a*	*n/a*	9.5×10^{7}
lmsfir	9	3	112	2	27	10534223	*n/a*	*n/a*	2.8×10^{8}
matmult	2	3	7	1	912	*n/a*	*n/a*	*n/a*	912
latnrm	11	3	75	3	9	1896502	$> 10^{15}$	*n/a*	$> 10^{22}$
lpc	12	7	85	2	63594	$> 10^{20}$	*n/a*	*n/a*	$> 10^{25}$
ludcmp	14	10	187	3	36	$> 10^{20}$	$> 10^{25}$	*n/a*	$> 10^{46}$
radar	17	20	153	3	400	$> 10^{20}$	$> 10^{25}$	*n/a*	$> 10^{48}$

Figure 5.8: Search space statistics

Figure 5.8 summarizes the size of the legal polytopes for different benchmarks, for all schedule dimensions. We consider 10 SCoPs extracted from classical benchmarks. The first eight are UTDSP benchmarks [76] directly amenable to polyhedral representation: `compress-dct` is an image compression kernel (8x8 discrete cosine transform), `edge-convolve2d` is an edge detection kernel (different

from Ring-Roberts), `fir` is a Finite Impulse Response filter, `lmsfir` is a Least Mean Square adaptive FIR filter, `iir` is an Infinite Impulse Response filter, `matmult` is a matrix multiplication kernel, `latnrm` is a normalized lattice filter, and `lpc (LPC_analysis)` is the hot function of a linear predictive coding encoder. We considered two additional benchmarks: `ludcmp` solves simultaneous linear equations by LU decomposition, and `radar` is an industry code for the analysis of radar pulses. For each benchmark, we report the number of (complex) instructions carrying array accesses (#Inst), the number of loops (#Loops), dependences (#Dep), schedule dimension (#Dim), and the total number of points for those dimensions (still only legal schedules) where $> 10^n$ provides a conservative lower bound when it was not possible to compute the exact space size in a reasonable amount of time.

5.2.3 Scanning the Search Space Polytopes

The algorithm presented in Section 5.2.2 constructs one polytope per dimension of the schedule. Picking one point in every polytope $\mathcal{T}_d$ fully describes one multidimensional schedule, hence one program version: the generated imperative codes will be distinct if the scheduling matrices are distinct. To construct a program version, that is a schedule, we need to scan the legal polytopes. This is reminiscent of the classical polyhedron scanning problem [65, 10]; however, none of the existing algorithms scale to the hundreds of dimensions we are considering. Fortunately, our problem happens to be simpler than "static" loop nest generation: we only need to "dynamically" enumerate every integral point that respects the set of constraints.

Each program version is represented by a unique scheduling matrix Θ. The first columns are schedule coefficients associated with each loop iterator surrounding a statement in the original program ($\vec{\imath}$), for all statements. The next set of columns are schedule coefficients associated with global parameters ($\vec{p}$), for all statements. The last column are the schedule coefficients associated with the constant (c), for all statements.

Since we represent legal schedules as multidimensional affine functions, each row Θ_d of the scheduling function corresponds to an integer point in the polytope of legal coefficients $\mathcal{T}_d$, built explicitly for this dimension. A program version in the optimization space can thus be represented as follows, for a SCoP of t statements, a schedule of dimension s, and the iteration vector $\vec{x}$:

$$\Theta.\vec{x} = \begin{pmatrix} \vec{\imath}_1^1 & \cdots & \vec{\imath}_t^1 & \vec{p}_1^1 & \cdots & \vec{p}_t^1 & c_1^1 & \cdots & c_t^1 \\ \vdots & & & & & & & & \vdots \\ \vec{\imath}_1^s & \cdots & \vec{\imath}_t^s & \vec{p}_1^s & \cdots & \vec{p}_t^s & c_1^s & \cdots & c_t^s \end{pmatrix} . \begin{pmatrix} \vec{x}_1 \\ \vdots \\ \vec{x}_t \\ \vec{n}_1 \\ \vdots \\ \vec{n}_t \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

To build each row Θ_d, we scan the legal polytope $\mathcal{T}_d$, by successively instantiating values for each coefficient in a predefined order.[1] Fourier-Motzkin elimination — a.k.a. projection — [101] provides a representation of the affine constraints of a polytope suitable for its dynamic traversal. Computing the projection of all variables of a polytope $\mathcal{T}_d$ results in a set of constraints defining the same polytope,

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

but where it is guaranteed that for a point $v \in \mathcal{T}_d$, the value of the kth coordinate v_k *only depends* on $v_1, \ldots, v_{k-1}$, that is the affine inequalities involve only $v_1, \ldots, v_k$. Thus, the sequential order to build coefficients is simply the reverse order of the Fourier-Motzkin elimination steps. This scheme guarantees that provided a value in the projection of $v_1, \ldots, v_{k-1}$, a value exists for v_k, for all k.[2] In its basic form, the Fourier-Motzkin algorithm is known to generate many redundant constraints; these redundancies reduce its scalability on large polyhedra. Instead, we use our modified, redundancy-aware projection algorithm which is described in Chapter 4. In practice, this modified algorithm scales to hundreds of variables (schedule coefficients) in the original system. It is applied on each polytope $\mathcal{T}_d$ generated.

5.3 Related Work

The growing complexity of architectures became a challenge for compiler designers to achieve the peak performance for every program. In fact, the term *compiler optimization* is now biased since both compiler writers and users know that those program transformations can result in performance degradation in some scenarios that may be very difficult to understand [27, 18, 110]. Iterative compilation aims at selecting the best parameterization of the optimization chain, for a given program or for a given application domain. It typically affects optimization flags (switches), parameters (e.g., loop unrolling, tiling), phase ordering, the heuristic itself, or the hybridization of multiple heuristics [27, 18, 5, 71, 3, 86, 107, 22, 69]

This thesis studies a different search space: instead of relying on existing compiler options to transform the program, we statically construct a set of candidate program versions, considering the distinct result of numerous legal transformations in a particular class. Building an actual optimization phase out of this search space is much easier than from the composition of multiple search spaces arising from short-sighted, local transformations. Our method is also complementary to other forms of iterative optimization which address the orchestration of existing heuristics. Furthermore, it is completely independent from the compiler back-end.

In recent years, the benefits of iterative compilation have been widely reported [67, 32, 33, 58]. Iterative compilation is often able to find optimization sequences that outperform the highest optimization settings in commercial compilers. Kulkarni et al. [71] introduce the VISTA system, an interactive compilation system which concentrates on reducing the time to find good solutions. Another system that attempted to speedup iterative compilation was introduced by Cooper et al. called ACME [31]. Triantafyllis et al. [109] develop an alternative approach to reduce the total number of evaluations of a new program. Here the space of compiler options is examined off-line on a per function basis and the best performing ones are classified into a small tree of options.

Because iterative compilation relies on multiple, costly "runs" (including compilation and execution), the current emphasis is on improving the profiling cost of each program version [71, 48], or the total number of runs, using, e.g., genetic algorithms [70] or machine learning [3, 22]. Our heuristic is tuned to the rich mathematical properties of the underlying *polyhedral* model of the search space.Combining it with machine learning techniques seems promising and is the subject of our ongoing work.

Several researchers have also looked at using machine learning to construct heuristics that control a single optimization. Stephenson *et al.* [107] used genetic programming (GP) to tune heuristic priority functions for three compiler optimizations: hyperblock selection, register allocation, and data prefetching within the Trimaran's IMPACT compiler. Cavazos *et al.* [23] describe the use of supervised learning to

[2] The case of holes in $\mathbb{Z}$-polyhedra is handled through a schedule completion algorithm described in the next section.

control whether or not to apply instruction scheduling.

Iterative optimization has been used effectively on a variety of compilation and parallelization problems and its applicability and practicality has been demonstrated beyond the academic world [94]. Although multidimensional affine scheduling is an obvious target for iterative optimization, its profitability is one of the most difficult to assess, due to (1) the model's intrinsic expressiveness (the downside of its effectiveness) and (2) its lack of analytical models for the impact of transformations on the target architecture. Hence, related work has been very limited up to this point. To the best of our knowledge, Nisbet pioneered research in the area with one of the very first papers in iterative optimization. He developed the GAPS framework [87] which used a genetic algorithm to traverse a search space of affine schedules for automatic parallelization. In addition, Long and O'Boyle [80] considered a search space of transformation sequences represented in the UTF framework [63]. Both of these approaches suffer from under-constraining the search space by considering all possible schedules, including illegal ones. Downstream filtering approaches do not scale, due to the exponentially diminishing proportion of legal schedules with respect to the program size. For instance, Nisbet obtains only $3-5\%$ of legal schedules for the ADI benchmark (6 statements). Moreover, under-constraining the search space limits the possibility to narrow the search to the most promising subspaces.

Long et al. also tried to define a search space based on the Unified Transformation Framework [82, 81], targeting Java applications. Long's search space includes a potentially large number of redundant and/or illegal transformations, that need to be discarded after a legality check, and the fraction of distinct and legal transformations decreases exponentially to zero with the size of program to optimize. On the contrary, we show how to build and to take advantage of a search space which, by construction, contains no redundant and no illegal transformation.

The polyhedral model is a well studied, powerful mathematical framework to represent loop nests and their transformations, overcoming the limitations of classical, syntax-driven models. Many studies have tried to assess a predictive model characterizing the best transformation within this model, mostly to express parallelism [79, 42] or to improve locality [125, 37, 84]. However such models do not scratch the full complexity of the target architecture and the interference of the back-end compiler phases, because they abstract away most of the fine-grain architectural properties in their cost model for the selection of an affine transformation. This yields sub-optimal results even on simple kernels, as well as a poor performance portability on different architectures.

Chapter 6

Performance Distribution of Affine Schedules

In the previous chapter, we formally defined how to build a singular search space where each point corresponds to a distinct legal program version. We also adapted this space in such a way that a scan becomes possible in any case. In the following, we will actually traverse the search space to evaluate its potential for program optimization.

In Section 6.1 we focus on the exhaustive traversal of the search space for programs that accept a one-dimensional schedule. We highlight critical results about the performance distribution, and its connection with the compiler and the target machine. In Section 6.2 we provide an extensive study for the general case of multidimensional schedules. We provide and experimentally validate a powerful subspace decomposition of the search space, a mandatory step towards devising efficient traversal heuristics.

6.1 Scanning the Optimization Search Space

6.1.1 Experimental Setup

The experimental protocol is as follows. For each point of the search space, (1) generate the kernel code with CLooG[1] (2) add input initialization and measure tools, to produce a C compilable unit (3) Compile it provided a compiler and its optimization options (4) run the program on the target architecture and gather the results. In order to be consistent, the original code is included in this procedure starting at the second step.

We ran our experiments on an Intel workstation based on Xeon 3.2GHz, 16KB L1, 1024KB L2 caches. We used four different compilers: GCC 3.4.2, GCC 4.1.1, Intel ICC 9.0.1 and PathScale EKOPath 2.5. We used hardware counters to measure the number of cycles used by various programs. In order to avoid interferences with other programs and the system, we set the system scheduler policy to FIFO for every test. The kernel benchmark set is the one presented in Section 5.2.1. The time to compute the performance of each version in the space is connected to the execution time of each candidate. It can be a matter of a few minutes for matMult (912 points executing fast) or more than one hour for locality (thousands of points).

[1]CLooG version 0.14.0, with default options

6.1.2 Exhaustive Space Scanning

Because our search space is only based on legal schedules, the number of solutions for kernel benchmarks is small enough to make it possible to achieve an exhaustive search in a reasonable amount of time. Figure 6.1 and Figure 6.2 summarize our results. The Benchmark column states the input program; the Compiler column shows the compiler used to build each program version of the search space (GCC version was 4.1.1); the Options column gives the full compiler options; the Parameters column shows the values of the global parameters (e.g., for array sizes, parameters are chosen to exceed L2 cache size); the #Improved column shows the number of version that achieves a better performance than the original program (the total number of versions is shown in Figure 5.5); the ID best gives the unique "identifier" of the best solution; lastly, the Speedup column gives the speedup achieved by the best solution with respect to the original program performance. On average, one second is needed to explore a point (code generation, compilation and run of the target version).

The two main results shown by this figure are, first of all, that the best program version highly depends on both compiler and compiler options. Even considering the several "best" solutions, there is typically no intersection between the set of best transformations for two pairs compiler/compilation-options. Second, significant speedups are achieved, demonstrating the interest of the method for optimizing compilation. In few cases, a 0% speedup is achieved, meaning that the original code was already optimal for our experimental setup and model. On average, the method leads to a 35.4% speedup, or to 14.9% excluding the extreme results of matrix-multiply kernel which is known to be a good candidate for such study. A global observation is the correlation between observed speedups and locality improvements and / or transformations enabled in the back-end compiler by our program versions.

6.1.3 Intricacy of the Best Program Version

Another interesting result is the form of the best transformed programs since they typically appear to be quite complex. Most of the time, it was not possible to easily understand which part of the transformation sequence was responsible for the speedup since a significant part of the answer was related to the compiler design. We also noticed that optimization algorithms based on formal representations were sometimes far away from the optimal solution. A very simple but striking example is shown in Figure 6.3.

The simple, supposed optimal locality transformation in our class suggests a schedule of (i) for $S1$ and $(j+n)$ for $S2$ using the *Chunking* technique from Bastoul [13], which results in maximizing the reuse of the array a. The very best schedules were in fact $(i-j)$ and $(i+j-n+1)$ (the code generated by our framework is given in Figure 6.3). While the supposed optimal schedules generate a speedup of 147% with $n = 100$ and $m = 500k$ using GCC 3.4, the very best schedules generate a speedup of 398% (with a similar number of L1 and L2 cache-misses but a heavily reduced data TLB misses).

The relation with the compiler is described further in section 6.1.4. Section 6.1.5 deals with the effect of compiler options and lastly, we discuss the performance distribution in section 6.1.6.

6.1.4 The Compiler as an Element of the Target Platform

Our iterative optimization scheme is independent from the compiler and may be seen as a higher level to classical iterative compilation. In the same way as a given program transformation may better exploit a feature of a given processor, it also may enable more aggressive options of a given compiler. Because

Benchmark	Compiler	Options	Parameters	#Improved	ID best	Speedup
h264	PathCC	-Ofast	none	11	352	36.1%
	GCC	-O2		19	234	13.3%
	GCC	-O3		26	250	25.0%
	ICC	-O2		27	290	12.9%
	ICC	-fast		0	N/A	0%
fir	PathCC	-Ofast	N=50000	240	72	6.0%
	GCC	-O2		259	192	15.2%
	GCC	-O3		119	289	13.2%
	ICC	-O2		420	242	18.4%
	ICC	-fast		315	392	3.4%
fft	PathCC	-O2	N=256 M=256 O=8	21	267	7.2%
	GCC	-O2		10	285	0.9%
	GCC	-O3		11	289	1.8%
	ICC	-O2		17	260	6.9%
	ICC	-fast		20	112	6.4%
lu	PathCC	-Ofast	N=1000	100	224	6.5%
	GCC	-O2		321	339	1.6%
	GCC	-O3		330	337	3.9%
	ICC	-O2		281	770	9.0%
	ICC	-fast		262	869	8.7%
gauss	PathCC	-Ofast	N=150	212	4	3.1%
	GCC	-O2		204	2	1.7%
	GCC	-O3		52	2	0.01%
	ICC	-O2		63	288	0.05%
	ICC	-fast		15	39	0.03%
crout	PathCC	-Ofast	N=150	0	N/A	0%
	GCC	-O2		132	638	3.6%
	GCC	-O3		56	628	1.7%
	ICC	-O2		37	625	0.5%
	ICC	-fast		63	628	2.9%

Figure 6.1: Search space statistics for exhaustive scan (1/2)

production compilers have to generate a target code in any case in a reasonable amount of time, their optimizations are very fragile, i.e. a slight difference in the source code may enable or forbid a given optimization phase.

To study this behavior and estimate how a higher level iterative optimization scheme may lead to better performances, we performed a exhaustive scan of our search space for various programs and compilers with aggressive optimization options. We illustrate our results in Figure 6.2, studying the matrix multiplication kernel in more details in Figure 6.4 (this benchmark has been extensively studied, and is a typical target of aggressive optimizations of production compilers).

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

Figure 6.4 shows significant speedups achieved by the best transformations for each back-end com-

Benchmark	Compiler	Options	Parameters	#Improved	ID best	Speedup
`matmult`	PathCC	-Ofast	N=250	402	283	308.1%
	GCC	-O2		318	573	243.6%
	GCC	-O3		345	143	248.7%
	ICC	-O2		390	311	56.6%
	ICC	-fast		318	641	645.4%
`MVT`	PathCC	-Ofast	N=2000	5652	4934	27.4%
	GCC	-O2		3526	13301	18.0%
	GCC	-O3		3601	13320	21.2%
	ICC	-O2		5826	14093	24.0%
	ICC	-fast		5966	4879	29.1%
`locality`	PathCC	-Ofast	N=10000, M=2000	6069	5430	47.7%
	GCC	-O2		30	5494	19.0%
	GCC	-O3		589	4332	6.0%
	ICC	-O2		3269	2956	38.4%
	ICC	-fast		4614	3039	54.3%

Figure 6.2: Search space statistics for exhaustive scan (2/2)

piler. Such speedups are not uncommon when dealing with the matrix-multiplication kernel. The important point is that we do not perform any tiling on the input code (it requires multi-dimensional schedules and modification of the polyhedral representation), contrary to nearly all other works (see [126, 4] for useful references). Yet, we do not prevent the backend compiler applying itself further optimizations, which potentially includes tiling.

In general, it was possible to check using PathScale EKOPath that many optimization phases have been enabled or disabled, depending on the version generated from our exploration tool. The enabling transformation aspect of our method is brought to light with for instance the `h264` benchmark: the EKOPath compiler fuses 4 times in the original version but only once with the best found one, but was able to vectorize 3 times more with our transformation. Nevertheless it is technically hard to know precisely the contribution of the one-dimensional schedule (which has a high potential, by itself, as an optimizing transformation) with respect to the enabled compiler optimizations. In the `matmult` case, Interchanging loops on k and i is the core of the transformation embedded in all best schedules found. This drastically improves locality: for instance, with ICC -fast, the number of L1 and L2 cache-misses is comparable for the original code and the best found version, but the number of data TLB misses goes from 15M to 164k, diminishing with a similar ratio the number of floating point operations executed (the results are consistent whether the matrices are allocated with `malloc` or directly on the stack). This encourages the potential of a combination with tiling.

But more transformations are embedded in the schedules, and another striking result is the high variation of the best schedules depending on the compiler. For instance the lack of the j iterator in $\theta_{S1}(\vec{x}_{S1})$ for GCC or the lack of the n parameter $\theta_{S2}(\vec{x}_{S2})$ for ICC. These results, which are consistent with the other tested programs, emphasize the need of a transformation *specifically built for a given compiler* to achieve the best possible performance. One possible explanation is the difference between optimization phases in the different back-end compilers. Compilers have reached such a level of complexity that it is no longer possible to model the effects of downstream phases on upstream ones. Yet it is mandatory to rely on the downstream phases of a back-end compiler to achieve a decent performance, especially those which cannot be embedded naturally in the polyhedral model.

```
  S1(i): a[i] = i
  S2(i,j): b[j] = (b[j] - a[i]) / 2

Original code:
 for (i = 0; i <= M; i++) {
 | S1(i);
 | for (j = 0; j <= N; j++) {
 | | S2(i,j);
 | }
 }

Chunked code:
 for (t = 0; t <= M; t++) {
 | S1(t);
 }
 for (t = M; t <= M+N; t++) {
 | for (i = 0; i <= M; i++) {
 | | S2(i,t-M);
 | }
 }
```

```
best transformation:
 S1(0);
 for (t = -M+1; t <= 0; t++) {
 | for (i = max(0, t+M-N-1); i <= t+M-1; i++) {
 | | S2(i,t-i+M-1);
 | }
 | S1(t+M);
 }
 for (t = 1; t <= N+1; t++) {
 | for (i = max(t+M-N-1, 0); i <= M; i++) {
 | | S2(i,t-i+M-1);
 | }
 }
```

Figure 6.3: Intricacy of transformed code

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

Figure 6.4: Results for the `matmult` example

6.1.5 On the Influence of Compiler Options

Experiments have shown a relation between the best transformations and the compiler options. For instance, in the `matmult` kernel benchmark case with the ICC compiler used with the aggressive `-fast` option, the best transformation yields a 4.5% slowdown when it is compiled with `-O2` and compared to the best one found for this compiler option. This behavior was observed on all the tested programs. Finding the best compiler options is the subject of many research works in iterative compilation (see section 8.5). Studying this aspect is out of the scope of the present paper but those results are a sign that combining our method with existing iterative compilation techniques is a promising way.

6.1.6 Performance Distribution

Exhaustive scanning of all program versions is feasible on (small) kernels, and we can observe the complete performance distribution. Figure 6.5 shows this distribution for `matmult` and `locality` which are compiled with `GCC 4.1.1 -O2`. In Figure 6.6 the left `crout` is compiled with `ICC -fast`, and the second with `GCC 4.1.1 -O3`. Each graph represents the computation time of every point in the search space as a function of its number in the scanning order. The horizontal line shows the performance of the original program: every point below this line corresponds to a more efficient program version.

Although the scanning order may be a weird choice for such representation, it shows that the performance distribution is not totally random.[2]

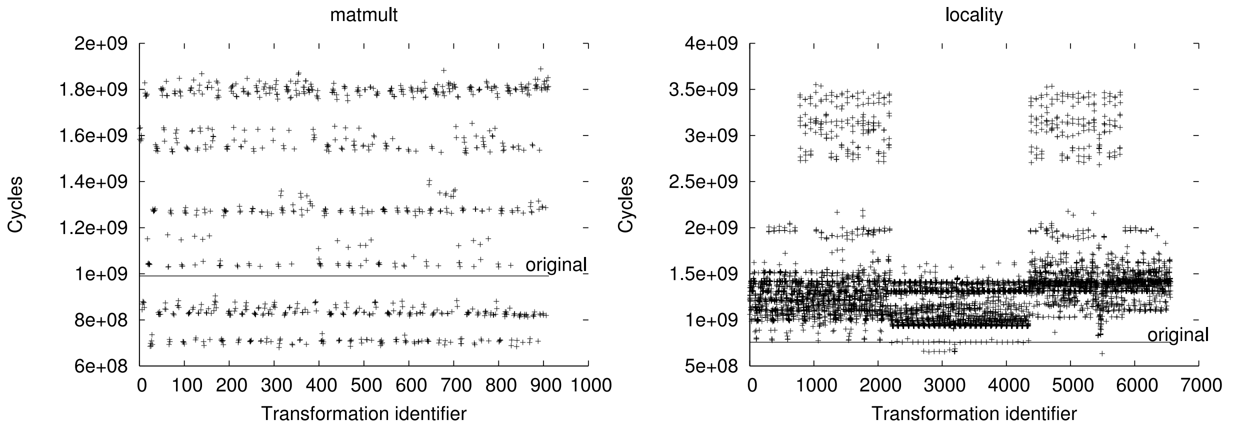


Figure 6.5: Performance distributions for `matmult` and `locality` with GCC -O3

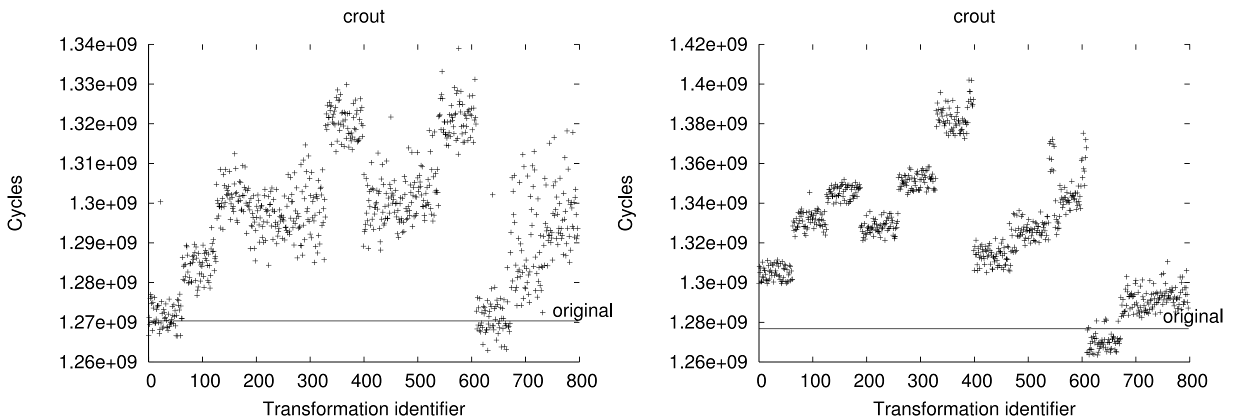


Figure 6.6: Performance distributions for `crout` with ICC -fast and GCC -O3

From these observations, we conclude that:

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

The impact of the compiler on the distribution is emphasized on the `crout` example, in Figure 6.6. Here we compare, for an identical original program (hence an identical optimization search space), the distribution on `ICC -fast` and `GCC 4.1.1 -O3` on the `crout` kernel benchmark. Hence, understanding performance regularities may help to find *hot* regions in the search space, thus avoiding useless runs in low-interest regions and diminishing-return searches among nearly optimal solutions.

[2]It is not an absurd ordering though: the scanning procedure could be seen as a very deep loop nest were the outer loop iterates on values of the first iterator coefficient of the first statement and the inner loop iterates on values of the constant coefficient of the last statement.

6.2 Extensive Study of Performance Distribution

The polyhedral representation of programs offers a compact way of representing arbitrarily complex sequences of transformations, significantly increasing the expressiveness of the search space. Moreover, the design of traversal methods for such spaces is facilitated by the algebraic properties of the model. For instance it is possible to consider only legal sequences, dramatically narrowing the search. We propose to go deeper and expose static characteristics of the space correlated to performance distribution. *Considering the search space constructed with the technique described in Chapter 5*, we extensively study the performance distribution of some representative benchmarks to assess the following hypotheses.

1. It is possible to statically order the impact on performance of transformation coefficients, that is, decompose the search space in subspaces where the performance variation is maximal or reduced.
2. The more a schedule dimension impacts a performance distribution, the more it is constrained.

As a result of this hypothesis, traversal techniques can be designed to focus on the most promising subspaces first, notably increasing the efficiency of the search method.

6.2.1 Experimental Protocol

For each tested point of the search space, we generate the corresponding C code with `CLooG` [10], add all the required instrumentation to the code, then compile and run it on the target machine. Our target architecture is an AMD Athlon X64 3700+ (single core), running at 2.4GHz (configured with 64KB+64KB L1 cache and 1024k L2 cache). The system is Mandriva Linux and the native compiler is GCC 4.1.2. All generated programs (as well as the original codes) were compiled using the following optimization settings, known to bring excellent performance for this platform: `-O3 -msse2 -ftree-vectorize`. The ACPI is not influential on this setting: the processor frequency is set to its maximum. The performance data are collected using hardware counters, using the PAPI library. We collected counters for cycles, L1 and L2 hits and misses, and branches taken and mispredicted. To limit OS interference to the minimum, all program versions are run with real-time priority scheduler and averaged over 100 executions.

6.2.2 Study of the `dct` Benchmark

The `dct` benchmark presented in Figure 6.7 computes a 32x32 Discrete Cosine Transform (M=32). This well known kernel is a good candidate for aggressive optimizations, and representative of several challenges for compilers. It is imperfectly nested, has 35 dependences and exposes possible multi-level fusion. Also, the `cos1` array can be reused, by means of a complex transformation sequence.

The `latnrm` benchmark presented in Figure 6.8 is a normalized lattice filter, and will be studied in Section 6.2.3.

Statistics of the search space for `dct`

The space of legal affine multidimensional schedules is built according to the algorithm presented in Section 5.2.2. This technique builds a search space where 3 sequential dimensions are necessary to respect the program dependences (the Θ matrix has 3 rows). Its statistics are summarized in Figure 6.9.

```
for (i = 0; i < M; i++) {
  for (j = 0; j < M; j++) {
    temp2d[i][j] = 0.0;
    for (k = 0; k < M; k++) {
      temp2d[i][j] += block[i][k] *
                      cos1[j][k];
    }
  }
}
for (i = 0; i < M; i++) {
  for (j = 0; j < M; j++) {
    sum = 0.0;
    for (k = 0; k < M; k++) {
      sum += cos1[i][k] * temp2d[k][j];
    }
    block[i][j] = ROUND(sum2);
  }
}
```

Figure 6.7: Source Code for `dct`

```
for (i = 0; i < M; i++) {
    top = data[i];
    for (j = 1; j < N; j++) {
        left = top;
        right = internal_state[j];
        internal_state[j] = bottom;
        top = coefficient[j-1] * left -
              coefficient[j] * right;
        bottom = coefficient[j-1] * right +
                 coefficient[j] * left;
     }
    internal_state[N] = bottom;
    internal_state[N+1] = top;
    sum = 0.0;
    for (j = 0; j < N; j++)
      sum += internal_state[j] *
             coefficient[j+N];
    outa[i] = sum;
  }
```

Figure 6.8: Source Code for `latnrm`

For each schedule dimension, we report the *degree of freedom* (that is, the number of *different legal schedules*) decomposed in 3 different classes. The $\vec{\imath}$ class represents all the schedules with a distinct $\vec{\imath}$ prefix (that is, where iterator coefficients are going to be different, typically distinct legal combinations of interchange, skewing, reversal); then respectively for the $\vec{\imath}+\vec{p}$ class (adding fusion, distribution); and the $\vec{\imath}+\vec{p}+c$ class (adding peeling, shifting). Finally, the size of the search space for the entire program is shown in the Total combined row, for each 3 classes (multiplying the degree of freedom for each schedule dimension).

Schedule dimension	$\vec{\imath}$	$\vec{\imath}+\vec{p}$	$\vec{\imath}+\vec{p}+c$
Dimension 1	39	66	471
Dimension 2	729	19683	531441
Dimension 3	60750	1006020	64855485
Total combined	1.7×10^9	1.3×10^{12}	1.6×10^{16}

Figure 6.9: Search Space Statistics for `dct`

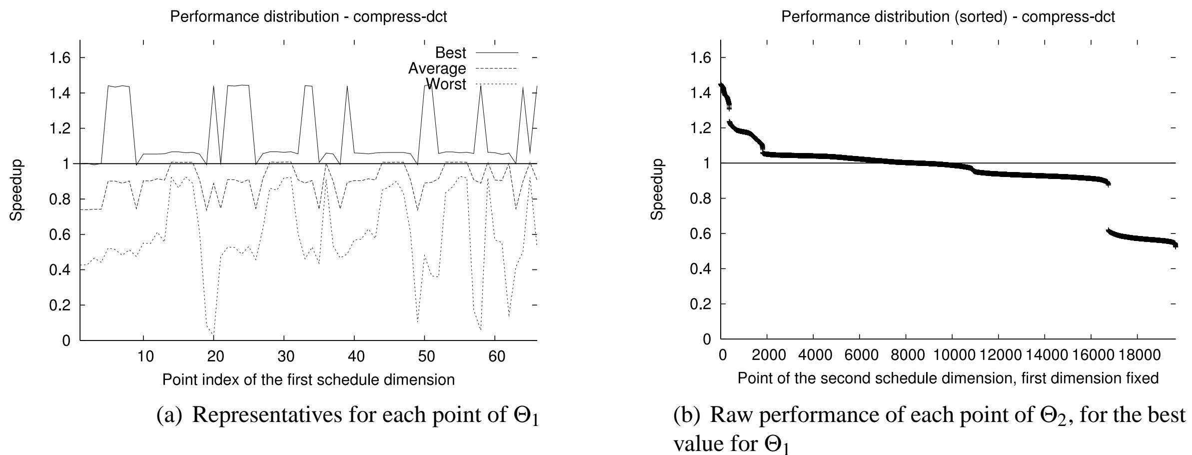
It is worth recalling that each program version corresponds to an arbitrarily complex sequence of transformations applied to the original program. It is possible to limit the degree of freedom to (a part of) the $\vec{\imath}$ or $\vec{\imath}+\vec{p}$ classes, by simply relying on our completion algorithm to find the minimal set of complementary transformations (contained in the larger classes) to make the current sequence legal. In this case we do not explore the numerous possibilities of making this sequence legal, but instead use the completion algorithm to generate only one of them.

Performance distribution

To limit the set of tested program versions, we rely upon two empirical observations. First, it is expected that the degree of freedom for peeling and shifting will have a low impact on the performance distribution, as shown in the previous section, hence we can safely limit the traversal to the $\vec{\imath}+\vec{p}$ class. Second, for the `dct` benchmark, it is expected that the third schedule dimension will have a low impact on performance: it will only affect the inner-most scheduling of two statements with a regular memory pattern, thus very

little improvement can be expected.[3] Eventually, we consider a search space of 1.29×10^6 different program versions, where each schedule coefficient that is not explored is computed with the completion algorithm.

Figure 6.10 shows the performance distribution of all versions generated for the `dct` program. Figure 6.10(a) plots the *best*, *worst*, and *average* performance for each of the 66 possible values for Θ_1 (represented in the x axis). For each of these values, we evaluated the 19683 possible values for Θ_2, and reported the performance. The performance of the original code is represented by the bold horizontal bar: each point above this bar improves the original code. Figure 6.10(b) plots the raw performance (sorted from the best to the worst) of all the 19683 points of Θ_2, using the value of Θ_1 of the best found version.



(a) Representatives for each point of Θ_1

(b) Raw performance of each point of Θ_2, for the best value for Θ_1

Figure 6.10: Performance Distribution for `dct`

The first observation is that an important speedup can be discovered: the best optimization achieves a speedup of 44.17%. Also, as what was pointed out for the case of one-dimensional schedules, several program versions achieve a similar performance.

The difficulty in reaching the best improving points in the search space is emphasized by their extremely low proportion: only 0.14% of points achieve at least 80% of the maximal speedup, while only 0.02% achieve 95% and more. Conversely, 61.11% degrade performance of the original code, while in total 10.88% degrade the performance by a factor 2 or more. Hence in this context it is expected that pure random approaches will fail to converge quickly to the best speedup.

We note that there are several values for the first schedule dimension from which it is impossible to attain the maximal performance. However, the maximal performance is attainable from more than one point in the first dimension. We conclude that effectively searching for points in Θ_1 is important in obtaining good performance, but cannot be the only criterion in designing search techniques when performing iterative optimization in the polyhedral representation.

[3]We performed sampling also in the $\vec{\imath}+\vec{p}+c$ class as well as for the third schedule dimension: it always confirmed these assumptions.

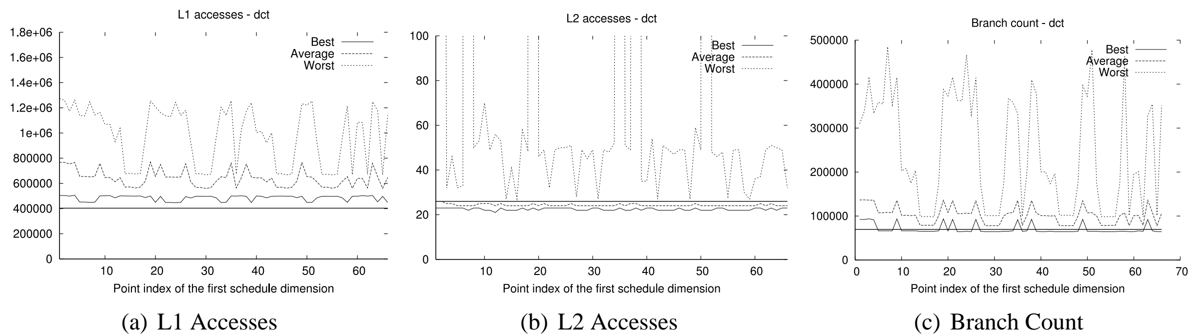
Statistical analysis

This section describes a finer grain analysis by capturing the relative impact of the schedules coefficients on the performance distribution. We first compute the variance of each schedule coefficient on the set of versions achieving at least 80% of the maximum speedup. Coefficients with little to no variance among points with good speedups mean that those coefficients are important in obtaining that good performance. We observe that 7 (out of 12) coefficients of the $\vec{\imath}$ class of Θ_1 have the same value, as well as 2 (out of 5) coefficients of the $\vec{p}$ class. Also, 3 (out of 12) coefficients of the $\vec{\imath}$ class of Θ_2 have a very low variance, emphasizing that the second dimension Θ_2 plays an important role in obtaining a good characterization of the performance distribution. The impact of the coefficients with low variance on the complete distribution shape is confirmed by correlating the performance of a program version with a non-optimal value of these coefficients. For example, we found that slight changes to any of the $\vec{\imath}$ low variance coefficients of Θ_1 translated into major performance variations.

We observe that a relevant ordering of the impact of the several classes of coefficients is $\vec{\imath} > \vec{p} > \vec{c}$, and studying the variance of the coefficients confirmed our first hypothesis stated in Section 6.2.

Hardware counters details

Figure 6.11 gives more details on source of performance improvements and degradations. It reports the behavior for the *L1 accesses*, *L2 accesses* and *Branch count* metrics.[4] The performance of the original code is represented by a bold horizontal bar.



(a) L1 Accesses (b) L2 Accesses (c) Branch Count

Figure 6.11: Hardware Counters Distribution for `dct`

The metric that seems to capture the performance distribution shape best is the *L1 accesses* curve. We observe that all transformations access the L1 cache more than the original code does. The transformed code performs at least 8% more L1 accesses than the original code. Also, the lowest L1 accesses points corresponds exactly to the peaks of highest speedup reported in Figure 6.10(a). On the other hand, several transformed program versions access the L2 cache less than the original code. Hence, the criterion in terms of memory accesses for optimal performance is to minimize L1 and L2 accesses. Note, we increase the number of L1 accesses as compared to the original code because there are more hits to the L1 cache, thereby minimizing L2 accesses. The last reported performance counter statistic is Branch count. We can correlate this statistic with the control statements added in the transformed code. The polyhedral code generation algorithms are likely to generated many complicated control statements (if and modulos)

[4] Accesses = hits + misses, count = taken + mispredicted

when highly complex transformations are applied. While not directly correlated to the performance distribution itself, this metric shows that the space contains many complicated versions, and in most cases a transformation sequence leads to more branches than the original code.

Discussion of the performance counter statistics

The best performing transformations reduce the numbers of stall cycles by a factor of 3, while improving the L2 hit/miss ratio by 10%. Transformation sequences achieving the optimal performance are not obvious at first glance: they involve complex combinations of skewing, reversal, distribution and index-set splitting. These transformations address specific performance anomalies of the loop nest, but they are often associated with the interplay of multiple architecture components. Overall, our results confirm the potential of iterative optimization to find program versions that better exploit the complex behavior of current superscalar processors. Also, we have extended iterative optimization to optimization problems far more complex than those commonly solved in adaptive compilation.

6.2.3 Evaluation of Highly Constrained Benchmarks

We established in the previous sections a connection between the $\vec{\imath}$ class and the dispersion of the performance distribution, on a representative benchmark offering a large degree of freedom for scheduling. In this section, we study the influence of a strong limitation of the degree of freedom of the $\vec{\imath}$ class. In particular, such a situation may derive from the greedy algorithm of Section 5.2.2 which tends to reduce the degree of freedom for the $\vec{\imath}$ class of the first dimension.

Search space statistics on more examples

In the following we focus on four representative benchmarks extracted from the UTDSP suite. Namely `latnrm`, a normalized lattice filter (shown in Figure 6.8); `fir`, Finite Impulse Response filter; `lmsfir`, a Least Mean Square adaptive FIR filter; and `iir`, an Infinite Impulse Response filter. Figure 6.12 shows the search space statistics for the first schedule dimension for the $\vec{\imath}$, $\vec{p}$, and c classes. We also report, for each benchmark, the number of statements (# St.), the number of dependences (# Deps.) and the number of schedule dimensions (# Dim.) needed to represent the program.

Benchmark	# St.	# Deps.	# Dim.	$\vec{\imath}$	$\vec{\imath}+\vec{p}$	$\vec{\imath}+\vec{p}+c$
`latnrm`	11	75	3	1	9	27
`fir`	4	36	2	1	9	18
`lmsfir`	9	112	2	1	9	27
`iir`	8	66	3	1	9	18

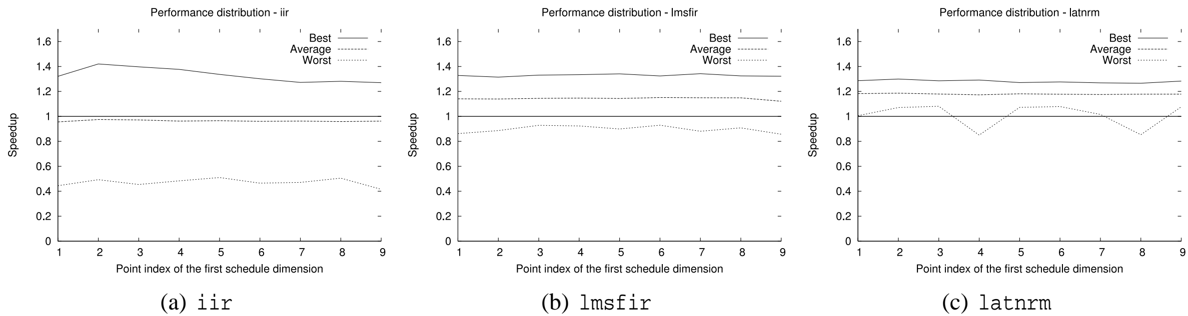
Figure 6.12: Search Space Statistics

We observe for each benchmark that the degree of freedom of the $\vec{\imath}$ class, for the first dimension, is null: there is only one sequence of interchange, reversal and skewing available for the first schedule dimension in the search space. This situation is not connected to usual program indicators such as number of statements or dependences, it is necessary to build the search space to detect this static property. Moreover, the four benchmarks we consider are syntactically different, and representative of many kernels in embedded computing.

We show in the following how this lack of degree of freedom translates into regularities of the performance distribution, and in performance improvements.

Performance distribution

We conducted for each of these benchmarks the same study as presented in Section 6.2.2. We exhaustively traverse the $\vec{\imath}+\vec{p}$ class, for the first two schedule dimensions. Figure 6.13 shows the performance distributions from this search.



(a) `iir` (b) `lmsfir` (c) `latnrm`

Figure 6.13: Performance Distribution for 3 UTDSP benchmarks

Again, we see there is significant speedup to be discovered: more than 30% speedup can be achieved for each of these benchmarks. Hence, for these benchmarks, the limited degree of freedom of the first schedule dimension does not restrict significant speedups.

The performance distribution is almost flat, another evidence of the impact of transformation coefficient freedom. We can conclude that the degree of freedom in the $\vec{\imath}$ class translates into variations in the performance distribution.

We also conducted variance studies to capture the relative impact of schedule coefficients. We observe that the impact on performance distribution of the $\vec{p}$ coefficients is lower than the $\vec{\imath}$ ones, while the impact of the c coefficients is almost negligible. Overall, all the conducted experiments confirm our initial hypothesis from Section 6.2.

6.2.4 Addressing the Generalization Issue

In order to assess the hypothesis that schedule coefficients can be ordered with respect to their impact on performance, we need to distinguish two different concerns: the generalization to other programs, and the generalization to other architectures.

Generalization to other programs We showed a positive correlation between the amount of variation in the performance distribution and the degree of freedom in the $\vec{\imath}$ class, especially for the first schedule dimension. Hence, it is expected that a traversal of each possible value for this class will be necessary in order to guarantee the maximal performance is achieved. Nevertheless, in the general case, particular fusions and distributions can achieve a dramatic impact on performance, and the full $\vec{\imath}+\vec{p}$ class is of interest for traversal. Finally, traversing the degree of freedom for peeling and shifting is almost useless, as the aim of those transformations usually is program legality and not program performance.

Generalization to other architectures Generalizing the results obtained on AMD Athlon64 to other architectures must be done with care. First, even if it is clear that the $\vec{\imath}$ class will still have a large impact on performance regardless of the architecture, one has to pay attention to fusion and distribution: which are important transformations for several embedded architectures. Hence, motivating the traversal of the degree of freedom offered by the $\vec{\imath}+\vec{p}$ class. We also conducted experiments on the ST231 embedded VLIW processor, though different from the AMD Athlon, we still observed a similar impact of the $\vec{\imath}$ class to the shape of the performance distribution. Although smaller speedups were found (the regular VLIW architecture is easier to model and well exploited by the STMicroelectronics compiler), our framework is still able to discover performance improvements on all tested benchmarks, with an average of 13.3%.

An important factor on modern and upcoming architectures is the influence of frequency scaling mechanisms such as the Advanced Configuration and Power Interface (ACPI). Considering for instance that only the processor frequency is decreased by the operating system, then the influence of locality effects may be reduced: the penalty to load a data element is reduced. In that context, an appropriate approach is to run the iterative process for a few representative configurations (e.g., maximal performance, average and minimal power). The same observations on the benefit of exploring first the $\vec{\imath}$ class then also the $\vec{\imath}+\vec{p}$ holds whatever the power configuration. As a best transformation is found for each use scenario, versioning is used in the generated code. However we believe that in the general case where performance is critical, limiting to the version found when the processor performance is maximal is satisfactory.

Performance distribution discussion From this study of the performance distribution of several programs, we deduce the following facts.

1. The degree of freedom in the $\vec{\imath}$ class of Θ_1, the first row of Θ, translates into variation in the performance distribution.
2. When the degree of freedom in the $\vec{\imath}$ class of Θ_1 is nonexistent, the performance distribution is almost flat.
3. The impact of coefficients on performance is ordered: Θ_1 impacts performance more than Θ_2, and inside a schedule row, $\vec{\imath}$ coefficients impact performance more than $\vec{p}$ and c.

Chapter 7

Efficient Dynamic Scanning of the Search Space

7.1 Introduction

Applying iterative optimization to the polyhedral model provides a significant breakthrough to the challenges of expressiveness and applicability. It enables searching in a space where every point is relevant: each point corresponds to a legal, distinct program version resulting in the application of an arbitrarily complex sequence of transformations. Since it is impractical to explore the whole search space on large benchmarks, we propose heuristics to enumerate only a high-potential subspace, using the properties of the polyhedral model to characterize the highest potential and narrowest one. We first present a heuristic adapted to the case of one-dimensional schedules in Section 7.2, which is able to discover the space-optimal point in our experiments without having to traverse the full set.

We then extend our approach for one-dimensional schedule search to the case of multidimensional schedules in Section 7.3, offering a feedback-driven iterative heuristic tailored to the search space properties of the polyhedral model. Although it quickly converges to good solutions for small kernels, larger benchmarks containing higher dimensional spaces are more challenging and our heuristic misses opportunities for significant performance improvement. Thus, we introduce in Section 7.4 the use of a genetic algorithm with specialized operators that leverage the polyhedral representation of program dependences. We provide experimental evidence that the genetic algorithm effectively traverses huge optimization spaces, achieving good performance improvements on large loop nests with complex memory accesses.

7.2 Heuristic Search for One-Dimensional Schedules

7.2.1 Decoupling Heuristic

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

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

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

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

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

The heuristic can be sketched in 5 steps.

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

7.2.2 Discussion

Figure 7.1 details a run of our decoupling heuristic (with a filtering level of 5% and a static limit of 10 points per coefficient type, see below), and compares it with a plain random search for three of our benchmarks. It shows the relative percentage of the best speedup achieved as a function of the number of iterative runs. The decoupling heuristic (the *DH* plot) yields much faster convergence, bringing to light the correlation between the speedup and the $\vec{\imath}$-coefficients. On these tested examples, one may achieve over 98% of the maximum speedup within less than 20 iterations.

On the other hand, we observed the heuristic behavior to be comparable to a full random driven approach (the *R* plot), as Figure 7.1 shows for the `matmult` kernel. Not surprisingly, as soon as the density of good transformations is large, a random space scan may converge faster than our enumeration-based method. For the `MVT` kernel, even if there is a large set of improved versions in the search space, the low density of good ones is emphasized by the poor convergence of the random-driven approach.

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

1. impose a static or dynamic limit to the number of runs, which should be coupled to an exploration

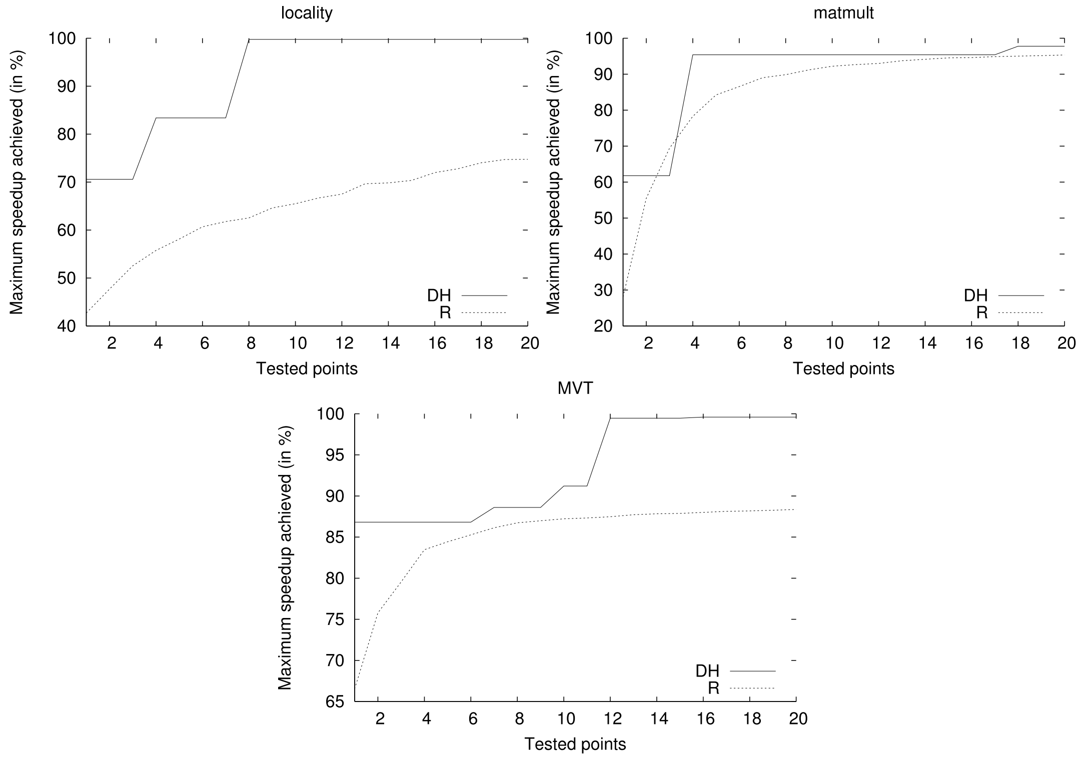


Figure 7.1: Comparison between the random and the decoupling heuristics

strategy starting with coefficients as close as possible to 0 (remember 0 may not correspond to any legal schedule);

2. replace an exhaustive enumeration of the $\vec{\imath}$ combinations by a limited set of random draws in the $\vec{\imath}$ space.

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

7.3 Heuristic Search for Multidimensional Schedules

7.3.1 Schedule Completion Algorithm

For SCoPs with more than 3 or 4 statements, the space construction algorithm leads to very large search spaces, challenging any traversal procedure. It is possible to focus the search on some coefficients of the schedule with maximal impact on performance, postponing the instantiation of a full schedule in a second heuristic step. We show that such a two-step procedure can be designed without breaking the fundamental legality property of the search space. This approach will be used extensively to simplify the optimization problem.

We rely on the previous projection pass to guarantee it is always possible to complete or even correct any point, by slightly modifying its coordinates, to make it lie within a given polytope $\mathcal{T}_d$. Our *completion*

algorithm is sketched in the following. Given a point v in a n-dimensional space with some coordinates which have been set (for instance, with a heuristic search procedure as presented below), and some other undefined coordinates:

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

Therefore it is possible to partially build a schedule prefix, e.g., values for the $\vec{\imath}$ coefficients, leaving the other coefficients undefined. Then, applying this completion algorithm will result in finding the minimal amount of complementary transformations to make the transformation lie in the computed legal space. The completion algorithm motivates the order of coefficients in the Θ matrix. We showed that the most performance impacting transformations (interchange, skewing, reversal) are embedded in the first coefficients of Θ — the $\vec{\imath}$ coefficients; followed by coefficients usually involved in fusion and distribution — the $\vec{p}$ coefficients; and finally the less impacting c coefficients, representing loop shifting and peeling. The completion algorithm finds complementary transformations in order of least to most impacting, as it will not alter any vector prefix if a legal vector suffix exists in the space.

Three fundamental properties are embedded in this completion algorithm:

1. if $v_1, \dots, v_k$ is a prefix of a legal point v, a completion is always found;
2. this completion will only update $v_{k+1}, \dots, v_{d_{\max}}$, if needed;
3. when $v_1, \dots, v_k$ are the $\vec{\imath}$ coefficients, the heuristic looks for the smallest absolute value for the $\vec{p}$ and constant coefficients, which corresponds to maximal (nested) loop fusion — relative to the $\vec{\imath}$ coefficients.

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

While it is possible to exhaustively traverse the constructed space of legal versions for small SCoPs, in the case of one-dimensional schedules, it becomes intractable in the multidimensional case. We have given earlier a preliminary answer by means of a heuristic to narrow this space and accelerate the traversal for the case of one-dimensional schedules. We build on this result to design a powerful heuristic suitable for the multidimensional case.

7.3.2 A Multidimensional Decoupling Heuristic

Our approach is called the *decoupling heuristic* as it leverages the completion algorithm of Section 7.3.1 to stage the exploration of large search spaces. It derives from the observation of the performance distribution from Chapter 6, where density patterns hinted that not all schedule coefficients have a significant

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

impact on performance. The principle of the decoupling heuristic for one-dimensional schedules is (1) to enumerate different values for the $\vec{\imath}$ coefficients, (2) to instantiate full schedules with the completion algorithm, and (3) to select the best completed schedules and further enumerate the different coefficients for the $\vec{p}$ part.

A direct extension to the multidimensional case exhibits two major drawbacks. First, the relative performance impact of the different schedule dimensions must be quantified. Second, an exhaustive enumeration of $\vec{\imath}$ coefficients for all dimensions is out of reach, as the number of points exponentially increases with the number of dimensions. Figure 5.8 illustrates this assertion by summarizing the size of the legal polytopes for different benchmarks, for all schedule dimensions.

Relations between schedule dimensions To extend the decoupling approach to multidimensional schedules, we need to integrate interactions between dimensions. For instance, to distribute the outer loop of a nest (which can improve locality and vectorization [4]), one can operate on the $\vec{p}$ and c parts of the schedule for the first dimension (a parametric shift). On the other hand, altering the $\vec{\imath}$ parts will lead to the most significant changes in the loop controls. Indeed, the largest performance variation is usually captured through the $\vec{\imath}$ parts, and a careful selection of those coefficients is mandatory to attain the best performance; conversely, it is likely that the best performing transformations will share similar $\vec{\imath}$ coefficients in their schedules.

Furthermore, the first dimension is highly constrained in general, since all dependences need to be — weakly or strongly — considered. Conversely, the last dimension is the least constrained and often carries only very few dependences.[2]

The decoupling heuristic in a nutshell We conducted an extensive experiment showing that Θ_1 (the first time dimension of the schedule) is a major discriminant of the overall performance distribution. Therefore, the heuristic starts with an exploration of the different legal values for the coefficients of Θ_1, and the completion algorithm is called to compute the remaining rows of Θ. Furthermore, this exploration is limited to the subspace associated with the $\vec{\imath}$ coefficients of Θ_1 (and the remaining coefficients of Θ_1 are also computed with the completion algorithm), except if this subspace is smaller than a given constant L_1 ($L_1 = 50$ in our experiments). L_1 drives the exhaustiveness of the procedure: the larger the degree of freedom, the slower the convergence. By limiting to the $\vec{\imath}$ class we target only the most performance impacting subspaces.

To enumerate points in the polytopes, we incrementally pick a dimension then *pick an integer* in the polyhedron's projection onto this dimension. Note that the full projection is computed once and for all by the Fourier-Motzkin algorithm presented in Section 5.2.3, *before* traversal. Technically, to enumerate integer points of the subspace composed of the first m columns of $\mathcal{T}_d$, we define the following recursive procedure to build a point v:

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

1. compute the lower bound lb and the upper bound ub of v_k in $\mathcal{T}_d$, given the coordinate values for $v_1 \dots v_{k-1}$;

2. for each $x \in [lb, ub]$:

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

(a) set $v_k = x$,

(b) if $k < m$ call EXPLORE $(v, k+1, \mathcal{T}_d)$ else output v.

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

Then, the heuristic selects the x% best values for Θ_1 ($x = 5$% in our experiments), it proceeds with the exploration of values for coefficients of Θ_2 with the selected values of Θ_1, and recursively until the last but one dimension of the schedule. The last dimension (corresponding to the innermost nesting depth in the generated code) is not traversed, but completed with a single value: exploring it would yield a huge number of iterations, with limited impact on the generated code, and negligible impact on performance. Eventually, the exploration is bounded with a static limit (1000 evaluations in our experiments).

7.3.3 Experiments on AMD Athlon

Figure 7.3 shows the results for `dct` along with seven other kernels from the UTDSP suite that were amenable to the polyhedral representation without code modification. Note that here `matmult` is a 2 statement matrix multiplication, for 10×10 matrices. See Chapter 6 for an extensive study of this kernel on larger dataset sizes.

We report the number of statements (# Stm.), the size of the $\vec{i}$ class for the first schedule dimension, the size of the search space considered (Space), the run number at which the best performing version was found (Id Best: the lower, the earlier), and the speedup achieved (Speedup). The overhead of picking a point in the search space and building its syntactic representation is negligible in comparison to the execution time of the program version, and several points can be tested within a second. Finally, the procedure is fully automated.

Similarely as in Chapter 6, our target architecture is an AMD Athlon X64 3700+ (single core), running at 2.4GHz (configured with 64KB+64KB L1 cache and 1024k L2 cache). The system is Mandriva Linux and the native compiler is GCC 4.1.2. All generated programs (as well as the original codes) were compiled using: `-O3 -msse2 -ftree-vectorize`.

	dct	matmult	lpc	edge-c2d	iir	fir	lmsfir	latnrm
#Inst.	5	2	12	3	8	4	9	11
i	39	76	243	1	1	1	1	1
Space	1.6×10^{16}	912	$> 10^{25}$	5.6×10^{15}	$> 10^{19}$	9.5×10^{7}	2.8×10^{8}	$> 10^{22}$
Id Best	46	16	489	11	34	33	51	6
Speedup	57.1%	42.87%	31.15%	5.58%	37.50%	40.24%	30.98%	15.11%

Figure 7.2: Heuristic Performance for AMD Athlon

The heuristic succeeds in discovering an average speedup of 32.56% for the 8 tested benchmarks. All the best versions are the result of the application of a complex transformation sequence, syntactically very different from the original code. Analysis of the performance counters for these transformations shows improvements in memory behavior, combined with a better workload of the processor units which is likely to be the result of hard to predict interaction between the compiler optimizations and the processor features.

The limited performance improvement for `edge` is directly correlated to the code structure: this benchmark performs a convolution of a 3x3 kernel, and is an excellent candidate for optimization with

loop unrolling — a transformation not embedded in our search space. Our technique is fully compatible with other iterative search techniques such as parameters tuning [3], and it is expected that this combination would bring excellent performance.

For the case of highly constrained benchmarks, we also specifically studied the performance of a single statically computed schedule, corresponding to applying the completion heuristic on a fully uninitialized schedule. For this class of benchmarks, this schedule performs very well, and succeeds in discovering 75%–99% of the maximum speedup available in the space. We recall that this schedule is computed in a deterministic fashion from the space $\mathcal{T}$, by using the completion algorithm on all coefficients. Hence this schedule is generated *without any performance evaluation* of other schedules. The proposed heuristic can be coupled with the detection of the special case where the size of the $\vec{\imath}$ class on the first dimension is 1,to avoid traversing a space leaving little room for further improvements (as it is expected that the performance distribution will be almost flat). This approach leads to a an average 17.8% speedup on the 5 benchmarks where this criteria applies, without any further evaluation required.

7.3.4 Extensive Experimental Results

We now consider three target architectures. The AMD Alchemy Au1500 is an embedded SoC with a MIPS32 core (Au1) running at 500MHz. We used GCC 3.2.1 with the -O3 flag (version of GCC and option with peak performance numbers, according to the manufacturer). The STMicroelectronics ST231 is an embedded SoC with a 4-issue VLIW core running at 400MHz and a blocking cache. We used st200cc 1.9.0B (Open64) with the flags `-O3 -mauto-prefetch -OPT:restrict`. The AMD Athlon X64 3700+ has a 1MB L2 cache and runs at 2.4GHz. It runs Mandriva Linux and the native compiler is GCC 4.1.1. We used the following optimization flags for this platform which are known to bring excellent performance: `-O3 -msse2 -ftree-vectorize` . For this particular machine, hardware counters were used to collect fine-grained cycle counts, and we used a real-time priority scheduler to minimize OS interference. We used the average of 10 runs for all performance evaluations.

We implemented an instancewise dependence analysis, the construction of the space of legal transformations, and the efficient scanning algorithms introduced in this thesis.[3] We used free software such as `PipLib` [39, 122] (a polyhedral library and parametric integer linear programming solver) and `CLooG` [10] (an efficient code generator for the polyhedral model). For each tested point in the search space, (1) we generated the kernel C code with `CLooG`,[4] (2) then we integrated this kernel in the original benchmark along with instrumentation to measure running time (we use performance counters when available), (3) we compiled this code with the native compiler and appropriate options, (4) and finally ran the program on the target architecture and gathered performance results. The original code is included in this procedure starting at the second step, for appropriate performance comparison. The full iterative compilation and execution process takes a few seconds using our heuristic, and up to a few minutes using the GA described in Section 7.4 for the largest benchmark (up to 1000 tested versions). The time to compute the legal space and to generate points is negligible with respect to the total running time of the tested versions.

Results Figure 7.3 shows the results for the three architectures we considered. We report the total numbers of tested versions (*Tested*), the run index of the best performing version (*Id Best*; the lower,

[3] LETSEE, the LEgal Transformation SpacE Explorator, available at http://letsee.sourceforge.net.

[4] We use CLooG version 0.14.0 with default options.

		compress-dct	edge	iir	fir	lmsfir	matmult
AMD Athlon	Tested	480	243	1000	77	1000	81
	Id. Best	**19**	**11**	**34**	**33**	**51**	**16**
	Perf. Imp.	37.11%	5.58%	37.50%	40.24%	30.98%	42.87%
ST231	Tested	480	243	1000	77	1000	81
	Id. Best	**39**	**12**	**6**	**2**	**9**	**16**
	Perf. Imp.	15.11%	3.10%	24.91%	17.96%	10.17%	17.91%
Au1500	Tested	480	243	1000	77	1000	81
	Id. Best	**30**	**17**	**38**	**27**	**11**	**17**
	Perf. Imp.	22.37%	2.51%	3.12%	14.00%	15.80%	20.18%

		latnrm	lpc	ludcmp	radar	Average
AMD Athlon	Tested	1000	1000	1000	1000	
	Id. Best	6	489	37	405	
	Perf. Imp.	15.11%	31.15%	4.50%	6.42%	25.14%
ST231	Tested	1000	1000	1000	1000	
	Id. Best	13	158	391	709	
	Perf. Imp.	2.61%	1.99%	6.33%	4.12%	10.42%
Au1500	Tested	1000	1000	1000	1000	
	Id. Best	43	82	175	454	
	Perf. Imp.	15.19%	14.08%	3.66%	3.39%	11.43%

Figure 7.3: Results of the decoupling heuristic for AMD Athlon, ST231 and Au1500

the earlier), and the performance improvement of execution time in percentage (*Perf. Imp.*). We also imposed a static limit of evaluating 1000 data points in the search space.[5]

Discussion Optimizing static control parts makes the optimization insensible to the dataset, because the control flow is not changed whatever the data. However, iterative search is sensitive to the dataset *size*. To guarantee optimality, one should perform the search for several values of the size of the dataset, because there is no guarantee that the best transformation is identical when the dataset size changes. Here, all UTDSP experiments use the reference dataset size. Increasing data size would emphasize locality effects, yielding actually an even better performance improvements. E.g., `matmult` on Athlon with $n = 250$ yields 361% performance improvement, $n = 64$ yields 318% improvement, whereas the reference value $n = 10$ yields 43% improvement. However, changing the dataset size may theoretically also require to search again for the best transformation. This would lead to generate different optimized versions of the code for different dataset size, as most of the auto-tuning libraries such as ATLAS do.

Our results show significant improvement on all kernels of the UTDSP suite. In addition, about 50 runs were sufficient for kernels with less than 10 statements (all but `lpc` and `radar`).

For all benchmarks, the best program version is syntactically very far from the original one.

A good illustration of this is given for the `Ring-Roberts` running example, which achieves a 47% performance improvement on a full HD image on AMD Athlon; hardware counter details show a 54% reduction of the L1 hit/miss ratio and a 51% of the data TLB misses. This complex transformation is the result of multidimensional shifting and peeling of the iterations preventing fusion, and the complete fusion of the remaining iterations.

We also noticed that performance improvements are often the result of indirect enabling of back-end compiler optimizations (e.g., vectorization or scalar promotion), in addition to the direct impact on hardware components (e.g., locality). Modern compiler optimization heuristics are still fragile, and the interactions between optimization phases are not captured in their design. Predicting this interaction on

[5]This matches the maximum number of versions considered by the genetic algorithm in Section 7.4.

non-trivial codes is still out of reach, and slight syntactic differences can trigger different optimization results. Testing different source code having the same semantics is one way to circumvent the compiler's optimization unpredictability.

In addition, the best iteratively found transformation for a given benchmark is different when considering a different target architecture. This is due to different interactions with the compiler, as well as different architectural features to optimize for. Note that it is not a consequence of working with more expressive schedules: we already highlighted a similar pattern for the case of one-dimensional schedules. It confirms the complexity of the optimization problem and the relevance of a feedback-directed approach.

The heuristic heavily relies on the observation that the first dimension of the schedule contains very few points — it traverses this dimension exhaustively. However, exhaustive enumeration is only possible for small kernels, such as most UTDSP benchmarks. Unfortunately, for larger programs like `lpc`, `ludcmp`, `radar`, and to some extent on `latnrm`, this approach does not scale.

To address this scalability issue, we substitute the exhaustive search with a traversal driven by a genetic algorithm.

7.4 Evolutionary Traversal of the Polytope

This section introduces novel genetic operators tailored to the traversal of polytopes of legal affine schedules.

Genetic algorithms (GA) [52] are known for their genericity: we chose an evolutionary approach because of the natural encoding of the geometric properties of the search space into crossover and mutation operators. The two main properties are the following:

1. to enforce legality and uniqueness of the program versions, the genetic operators must be closed on the search space polytope; we construct dedicated mutation and crossover operators satisfying this property;
2. unlike random search, the traversal is characterized by its non-uniformity (from the initial population and the crossovers); this is utterly important as the largest part of the search space is generally plagued with poor or similar performing versions.

Genetic algorithms have often been used in program optimization. Our contribution is to reconcile fine-grain control of transformation heuristics — as opposed to optimization flag or pass selection [109, 3] — with the guaranteed legality of the transformed program — as opposed to filtering approaches [87, 80, 81] or always-correct transformations [107, 71].

7.4.1 Genetic Operators Design

Using classical GA operators would not be an efficient way to generate data points in our search space. This is because legal schedules lie in affine bounds that are strongly constrained and changing them at random has a very low probability of preserving legality. Moreover, in general, this probability decreases exponentially with the space dimension [87]. We thus need to understand the properties of the space of legal schedules, and to embed them into dedicated GA operators.

Some properties of affine schedules The construction algorithm outputs one polytope per schedule dimension. We can deduce numerous properties on these polytopes, either deriving from the construction algorithm or from affine scheduling itself. In the following, the term *affine constraint* refers to any dependence, iteration domain, or search bound constraint on coefficients of the schedule.

1. No affine constraint involves coefficients from different rows of Θ, since those coefficients are computed from distinct polytopes. Of course, multiple coefficients inside a row can be involved in a constraint.

2. Multiple coefficients involved in a constraint are called *dependent*. Each row can be partitioned into *classes of dependent coefficients*, where no constraint involves coefficients from different classes. For example, in the polyhedron defined by $\{x_1 + x_2 \geq 0 \wedge x_3 \geq 0\}$ we say that the set $\{x_1, x_2\}$ is independent from the set $\{x_3\}$. Legality preservation is local to each class of dependent coefficients.

We design novel genetic operators exploiting and preserving these properties.

Initialization We first introduce an individual with a statically computed schedule, built by applying the completion algorithm on a fully-undefined schedule. This choice shares its motivation with the decoupling heuristic in Section 7.3.2.

The rest of the population is initialized by performing aggressive mutations on this static schedule; we generate 30 to 100 individuals, depending on the space dimension. The initial population is heavily biased towards a particular subspace (typically the subspace of the $\vec{\imath}$ coefficients), emphasizing the non-uniformity of the traversal.

Mutation The mutation operator starts with the computation of a probability distribution for the modification of each schedule coefficient. This probability is driven by three factors; the first one derives directly from the heuristic of the one-dimensional case:

- coefficients of the iteration vectors have a dramatic impact on the structure of the generated code; minor modifications trigger wild jumps in the search space;

- coefficients with few linear dependences with others may require more mutations to trigger significant changes, e.g., modifying their value will not require updating many other coefficients to make the point legal;

- lower dimensions and especially the scalar ones usually have a lower impact on performance.

In addition, we weigh the probabilities with a uniform annealing factor, to tune the aggressiveness of the mutation operator along with the maturation of the population.

We randomly pick a value *within the legal bounds for this coefficient*, and according to the distribution of probabilities. As this mutation may cause other coefficients to become incorrect, we then update the schedule with the completion algorithm depicted in Section 7.3.1; it is a simple update because the schedule prefix can be kept in the legal space, computing mutated coefficients in the reverse order of Fourier-Motzkin elimination.

We also experimented with a simpler mutation operator, where the bounds to pick mutated values where not adjusted to the corresponding polytope of legal versions, applying our correction mechanism a posteriori. This approach did not prove very effective as coefficients are often correlated or severely constrained: randomly picking values for multiple correlated coefficients often leads to identical schedules after correction. Only an incremental application of the correction mechanism avoids the generation of many duplicates (which strongly degrade the effectiveness of the mutation operator).

Crossover We propose two operators. The *row* crossover aims to compensate the row-wise scope of the mutation operator. Given two individuals represented by Θ and Θ', the row crossover operator randomly picks rows of either Θ or Θ' to build a new individual Θ''. This operator obviously preserves legality since there are no dependences between rows. Since the mutation operates within a schedule dimension, it may succeed in finding good candidates for a given row of Θ or Θ', but may mix these with ineffective rows. Combining these rows may lead to a good schedule, with a much higher probability than with mutation alone.

The *column* crossover is dedicated to crossing independent classes of schedule coefficients (represented by sets of columns not connected by any affine constraint); this operator is quite original and specific to the geometrical properties of the search space. It can be seen as a finer-grained crossover operator. From two individuals Θ and Θ', it randomly selects an independent class from either parent — at every dimension — to build Θ''. When there is only one independent class in a given schedule dimension this operator behaves like the row crossover. This situation is more likely to occur for outer schedule dimension(s). But inner schedule dimensions are less constrained, because numerous dependences have been solved at previous level. Hence, the probability of having independent statements (that is, independent classes of schedule coefficients) increases with the scheduling level.

We rely on the geometric properties of the polytope to compute classes of dependent coefficients for a given schedule dimension. These classes are computed immediately after the search space polytope is built, and do not change during traversal. This operator preserves legality as it only modifies independent sets of schedule coefficients. Dependences constrain schedule coefficients in pairs of statements. Several transitive steps are needed to characterize all correlations between coefficients in a dependent class. This operator carefully refines the grain of schedule transformations, while preserving legality.

Selection The selection process uses the best half of the current population for the next generation. Mutation and crossover are applied on these individuals to generate a new full population. Instead of considering only running time, a better option might be to combine multiple metrics, including performance predictors (to avoid running the code) or multiple hardware counters.

7.4.2 Experimental Results

Figure 7.4 summarizes the results of the genetic algorithm applied to all benchmarks for the three architectures presented in the previous section, with the same experimental setup. Row Heuristic/GA shows the fraction of the performance improvement achieved by the decoupling heuristic w.r.t. the genetic algorithm, and fractions are averaged for the benchmarks of less than 10 statements versus more than 10. We initialized the population with 30 to 100 individuals, and performed at most 10 generations; therefore, the maximum number of runs for each program was 1000.

Comparing these results with the table in Figure 7.3 shows the efficiency and scalability of our

Architecture	compress-dct	edge	iir	fir	lmsfir	matmult
AMD Athlon	44.17%	7.86%	32.18%	40.70%	24.23%	42.87%
Heuristic/GA	84.31%	82.26%	93.58%	98.86%	80.71%	100%
ST231	18.42%	3.29%	27.40%	18.81%	8.63%	17.91%
Heuristic/GA	83.33%	94.52%	90.91%	95.48%	85.14%	100%
AMD Au1500	25.11%	3.03%	4.07%	14.10%	19.18%	22.67%
Heuristic/GA	89.21%	82.83%	78.29%	99.50%	81.64%	88.93%

Architecture	latnrm	lpc	ludcmp	radar	Average
AMD Athlon	28.23%	45.84%	69.63%	40.18%	37.58%
Heuristic/GA	**53.57%**	**67.68%**	**6.52%**	**16.05%**	89.95% / **35.95%**
ST231	0.86%	3.44%	5.96%	28.32%	13.30%
Heuristic/GA	**92.30%**	**32.20%**	**22.26%**	**30.82%**	91.56% / **44.39%**
AMD Au1500	27.01%	17.43%	15.71%	30.87%	17.91%
Heuristic/GA	**55.55%**	**82.35%**	**16.56%**	**10.91%**	86.73% / **41.35%**

Figure 7.4: Results of the Genetic Algorithm. The decoupling heuristics succeeds in discovering 78-100% of the performance improvement achieved by GA for all benchmarks of less than 10 statements. For larger benchmarks, the GA performs 2.46× better in average, and up to 16× better.

method. The genetic algorithm (GA) achieves good performance improvements for the larger kernels; these improvements are much better than those of the decoupling heuristic for the larger benchmarks. On the other hand, the decoupling heuristic exposes 78–100% of the improvement obtained with GA within the first 50 runs, for all kernels of less than 10 statements

Results are better on AMD Athlon than on embedded processors, probably because the architecture is more complex: a good interaction between architectural components is harder to achieve and brings higher improvements. Conversely, the ST231 and AMD Au1500 have a predictable behaviour, more effectively harnessed by the back-end compiler, and showing less room for improvement; yet our results are still significant for such targets.

We report a detailed study of the representative `compress-dct` benchmark, on AMD Athlon. Figure 7.5 summarizes the results, and confirms the huge advantage of the GA given the statistically sparse and chaotic occurrence of performance-enhancing schedules. Figure 7.5(a) shows the convergence of our GA approach versus a Random traversal in the space of legal schedules (only legal points are drawn). The GA algorithm ran for 10 generations from an initial population of 50 individuals. Both plots are an average of 100 complete runs. Figure 7.5(b) reports the performance distribution of the legal space. We exhaustively enumerate and evaluate all points with a distinct value for the $\vec{\imath} + \vec{p}$ coefficients of the first schedule dimension, combined with all points with a distinct $\vec{\imath}$ value for the second one; a total of 1.29×10^6 schedules are evaluated. For each distinct value of the first schedule dimension (plotted in the horizontal axis), we report the performance of the Best schedule, the Worst one, and the Average for all tested values of the second schedule dimension. Figure 7.5(c) shows the performance distribution for all tested points of the second schedule dimension, provided a single value for the first one, sorted from the best performing one to the worst (the best performing schedule belongs to this chart).

Figure 7.5(a) shows that our GA converges much faster than random search. Random search achieves only 18% performance improvement, after 500 runs, while the GA takes only 120 runs to match this result. The GA converges towards 44.1% performance improvement after 350 runs, at the 7$^{\text{th}}$ generation, before the imposed limit of 10 generations. This the maximum performance improvement available, as shown by the exhaustive search experiments in Figure 7.5(b). The effectiveness of the genetic operators

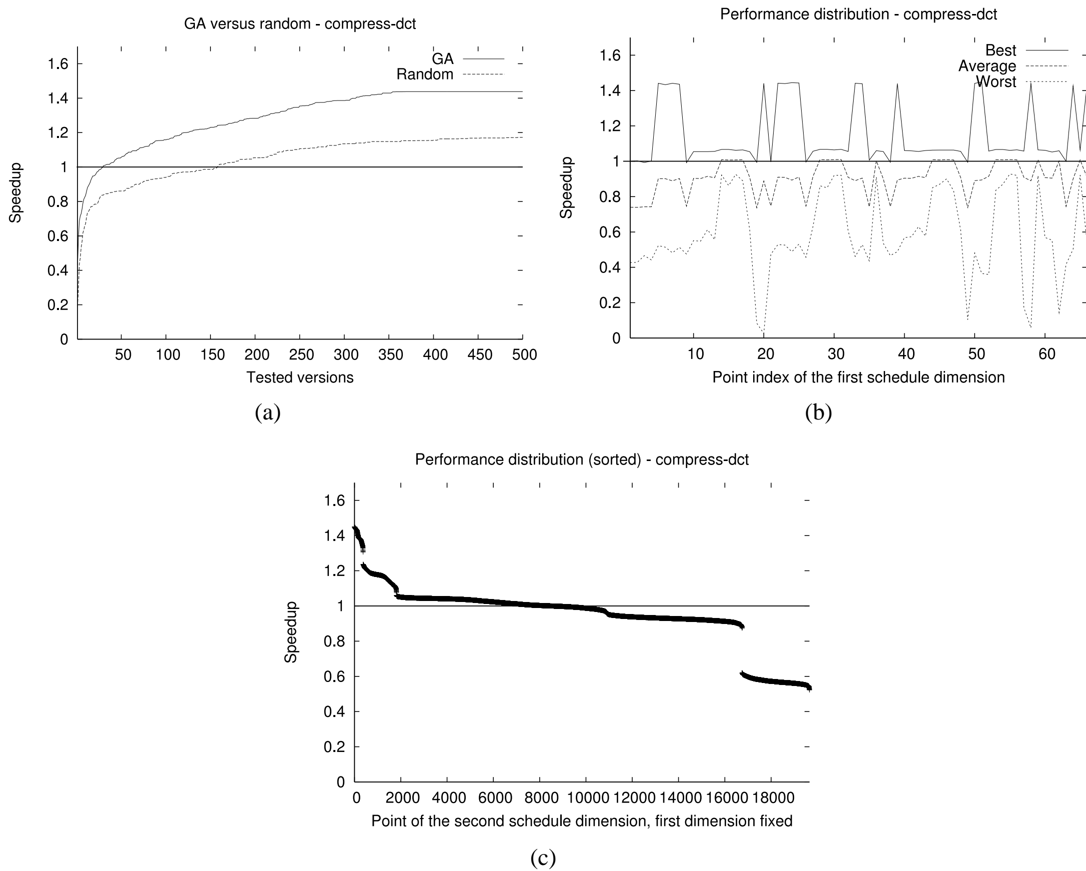


Figure 7.5: Performance Distribution of `compress-dct`, AMD Athlon. GA discovers the maximum performance improvement available in the search space.

is illustrated by the lack of correlation of the performance improvements and the actual performance distribution. Conversely, random traversal follows the shape of the performance distribution, and on average is not able to reach the best performing schedules — as their density in the space is very low. The difficulty to reach the best points in the search space is emphasized by their extremely low proportion: only 0.14% of points achieve at least 80% of the maximal performance improvement, while only 0.02% achieve 95% and more, as observed in Chapter 6.

Finally, we studied the behavior of multiple schedules for the `compress-dct` benchmark, analyzing hardware counters on Athlon. This study highlights complex interactions between the memory hierarchy (both L1 and L2 accesses are minimized to achieve good performance), vectorization, and the activity of functional units. The best performing transformation reduces the numbers of stall cycles by a factor of 3, while improving the L2 hit/miss ratio by 10%. Transformation sequences achieving the optimal performance are opaque at first glance: they involve complex combinations of skewing, reversal, distribution and index-set splitting. These transformations address specific performance anomalies of the loop nest, but they are often associated with the interplay of multiple architecture components. Moreover, we observe that the best optimizations are usually associated with more complex control flow than the original code. The number of dynamic branches is increased in most cases, although stall cycles are heavily reduced due to locality and ILP improvements.

Our results confirm the potential of iterative optimization to accurately capture the complex behavior of the processor and compiler, and extends its applicability to optimization problems far more complex than those typically solved in adaptive compilation.

Chapter 8

Iterative Selection of Multidimensional Interleavings

8.1 Introduction

Selecting the appropriate combination of loop fusion, loop distribution and code motion is a key performance factor, and a highly target-specific problem — the best combination varies across architectures. Yet this selection is very hard, in terms of expressiveness, semantics preservation, and profitability. Worse, deciding which sequence of enabling transformations — required for semantics preservation — to obtain the selected combination of loop fusion and distribution is a hard combinatorial problem.

For the first time, we address this fundamental challenge in its most general setting, offering to the optimizer the choice of selecting fusions and distributions among a space of all, distinct and semantics-preserving transformations. We propose a level-by-level decomposition of the problem to exhibit necessary and sufficient conditions. Compared to the state-of-the-art in loop fusion, we consider arbitrarily complex sequences of enabling transformations, in the multidimensional case. This generalization of loop fusion, called *fusability*, results in a dramatic broadening of the expressiveness (hence of expected effectiveness) of the optimizer.

To make the characterization of semantics-preserving transformations tractable, we have introduced the first affine encoding of all statement interleavings at a given loop level in Chapter 3. We now demonstrate key properties about fusability of loops at the statement level, and study the transitivity of this relation along with enabling transformations. We first state in Section 8.2 the optimization problem in the polyhedral framework by giving a concrete example for the need of different loop fusion / distribution choices for different target architectures. In Section 8.3 we present key results to reduce the problem of deciding the fusability of statements to the existence of compatible pairwise loop permutations. We further improve the tractability through the design of an objective function to predict complex sequences of enabling transformations for fusion, that include coarse-grain parallelism and tiling. Finally, we present a complete and tractable optimization algorithm to select profitable interleavings, before its experimental evaluation in Section 8.4. Our experiments demonstrate the effectiveness of this approach, both in obtaining solid performance improvements over existing auto-parallelizing compilers, and in achieving portability of performance on various modern multi-core architectures.

8.2 Problem Statement

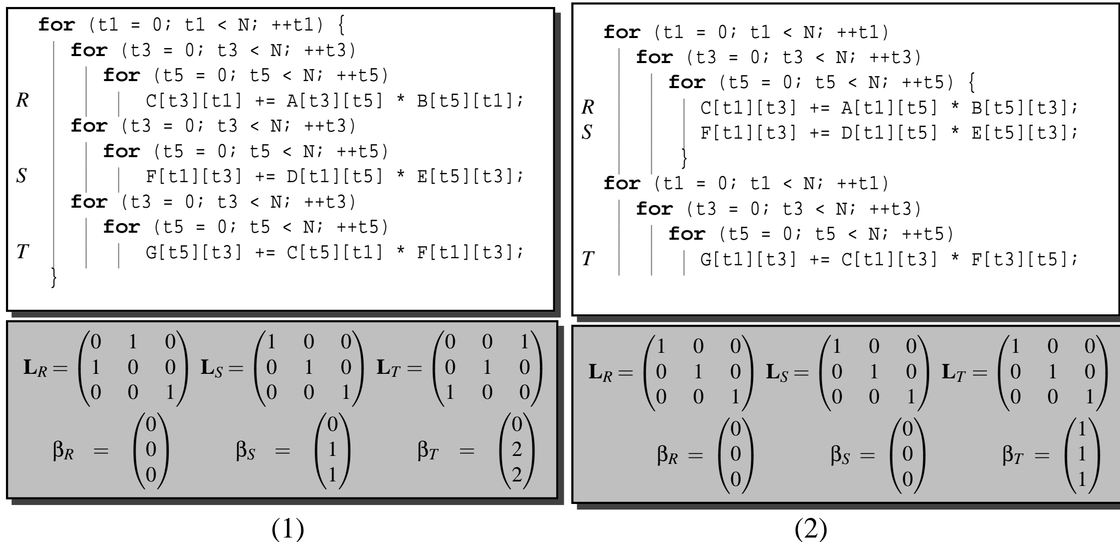
8.2.1 Motivating Example

Let us illustrate the optimization challenge and the associated formalism on a simple example, again a series of three matrix-products, ThreeMatMat, shown in Figure 8.1.

```
  for (i1 = 0; i1 < N; ++i1)
    for (j1 = 0; j1 < N; ++j1)
      for (k1 = 0; k1 < N; ++k1)
R       C[i1][j1] += A[i1][k1] * B[k1][j1];
  for (i2 = 0; i2 < N; ++i2)
    for (j2 = 0; j2 < N; ++j2)
      for (k2 = 0; k2 < N; ++k2)
S       F[i2][j2] += D[i2][k2] * E[k2][j2];
  for (i3 = 0; i3 < N; ++i3)
    for (j3 = 0; j3 < N; ++j3)
      for (k3 = 0; k3 < N; ++k3)
T       G[i3][j3] += C[i3][k3] * F[k3][j3];
```

Figure 8.1: Running example: $C = AB$, $F = DE$, $G = CF$

We set $N = 512$, and computed 5 different versions of ThreeMatMat resorting to complex transformations that include tiling. We experimented on three high-end machines described in Section 8.4, using Intel ICC compiler for the AMD and Intel machines, and IBM XL/C for the Power5+, all with aggressive optimization flags. With the Power5+, none of the versions are able to outperform the native compiler. But with the Intel and AMD machines, we outperform ICC by a factor 2.28× and 1.84×, respectively. We observe that the best found version depends on the target machine: for the Intel, the best found version is shown in Figure 8.2(2), on which further polyhedral tiling and parallelization have been applied (not shown in Figure 8.2). But on the AMD machine distributing all statements and individually tiling them performs best, 1.23× better than 8.2(2).



$$\mathbf{L}_R = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \mathbf{L}_S = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \mathbf{L}_T = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad \beta_R = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \beta_S = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \beta_T = \begin{pmatrix} 0 \\ 2 \\ 2 \end{pmatrix} \quad (1)$$

$$\mathbf{L}_R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \mathbf{L}_S = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \mathbf{L}_T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \beta_R = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \beta_S = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \beta_T = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad (2)$$

Figure 8.2: Two possible legal transformations for $C = AB$, $F = DE$, $G = CF$

The challenge lies in the conjunction of a combinatorial transformation space and a poorly understood profitability model for those transformations. Figure 8.2 shows a few combinations of loop fusions:

one combination may expose better temporal reuse of the intermediate arrays `C` and/or `F`, while potentially damaging other important performance components. Of course, given a selected combination of loop fusions, many other decisions have to be taken: is it profitable to interchange some of the loops to exhibit fine-grain vector parallelism? is it possible to tile the iteration space (blocked-matrix-multiplication) to further improve temporal locality? Is such a tiling transformation compatible with the extraction of coarse-grain thread-level parallelism? Is there a way to select those transformations to benefit from both thread-level and vector parallelism? Of course, there is no well understood methodology or heuristic to answer such questions. The state-of-the-art provides only rough models of the impact of loop transformations on the actual execution, and it does not provide any effectiveness guarantee regarding the heuristics.

8.2.2 Challenges and Overview of the Technique

We want to provide a general and sound optimization framework based on the selection of multi-level statement interleavings. Several challenges must be tackled:

1. *Expression* of any possible multidimensional statement interleaving in a convex fashion, to enable the design of operation research algorithms (e.g., linear programs) for interleaving selection, and to enable an efficient enumeration of this set; this was covered in Chapter 3.
2. *Pruning* this set so that all and only semantics-preserving interleavings remain, in the most general framework of arbitrary transformations for interleaving construction; this was covered also in Chapter 3.
3. providing *tractable* techniques to perform program optimization based on interleaving selection, together with powerful optimizations such as loop tiling and parallelization.

Our technique relies on a level-by-level decomposition for all these problems. In a nutshell, we proceed from the outermost dimension (corresponding to the outermost loops) inwards, and prune the space of interleavings at each dimension. This results in a space of candidate interleavings at that level, for each of which we may exhibit an optimizing affine schedule enabling the effective transformation of the loop nest.

8.3 Optimizing for Locality and Parallelism

In Chapter 3 we defined a general framework for multi-level statement interleaving. We address now the problem of providing a complete optimization algorithm that integrates tiling and parallelization, along with the possibility to iteratively select different interleavings. Interleaving selection allows us to determine which statements are fused and which are not, a critical decision for performance.

The optimization algorithm proceeds recursively, from the outermost level to the innermost. At each level of the recursion, we *select* the associated schedule dimension by instantiating its values. We then build the set of semantics-preserving interleavings at that level, pick one and proceed to the next level until a full schedule is instantiated.

We first present additional conditions on the schedules to improve the performance of the generated transformation, by integrating parallelism and tilability as criteria. As we progressively instantiate the

schedule dimensions, constructing the set of legal interleaving at a given level is simplified: we have reduced the number of unknowns, as we know exactly which dependences have been solved at a previous level. We show how to construct this set without having to resort to testing the set of legal schedules for sets larger than a pair of statements. Finally we present the complete optimization algorithm.

8.3.1 Additional Constraints on the Schedules

Tiling (or blocking) is a crucial loop transformation for parallelism and locality. Bondhugula et al. developed a technique to compute an affine multidimensional schedule such that parallel loops are brought to the outer levels, and loops with dependences are pushed inside [20, 21]; at the same time, the number of dimensions that can be tiled are maximized. We extend and recast their technique into our framework.

Legality of tiling Tiling along a set of dimensions is legal if it is legal to proceed in fixed block sizes along those dimensions: this requires dependences to not be backward along those dimensions, thus avoiding a dependence path going out of and coming back into a tile; this makes it legal to execute the tile atomically. Irigoin and Triolet showed that a sufficient condition for a schedule Θ to be tilable [60], given R the dependence cone for the program, is that

$$\Theta.R \geq \vec{0}$$

In other words, this is equivalent to saying that all dependences must be weakly satisfied for each dimension Θ_k of the schedule. Such a property for the schedule is also known as Forward Communication Only property [53]. Note that schedule does not need to respect the FCO property to be legal: given a schedule dimension d at which a dependence is strongly satisfied, it does not need to be taken into account for the legality constraints for subsequent schedule dimensions. On the contrary, FCO requires the dependence to still be taken into account for subsequent dimensions, but enforcing only weak satisfaction (that is, $\Theta_k^S(\vec{x}_S) - \Theta_k^R(\vec{x}_R) \geq 0$) is enough. Considering the p first schedule dimensions, if they respect the FCO property then they can be permuted without breaking the program semantics [53, 20]. Tiling these p dimensions is thus legal.

Returning to Lemma 3.4, it is possible to add an extra condition such that the p first dimensions of the schedules are permutable. This gives a sufficient condition for the p first dimensions to be tilable. This translates into the following additional constraint on schedules, to enforce permutability of schedule dimensions.

Definition 8.1 (Permutability condition) *Given two statements R,S. Given the conditions for semantics-preservation as stated by Lemma 3.4. Their schedule dimensions are permutable up to dimension k if in addition:*

$$\forall \mathcal{D}_{R,S},\ \forall p \in \{1,\ldots,k\},\ \forall \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S},$$
$$\Theta_p^S(\vec{x}_S) - \Theta_p^R(\vec{x}_R) \geq \delta_p^{\mathcal{D}_{R,S}}$$

To translate k into actual number of permutable loops, the k associated schedule dimensions must express non-constant schedules (unless these dimensions could express only statement interleaving).

Rectangular tiling Selecting schedules such that each dimension is independent with respect to all others enables a more efficient tiling. Rectangular or close to rectangular blocks are achieved when possible, avoiding complex loop bounds in the case of arbitrarily shaped tiles. We resort to augmenting the constraints, level-by-level, with independence constraints. At this stage, this implies: to compute schedule dimension k, we need to have *instantiated* a schedule for all previous dimensions 1 to $k-1$. This comes from the fact that orthogonality constraints are not linear or convex and cannot be modeled as affine constraints directly. In its complete form, adding orthogonality constraints leads to a non-convex space, and ideally, all cases have to be tried and the best among those kept. When the number of statements is large, this leads to a combinatorial explosion. In such cases, we restrict ourselves to the sub-space of the orthogonal space where all the constraints are non-negative (that is, we restrict to have $\theta_{i,j} \in \mathbb{N}$). By just considering a particular convex portion of the orthogonal sub-space, we discard solutions that usually involve loop reversals or combination of reversals with other transformations; however, we believe this does not make a strong difference in practice. For the rest of this chapter, **we now fix** $\theta_{i,j} \in \mathbb{N}$.

Flexible permutability condition If it is not possible to express permutable loops for the first level, Bondhugula proposed to split the statements into distinct blocks to increase the possibility to find outer permutable loops [21]. Since our technique already supports explicitly the selection of any semantics-preserving possibility to split statements into blocks via the statement interleaving, we propose instead to enable the construction of *inner* permutable loops, by choosing to maximize the number of dependences solved at the first levels until we (possibly) find permutable dimensions at the current level. Doing so increases the freedom for the schedule at inner dimensions when it is not possible to express permutable loops at the outer levels. Maximizing the number of dependences solved at a given level was introduced by Feautrier [42] and we use a similar form:

$$S^i = \max \sum_{\mathcal{D}_{R,S}} \delta_k^{\mathcal{D}_{R,S}} \tag{8.1}$$

This cost function replaces the permutability condition, when it is not possible to find a permutable schedule for a given dimension k.

Dependence distance minimization There are infinitely many schedules that may satisfy the permutability criterion from Definition 8.1 as well as (8.1). An approach that has proved to be simple, practical, and powerful has been to find those schedules that have the shortest dependence components along them [20]. For polyhedral code, the distance between dependent iterations can always be bounded by an affine function of the global parameters, represented as a p-dimensional vector $\vec{n}$.

$$\mathbf{u}.\vec{n} + w \geq \Theta^S(\vec{x}_S) - \Theta^R(\vec{x}_R) \quad \langle \vec{x}_R, \vec{x}_S \rangle \in \mathcal{D}_{R,S} \qquad \mathbf{u} \in \mathbb{N}^p, w \in \mathbb{N} \tag{8.2}$$

The permutability and bounding function constraints are recast through the affine form of the Farkas Lemma such that the only unknowns left are the coefficients of Θ and those of the bounding function, namely $\mathbf{u}$, w. Coordinates of the bounding function are then used as the minimization objective to obtain the unknown coefficients of Θ.

$$\text{minimize}_{\prec} \; (\mathbf{u}, w, \ldots, \theta_{i,1}, \ldots) \tag{8.3}$$

The resulting transformation is a complex composition of multidimensional loop fusion, distribution, interchange, skewing, shifting and peeling. Eventually multidimensional tiling is applied on all permutable bands, and resulting tiles can be executed in parallel or at worse with a pipeline-parallel schedule [21]. Tile sizes are computed such that data accessed by each tile roughly fits in the L1 cache.

8.3.2 Computation of the Set of Interleavings

As we now proceed level-by-level, we face a simpler problem when considering the construction of the set of interleavings. We have *selected* a schedule for the previous levels, and so we know exactly the polyhedral dependence graph to consider for the current level.

We look at the problem of deciding if a given set of statements is fusable at level k, given the schedule constraints from Definition 8.1 and a schedule for levels 1 to $k-1$. The solution put forward in Chapter 3 resorts to testing the existence of a semantics-preserving schedule leading to fusion *for the whole set of statements*. As we are now considering a more constrained problem, we propose a technique based on checking the existence of compatible permutations along all *pairs of statements* in the set.

Fusability is the capability to exhibit a semantics-preserving schedule such that some of the instances are fused according to Definition 3.6. First let us remark that fusability is not a transitive relation. As an illustration, consider the sequence of matrix-by-vector products $x = Ab,\ y = Bx,\ z = Cy$. While it is possible to fuse them 2-by-2, it is not possible to fuse them all together. When considering fusing loops for $x = Ab,\ y = Bx$, one has to permute loops in $y = Bx$. When considering fusing loops for $y = Bx,\ z = Cy$, one has to keep loops in $y = Bx$ as is.

Let us now propose a decomposition of *one-dimensional schedules* in two sub-parts, with the objective of isolating loop permutation from the other transformations embedded in the schedule. One can decompose a one-dimensional schedule Θ^R_k with coefficients in $\mathbb{N}$ into two sub-schedules μ^R and λ^R such that:

$$\Theta^R_k = \mu^R + \lambda^R,\ \ \mu^R_i \in \mathbb{N},\ \ \lambda^R_i \in \mathbb{N}$$

without any loss of expressiveness. Such a decomposition is always possible because of the distributivity of the matrix multiplication over the matrix addition. For our purpose, we are interested in modeling one-dimensional schedules which are *not* constant schedules. This is relevant as we do not want to consider building a schedule for fusion that would translate only into statement interleaving. On the contrary we aim at building a schedule that performs the interleaving of statements *instances*, hence the linear part of the schedule must be non-null. For R surrounded by d loops, we enforce μ to be a linear form of the d loop iterators:

$$\mu^R(\vec{x}_R) = \begin{pmatrix}\mu^R_1 & \dots & \mu^R_d & \vec{0} & 0\end{pmatrix} . \begin{pmatrix}i_1 & \dots & i_d & \vec{n} & 1\end{pmatrix}^t$$

To model non-constant schedules, we add the additional constraint $\sum_{i=1}^{d} \mu^R_i = 1$. Note that by constraining μ to have only one coefficient set to 1, this does not prevent to model any compositions of slowing or skewing: these would be embedded in the λ part of the schedule, as shown in the example below.

The μ part of the schedule models different cases of loop permutations. For instance for statement R surrounded by 3 loops in the illustrating example, μ^R can take only three values:

$$\mu^R(\vec{x}_R) = \begin{pmatrix}1 & 0 & 0 & 0 & 0\end{pmatrix} . \begin{pmatrix}i & j & k & N & 1\end{pmatrix}^t = (i)$$
$$\mu^R(\vec{x}_R) = \begin{pmatrix}0 & 1 & 0 & 0 & 0\end{pmatrix} . \begin{pmatrix}i & j & k & N & 1\end{pmatrix}^t = (j)$$
$$\mu^R(\vec{x}_R) = \begin{pmatrix}0 & 0 & 1 & 0 & 0\end{pmatrix} . \begin{pmatrix}i & j & k & N & 1\end{pmatrix}^t = (k)$$

while λ^R can take arbitrary values. For better illustration let us now build the decomposition of the schedule $\Theta_k^R = (2.j+k+2)$. Θ_k^R is the composition of a permutation, a non-unit skewing and a shifting, and can be decomposed as follows:

$$\Theta_k^R(\vec{x}_R) = \begin{pmatrix}0 & 2 & 1 & 0 & 2\end{pmatrix}.\begin{pmatrix}i & j & k & N & 1\end{pmatrix}^t = (2.j+k+2)$$

$$\mu^R(\vec{x}_R) = \begin{pmatrix}0 & 1 & 0 & 0 & 0\end{pmatrix}.\begin{pmatrix}i & j & k & N & 1\end{pmatrix}^t = (j)$$
$$\lambda^R(\vec{x}_R) = \begin{pmatrix}0 & 1 & 1 & 0 & 2\end{pmatrix}.\begin{pmatrix}i & j & k & N & 1\end{pmatrix}^t = (j+k+2)$$

$$\Theta_k^R(\vec{x}_R) = (\mu^R+\lambda^R)(\vec{x}_R) = (2.j+k+2)$$

One may note that another possible decomposition is $\mu^R(\vec{x}_R) = (k)$, $\lambda^R(\vec{x}_R) = (2j+2)$. In general, when the schedule contains skewing it is possible to embed either of the skewing dimensions in the μ part of the schedule. For the sake of coherency we add an extra convention for the decomposition: μ matches the first non-null iterator coefficient of the schedule. Returning to the example, $\mu^R(\vec{x}_R) = (j)$, $\lambda^R(\vec{x}_R) = (j+k+2)$ is thus the only valid decomposition of Θ_k^R.

Note that this decomposition prevents modeling of compositions of loop permutations in the λ part. For λ to represent a loop permutation, λ must have values in $\mathbb{Z}$, as shown in the following example:

$$\mu^R(\vec{x}_R) = \begin{pmatrix}1 & 0 & 0 & 0 & 0\end{pmatrix}.\begin{pmatrix}i & j & k & N & 1\end{pmatrix}^t = \begin{pmatrix}i\end{pmatrix}$$
$$(\mu^R+\lambda)(\vec{x}_R) = \begin{pmatrix}j\end{pmatrix} \Rightarrow \lambda = \begin{pmatrix}-1 & 1 & 0 & 0 & 0\end{pmatrix}$$

which is not possible as we have constrained $\lambda_i \in \mathbb{N}$. Hence, when considering arbitrary compositions of permutation, (parametric) shifting, skewing and peeling, the $\mu+\lambda$ decomposition separates permutation (embedded in the μ part of the schedule) from the other transformations (embedded in the λ part of the schedule). We now show it is possible to determine if a set of statements are fusable only by looking at the possible values for the μ part of their schedules.

Considering three statements R,S,T that are fusable while preserving the semantics at level k. Then there exist $\Theta_k^R = \mu^R+\lambda^R$, $\Theta_k^S = \mu^S+\lambda^S$, $\Theta_k^T = \mu^T+\lambda^T$ leading to fusing those statements. Considering now the sub-problem of fusing only R and S, we build the set $\mathcal{M}_{R,S}$ of all possible values of μ^R,μ^S for which there exist a λ^R,λ^S leading to fuse R and S. Obviously, the value of μ^R,μ^S leading to fusing R,S,T are in $\mathcal{M}_{R,S}$, and μ^S,μ^T are also in $\mathcal{M}_{S,T}$. Similarly μ^R,μ^T are in $\mathcal{M}_{R,T}$. We derive a sufficient condition for fusability based on pairwise loop permutations for fusion.

Lemma 8.1 (Pairwise sufficient condition for fusability) *Given three statements R,S,T such that they can be 2-by-2 fused and distributed. Given $\mathcal{M}_{R,S}$ (resp. $\mathcal{M}_{R,T}$, resp. $\mathcal{M}_{S,T}$) the set of possible tuples μ^R,μ^S (resp. μ^R,μ^T, resp. μ^S,μ^T) leading to fusing R and S (resp. R and T, resp. S and T) such that the full program semantics is respected. R,S,T are fusable if there exists μ^R,μ^S,μ^T such that:*

$$\mu^R,\mu^S \in \mathcal{M}_{R,S}$$
$$\mu^R,\mu^T \in \mathcal{M}_{R,T}$$
$$\mu^S,\mu^T \in \mathcal{M}_{S,T}$$

Proof. Given the schedule $\Theta_k^R = \mu^R+\lambda^R, \Theta_k^S = \mu^S+\lambda^S$ leading to fusing R and S, $\Theta_k^{'R} = \mu^R+\lambda^{'R}, \Theta_k^T = \mu^T+\lambda^T$ leading to fusing R and T, and $\Theta_k^{'S} = \mu^S+\lambda^{'S}, \Theta_k^{'T} = \mu^T+\lambda^{'T}$ leading to fusing S and T, such that they all preserve the full program semantics.

The schedule $\Theta_k^{*R} = \mu^R + \lambda^R + \lambda'^R, \Theta_k^{*S} = \mu^S + \lambda^S + \lambda'^R$ is legal, as adding λ'^R consists in performing additional compositions of skewing and shifting, which cannot make the dependence vectors lexicographically negative. It cannot consist in performing a parametric shift (resulting in a loop distribution), Θ_k^R is a schedule fusing R and S and $\Theta_k'^R$ is a schedule fusing R and T. As Θ_k^{*R} is a non-constant schedule, it leads to fusing R and S. Generalizing this reasoning we can exhibit the following semantics-preserving schedule leading to the fusion of R, S, T:

$$\Theta_k^{*R} = \mu^R + \lambda^R + \lambda'^R + \lambda^S + \lambda'^S + \lambda^T + \lambda'^T$$
$$\Theta_k^{*S} = \mu^S + \lambda^R + \lambda'^R + \lambda^S + \lambda'^S + \lambda^T + \lambda'^T$$
$$\Theta_k^{*T} = \mu^T + \lambda^R + \lambda'^R + \lambda^S + \lambda'^S + \lambda^T + \lambda'^T$$

As all statements are fused 2-by-2, they are fused all together. As the three statements can be distributed 2-by-2, there is no dependence cycle. ■

To stress the importance of Lemma 8.1, let us return to the illustrating example. We can compute the pairwise permutations for fusion sets at the outermost level:

$$\mathcal{M}_{R,S} = \{(i,i);(i,j);(i,k);(j,i);(j,j);(j,k);(k,i);(k,j);(k,k)\}$$
$$\mathcal{M}_{R,T} = \{(i,i);(j,k)\}$$
$$\mathcal{M}_{S,T} = \{(i,k);(j,j)\}$$

These sets are computed by iteratively testing, against the set of constraints for semantics-preservation augmented with fusion and orthogonality constraints, for the existence of solutions with a non-null value for each of the coefficients associated with the statement iterators. Technically, we do not need to compute $\mathcal{M}_{R,S}$ as the two statements are independent, and are trivially fusable. Hence $\mathcal{M}_{R,S}$ does not contribute to the fusability of R, S, T. Here we can decide that R, S, T are fusable, as the solution $\mu^R = j,\ \mu^S = i,\ \mu^T = k$ respects the conditions from Lemma 8.1. This solution is presented in Figure 8.2(1). Note that fusability at level d does not imply fusability at level $d+1$, although the number of dependences to consider can only decrease for level $d+1$. This is because again we add orthogonality constraints, providing stronger conditions on the remaining schedule dimensions.

To improve further the tractability, we rely on two more standard properties on fusion. Given two statements R and S:

1. if R and S are not fusable, then any statement on which R transitively depends on is not fusable with S and any statement transitively depending on S;
2. reciprocally, if R and S must be fused, then any statement depending on R and on which S depends must also be fused with R and S.

These properties cut the number of tested sequences dramatically, in particular, in highly constrained programs such as loop-intensive kernels. They are used at each step of the optimization algorithm. Note that it was not profitable to rely on these properties for the general pruning algorithm: computing the existence of dependent statements at a given level is a combinatorial problem when the schedule at previous levels is not yet known.

8.3.3 Optimization Algorithm

We now present our optimization algorithm. The algorithm explores possible interleavings of dimension *maxExploreDepth*, and generates a collection of program schedules, each of them being a candidate for

the optimization. We use iterative compilation to select the best performing one. For each candidate program schedule we generate back a syntactic C code, compile it and run it on the target machine.

The structure and principle of the optimization algorithm, shown in Figure 8.3, matches that of the pruning algorithm of Figure 3.4, as it also aims at computing a set of feasible interleavings at a given level. It is in essence a *specialization* of the pruning algorithm for our optimization problem instance. To decide the fusability of a set of statements, we put the problem in a form matching the applicability conditions of Lemma 8.1. We merge nodes that must be 2-by-2 fused to guarantee that we are checking for the strictest set of program-wise valid μ values when considering fusability.

```
OptimizeRec: Compute all optimizations
Input:
  Θ: partial program optimization
  pdg: polyhedral dependence graph
  d: current level for the interleaving exploration
  n: number of statements
  maxExploreDepth: maximum level to explore for interleaving
Output:
  Θ: complete program optimization

1   G ← newGraph(n)
2   F^d ← O
3   unfusable ← ∅
4   forall pairs of dependent statements R,S in pdg do
5      T_{R,S} ← buildLegalOptimizedSchedules({R,S}, Θ, d, pdg)
6      if mustDistribute(T_{R,S}, d) then
7         F^d ← F^d ∩ {e_{R,S} = 0}
8      else
9         if mustFuse(T_{R,S}, d) then
10           F^d ← F^d ∩ {e_{R,S} = 1}
11        end if
12        F^d ← F^d ∩ {s_{R,S} = 0}
13        M_{R,S} ← computeLegalPermutationsAtLevel(T_{R,S}, d)
14        addEdgeWithLabel(G, R, S, M_{R,S})
15     end if
16  end for
17  forall pairs of statements R,S such that e_{R,S} = 1 do
18     mergeNodes(G, R, S)
19     updateEdgesAfterMerging(G, R, S)
20  end for
21  for l ← 2 to n−1 do
22     forall paths p in G of length l such that
         there is no prefix of p in unfusable do
23        if ¬ existCompatiblePermutation(nodes(p)) then
24           F^d ← F^d ∩ {∑_p e_{pairs in p} < l−1}
25           unfusable ← unfusable ∪ p
26        end if
27     end do
28  end for
29  forall i ∈ F^d do
30     Θ_d ← computeOptimizedSchedule(Θ, pdg, d)
31     if d < maxExploreDepth then
32        OptimizeRec(Θ, pdg, d+1, n, maxExploreDepth)
33     else
34        finalizeOptimizedSchedule(Θ, pdg, p)
35        return Θ
36     end if
37  end for
```

Figure 8.3: Optimization Algorithm

Procedure `buildLegalSchedules` computes, for a given pair of statements R,S, the set $\mathcal{L}_{R,S}$ of semantics-preserving schedules as described by Lemma 3.4. When $d > 1$, then some of the schedule

coefficients have already been instantiated, those in Θ_k, $k < d$. For such case, the corresponding coefficients in $\mathcal{L}_{R,S}$ are explicitly set to their instantiated value in Θ, and for dependences which are strongly solved by Θ_k the associated Boolean variables $\delta_k^{\mathcal{D}_{R,S}}$ are set to 1.

Procedure `mustDistribute` checks if it is legal to fuse the statements R and S. To perform the check, $\mathcal{L}_{R,S}$ is augmented with additional constraints:

- the fusion constraints up to level d according to Definition 3.6;
- the permutability constraints up to level d from Definition 8.1;
- the linear independence constraints (i.e., orthogonality constraints as discussed in the previous section) up to level d.

If there is no solution in the augmented set of constraints, then the statements cannot be fused at that level and hence must be distributed.

Procedure `mustFuse` checks if it is legal to distribute the statements R and S. The check is performed by augmenting $\mathcal{L}_{R,S}$ with the permutability and linear independence constraints up to level d, together with the insertion of a splitter at level d. If there is no solution in this set of constraints, then the statements cannot be distributed at that level and hence must be fused.

Procedure `computeLegalPermutationsAtLevel` computes $\mathcal{M}_{R,S}$ the set of all valid permutations μ^R, μ^S leading to fusion. This procedure operates on the same set as `mustDistribute`, that is, $\mathcal{L}_{R,S}$ augmented with the fusion and permutability constraints. To check if a given permutation μ^R, μ^S is valid and leading for fusion at level d, the set of constraints is tested for the existence of a solution where the schedule coefficients of row d corresponding to μ^R and μ^S is not 0. This is sufficient to determine the existence of the associated λ part. Returning to the ThreeMatMat example, and considering statements R and S. The two prototype schedules for R and S at level d are:

$$\Theta_d^R = \theta_{d,1}^R.i + \theta_{d,2}^R.j + \theta_{d,3}^R.k + \theta_{d,4}^R.N + \theta_{d,5}^R.1$$
$$\Theta_d^S = \theta_{d,1}^S.i + \theta_{d,2}^S.j + \theta_{d,3}^S.k + \theta_{d,4}^S.N + \theta_{d,5}^S.1$$

To determine all pairs μ^R, μ^S leading to fusion at level d, we successively test for the existence of a solution in the augmented set of constraints where $\theta_{d,1}^R > 0, \theta_{d,1}^S > 0$, then $\theta_{d,1}^R > 0, \theta_{d,2}^S > 0$, and so on for the 9 different pairs. Then, each time a solution do exist, the corresponding tuple μ^R, μ^S (e.g., $\mu^R = i, \mu^S = i$) is inserted in $\mathcal{M}_{R,S}$.

Procedure `updateEdgesAfterMerging` modifies the graph edges after the merging of two nodes R and S such that: (1) if for T there was an edge $e^{T\to R}$ and not $e^{T\to S}$ or vice-versa, $e^{T\to R}$ is deleted, and $e_{S,T} = 0$, this is to remove triplets of trivially unfusable sets; (2) if there are 2 edges between T and RS, one of them is deleted and its label is added to the remaining one existing label; (3) the label of the edge $e^{R\to S}$ is added to all remaining edges to/from RS, and $e^{R\to S}$ is deleted.

Procedure `existCompatiblePermutation` collects the sets $\mathcal{M}$ of the pairs of statements connected by the path p, and tests for the existence of μ values according to Lemma 8.1. If there is no compatible permutation, then an additional constraint is added to $\mathcal{F}^d$ such that it is not possible to fuse the statements in p all together. The constraint sets that the $e_{i,j}$ variables, for all pairs i, j in the path p, cannot be set to 1 all together.

Procedure `computeOptimizedSchedule` instantiates a schedule Θ_d at the current dimension d, for all statements. The interleaving is given by i, and for each group of statements to be fused under a common loop, a schedule is computed to maximize fusion and to enforce permutability if possible. To select the coefficient values we resort to the objective function (8.3).

Procedure `finalizeOptimizedSchedule` computes the possibly remaining schedule dimensions, when $maxEploreDepth$ is lower than the maximum program loop depth. Note that in this case, maximal fusion is used to select the interleaving, hence we do not need to build a set of interleavings for the remaining depths.

In practice this algorithm proved to be very fast, and for instance computing the set $\mathcal{F}^1$ of all semantics-preserving interleavings at the first dimension takes less than 0.5 second for the benchmark ludcmp, pruning $\mathcal{I}^1$ from about 10^{12} structures to 8, on an initial space with 182 binary variables to model all total preorders.

Let us go back to the illustrating example. We have set the value of $maxEploreDepth$ to 1. A graph G is constructed with 3 nodes, and two edges. The edge $e^{R\to T}$ is labeled with $\mathcal{M}_{R,T}$, and $e^{S\to T}$ with $\mathcal{M}_{S,T}$. There is no edge $e^{R\to S}$ as the statements are not dependent. All statements can be 2-by-2 fused and distributed, so there is no additional constraint on the $p_{R,S}, p_{S,T}, p_{R,T}$ variables nor $e_{R,S}, e_{S,T}, e_{R,T}$. But T depends on R and S, hence T cannot be executed before them: $s_{R,T}$ and $s_{S,T}$ are set to 0. There is only one path of length 2, testing for the fusability of R, S, T which are indeed fusable. The resulting interleaving space contains 5 possibilities, from all fused to all distributed, plus combinations where R is executed after S. For each case a complete optimization is computed, the resulting programs (before tiling) for three of them are shown in Figure 8.2.

8.3.4 Search Space Statistics

Several $e_{i,j}$ and $p_{i,j}$ variables are set during the pruning of $\mathcal{O}$, so several consistency constraints are made useless and are not built, significantly helping to reduce the size of the space to build. Table 8.4 illustrates this by highlighting, for our benchmarks considered, the properties of the polytope $\mathcal{O}$ in terms of the number of dimensions (#dim), constraints (#cst) and points (#points) when compared to $\mathcal{F}^1$, the polytope of possible interleavings *for the first dimension only*. For each benchmark, we list #loops the number of loops, #stmts the number of statements, #refs the number of array references, as well as the time to build all candidate interleavings from the input source code (that is, including all analysis) on an Intel Xeon 2.4 GHz. The number of candidates that end up being tested during the iterative process is reported (#Tested), as explained in the following section. We also report the dataset size we used for the benchmarks (Pb. Size).

				$\mathcal{O}$			$\mathcal{F}^1$					
Benchmark	#loops	#stmts	#refs	#dim	#cst	#points	#dim	#cst	#points	#Tested	Time	Pb. Size
advect3d	12	4	32	12	58	75	9	43	26	52	0.82s	300x300x300
atax	4	4	10	12	58	75	6	25	16	32	0.06s	8000x8000
bicg	3	4	10	12	58	75	10	52	26	52	0.05s	8000x8000
gemver	7	4	19	12	58	75	6	28	8	16	0.06s	8000x8000
ludcmp	9	14	35	182	3003	$\approx 10^{12}$	40	443	8	16	0.54s	1000x1000
doitgen	5	3	7	6	22	13	3	10	4	8	0.08s	50x50x50
varcovar	7	7	26	42	350	47293	22	193	96	192	0.09s	1000x1000
correl	5	6	12	30	215	4683	21	162	176	352	0.09s	1000x1000

Figure 8.4: Search space statistics

Finally, for each candidate fusion structure, we also test with and without the application of poly-

hedral tiling (hence the number of tested candidates being twice the number of points in $\mathcal{F}_1$). The motivation is twofold. Firstly, tiling may be detrimental as it may introduce complex loop bounds and the computation overhead may not be compensated by the locality improvement. Secondly, tiling may prevent the compiler from performing aggressive, low-level optimizations, as current production compilers optimization heuristics are still very conservative, in particular when loop bounds are complex as in polyhedraly tiled code.

The final number of tested versions is shown in Figure 8.4, in the #Tested column.

8.4 Experimental Results

Studies performed in Chapter 6 on the performance impact of selecting schedules at various levels highlighted the much higher impact of carefully selecting outer loops. Hence, the selection of the statement interleaving at the outermost level captures the most significant difference in terms of locality and communication. We choose to limit the recursive traversal of interleavings to the outer level only, and show that we are still obtaining significant performance improvement and a wide range of transformed codes. Nevertheless, when the number of candidates in $\mathcal{F}^1$ is very small, typically because of several loop-dependent dependences at the outer level, it is relevant to build $\mathcal{F}^2$ and further. One can choose to enumerate the next dimension if there are 2 or less candidates at the current dimension, mostly to offer freedom for the iterative search while still controlling the combinatorial nature of the recursive search. Note that in the experiments presented in this paper we traverse exhaustively only $\mathcal{F}^1$.

The automatic optimization and parallelization process has been implemented in POCC, the *Polyhedral Compiler Collection*, a complete source-to-source polyhedral compiler based on available free software such as CLOOG, CLAN, CANDL, PIPLIB and POLYLIB. Specifically, the search space construction has been implemented in the LETSEE optimizer and the transformations for tiling and parallelization are computed by the PLUTO optimizer. In the generated programs, parallelization is obtained by marking transformed loops with OpenMP pragmas. In addition, when compiling with ICC, intra-tile parallel loops are moved to the innermost position and marked with `ivdep` pragmas to facilitate compiler auto-vectorization, when possible.

8.4.1 Experimental Setup

We experimented on three high-end machines: a 4-socket Intel hexa-core Xeon E7450 (Dunnington) at 2.4 *GHz* with 64 *GB* of memory (24 cores, 24 hardware threads), a 4-socket AMD quad-core Opteron 8380 (Shanghai) at 2.50 *GHz* (16 cores, 16 hardware threads) with 64 *GB* of memory, and an 2-socket IBM dual-core Power5+ at 1.65 *GHz* (4 cores, 8 hardware threads) with 16 *GB* of memory. All systems were running Linux 2.6.x. We used Intel ICC 11.0 with options `-fast -parallel -openmp` referred to as icc-par, and IBM/XLC 10.1 compiled for Power5 with options `-O3 -qhot=nosimd -qsmp -qthreaded` referred to as xlc-par.

We consider 8 benchmarks, typical from compute-intensive sequences of algebra operations. atax, bicg and gemver are compositions of BLAS operations [88], ludcmp solves simultaneous linear equations by LU decomposition, advect3d is an advection kernel for weather modeling and doitgen is an in-place 3D-2D matrix product. correl creates a correlation matrix, and varcovar creates a variance-covariance matrix, both are used in Principal Component Analysis in the StatLib library. Problem sizes are reported in column Pb. Size of Figure 8.4.

The time to compute the space, pick a candidate and compute a full transformation is negligible with respect to the compilation and execution time of the tested versions. In our experiments, the full process takes a few seconds for the smaller benchmarks, and up to about 2 minutes for correl on Xeon.

8.4.2 Performance Improvement

In Figure 8.5, we report for all benchmarks the speedup of our iterative technique (iter-*xx*) normalized to the best single-threaded version produced by the native compiler (ICC for Intel and Opteron, XLC for Power5+). We also compare the performance improvement obtained over maximal fusion as proposed by Bondhugula [21] and over ICC/XLC with automatic parallelization (icc-par or xlc-par) in Figure 8.6.

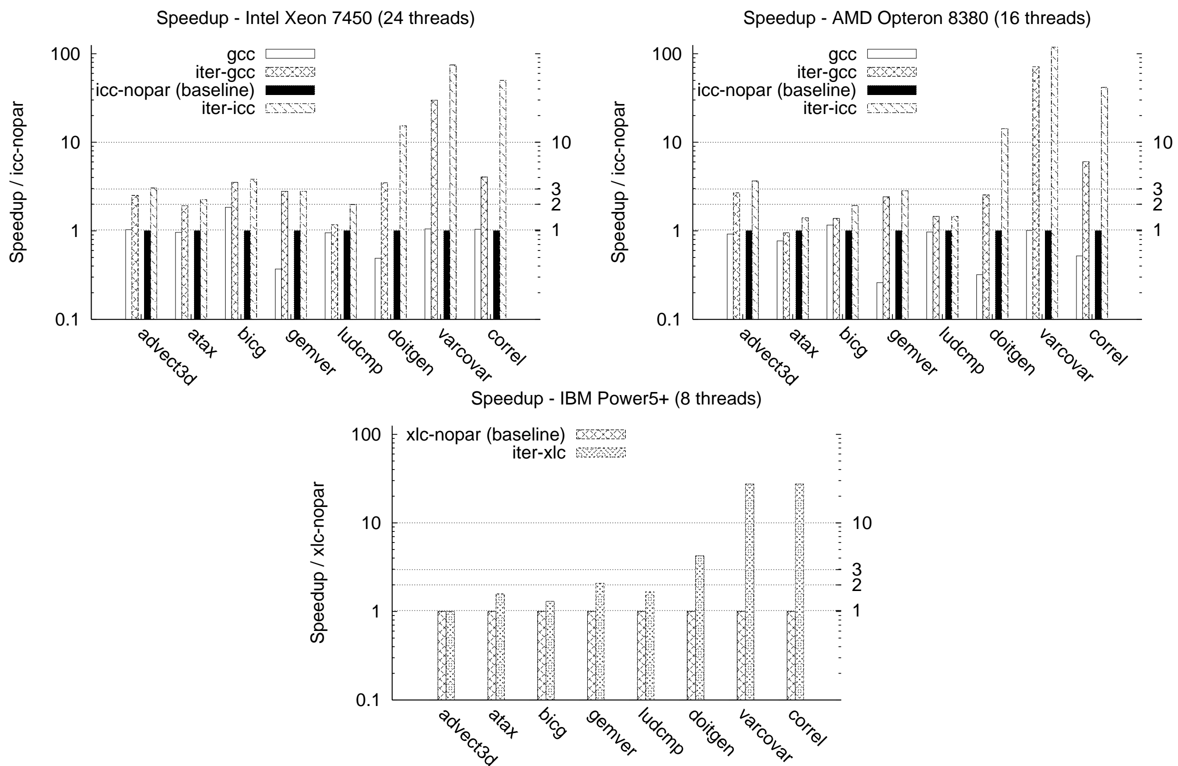


Figure 8.5: Speedup for Xeon, Opteron and Power5+ processors over best single-threaded version

For doitgen, correl and varcovar, three compute-bound benchmarks, our technique exposes a program with a significant parallel speedup of up to 112× on Opteron. Our optimization technique goes far beyond parallelizing programs, and for these benchmarks locality and vectorization improvements were achieved by our framework. For advect3d, atax, bicg, and gemver we also observe a significant speedup, but this is limited by memory bandwidth as these benchmarks are memory-bound. Yet, we are able to achieve a solid performance improvement for those benchmarks over the native compilers, of up to 3.8× for atax on Xeon and 5× for advect3d on Opteron. For ludcmp, although parallelism was exposed, the speedup remains limited as the program offers little opportunity for high-level optimizations. Yet, our technique outperforms the native compiler, by a factor up to 2× on Xeon.

For Xeon and Opteron, the iterative process outperforms ICC with auto-parallelization, with a factor between 1.2× for gemver on Intel to 15.3× for doitgen. For both of these kernels, we also compared with

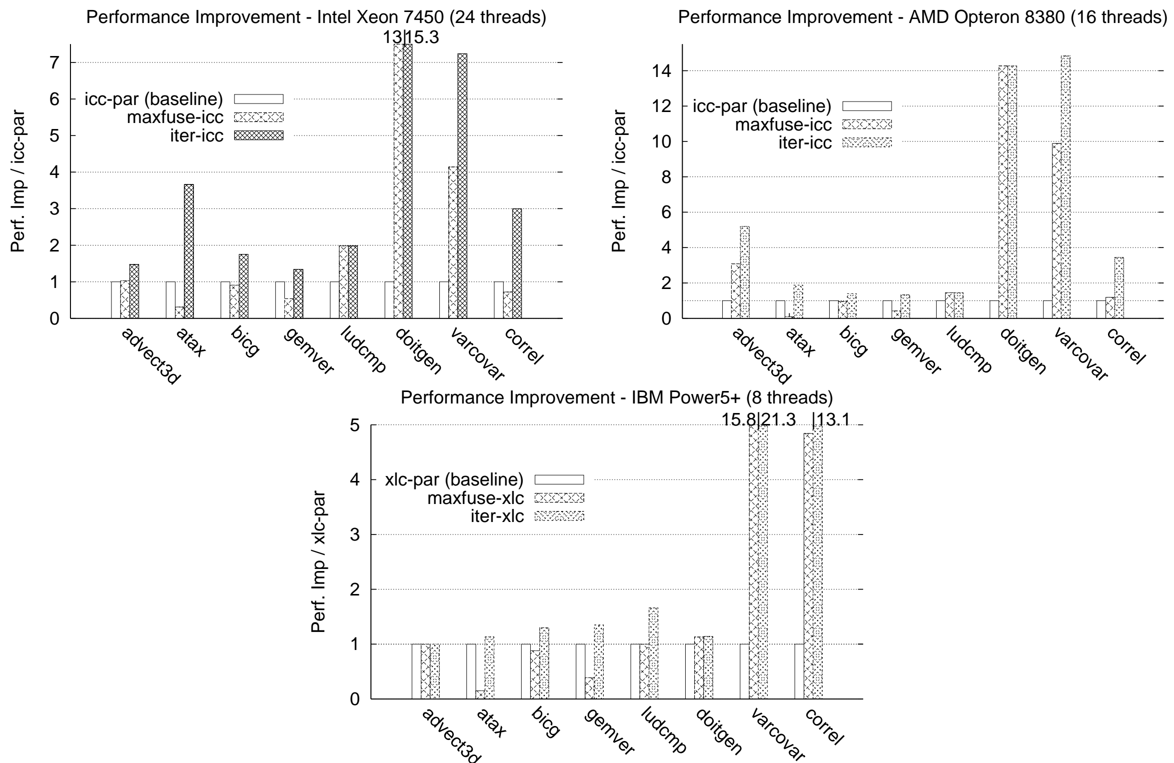


Figure 8.6: Performance improvement over maximal fusion, and over the the reference auto-parallelizing compiler

an implementation using Intel Math Kernel Library (MKL) 10.0 and AMD Core Math Library (ACML) 4.1.0 for the Xeon and Opteron machines respectively, and we obtain a speedup of 1.5× to 3× over these vendor libraries. For varcovar, our technique outperforms the native compiler by a factor up to 15×. Although maximal fusion significantly improved performance, the best iteratively found fusion structure provides a much better improvement, up to 1.75× better. Maximal fusion is also outperformed for all but ludcmp. This highlights the power of the method to discover the right balance between parallelism (both coarse-grain and fine-grain) and locality.

On Power5+, on all but advect3d the iterative process outperforms XLC with auto-parallelization, by a factor between 1.1× for atax to 21× for varcovar.

For the sake of completeness, we also provide the best performance in GFLOP/s for all our benchmarks in Figure 8.7. Benchmarks are superscripted with ^{d} when the data type is *double*, and with ^{f} for *float*.

	advect3d^f	atax^d	bicg^d	gemver^d	ludcmp^d	doitgen^d	correl^f	varcovar^f
Xeon E7450 (24 cores)	0.47	2.13	2.13	2.20	1.33	44.64	16.71	50.05
Opteron 8380 (16 cores)	0.53	1.42	1.70	2.66	0.75	31.25	11.14	33.36
Power5+ (4 cores)	0.34	1.16	1.15	1.42	0.48	10.41	7.16	14.30

Figure 8.7: Best Performance obtained, in GFLOP/s

8.4.3 Performance Portability

Beyond absolute performance improvement, another motivating factor for iterative selection of fusion structures is performance portability. Because of significant differences in design, in particular in SIMD units' performance and cache behavior, a transformation has to be tuned for a specific machine. This leads to a significant variation in performance across tested frameworks.

To illustrate this, we emphasize the gemver kernel shown in Figure 8.8.

```
    for (i = 0; i < M; i++)
       for (j = 0; j < M; j++)
S1      A[i][j] = A[i][j] + u1[i] * v1[j]
                  + u2[i] * v2[j];
    for (i = 0; i < M; i++)
       for (j = 0; j < M; j++)
S2      x[i] = x[i] + beta * A[j][i] * y[j];
    for (i = 0; i < M; i++)
S3    x[i] = x[i] + z[i];
    for (i = 0; i<M; i++)
       for (j = 0; j<M; j++)
S4      w[i] = w[i] + alpha * A[i][j] * x[j];
```

Figure 8.8: gemver original code

We show in Figure 8.9 the relative performance normalized with respect to icc-par of gemver, for Intel and Opteron. The version index is plotted on the x axis, 1 is max-fuse and 8 is maximal distribution.

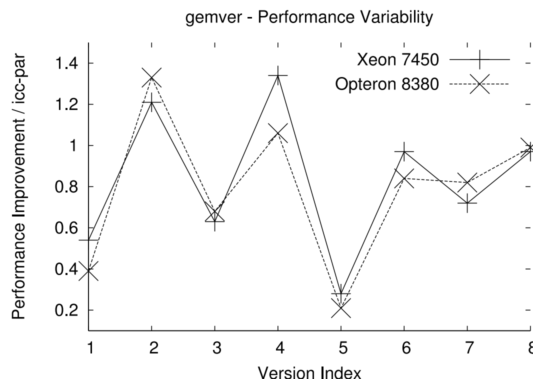


Figure 8.9: Performance variability for gemver

For Xeon, the best version is 4, corresponding to the fusion structure in Figure 8.10. It performs 10% better than version 2 — version 2 corresponds to the fusion structure where $S1$, $S2$ and $S3$ are fused, and $S4$ is in the next loop nest — which is the optimal fusion for Opteron. And for the Opteron, version 4 performs 20% slower than 2. Note that on gemver the performance distribution for Power5+ is very similar as for Xeon.

Performance variation is also exhibited by maximal fusion results over the three architectures. For ludcmp, while it is the best performing one on Xeon and Opteron, it is not on Power5+. Such a pattern can also be observed for doitgen.

The trade-off between coarse-grain parallelization and vectorization is very difficult to capture, as it also depends on the capability of the back-end compiler to perform vectorization. One has to capture the interplay between distinct optimization passes, something missing in present day compilers. Moreover,

```
    for (i = 0; i < M; i++)
       for (j = 0; j < M; j++) {
S1      A[i][j] = A[i][j] + u1[i] * v1[j]
                  + u2[i] * v2[j];
S2      x[j] = x[j] + beta * A[i][j] * y[i];
    }
    for (i = 0; i < M; i++)
S3    x[i] = x[i] + z[i];
    for (i = 0; i<M; i++)
       for (j = 0; j<M; j++)
S4      w[i] = w[i] + alpha * A[i][j] * x[j];
```

Figure 8.10: Optimal fusion for `gemver` on Xeon

accurate profitability models have to be relied upon, and their design remains a major challenge for compiler designers. Tuning the trade-off between fusion and distribution is a relevant technique to address the performance portability issue. Our technique is able to automatically adapt to the target framework, and successfully discovers the optimal fusion structure, whatever the specifics of the program, compiler and architecture.

8.5 Related Work

Several heuristics for loop fusion and tiling have been proposed for the construction of loop-nest optimizers [124, 68, 115, 95]. Those heuristics have also been revisited in the context of complex architectures with non-uniform memory hierarchies and heterogeneous computing resources [100]. The polyhedral model is complementary to those efforts, opening many more opportunities for the construction of loop nest optimizers and parallelizing compilers. It is currently being integrated in production compilers, including GCC and IBM XL.

The tiling hyperplane method has proved to be very effective in integrating loop tiling into polyhedral transformation sequences [60, 99]. However, the state-of-the-art model-driven technique proposed by Bondhugula et al. [20, 21] lacks a portable heuristic to select good loop fusion structures. But despite the weaknesses of its target-independent optimization model, it does identify interesting parallelism-locality trade-offs. It is also unable to compute an enabling schedule for fusion in the presence of parametric dependences. The techniques presented in this chapter inherits from the Tiling hyperplane benefits, and removes the limitations of Bondhugula's approach.

Powerful semi-automatic polyhedral frameworks have been designed as building blocks for compiler construction or (auto-tuned) library generation systems [64, 30, 51, 24, 108]. They capture fusion structures, but neither do they define automatic iteration schemes nor do they integrate a model-based heuristic to construct profitable parallelization and tiling strategies.

Iterative compilation has proved its efficiency in providing solid performance improvements over a broad range of architectures and transformations [18, 107, 3, 82, 95, 44, 119, 100]. However, none of the previous works achieved the expressiveness and application of complex transformation sequences presented in this chapter, along with a focused search on semantics-preserving candidates only.

Chapter 9

Future Work on Machine Learning Assisted Optimization

9.1 Introduction

For the past decade, compiler designers have looked for automated techniques to improve the quality and portability of the optimization heuristics implemented in compilers [23, 107, 110, 3]. Machine learning based approaches have been popularized in recent years, and arise as research projects in production compilers such as GCC. Fursin and the UNIDAPT group deployed iterative as well as collective optimization frameworks into GCC, to allow the compiler benefiting from the results of previous off-site compilation processes [49, 47, 50]. Building auto-tuning compilers is a promising direction to increase the productivity of compiler writers, and the quality of the generated code.

Many learning-based compiler techniques typically aim at performing a classification of the search space. A relevant application is to validate and even refine the subspace partitioning technique we have described in Chapter 6, but more standard is the application to the decision of applying a dedicated optimization heuristic or not. Yet the ultimate objective of machine learning assisted compilation is *given a new program, to determine which optimization should be applied on it.* In this spirit, we aim at going further and rely on supervised learning techniques to infer a *decision* based on the previous samples that have been seen during the training phase.

Previous work poorly addressed the problem of using machine learning to select an affine transformation for the program. It was at best limited to improve the search efficiency, with Genetic Algorithms for instance (see Nisbet for a basic approach [87] or Chapter 7 for a semantics-preserving one). Part of the reason can be found in the increased complexity of selecting *fine-grain optimizations*. To benefit from the polyhedral representation, one has to consider building a (possibly partial) transformation for each statements. This makes the problem more complicated than with *coarse-grain optimizations*, where the task of the process is to select a transformation to apply to the full program. For such cases, the standard output of the algorithm is a sequence of transformations (e.g., "tile + unroll") which is blindly transmitted to a black-box which then decides the application of the sequence. Problems such as semantics-preservation are not taken into account by the transformation selection process, and it is not possible to finely tune on which specific locations this transformation is applied.

We propose in this chapter to present some of the key ideas to achieve an automatic, learning-based fine-grain auto-tuner. Since we rely on the polyhedral program representation to abstract away the syn-

tactic limitations of a standard representation, this leads to a significantly more precise and powerful optimization framework. This also leads to new challenges which must be tackled. To harness the power of polyhedral optimization for machine-learning assisted compilation, it is required to revisit the main building blocks and adapt them. We start in Section 9.2 by presenting numerous indicators which can be used to relate programs. We rely on the algebraic nature of the polyhedral representation to devise new *program features* which are decoupled from the standard syntactic ones. Another major problem for a learning process is the knowledge representation, that is, how the information to be generated by the model is stored. We highlight in Section 9.3 the limits of a syntactic-based representation of transformations, and we present a generalized approach to model program optimization. Any off-line learning framework requires a training phase on a substantial and representative set of benchmarks. We build on the framework presented in this thesis to present in Section 9.4 a technique to generate a very large set of input programs. Finally, we gather all these ideas to give an abstract picture of the functioning of a machine learning process to build fine-grain optimizations in Section 9.5.

9.2 Computing Polyhedral Program Features

Program features are a convenient mean to describe a program with a fixed set of values. For compiler-based machine learning, it requires programs to be represented as a set of features that serve as inputs to a machine learning tool. Since the first experiments of ML in compilation, it has been required to model any input program in a normalized fashion with a fixed-length set of program features. Recent work shows it is a promising direction to automatically detect these features [75], yet we focus here to the task of exhibiting relevant features based on the polyhedral representation.

9.2.1 A Grain for Each Goal

Depending on the application for the learning process, different grains of features are relevant. Consider for instance the task of learning an unrolling heuristic. Unrolling happens at the loop level, so features of interest are defined at the granularity of the loop: information such as *loop nest depth* is meaningful [106]. Consider now the problem of deciding a sequence of optimizations for a full program part. The same feature cannot be reused as-is: one can expect several loop nests, and representatives like *maximal loop nest depth* or *average loop nest depth* must be used to keep a fixed number of features.

As we address the problem of the optimization of static control parts, it is needed to exhibit coarse-grain features which are normalized on the full program part. Although this intuitively contradicts our goal of performing fine-grain optimization, we show in Section 9.3 that the representation of transformation we select does not suffer from this limitation. Moreover, we present in the following section that abstract polyhedral features can provide an efficient characterization of the program without resorting to the finest syntactic features.

9.2.2 Some Standard Syntactic Features

Some standard syntactic features remain a very good discriminant for the program. They enable a fast classification of the input programs into major categories. We do not aim here to present an exhaustive list of syntactic features. Instead, we give an intuition of the elementary ones and left to the designer the task of building composite features and to evaluate the variance of those on the training set. Let us note

that a valid technique to determine which features are relevant for the learning model is to compute a very rich set of features for all the programs in the training set, and eliminate the useless ones for instance with Principal Component Analysis.

An overview of some syntactic features is shown in Figure 9.1. It is left to the reader to complete it to a more extensive list, but we believe it is enough to give a good intuition of the kind of syntactic features that can be extracted from the program.

# of loops	# of outer loops	# of branches
# of arrays read	# of arrays written	# of scalars read
# of scalars written	# of statements	# of operations
# of FP instructions	max loop depth	avg. loop depth
avg. stmts / outer loop	avg. loops / outer loop	avg. inst. / stmt

Figure 9.1: Some syntactic program features

9.2.3 Polyhedral Features

The polyhedral program representation abstracts away the actual computation which is performed by the statements. Instead, the program is described as a set of dependences between each executed instance of the statements. Syntactic features are required to get information about the actual computation which is performed — think for instance to Floating Point instructions in the program. On the other hand, this algebraic representation gives the finest information about *program data dependences* which actually dictates how the program can be transformed. We propose to extract two categories of features. The first involves metrics about the dependences themselves, the second metrics about transformations which can be done on the program.

Dependences The polyhedral dependence graph (PDG for short) is a unique characterization of the program restructuring possibilities. In other words, if two different programs have the same PDG, then exactly the same transformations can be performed on them. We propose a series of metrics to represent the PDG. We are typically interested in having synthetic information on dependent iterations. We heavily rely on the possibility of computing the volume of polyhedra [9] to compute the actual number of instances in dependence for a given dependence polyhedron. In order to have normalized measures, we will store a ratio between the statement iterations and the statement iterations in dependence. For instance, for the MatVect kernel, dependence polyhedron $\mathcal{D}^1_{R,S}$ has a representative of 2: $dim(\mathcal{D}^1_{R,S}) = 2$, and the minimal loop depth of R and S is 1. Note that in the case of parametric loop bounds, an enumerator for the volume of a polyhedron can be computed as an Ehrhart quasi-polynomial in the form of the parameters [28, 102, 117]. For this case, using the normalization will in general allow removing the parameters from the picture: we simply keep the maximal degree of the polynomial and compare it with the loop depth.

We propose the following metrics.

- number of dependence polyhedra (total, and for each dependence kind: read-after-read, read-after-write, write-after-read and write-after-write);
- average number of dependence per statement (for all kinds of dependences);
- average number of dependence per outer loop (for all kinds of dependences);

- maximal, average and minimal dependence ratio (as presented above) for the program.

Program Transformation We also propose to provide the learning model with metrics about the possibilities to transform the program. One of the main benefit of the polyhedral representation is to allow computing beforehand, via the expression of an affine schedule for the program, if the program can be tiled, parallelized or simdized for instance. To compute the existence of such transformations, we propose to use the state-of-the-art algorithm for tiling and parallelization in the polyhedral model: Bondhugula's Tiling Hyperplane method [20, 21]. We also propose to use the framework presented in Chapter 3 and Chapter 8 to evaluate the possibilities of fusion and distribution for the program. We believe this information about how the program can be restructured is a critical metric to classify the programs. Intuitively, we can roughly partition programs into the sequential and parallel classes, and it is useless for the learning model to learn how to do this partitioning. We propose to go even further, and to investigate more categories, as described in the following.

- Maximal number of fusable statements / total number of statements
- Maximal number of distributable statements / total number of statements
- Maximal number of statements under tiled loops / total number of statements
- Maximal number of statements under parallel tiled loops / total number of statements
- Maximal tiling depth
- Maximal number of coarse-grain parallel outer loops
- Maximal number of fine-grain parallel inner loops

9.3 Knowledge Representation

The polyhedral optimization search space encompasses arbitrarily complex sequences of transformations, this is the most complete approach in terms of expressiveness. A benefit is the increasing portability of the approach: as the largest set of transformations is considered, it is expected that the best transformation whatever the architecture lies in the space. But the downside of this expressiveness is the modeling of the transformation itself. Considering an optimization which was effective for a program in the training set, one wants to apply it on a new program which has similar features. This problem of optimization knowledge representation is not trivial in the polyhedral representation.

When considering a sequence-based representation, the information about the optimization is stored as a fixed-length vector of transformation primitives. Using a scheduling matrix makes more difficult the task of representing an optimization in uniform way.

1. Given a schedule, it is not possible to apply it as-is on another program. The schedule dimensions are different between two programs that do not share the same number of statements, loops per statements, and global parameters.
2. Converting a schedule back to a complete sequence of primitives is irrelevant. The equivalent sequence is of arbitrary length, and polluted with complementary transformations for legality which introduces a cumbersome noise in the learning space.

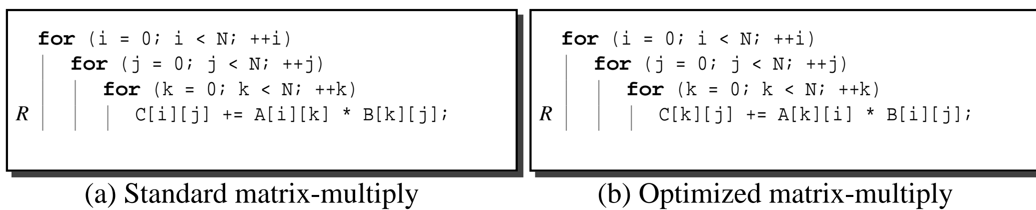
It is required to devise a mechanism to store the transformation associated with a program version such that the learning model can use this information to decide how to optimize another program. Two directions can be followed, the first based on modeling transformation primitives, and the second by relying on the optimization characteristics.

9.3.1 Dominant Transformation Extraction

The first and most intuitive approach is to extract a sequence-based representation from the affine schedule, but by distinguishing performance-related transformations from semantics-related ones. In Chapter 6 we have shown that a relevant subspace decomposition gives the lowest priority to shifting and peeling. Hence one can focus on rebuilding a sequence from the schedule based only on loop interchange, loop fusion / distribution, loop skewing and loop tiling (with tiling depth). These primitives would be defined in an abstract way and not for a pair of statements: for instance one can have $fuse(R,S)$, $tile(R,S)$ in the generated program, those would be abstracted as $fuse, tile$. Such an approach has been tested in the larger context of multidimensional (including illegal) affine schedules by Long et al. [80, 82].

When considering the problem from the knowledge representation point of view, it is in essence a very fragile approach. We are trying to learn how to fix the program instead of learning what we need to fix. Consider this naive analogy: should we learn that on a desert island we have to scratch matches to survive, or learn that we need to make a fire? Because if we don't have matches but only a lighter, should we irrevocably die?

Returning to a compilation language, consider the almost naive example in Figure 9.2. It is two matrix multiply programs, one written in the standard way and the other with an optimized loop order. Consider we are learning on a SIMD-capable single-core machine. Say for instance that training on the first example we learn that *interchange*(i,k) is the good transformation. If the model is asked to optimize the second program, it will also apply *interchange*(i,k) but the transformation will break the performance. But if we had learned that given such a program, *enable SIMD* was the correct way to optimize it then we would have efficiently transformed the second candidate too, by simply not changing the loop order.



(a) Standard matrix-multiply (b) Optimized matrix-multiply

Figure 9.2: Two matrix-multiply kernels

The efficiency of the sequence-based approach lies in the statistic probability to encounter only programs that are extremely close to the ones encountered during the training phase. The *specificity* of a primitive sequence is very high in the context of fine-grain optimization. One must be able to adapt the sequence to the new program. This leads to critical decision problems such as which transformation to adapt, and to which extent. Going back to the previous example, the process has to decide to select another interchange (which one?) or another transformation to optimize the program. Building such a process is an open problem up to date. The other possibility is to provide a training set rich enough to cover almost all cases, this is obviously unrealistic when considering all loop transformations together. This motivates the proposal of another technique to model knowledge about transformation.

9.3.2 Optimized Program Abstract Characteristics

Another approach is to consider several metrics that can be statically computed on the schedule resulting in the best optimization. The knowledge is then represented as the distance between the value for this metric, computed from the optimizing schedule, and the optimal value that is only program-dependent. The task of the learning model is to determine how metrics must be prioritized, and how they should be maximized with respect to the theoretical optimal for the program.

- *Parallelism.* Both coarse-grain and fine-grain parallelism may be extracted on a program. But depending on the program and on the target machine, the best performance does not necessarily come with the exploitation of the maximal degree of parallelism available in the input program. For instance, the transformation required to extract parallelism may exhibit too much control overhead and thus not be beneficial for performance. This situation typically occurs on loops with few iterations or non-uniform dependence patterns. The model can learn, given the maximal degree of parallelism for the program, what is the profitable amount of parallelism to extract.
- *Memory.* Several metrics do exist to characterize the memory behavior of a program. *Locality* of the memory accesses can be monitored, on a per-access basis. Furthermore, the frequency of an access can be computed thus leading to a precise characterization of the importance of the memory optimization. Consider for instance maximizing locality. Excessive fusion may interfere with hardware prefetching: processors have a limited number of hardware prefetch streams. In addition, after fusion, too many data spaces end up using the same cache, reducing the effective cache capacity of each statement, conflict misses are also likely to increase. The model can learn, on a per-access basis, to which extent locality was maximized for it – with respect to the best locality that can be achieved.

Instead of relying on complex and often inaccurate machine models, we propose to take the reverse approach and learn the machine characteristics in practice from the best performing schedules for a program. We also propose to link this information to the input program by abstracting away the connection to the schedule coefficients. The information stored is no longer *which coefficients values* are giving good performance, but *how to compute coefficients values* to efficiently optimize the program.

To complement such an approach, it is interesting to test also for several candidates in the subspaces corresponding to the different values for each metric. The goal is to determine for the input program if a given metric is representative of the program performance. This helps to order the importance of the metrics on the target machine.

We build on the framework presented in this thesis to present in Section 9.4 a technique to generate a very large set of input programs.

9.4 Training Set and Performance Metrics

To guarantee the efficiency of a learning-based approach it is required to train the model on a representative and large set of benchmarks. The training phase consists in running the iterative compilation process on several candidate programs, to determine which optimization(s) performs best for it. The information about the performance of a given transformation is analyzed by the learning model to successively refine its decision mechanism about program optimization. Although several previous works use standard

benchmark suites [107, 3], we require to learn on static control program parts. We propose to decouple the quest of a benchmark suite into two steps.

9.4.1 Starting Benchmarks

We use as a starting basis the set of kernels and programs that have a purely static control part in the hot spot(s) of the code. Many of them have been used all along this thesis, in particular in Chapter **??** to Chapter 8 and span most of the standard patterns in extended linear algebra. The reader may refer to Figure 5.5, Figure 5.8 and Figure 8.4 to get information about around 30 benchmarks fitting the requirements of static control parts. Note that all benchmarks (except `radar`) are publicly available in the distribution of the PoCC compiler.

In addition, it is required to gather programs that have not been explicitly dealt with in this thesis but represent a significant source of interest for the high performance community. These are typically stencil computations [21] and signal processing codes. We believe the community of polyhedral compilation is crucially missing a coherent and shared set of benchmarks. It is a short term planned effort to provide such a suite, which would enable a fair comparison of the optimization algorithms and would be a starting training set for learning algorithms.

9.4.2 A Potentially infinite Set of Training Benchmarks

We have presented in this thesis algorithms and tools to build and traverse very large and expressive optimization spaces. These algorithms can also be used to generate new input benchmarks. As for each candidate version another syntactic code may be generated, starting from the collection of benchmarks presented above a potentially infinite collection of programs can be produced. For each input benchmark, thousands of versions can be generated by applying arbitrary legal transformations on them. The syntactic code which is produced is then brought back to a polyhedral representation, creating another benchmark.

The advantages of this technique are twofold. First, as we consider several transformations of the input program, a validating result for the training process is to obtain a performance similar to the optimized original one on all its variations. Second, the variability of the generated versions is extremely large. We can create arbitrarily complex problems in the input code, increasing the challenge for the optimization algorithms. As we use several times a program doing the same computation, only written in a different way, there is a risk of over-fitting for the learning algorithm. Standard techniques such as cross-validation are mandatory.

9.4.3 Performance Metrics

To gather information about the performance of a given optimization, several metrics should be considered. Depending on the target architecture they may not be prioritized the same way. For instance, in high-performance computing the dominant metrics are execution time and power consumption, but in embedded computing code size may be critical. We propose the following metrics.

- *Execution time.* Total number of cycles per core, total execution time for the program.

- *Memory behavior.* Number of cache hit / cache miss, for each layer of the memory hierarchy, and for instruction and data caches. Translation Lookaside Buffer (TLB) hit / miss is also a relevant metric.
- *Parallelism.* Number of coarse-grain parallel loops and number of vectorizable loops. For each parallel loop the number of iterations that can be run in parallel should be reported.
- *Code size.* Size of the program, in its compiled form.

Several techniques do exist to collect dynamic information about the program execution. In this thesis we have used the Performance Application Programming Interface (PAPI) library[1] which has proved to provide precise information based on performance counters available directly on the chip. For the static characteristics such as parallelism one can simply inspect the syntactic transformed code and apply a parallelism detection step based on the dependence graph, on each loop. It is worth noting that not all production compilers support annotations about dependence information, and so may not be able to parallelize a loop that is indeed detected and annotated as parallel in the output syntactic program. At the time of writing of this thesis, the practical user is encouraged to perform coarse-grain parallelization via OpenMP pragmas for instance, and to carefully monitor if the compiler was able to detect SIMD loops exposed in the program.

9.5 Putting It All Together

To build an efficient auto-tuned compiler we rely on an initial training stage, during which we make a chosen model learn how to optimize programs. We glue in Section 9.5.1 the blocks depicted previously and outline the procedure to train the model. Once the model is trained — or equivalently once the compiler has finished to install — the compiler is ready for use. When asked to compile a new program, an optimization for it has to be computed. We outline this process in Section 9.5.2.

9.5.1 Training Phase

During the training phase, the goal is to learn how to characterize the importance of abstract metrics such as parallelism and locality on the program performance. To achieve this objective, we propose to test a wide spectrum of transformations for each input program. The following procedure depicts the process for a given input training program:

1. compute the vector of optimal values OV for the abstract metrics selected to represent the program (parallelism and locality metrics);
2. pick a semantics-preserving transformation for the program;
3. compute the vector of abstract metrics values TV for the transformed program;
4. run the transformed code on the target machine, collect its performance;
5. feed the learning model with OV, TV and the performance.

[1] `http://icl.cs.utk.edu/papi`

The model is expected to learn, for this particular program, how critical it is to exhibit parallelism (both inner and outer level), and how critical it is to improve the program locality. For instance, if an equivalent quantity of schedules with poor and good locality gives a very good performance, then it is determined that locality improvement does not drive the choice of the optimization.

As the metrics are defined for the training phase at the finest granularity (for instance, on a per memory access basis), the information must be generalized to program-independent metrics. Considering a program with 4 loops, the model would decide for instance that it is of high importance to simdize loops 1 and 3, while the four loop nests are simdizable (optimum is then 4). This generalization is an open problem, but one care-free solution to evaluate is to simply consider the average value of each individual metrics, here that half of the loops should be simdized.

The process is repeated for each training program, using of course standard cross-validation techniques as well as feeding with randomly transformed versions of the initial set of benchmarks, as explained above.

This proposal of training process can be very long. We believe that time constraints for this learning phase are not relevant at this stage. Hence, we consider we may test for an almost unlimited number of candidate transformations for a given program. It is expected that tuning this stage is a critical matter to reduce the training time, but it is left as a future work to investigate the solutions to do so.

9.5.2 Compiling a New Program

The final step is to build a transformation for a new program. So far, we have described in a very abstract way how the information is stored and used to optimize a new program. We now detail this procedure with an example.

Consider again the example of Figure 9.2, where we have learned on the standard matrix multiply code (left), and the compiler is now asked to compile the optimized one (right). Remember we are using a SIMD-capable single-core machine. The syntactic and polyhedral features of the program are computed as described in Section 9.2, and the model correlates the new program with the standard matrix-multiply. During the training phase, it has been determined that SIMD is critical (all statements must be simdized to get good performance, the metrics value is 1), coarse-grain parallelism does not impact performance (metrics value is 0).

Now to compute the transformation for the new program, one can proceed as follows. We first build the set of all legal affine multidimensional schedules for the program. We then refine the set with additional constraints, such as inner-most parallelism in this case. In the general case, this leaves us with a large family of candidate transformations that respects the criteria. Two options can be followed. First, one can resort to iterative compilation in the subset of computed transformations. As we have pruned the space and isolated a subspace of potentially good transformations, it is expected that such a process can converge quickly towards a good performing one. The other possibility is to resort to additional cost functions to instantiate the schedule coefficients, as proposed in Chapter 8.

Clearly, the decision process of performing the final section of the schedule is another open problem, which is of critical importance. For instance, what happens if we can only vectorize statements 1 and 3, or 2 and 4, which one do we select? As a future work of this thesis we wish to investigate the implementation and evaluation of the framework proposed in this chapter. We acknowledge that many open problems remain to be solved, in particular the process of computing a transformation for a new program deserves significant research to attain a high efficiency in the general case. Still we believe we

have provided some significant blocks for machine-learning assisted compilation, and as a starter this infrastructure can be used to speed the iterative compilation process.

Chapter 10

Conclusions

From the age of punched tape computers to present-day power efficient multi-core heterogeneous platforms, a dominant task has been to use the computational resources efficiently. This task, initially resorting to programmers, is progressively relying more and more on *optimizing compilers*. For decades, chip architects have used the most advanced ideas to beat the memory wall, but they are now facing hard the power wall. As a consequence chips are going multi-core, requiring compilers to automatically parallelize sequential legacy applications. But another problematic consequence of the increased chip complexity is the performance gap between standard compiler output and the machine's theoretical capabilities.

Present day compilers usually fail to model the complex interplay between different optimizations and their effect on all the different processor architectural components. Also, the complexity of current hardware has made it impossible for compilers to accurately model architectures analytically. Thus, empirical search has become a valid alternative to achieve portable high performance on most modern architectures.

Iterative compilation consists in successively testing for different candidate optimizations on the target machine, measuring their actual performance instead of relying on often inaccurate performance models. Most iterative compilation techniques target compiler optimization flags, parameters, decision heuristics, or phase ordering. We take a more aggressive stand, aiming for the construction and tuning of complex sequences of transformations.

We observe that previous iterative compilation processes have been limited by the expressiveness of the transformation framework they use. In most related work the key contribution lies in the design and analysis of efficient search techniques, more than in the quest for a unified framework for program optimization. The search space is usually made of fixed-length sequences of the optimization primitives offered by the native compiler (for instance, loop unroll, loop tiling, loop interchange). Several problems are encountered by such approaches.

- *Legality of the sequence.* Some generated sequences may not respect the program semantics, and thus not be applied. And since limiting to always legal transformations is dramatically reducing the optimization range, it is no better solution.

- *Uniqueness of the sequence.* Some sequences may lead to identical code due to the commutativity and associativity properties of several loop transformations. Bounding the size of the sequence does not help.

- *Applicability of the sequence.* Given a sequence of transformation that is not semantics-preserving, one may in many situation be able to compute a series of complementary transformations to make the considered sequence legal. Most existing techniques miss the ability and expressiveness of the polyhedral representation to be able to compute such complementary sequences.

We present in this thesis a unified search space where all those critical problems are solved. The polyhedral program representation is the most powerful tool to model arbitrarily complex loop transformations into a single search space. In addition, we propose to use an *all-in-one approach* for the search space construction, to significantly help any search process to focus on relevant candidates only. For a given search space, we guarantee as a property for all candidates that (1) they each preserve the program semantics; (2) they each lead to distinct transformed programs; and (3) expressiveness is maximized by considering all combinations of transformations in a given category.

We presented in Chapter 3 a *convex representation of the set of all, legal and distinct multidimensional schedules* with bounded coefficients, as a starting block for the search space construction. By encompassing all possible affine loop transformations into a single loop optimization step, we have dramatically extended the expressiveness compared to standard iterative compilation approaches. Loop fusion and loop distribution are two key transformations which application should be considered out of the other complementary transformations — those transformations required to actually fuse or distribute statements, such as skewing and peeling. We have contributed in Chapter 3 an affine representation of the set of *all, legal and distinct multidimensional statement interleavings*. This generalization of loop fusion, called fusability, results in a dramatic broadening of the expressiveness (hence of expected effectiveness) of the optimizer.

Several other building blocks for the design of a robust and portable iterative compilation process are required. A critical concern is scalability, in other words, the ability of the iterative process to effectively converge towards a good solution. In this spirit, we focused on providing all the required mechanisms for an efficient search space construction and traversal.

We presented in Chapter 5 efficient techniques to build search spaces of multidimensional schedules. We have addressed the problem of *building practical search spaces*, navigating the trade-off between expressiveness and optimality of the solution versus tractability of the space construction and its traversal. To assess the relevance of our approach, we have extensively evaluated the *performance distribution of affine multi-dimensional schedules* contained in this space in Chapter 6. We evaluated numerous programs on distinct architectures typical from desktop and embedded processors (x86, VLIW, MIPS32) and concluded key observations on the performance distribution. We have experimentally validated a partitioning of the space of affine schedules, by *ordering the performance impact of several classes of schedule coefficients*.

We proposed several heuristics to enumerate only these high-potential subspaces in the set of affine schedules, to discover efficient program transformations in Chapter 7. We have provided a *schedule completion mechanism* leveraging the static and dynamic characteristics of the search space, enabling to complete or correct any partial schedule to make it lie in the space of legal transformations. We contributed a heuristic to discover the *wall-clock optimal schedule* for the case of one-dimensional schedules, and extended this work to the case of multidimensional schedules. To further improve scalability on the largest search spaces, we contributed the *first genetic operators for loop transformations which are closed under semantics-preservation*. We provided experimental evidence of the efficiency of the iterative process in optimizing programs, by testing on three different single-core architectures. Our processes systematically outperform the native compilers, including Intel ICC, whatever the target architecture by up to an

order of magnitude faster.

Tiling (or blocking) is a crucial loop transformation for parallelism and locality. The downside is that it is not an affine transformation, as it requires to alter the polyhedral representation to be performed. It is known that tiling is legal if loops are permutable, and efficient algorithms for parallelism via tiling in the polyhedral model have recently been proposed by Bondhugula. We extended his approach in Chapter 8 by allowing for an *iterative search of multidimensional statement interleavings*. We have tackled the main limitation of this model-driven approach, by offering the optimizer the possibility to adapt for any architecture the fusion structure of the program. Furthermore, we have removed the applicability constraints of this technique, by enabling the discovery of permutable loops in the presence of parametric dependences. As a result, parallel tiled optimized code is produced, and our experiments on three high-end modern multi-core machines (8, 16 and 24 hardware threads) concluded improvements of up to *two orders of magnitude faster programs* when compared to the native auto-parallelizing compilers or the initial model-driven approach.

The last building block for the design of an efficient iterative optimizer in the polyhedral model relates to the manipulation of large polyhedra. As we have designed search spaces as convex sets, we face the problem of scanning polytopes of high dimensionality, orders of magnitude larger than those manipulated by other polyhedral frameworks. To reach scalability, we have exhibited the key properties required on the space to enable the design of a *linear-time scanning procedure* of the search space. We have motivated the chosen representation for these polyhedra, and built simplification mechanisms to perform a *redundancy-less projection with the Fourier-Motzkin algorithm*. Our experiments showed that redundancy was the dominant bottleneck of this algorithm, and with a redundancy control is a scalable technique to perform polyhedral projection on our problem instances.

We have gathered all the theoretical and practical contributions of this thesis into a set of software applications dedicated to polyhedral compilation. During this thesis we have developed *FM, the Fourier-Motzkin Library* to enable the manipulation of high-dimensionality polyhedra. All search space construction and traversal techniques have been implemented in *LetSee, the Legal Transformation Space Explorator*. To easily use and evaluate these tools, we also developed *PoCC, the Polyhedral Compiler Collection*, which offers a full source-to-source iterative compiler in the polyhedral model. All these tools are freely available for download and are already used by several other research projects.

As a result, we have designed, implemented and evaluated a *machine-independent optimizer computing machine-dependent optimizations*. Still, there exists numerous possible improvements for our techniques. Specifically, we identify the following sources for improvement.

- *Knowledge transfer.* In the processes we have designed, no knowledge is extracted to help the compilation of another program. Each time the iterative process completes, the optimization information coming from this compilation is lost.

- *Iterative search.* The search process may require numerous runs before converging. An ultimate goal is to reduce to the minimum the number of runs. And an alternative is to evaluate the potential of substituting the program execution step with an off-line trained model for performance evaluation.

- *Scalability.* We experimentally observed that the complexity of the search process is connected with the number of polyhedral statements in the program. As the size of the program increases, so does the compilation time.

The scalability issue is directly connected to the iterative search processing time. If we can exhibit techniques to control the iterative search to operate successfully on a fixed and small number of runs (say, at most 10 runs and at best 0), then scalability would *de facto* be achieved too. In Chapter 9 we presented novel ideas to *harness the power of polyhedral optimization in a machine learning assisted compiler*. We believe that the knowledge transfer, the iterative search and the scalability issues can all be solved within a single approach for optimization based on machine learning. We provided numerous ideas to reach this goal, and a short term objective is to build upon these concepts to design an even more effective compiler for the current and upcoming chip generations.

Appendix A

Correctness Proof for $\mathcal{O}$

We restate the expression of $\mathcal{O}$, the affine set of all, distinct total preorders of n elements. For $1 \le i < n$, $i < j \le n$, $\mathcal{O}$ is:

$$\begin{cases}
\left.\begin{array}{rr} & 0 \le p_{i,j} \le 1 \\ & 0 \le e_{i,j} \le 1 \end{array}\right\} & \text{Variables are binary} \\
\left.\begin{array}{rr} & p_{i,j} + e_{i,j} \le 1 \end{array}\right\} & \text{Relaxed mutual exclusion} \\
\left.\begin{array}{lr} \forall k \in]j,n] & e_{i,j} + e_{i,k} \le 1 + e_{j,k} \\ & e_{i,j} + e_{j,k} \le 1 + e_{i,k} \end{array}\right\} & \text{Basic transitivity on } e \\
\left.\begin{array}{lr} \forall k \in]i,j[& p_{i,k} + p_{k,j} \le 1 + p_{i,j} \end{array}\right\} & \text{Basic transitivity on } p \\
\left.\begin{array}{lr} \forall k \in]j,n] & e_{i,j} + p_{i,k} \le 1 + p_{j,k} \\ & e_{i,j} + p_{j,k} \le 1 + p_{i,k} \\ \forall k \in]i,j[& e_{k,j} + p_{i,k} \le 1 + p_{i,j} \end{array}\right\} & \text{Complex transitivity on } p \text{ and } e \\
\left.\begin{array}{lr} \forall k \in]j,n] & e_{i,j} + p_{i,j} + p_{j,k} \le 1 + p_{i,k} + e_{i,k} \end{array}\right\} & \text{Complex transivity on } s \text{ and } p
\end{cases}$$

We want to prove that *the set $\mathcal{O}$ contains one and only one point per distinct total preorder of n elements.*

Proof. We first prove that $\mathcal{O}$ contains all and only total preorders, before proving the uniqueness of preorders in $\mathcal{O}$.

From the encoding through s, p and e variables chosen, at least all total preorders are represented in the initial set

$$\left\{ \begin{array}{c} 0 \le p_{i,j} \le 1 \\ 0 \le e_{i,j} \le 1 \\ 0 \le s_{i,j} \le 1 \end{array} \right\}$$

This is because all possible combinations for elements i, j can be represented ($i < j$, $i > j$ or $i = j$) with the proposed representation. We now show that the successive constraints added to prune the set remove all points that are not a valid total preorder. To prove so, we rely on the fact that a total preorder relation is a relation which is total, transitive and symetric. Hence, we prove that our constraints are sufficient to guarantee to preserve the totality, the transitivity and the reflexivity of the relation.

Totality: Given x, y two elements of a set S of n elements on which the total preorder relation is defined. Without any loss of generality and for the rest of the proof, we assume that S is the set of consecutive integers from 1 to n. Given a, b their position identifier as specified by the preorder. Totality gives:

$$a \preceq b \ \vee \ b \preceq a$$

Either $a < b$, $a = b$, $b < a$ or $b = a$ which is equivalent to $a = b$. This is guaranteed by the relaxed mutual exclusion inequality.

Transitivity: Given x, y, z three elements and a, b, c their respective partition identifier. Transitivity gives:

$$a \preceq b \ \wedge \ b \preceq c \ \Rightarrow \ a \preceq c$$

That is, one of the following configuration must occur:

1. $a < b \ \wedge \ b < c \ \Rightarrow \ a < c$
2. $a < b \ \wedge \ b = c \ \Rightarrow \ a < c$
3. $a = b \ \wedge \ b < c \ \Rightarrow \ a < c$
4. $a = b \ \wedge \ c < b \ \Rightarrow \ c < a$
5. $a = b \ \wedge \ b = c \ \Rightarrow \ a = c$
6. $b < a \ \wedge \ c < b \ \Rightarrow \ c < a$
7. $b < a \ \wedge \ b = c \ \Rightarrow \ c < a$

Converting 1. into our encoding gives:

$$p_{x,y} \ \wedge \ p_{y,z} \ \Rightarrow p_{x,z} \tag{A.1}$$

To generalize this constraint to the n possible elements, we must then consider the different possible values for x, y, z: we can have $x < y$ or $x > y$, $x < z$ or $x > z$, and $y < z$ or $z < y$. We start by focusing only on the case where $x < y < z$. (A.1) is written:

$$\forall \ 1 \leq i < k < j \leq n, \quad p_{i,k} \ \wedge \ p_{k,j} \ \Rightarrow p_{i,j} \tag{A.2}$$

This equation can be converted in an affine form in a deterministic fashion (one can use a Boolean table to ensure the constraint defines an equivalent logic as the implication), and once converted in an affine form corresponds to the basic transitivity of p coefficients inequalities shown in the definition of $\mathcal{O}$.

Converting 5. into our encoding gives:

$$e_{x,y} \ \wedge \ e_{y,z} \ \Rightarrow e_{x,z} \tag{A.3}$$

For this case, if $x < y < z$, then (A.3) is written:

$$\forall 1 \leq i < j < k \leq n, \quad e_{i,j} \ \wedge \ e_{j,k} \ \Rightarrow e_{i,k}, \tag{A.4}$$

If $y < x < z$, then (A.3) is written:

$$\forall 1 \leq i < j < k \leq n, \quad e_{i,j} \wedge e_{i,k} \Rightarrow e_{j,k}, \tag{A.5}$$

These equations once converted in an affine form correspond to the basic transitivity of e coefficients.

Converting 3. into our encoding gives:

$$e_{x,y} \wedge p_{y,z} \Rightarrow p_{x,z} \tag{A.6}$$

If $x < y < z$, then (A.6) is written:

$$\forall 1 \leq i < j < k \leq n, \quad e_{i,j} \wedge p_{i,k} \Rightarrow p_{j,k}, \tag{A.7}$$

If $y < x < z$, then (A.6) is written:

$$\forall 1 \leq i < j < k \leq n, \quad e_{i,j} \wedge p_{j,k} \Rightarrow p_{i,k}, \tag{A.8}$$

Converting 2. into our encoding gives:

$$e_{y,z} \wedge p_{x,y} \Rightarrow p_{x,z} \tag{A.9}$$

If $x < z < y$, then (A.9) is written:

$$\forall 1 \leq i < k < j \leq n, \quad e_{k,j} \wedge p_{i,k} \Rightarrow p_{i,j}, \tag{A.10}$$

Equations (A.7), (A.8) and (A.10) correspond to the complex transitivity constraints on the p and e variables.

Converting 6. into our encoding gives:

$$s_{x,z} \wedge p_{y,z} \Rightarrow s_{x,y} \tag{A.11}$$

If $x < y < z$, then (A.11) is written (thanks to the substitution coming from the mutual exclusion equation):

$$\forall 1 \leq i < j < k \leq n, \quad \neg e_{i,j} \wedge \neg p_{i,j} \wedge p_{j,k} \Rightarrow \neg p_{i,k} \wedge \neg e_{i,k}, \tag{A.12}$$

This equations corresponds to the complex transitivity constraints on the p and s variables.

Several cases have not been explicitely addressed, because they are either equivalent to the above-mentionned affine constraints, or non-contributing to the pruning of the space. Their enumeration and computing their equivalence with the presented cases is left to the motivated reader.

All necessary conditions for transitivity have been enforced in $\mathcal{O}$.

Reflexivity: Reflexivity is trivially satisfied in our encoding.

This concludes proving that all points in $\mathcal{O}$ is a total preorder, and that all total preorders are in $\mathcal{O}$.

We must now prove that there is only one point in $\mathcal{O}$ per distinct total preorder.

Uniqueness: To prove so, we show there exists a bijection $f : \mathcal{P} \to \mathcal{O}$ between the set of distinct total preorders $\mathcal{P}$ and $\mathcal{O}$.

We first prove that it is not possible that two distinct preorders are represented by the same point in $\mathcal{O}$. Suppose there exists two distinct total preorders p_1 and p_2 such that

$$f(p_1) = f(p_2) \wedge p_1 \neq p_2$$

By construction of the encoding, two distinct preorders result in at least one modification of a variable ($e_{i,j}$ and/or $p_{i,j}$) used to encode the preorder. Hence we must have:

$$f(p_1) = f(p_2) \Rightarrow p_1 = p_2$$

which is a contradiction.

To show that it is not possible to have two distinct points in $\mathcal{O}$ representing the same total preorder, we again rely on our encoding definition. This concludes the proof of Lemma 3.6. ∎

Personal Bibliography

Peer-reviewed international conferences:

- Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen and Cédric Bastoul. The Polyhedral Model is more widely applicable than you think. In *ETAPS Conference on Compiler Construction (CC'10)*, Paphos, Cyprus, March 2010. Springer Verlag.
- Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.
- Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *Fifth International Symposium on Code Generation and Optimization (CGO'07)*, pages 144–156, San Jose, California, March 2007. IEEE Computer Society Press.
- Nicolas Vasilache, Albert Cohen, and Louis-Noël Pouchet. Automatic correction of loop transformations. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*, pages 292–304, Brasov, Romania, September 2007. IEEE Computer Society Press.
- Thomas Claveirole, Sylvain Lombardy, Sarah O'Connor, Louis-Noël Pouchet, and Jacques Sakarovitch. Inside vaucanson. In *Implementation and Application of Automata, 10th International Conference (CIAA'05)*, volume 3845 of *Lecture Notes in Computer Science*, pages 116–128, Sophia Antipolis, France, 2006. Springer Verlag.

Peer-reviewed international journals:

- 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.

Peer-reviewed international workshops:

- K. Ibrahim, J. Jaeger, Z. Liu, L.N. Pouchet, P. Lesnicki, L. Djoudi, D.Barthou, F. Bodin, C. Eisenbeis, G. Grosdidier, O. Pene, and P. Roudeau. Simulation of the Lattice QCD and technological trends in computation. In *14th Workshop on Compilers for Parallel Computing (CPC'09), Zurich, Switzerland*, January 2009.

- Louis-Noël Pouchet, Cédric Bastoul, John Cavazos, and Albert Cohen. A note on the performance distribution of affine schedules. In *2nd Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion (SMART'08)*, Göteborg, Sweden, January 2008.

- Louis-Noël Pouchet, Cédric Bastoul, and Albert Cohen. Iterative optimization in the polyhedral model: the one-dimensional affine scheduling case. In *2nd HiPEAC Industrial Workshop*, Eindhoven, the Netherlands, October 2006.

Research Reports:

- Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Hybrid iterative and model-driven optimization in the polyhedral model. Report 6962, INRIA Research Report, June 2009.

- Mohamed-Walid Benabderrahmane, Cédric Bastoul, Louis-Noël Pouchet, and Albert Cohen. A conservative approach to handle full functions in the polyhedral model. Report 6814, INRIA Research Report, January 2009.

Other publications:

- Louis-Noël Pouchet, Cédric Bastoul, and Albert Cohen. Letsee: the legal transformation space explorator. Third International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES'07), L'Aquila, Italia, July 2007. Extended abstract, pp 247–251.

- Louis-Noël Pouchet. When iterative optimization meets the polyhedral model: One-dimensional date. Master's thesis, University of Paris-Sud 11, Orsay, France, 2006.

Bibliography

[1] The liège automata-based symbolic handler (LASH). Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.

[2] Ppl: The parma polyhedra library. `http://www.cs.unipr.it/ppl/`.

[3] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proc. of the Intl. Symposium on Code Generation and Optimization (CGO'06)*, pages 295–305, Washington, 2006.

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

[5] L. Almagor, K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, pages 231–239, New York, 2004.

[6] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50, June 1991.

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

[8] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. 40:210–226, 1997.

[9] A. I. Barvinok. Computing the ehrhart polynomial of a convex lattice polytope. *Discrete and Computational Geometry*, 12(1):35–48, December 94.

[10] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16, Juan-les-Pins, France, September 2004.

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

[12] 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.

[13] C. Bastoul and P. Feautrier. Improving data locality by chunking. In *Intl. Conf. on Compiler Construction (ETAPS CC 12)*, volume 2622, pages 320–335, Warsaw, Poland, April 2003.

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

[15] Anna Beletska. *Extracting synchronization-free parallelism with the slicing framework*. PhD thesis, Politecnico Di Milano, 2009.

[16] Anna Beletska, Wlodzimierz Bielecki, Albert Cohen, and Marek Palkowski. Synchronization-free automatic parallelization: Beyond affine iteration-space slicing. In *LCPC'22 Intl. Workshop on Languages and Compilers for Parallel Computing*, October 2009.

[17] Mohamed-Walid Benabderrahmane, Cédric Bastoul, Louis-Noël Pouchet, and Albert Cohen. A conservative approach to handle full functions in the polyhedral model. Technical Report 6814, INRIA Research Report, January 2009.

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

[19] Bernard Boigelot and Jean-François Degbomont. Partial projection of sets represented by finite automata, with application to state-space visualization. In *LATA '09: Proceedings of the 3rd International Conference on Language and Automata Theory and Applications*, pages 200–211, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International conference on Compiler Construction (ETAPS CC)*, April 2008.

[21] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2008.

[22] J. Cavazos, J. E. Moss, and M. F. P. O'Boyle. Hybrid optimizations: Which optimization algorithm to use. In *(CC'06)*, Vienna, Austria, April 2006.

[23] John Cavazos and J. Eliot B. Moss. Inducing heuristics to decide whether to schedule. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 183–194, New York, NY, USA, 2004. ACM.

[24] Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, U. of Southern California, 2008.

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

[26] N.V. 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.

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

[28] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: applications to analyze and transform scientific programs. In *ICS '96: Proceedings of the 10th international conference on Supercomputing*, pages 278–285, New York, NY, USA, 1996. ACM.

[29] Alan Cobham. On the base-dependence of sets of numbers recognizable by finite automata. *Theory of Computing Systems*, 3(2):186–192, June 1969.

[30] Albert Cohen, Sylvain Girbal, David Parello, M. Sigler, Olivier Temam, and Nicolas Vasilache. Facilitating the search for compositions of program transformations. In *ACM International conference on Supercomputing*, pages 151–160, June 2005.

[31] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. ACME: adaptive compilation made efficient. In *ACM SIGLPAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, pages 69–77, Chicago, IL, USA, 2005. ACM Press.

[32] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, Atlanta, GA, USA, July 1999. ACM Press.

[33] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002.

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

[35] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser, 2000.

[36] Alain Darte and Guillaume Huard. Loop shifting for loop parallelization. Technical Report RR2000-22, ENS Lyon, May 2000.

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

[38] Alain Darte, Georges-Andre Silber, and Frederic Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Proc. Letters*, 7(4):379–392, 1997.

[39] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[40] P. Feautrier. Dataflow analysis of scalar and array references. *Intl. J. of Parallel Programming*, 20(1):23–53, February 1991.

[41] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I: one dimensional time. *Intl. J. of Parallel Programming*, 21(5):313–348, October 1992.

[42] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, December 1992.

[43] P. Feautrier. Scalable and modular scheduling. Technical Report 19, École Normale Supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme, April 2004.

[44] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. Formal loop merging for signal transforms. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'05)*, pages 315–326. ACM Press, 2005.

[45] B. Franke and M. O'Boyle. Array recovery and high level transformations for dsp applications. In *CPC'10 Intl. Workshop on Compilers for Parallel Computers*, pages 29–38, Amsterdam, January 2003.

[46] Marc Le Fur. Scanning parameterized polyhedron using fourier-motzkin elimination. *Concurrency: Practice and Experience*, 8(6):445–460, November 1996.

[47] Grigori Fursin. Collective tuning initiative: automating and accelerating development and optimization of computing systems. In *Proceedings of the GCC Developers' Summit*, June 2009.

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

[49] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, Francois Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, and Michael O'Boyle. Milepost gcc: machine learning based research compiler. In *Proceedings of the GCC Developers' Summit*, June 2008.

[50] Grigori Fursin and Olivier Temam. Collective optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009)*, January 2009.

[51] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[52] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1989.

[53] M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. Facultät für mathematik und informatik, universität Passau, 2004.

[54] M. Griebl, P. Faber, and C. Lengauer. Space-time mapping and tiling – a helpful combination. *Concurrency and Computation: Practice and Experience*, 16(3):221–246, March 2004.

[55] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 106–111, 1998.

[56] Armin Größlinger, Martin Griebl, and Christian Lengauer. Introducing non-linear parameters to the polyhedron model. In *Proc. 11th Workshop on Compilers for Parallel Computers (CPC 2004)*, July 2004.

[57] Gautam Gupta and Sanjay Rajopadhye. The z-polyhedral model. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 237–248. ACM Press, 2007.

[58] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 123–132, Saint Louis, MO, USA, 2005. IEEE Computer Society.

[59] Tien Huynh, Catherine Lassez, and Jean-Louis Lassez. Practical issues on the projection of polyhedral sets. *Journal Annals of Mathematics and Artificial Intelligence*, 6(4):295–315, December 92.

[60] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329, New York, NY, USA, 1988. ACM.

[61] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: an overview of the pips project. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 244–251, New York, NY, USA, 1991. ACM.

[62] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, 1967.

[63] W. Kelly. *Optimization within a Unified Transformation Framework*. PhD thesis, Univ. of Maryland, 1996.

[64] W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, Department of Computer Science, University of Maryland at College Park, 1996.

[65] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Intl. Symp. on the frontiers of massively parallel computation*, pages 332–341, McLean, VA, USA, February 1995.

[66] 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, 1993.

[67] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 237–246, Philadelphia, PA, USA, 2000. IEEE Computer Society.

[68] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN'97 Conf. on Programming Language Design and Implementation*, pages 346–357, Las Vegas, June 1997.

[69] P. Kulkarni, W. Zhao, D. Whalley, X. Yuan, R. van Engelen, K. Gallivan, J. Hiser, J. Davidson, B. Cai, M. Bailey, H. Moon, K. Cho, Y. Paek, and D. Jones. Vista: Vpo interactive system for tuning applications. *ACM Transactions on Embedded Computing Systems*. To appear.

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

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

[72] Louis Latour. From automata to formulas: Convex integer polyhedra. *lics*, 00:120–129, 2004. A journal version was submitted to LMCS latour.06.lmcs.pdf.

[73] Louis Latour. Computing affine hulls over q and z from sets represented by number decision diagrams. In *Tenth International Conference on Implementation and Application of Automata (CIAA'05)*, volume 3845 of *LLNCS*, pages 213–224, 2005.

[74] Hervé Le Verge. A Note on Cherniakova's algorithm. Research Report RR-1662, INRIA, 1992.

[75] Hugh Leather, Edwin Bonilla, and Michael OâĂŹBoyle. Automatic feature generation for machine learning based optimizing compilation. In *IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO'09)*, 2009.

[76] Corinna Lee. UTDSP benchmark suite, 1998.
`http://www.eecg.toronto.edu/~corinna/DSP`.

[77] Richard Lethin, Allen Leung, Benoît Meister, Peter Szilagyi, Nicolas Vasilache, and David Wohlford. Final report on the the r-stream 3.0 compiler. Technical report, Reservoir Labs, Inc. Delivered to Air Force Research Laboratory, Rome, NY, 2008. For Contract F03602-03-C-0033.

[78] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Intl. J. of Parallel Programming*, 22(2):183–205, April 1994.

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

[80] S. Long and M.F.P. O'Boyle. Adaptive Java optimisation using instance-based learning. In *ACM Intl. Conf. on Supercomputing (ICS'04)*, pages 237–246, Saint-Malo, France, June 2004.

[81] Shun Long and Grigori Fursin. Systematic search within an optimisation space based on unified transformation framework. *IJCSE Intl. Journal of Computational Science and Engineering*. To appear.

[82] Shun Long and Grigori Fursin. A heuristic search algorithm based on unified transformation framework. In *Proc. of the 2005 Intl. Conf. on Parallel Processing Workshops (ICPPW'05)*, pages 137–144, Washington, DC, USA, 2005. IEEE Comp. Soc.

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

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

[85] Nimrod Megiddo and Vivek Sarkar. Optimal weighted loop fusion for parallel programs. In *symposium on Parallel Algorithms and Architectures*, pages 282–291, 1997.

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

[87] Andy Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In *HPCN Europe 1998: Proc. of the Intl. Conf. and Exhibition on High-Performance Computing and Networking*, pages 987–989, London, UK, 1998. Springer-Verlag.

[88] Boyana Norris, Albert Hartono, Elizabeth Jessup, and Jeremy Siek. Generating empirically optimized composed matrix kernels from MATLAB prototypes. In *Int. Conf. on Computational Science (ICCS'09)*, may 2009.

[89] M. Palkovič. *Enhanced Applicability of Loop Transformations*. PhD thesis, T. U. Eindhoven, The Netherlands, September 2007.

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

[91] Louis-Noël Pouchet. Fm, the Fourier-Motzkin library. Available for download at `http://www-rocq.inria.fr/~pouchet/software/fm`.

[92] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *ACM Intl. Conf. on Supercomputing (ICS'91)*, pages 4–13, Albuquerque, NM, USA, August 1991.

[93] William Pugh and Evan Rosser. Iteration space slicing and its application to communication optimization. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 221–228. ACM Press, 1997.

[94] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *J. of High Performance Computing and Applications, special issue on Automatic Performance Tuning*, 18(1):21–45, 2004.

[95] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proc. of the 20th Intl. Conf. on Supercomputing (ICS'06)*, pages 249–258. ACM press, 2006.

[96] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. J. of Parallel Programming*, 28(5):469–498, October 2000.

[97] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *The Journal of VLSI Signal Processing*, 1(2):95–113, October 1989.

[98] J. Ramanujam. Beyond unimodular transformations. *J. Supercomputing*, 9(4):365–389, 1995.

[99] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–230, 1992.

[100] Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally. A tuning framework for software-managed memory hierarchies. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'08)*, pages 280–291. ACM Press, 2008.

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

[102] Rachid Seghir and Vincent Loechner. Memory optimization by counting points in integer transformations of parametric polytopes. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 74–82, New York, NY, USA, 2006. ACM.

[103] S. Singhai and K. McKinley. A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality. *The Computer Journal*, 40(6):340–355, 1997.

[104] N. J. A. Sloane and Simon Plouffe. *The Encyclopedia of Integer Sequences*. Academic Press, 1995.

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

[106] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *ACM Intl. Symp. on Code Generation and Optimization (CGO'05)*, pages 123âĂŞ–134, 2005.

[107] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Notices*, 38(5):77–90, 2003.

[108] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable autotuning framework for computer optimization. In *IPDPS'09*, Rome, May 2009.

[109] S. Triantafyllis, M. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *J. of Instruction-level Parallelism*, volume 7, January 2005.

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

[111] Nicolas Vasilache. *Scalable Program Optimization Techniques in the Polyhedra Model*. PhD thesis, University of Paris-Sud 11, 2007.

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

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

[114] Nicolas Vasilache, Albert Cohen, and Louis-Noël Pouchet. Automatic correction of loop transformations. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'07)*, pages 292–302, Brasov, Romania, September 2007.

[115] S. Verdoolaege, F. Catthoor, M. Bruynooghe, and G. Janssens. Feasibility of incremental translation. Technical Report CW 348, Katholieke Universiteit Leuven Department of Computer Science, October 2002.

[116] Sven Verdoolaege. *The Integer Set Library*, 2008. http://www.kotnet.org/∼skimo/isl/user.html.

[117] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007.

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

[119] Yevgen Voronenko, Frédéric de Mesmay, and Markus Püschel. Computer generation of general size linear transform libraries. In *Intl. Symp. on Code Generation and Optimization (CGO'09)*, March 2009.

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

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

[122] D. K. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, Rennes, France, 1993.

[123] M. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of computer science, Stanford University, California, 1992.

[124] Michael Wolf, Dror Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286, 1996.

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

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

[127] Pierre Wolper and Bernard Boigelot. An automata-theoretic approach to presburger arithmetic constraints (extended abstract). In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 21–32, London, UK, 1995. Springer-Verlag.

[128] J. Xue. Transformations of nested loops with non-convex iteration spaces. *Parallel Computing*, 22(3):339–368, 1996.

[129] Y. Yaacoby and P.R. Cappello. Scheduling a system of affine recurrence equations onto a systolic array. In *Systolic Arrays, 1988., Proceedings of the International Conference on*, pages 373–382, May 1988.