Iterative Schedule Optimization for Parallelization in the Polyhedron Model

Stefan Ganser

April 29, 2019

Dissertation zur Erlangung des Doktorgrades
der Naturwissenschaften (Dr. rer. nat.)
eingereicht an der Fakultät für Informatik und Mathematik
der Universität Passau

Dissertation Submitted to
the Faculty of Computer Science and Mathematics
of the University of Passau
in Partial Fulfillment of Obtaining
the Degree of a Doctor of Natural Sciences

Betreuer / Supervisor: Prof. Christian Lengauer (PhD)
Externer Gutachter / External Examiner: Dr. Albert Cohen

UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Abstract

In high-performance computing, one primary objective is to exploit the performance that the given target hardware can deliver to the fullest. Compilers that have the ability to automatically optimize programs for a specific target hardware can be highly useful in this context. Iterative (or search-based) compilation requires little or no prior knowledge and can adapt more easily to concrete programs and target hardware than static cost models and heuristics. Thereby, iterative compilation helps in situations in which static heuristics do not reflect the combination of input program and target hardware well. Moreover, iterative compilation may enable the derivation of more accurate cost models and heuristics for optimizing compilers. In this context, the polyhedron model is of help as it provides not only a mathematical representation of programs but, more importantly, a uniform representation of complex sequences of program transformations by schedule functions. The latter facilitates the systematic exploration of the set of legal transformations of a given program.

Early approaches to purely iterative schedule optimization in the polyhedron model do not limit their search to schedules that preserve program semantics and, thereby, suffer from the need to explore numbers of illegal schedules. More recent research ensures the legality of program transformations but presumes a sequential rather than a parallel execution of the transformed program. Other approaches do not perform a purely iterative optimization.

We propose an approach to iterative schedule optimization for parallelization and tiling in the polyhedron model. Our approach targets loop programs that profit from data locality optimization and coarse-grained loop parallelization. The schedule search space can be explored either randomly or by means of a genetic algorithm.

To determine a schedule's profitability, we rely primarily on measuring the transformed code's execution time. While benchmarking is accurate, it increases the time and resource consumption of program optimization tremendously and can even make it impractical. We address this limitation by proposing to learn surrogate models from schedules generated and evaluated in previous runs of the iterative optimization and to replace benchmarking by performance prediction to the extent possible.

Our evaluation on the PolyBench 4.1 benchmark set reveals that, in a given setting, iterative schedule optimization yields significantly higher speedups in the execution of the program to be optimized. Surrogate performance models learned from training data that was generated during previous iterative optimizations can reduce the benchmarking effort without strongly impairing the optimization result. A prerequisite for this approach is a sufficient similarity between the training programs and the program to be optimized.

Contents

List of Symbols, Abbreviations, and Names

Symbol/Abbr./ Name	Description	Page
char.cone(P)	The characteristic cone of polyhedron P.	12
cols(A)	The number of columns of matrix A.	9
cone(S)	The convex cone generated by the set S of vectors.	11
conv.hull(S)	The convex hull of the set S of vectors.	11
dim(f)	The dimensionality of a linearly affine function $f : \mathbb{R}^n \to \mathbb{R}^m$ is m.	10
dim(S)	Let $S \subseteq \mathbb{R}^n$. We define $\dim(S) = n$.	10
F_{Depth}	A feature that expresses the depth of a schedule tree.	98
F_{DataLoc}	A schedule feature that expresses data locality.	105
F_{Leaves}	A feature that expresses the number of leaves in a schedule tree.	95
F_{MemAcc}	A schedule feature that characterizes statements' memory access pattern after the schedule transformation.	106
F_{Par}	A schedule feature that expresses coarse-grained parallelism.	102
F_{Seq}	A feature that expresses the number of sequence nodes in a schedule tree.	96
F_{SpIter}	A feature that expresses the sparsity of the iteration variable coefficients in a schedule.	99
F_{SpP}	A feature that expresses the sparsity of the structure parameter coefficients in a schedule.	100
F_{Tile}	A schedule feature that expresses the applicability of rectangular tiling.	103
lin.hull(S)	The linear hull of the set S of vectors.	11
lin.indep(S)	The set of vectors that are linearly independent of the vectors in S.	54
lin.space(P)	The lineality space of polyhedron P.	12
Γ	The depth that a schedule tree for the current SCoP can have. Uncounted are domain nodes and filter nodes.	97
Δ_{S_i}	The number of loops surrounding statement S_i or 1 in the absence of loops. As a simplification we use the value of I_{S_i}.	93
ν_S	Coefficient of 1 that is related to statement S in a schedule coefficient vector.	38

$\mathbb{R}^{m \times n}$	Space of all real matrices with m rows and n columns.	9
$\text{rows}(A)$	The number of rows of matrix A.	9
$\mathcal{L}$	The number of leaves of a given schedule tree.	93
$\mathcal{S}$	The number of sequence nodes in a given schedule tree.	93
$\mathcal{A}$	The total number of memory access relations in the SCoP. We distinguish between read and write accesses.	93
$\Lambda(S)$	Let $S = \{\vec{x} \mid \vec{x} \in \mathbb{R}^n \;\wedge\; m, n \in \mathbb{N} \;\wedge\; (p_1(\vec{x}) \vee p_2(\vec{x}) \vee ... \vee p_m(\vec{x})\}$ be some set of n-dimensional vectors. We define $\Lambda(S) = \{\{\vec{x} \mid \vec{x} \in \mathbb{R}^n \;\wedge\; p_i(\vec{x})\} \mid i \in \{1, 2, ..., m\}\}$.	11
`IsCstDim` : $\mathbb{N} \to \mathbb{B}$	Tests whether a given natural number is the index of a coefficient for the constant 1 in a schedule's coefficient space.	65
Θ_S	The schedule function of statement S.	20
$\Theta_S^{(d..e)}$	Addresses the partial schedule that consists of dimensions $d, d+1, ..., e$.	20
$\varepsilon x(P(x))$	Selects a value x for which $P(x)$ is true.	65
$\vec{\lambda}_S$	Coefficients for $\vec{i}_S$ in a schedule coefficient vector.	38
$\vec{\mu}_S$	Coefficients for $\vec{p}$ that are associated to statement S in a schedule coefficient vector.	38
$\vec{i}_S$	The vector of the iteration variables of the loops encasing statement S.	21
$\vec{p}$	Vector of a SCoP's structure parameters.	21
$\vec{u} \prec \vec{v}$	Vector $\vec{u}$ is lexicographically smaller than $\vec{v}$.	16
$A \cdot \vec{\beta}$	Matrix-vector product of matrix A and $\vec{\beta}$.	9
$A\vec{\beta}$	Horizontal coupling of vector $\vec{\beta}$ and matrix A.	9
$A^{(\bullet,j)}$	Column j of matrix A.	9
$A^{(i,\bullet)}$	Row i of matrix A.	9
$A^{(i,j)}$	Element in column j of row i of matrix A.	9
$C_\Theta^{1..d}$	Set of dependence polyhedra that are carried by dimensions $1, ..., d$, $d \in \mathbb{N}$ of schedule Θ.	79
$D_{O,T}$	Dependence polyhedron that models a set of dependences between statements O (origin) and T (target).	20
I	Given a SCoP, I is the set of its statements' iteration domains.	93
I_S	Iteration domain of statement S.	19
$read_S^A : \mathbb{Z}^n \to \mathbb{Z}^m$	Function that models a read access from statement S surrounded by n loops to m-dimensional array A.	20

$S_{O,T}$	Set of all coefficient vectors that represent one-dimensional schedules that satisfy $D_{O,T}$ strongly.	40
$U_{\Theta}^{1..d}$	Set of dependence polyhedra that are not carried by dimensions $1, \ldots, d, d \in \mathbb{N}$ of schedule Θ.	79
$W_{O,T}$	Set of all coefficient vectors that represent one-dimensional schedules that satisfy $D_{O,T}$ at least weakly.	40
$write_S^A : \mathbb{Z}^n \to \mathbb{Z}^m$	Function that models a write access from statement S surrounded by n loops to m-dimensional array A.	20
$x :: xs$	Prepends element x to list xs.	41
GA_B	Genetic algorithm with benchmarking.	91
GA_C	Genetic algorithm with classification.	110
CLANG	LLVM's front end for languages in the C language family (C, C++, OBJECTIVE C, etc.	92
ALU	Arithmetic-logic unit.	6
AST	Abstract syntax tree.	30
CART	Classification and regression tree.	24
GCC	GNU Compiler Collection: a C-compiler.	23
GRAPHITE	An extension of GCC for polyhedral loop optimization.	23
ILP	Integer linear program.	16
ISL	Integer Set Library: a library to represent and manipulate integer sets.	23
JNI	Java Native Interface.	114
JSCOP	A notation for a SCoP's mathematical representation that is based on JSON.	111
JSON	Java Script Object Notation.	111
JiT	Just-in-time.	29
LCD	Lowest common denominator.	49
LeTSeE	An iterative polyhedral schedule optimizer.	35
LIBBARVINOK	A program library by Verdoolaege et al. [157] that can count the number of points in a $\mathbb{Z}$-polytope.	52
LLVM IR	Intermediate language of LLVM.	114
LLVM	originally "Low-Level Virtual Machine": A compiler framework	2
MPI	The MESSAGE PASSING INTERFACE is a standard for inter-process communication in the context of distributed-memory parallelization.	117
NUMA	Non-uniform memory access.	149

OpenMP	Open Multi-Processing	1
PHR sampling	Pattern hit-and-run sampling: a technique to sample points from a discrete polytope according to a probability distribution.	47
PIP	Parametric integer program.	16
PLUTO+	An extension of the PLuTo scheduling algorithm.	23
PLuTo	An algorithm for polyhedral schedule optimization and a polyhedral source-to-source compiler	4
PoCC	A polyhedral loop optimizer.	23
Polly	A set of LLVM passes for polyhedral program optimization.	2
PolyBench	A set of benchmark programs that can be represented in the polyhedron model.	4
PPCG	Polyhedral Parallel Code Generation: a source-to-source compiler that can transform regions of C-programs to CUDA.	23
scikit-learn	A Python library for machine-learning.	113
SCoP	Static control part. A code region that is representable in the polyhedron model	18
SLURM	Simple Linux Utility Resource Manager: a workload management system.	116
SMT	Simultaneous Multi-Threading.	7
SVM	Support vector machine: a machine learning technique.	33
UTF	Unified Transformation Framework.	27
`GetCoeff`	Given a one-dimensional linearly affine function f and an index d, `GetCoeff` returns the dth coefficient in f.	65
`rand`	Function that selects randomly an element from a given set	41
`reverse`	Function that reverses a list.	41
`SetCoeff`	Given a one-dimensional linearly affine function f, an index d, and $\alpha \in \mathbb{Z}$ `SetCoeff` returns a copy of f with its dth coefficient set to α.	65
`IsParamDim` $: \mathbb{N} \to \mathbb{B}$	Tests whether a given natural number is the index of a structure parameter's coefficient in a schedule's coefficient space.	65

List of Tables

List of Figures

List of Algorithms

Listings

1 Introduction and Problem Statement

In 1975, Moore [102] predicted that the complexity, i.e., the number of components, of integrated circuits would double approximately every two years. In addition, the continuous miniaturization of processors let their power consumption recede and allowed a steady rise of the speed at which computations were carried out by accelerating processors' clocks. Meanwhile, this scaling schema, which is known as Dennard Scaling [35], has lost its validity due to physical limitations. All main processor manufacturers resort to multicore processors to be able to increase the speed of computations despite the inability to raise the clock speed [67]. To make software profit from the available performance, developers must write parallelized code or compilers must have the ability to automatically parallelize relevant parts of programs. Typically, parts of programs whose execution makes up a significant share of an application's total execution time and whose control flow is repetitive, such as in the case of a loop, are candidates for parallelization.

Additionally, other techniques have emerged to increase processors' performance. Let us mention two important ones. Multiple cache levels between main memory and the processor cores have been added to mitigate the effect of the von-Neumann bottleneck, that is, the limitation of performance by the main memory's access latency and the bandwidth of the memory bus. Another noteworthy technological innovation are vector instructions, which introduce an additional level of finely grained parallelism in modern processors.

Many innovations in processor technology are characterized by the fact that, to be able to profit from them, applications must be adapted accordingly. Naturally, to exploit coarse-grained parallelism, the application must be parallelized, for instance, by OpenMP (Open Multi Processing) parallelization of loops. Also, to profit from vector instructions, the application code must be vectorized. Consequently, to fully exploit the performance of which a given processor is capable, applications must be adapted accordingly. While software developers may do this by hand, they will likely find that different hardware requires different optimizations, not least because the processors' capabilities may differ, and will therefore end up creating multiple variants of their application. Moreover, some optimization goals may have conflicting demands [171], which may complicate the accomplishment of maximum performance. These aspects make optimizing compilers appealing that have the ability to restructure code such that, among other optimizations, it can be parallelized and vectorized and that it exploits the processor caches optimally. At best, these compilers have the ability to automatically adapt their optimizations to different applications and target hardware. Automatic program optimization may either optimize code written in a general-purposes programming language such as C, it may operate on a compiler's intermediate program representation, or it may start from a representation in a domain-specific high-level language. The advantage of starting from a domain-specific language is that domain knowledge can be taken into account and that the optimizer has the ability to generate code that optimally suits its requirements and the requirements of the target hardware. In the context of high-performance computing, such high-level domain-specific languages exist for several domains, for instance, partial differential equation solving [137], digital signal processing [57, 130], machine learning [151], and digital image processing [132].

Regardless of the optimizing compiler's input representation, powerful and adaptable performance and cost models are required to successfully optimize programs so as to exploit the performance that is available on the given target hardware. Moreover, a powerful program optimization requires a suitable representation of programs and program transformations. This representation should enable the systematic exploration of the set of legal, i.e., semantics-preserving transformation sequences for a given program. This enables an iterative optimization and search for an optimal program transformation sequence. The advantage of iterative optimization is that it requires little or no prior knowledge. This enables it to adapt easily to different programs to be optimized and different target hardware. The downside of iterative optimization is its time and resource consumption that results from the sampling of a potentially high number of candidate transformation sequences and from the effort that must be spent to evaluate these transformations. Frequently, this makes iterative optimization impractical.

We are convinced that spending the high effort of iterative program optimization is a useful approach to the search for effective cost models for optimizing compilers.

In the context of loop optimization, a useful abstraction for programs and program transformations is the polyhedron model [54]. The polyhedron model encodes programs and program transformations in a uniform mathematical representation. To a good extent, it is possible to abstract from the elements in the sequence of classical syntactical program transformations that corresponds to a single polyhedral transformation function, which is, for most kinds of loop transformations, a multi-dimensional linearly affine function. The set of classical syntactic transformations that can be represented in the polyhedron model comprises loop distribution/fusion [79], skewing [166], tiling [73], interchange [5], and index set splitting [64], among others. With its uniform representation of complex sequences of program transformations, the polyhedron model facilitates the systematic exploration of the set of legal transformations for a given program [122, 124]. The model's downside is that it imposes restrictions on the programs and program transformations that can be represented. Yet, particularly in the domain of numeric computations, many algorithms are expressible. Polyhedral optimization at run time can mitigate some of the model's limitations and widens its applicability [140].

We propose the iterative polyhedral loop optimizer POLYITE [pəˈlīt]. POLYITE relies on the LLVM (originally low-level virtual machine) compiler infrastructure [89] and LLVM's polyhedral optimizer POLLY. With POLYITE, we target primarily programs that profit from coarse-grained parallelism, general data locality optimization, and tiling. *Tiling* [73] is a loop nest transformation that can increase data locality and improve the effectiveness of loop parallelization. POLYITE's theoretical foundation is the approach to iterative optimization in the polyhedron model by Pouchet et al. [122, 124] and the domain-specific polyhedral schedule search space exploration for the optimization of stencil codes by Kronawitter [86]. From Kronawitter, we borrow a technique for data dependence analysis and a construction of constraints to bound the set of legal program transformations for a given program.

Pouchet et al. optimize the sequential execution time of programs that can be represented in the polyhedron model. They do not tile loop nests. The particularity of their approach is that it considers only legal transformations. Based on the observation that the size of the complete set of legal transformations for a given program is infinite and that even a reasonable subset of this set is too large for a complete exploration, their search space is narrowed to a specific subset. Pouchet et al. employ a decoupling traversal heuristics that enumerates the transformations in their multi-dimensional search space in a projection on some dimensions that are known to be strongly sensitive to performance and that avoids exploring less relevant search space dimensions and subsets of their search space. The decoupling heuristics and the structure of the search space are based on experience and statistical knowledge [123] and exploit the fact that they optimize for sequential execution and that no tiling will be added to the program transformations. Pouchet et al. rely on the measured execution time of the transformed code to determine a program transformation's profitability.

In addition to their decoupling heuristics, Pouchet et al. propose a genetic algorithm that permits the non-uniform traversal of the transformation search space as opposed to the partial enumeration by the decoupling heuristics, which traverses the search space with a uniform pattern.

A *genetic algorithm* is a technique for search-based optimization that is inspired by natural evolution. It starts from an initial population of candidate solutions that is often generated randomly. Subsequently, the genetic algorithm determines the fitness of each solution in the population. Pouchet et al. use execution time measurement as their fitness function. To build an offspring population from the current one, a genetic algorithm derives new solutions to the optimization problem from the ones that already exist. In many genetic algorithm designs, some solutions can survive from one population to the next. A solution's probability to survive or to produce an offspring by mutation or crossover depends on its fitness.

The genetic algorithm by Pouchet et al. uses tailored genetic mutation and crossover operators under which their search space of legal program transformations is closed.

Pouchet's approach is promising but has a limited ability to enable lucrative tiling and parallelization. The reason for narrowing the search space to a specific subset is its enormous size. Even if one bounds the coefficients of the multi-dimensional linearly affine function that represents a program transformation in the polyhedron model to reasonably small values and avoids the insertion of surplus dimensions into the transformation function, the search space's is exponential in the number of data dependences that occur in the program to be optimized [124]. Without bounding the maximum dimensionality of the schedule functions, the search space's size is non-convex and cannot be represented in its entirety. If one bounds the maximum dimensionality of the schedules, it is theoretically possible to express the set of all legal schedules for a given program as one convex set, but the dimensionality of the resulting search space likely poses the computational infeasibility of handling it [127].

Unfortunately, to optimize for tiling and parallelization, it is likely necessary to remove the restrictions that narrow the search space. We overcome the dilemma of high dimensionality not being able to represent the true entire search space at once by sampling subsets of it that are expressible in the same way as Pouchet et al. represent their search space. We refer to these subsets as *search space regions*. With our adaptation of the algorithm for search space construction by Pouchet et al. [124], we can sample search space regions. While, in principle, the selection of the regions is driven by chance, we have the ability to make some regions, such as regions that enable outer parallel loops, occur more likely than others.

Pouchet's decoupling heuristics for sampling builds on statistical knowledge that has only been validated for a sequential execution. We propose to use non-uniform random sampling as the basic sampling strategy. To this end, we describe and evaluate a number of potential sampling techniques. We find two of the techniques evaluated to be practical: one is a novel strategy proposed by us, which relies on Chernikova's algorithm (refer to Section 2.2.1.2). The other one relies on projection of polyhedra, for instance by the use of Fourier-Motzkin variable elimination [138], and is an adaptation for non-uniform random sampling of the sampling technique proposed by Pouchet et al. [124].

We adapt the genetic algorithm proposed by Pouchet et al. and propose a set of novel genetic operators because the operators presented by Pouchet et al. [124] would not be able to traverse our wider program transformation search space.

To remove unnecessary information and unwanted noise from our randomly generated program transformations and to facilitate their analysis and treatment by further optimization, such as tiling, we propose the transformation of their representation to polyhedral schedule trees [66] and a subsequent simplification.

As mentioned above, a major drawback of iterative program optimization is the effort that must be spent to evaluate candidate program transformations. Following Pouchet et al., we rely on benchmarking as our primary strategy to assess the fitness of program transformations. To increase the practicality of iterative optimization with Polyite, we propose a set of program transformation features based on which we can learn surrogate performance models that help to reduce the benchmarking effort. The surrogate performance models are learned from the results of previous iterative optimizations.

We evaluated our approach to iterative program optimization in the polyhedron model on the popular PolyBench 4.1 benchmark set [121]. PolyBench is a set of numeric computation kernels. All can be represented in the polyhedron model. In our evaluation, we compare Polyite to the widely used static polyhedral program optimization algorithm PLuTo [25, 28] as it is implemented in the Integer Set Library [152, 153, 156] and, thus, in Polly. PLuTo optimizes for data locality and the applicability of tiling. In PLuTo, parallelism is either a byproduct of optimizing data locality, or it can be targeted explicitly [156]. Recent advances by Zinenko et al. [171] extend a variant of PLuTo to account for spatial proximity of memory accesses and the tradeoff between data locality and parallelism explicitly. Further, we evaluate against an adaptation of the search space construction by Pouchet et al. [124] in Polyite. Note that Polyite cannot be used in combination with polyhedral loop vectorization. Despite this limitation, we found that, for several PolyBench programs, Polyite is competitive with a version of Polly that was contemporary at the time of writing and which used the PLuTo algorithm for optimization and had polyhedral vectorization enabled. This affects mainly programs which Polly cannot accelerate by vectorization. Besides the comparison to existing approaches of polyhedral program optimization, we evaluate a number of important parameters of our exploration technique with respect to their influence on the optimization result.

Our evaluation, which took coarse-grained loop parallelization with OpenMP and loop nest tiling into account, showed that Polyite has the ability to yield significantly higher speedups than the PLuTo scheduling algorithm. This finding applies to programs with long-running loops that operate on larger data sets since we used PolyBench with its extra large data set configuration. Random sampling of schedules from our larger search space outperforms random sampling from the search space that results from our adaptation of the search space construction by Pouchet et al. [124]. We found that the latter profits from amendments that facilitate tiling of loop nest's inner dimensions. Finally, we evaluated our surrogate performance models for program transformations. We used a leave-one-program-out schema to train models from transformations of $n - 1$ programs and applied the model to reduce the benchmarking effort spent by our genetic algorithm in an optimization of the remaining programs. We observed that, to a good extent, it is possible to reduce the benchmarking effort without reducing the maximum speedup in execution time of the transformed program reached seriously.

The rest of this thesis is organized as follows. Chapter 2 provides a short introduction to parallel computing and recalls the theoretical background of Polyite. Subsequently, Chapter 3 discusses work that is related to ours. Besides polyhedral program optimization techniques, we cover work that is in the context of iterative and compilation that uses machine learning. Chapter 4 recalls the search space construction by Pouchet et al. [124] and presents our amendment of the algorithm to sample regions of the space of legal program transformations. We continue by presenting and discussing a number of techniques for sampling program transformations from search space regions. Chapter 5 elaborates on our proposed transformation and simplification a of program transformation's representation. The presentation of our novel genetic operators for polyhedral program optimization (Chapter 6) and a set features of program transformations, together with an approach to integrate a performance predictor in our genetic algorithm (Chapter 7), follow. Chapter 8 provides an overview of our implementation Polyite. Chapter 9 describes our empirical evaluation of Polyite. Concluding remarks are made in Chapter 10, together with a short enumeration of open questions and directions.

2 Background

In this chapter, we present an overview of the fundamental aspects of parallel computing, with a focus on central processing units (CPUs), and the theoretical foundation of our work.

Section 2.1 introduces to parallel computing. We discuss briefly the main performance-related aspects of modern processor architectures and motivate the need for shared-memory (and distributed-memory) parallelism. In Section 2.2, we introduce the polyhedron model and its mathematical building blocks to the required extent. In Section 2.3, we introduce techniques of iterative (or search-based) optimization and machine learning on which our approach relies.

Readers who are familiar with any of these topics are invited to skip the respective sections.

2.1 Parallel Computing

In 1975, Moore [102] observed that the complexity of integrated circuits, which includes CPUs of general purpose computers, doubled approximately every two years. That is, the number of components on an integrated circuit (or number of transistors) doubles every two years. Dennard Scaling [49] permitted a steady rise of the clock speed. The *clock speed* is the frequency at which a processor executes machine instructions. While the size of transistors shrank, their power consumption receded, which permitted them to switch faster. In the following, we recall briefly the most important techniques that processor architects developed in order to increase the operation speed of CPUs besides simply increasing the clock speed. It is the combination of these techniques that complicates program optimization: different architectural effects may require opposing transformations [171].

Only in the early 2000s, Dennard Scaling lost its validity, which was due to physical limitations [35]. In particular, the power dissipation of contemporary processing units started to become difficult to handle. The main problem here is the large amount of energy that is turned into heat by processors that contain hundreds of millions of transistors. This makes their cooling difficult to handle. All main processor manufacturers chose multicore processors as their way to overcome this dilemma [67].

We start with a brief overview of techniques that exist besides multicore processing to increase the computing power of existing CPUs. We borrow from Hager and Wellein [67].

2.1.1 The Evolution towards Modern Multicore Processors

Contemporary processors use the *von-Neumann architecture*, which is named after John von Neumann, a Hungarian-American mathematician and computer scientist. In his document "First Draft of a Report on the EDVAC" [159], von Neumann originally described the idea of a stored-program computer architecture. Figure 2.1 illustrates this concept.

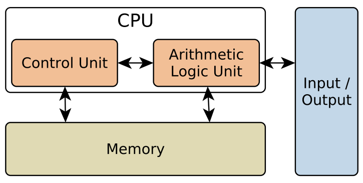


Figure 2.1: Basic operating schema of a von-Neumann processor: The CPU, which consists of a control unit and a processing unit, fetches commands and data from memory and stores computed results to memory. Input and output interfaces exist for communication with devices.

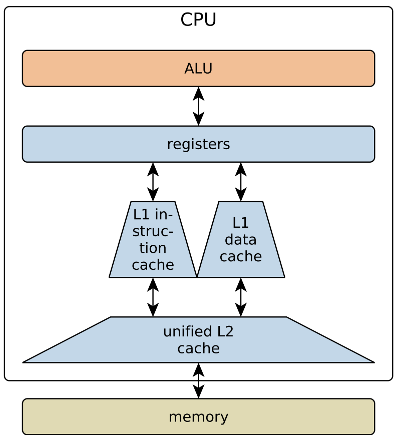


Figure 2.2: Illustration of a cache hierarchy with two levels: The upper level (L1) is split into two caches, one for instructions and one for data. The lower level (L2) is unified, that is, it contains both, instructions and data.

A *stored-program processor* fetches instructions and data from a (random access) memory. Instructions are processed and the results are stored back to memory. The actual computations are carried out by an arithmetic-logic unit (ALU). Input and output interfaces exist for communication with devices.

Processor Caches The von-Neumann architecture's major disadvantage is the fact that fetching data and instructions from memory can take much longer than the actual processing. Thus, the memory interface often limits the speed of computations. It is known as the *von-Neumann bottleneck*. At this point, the Roofline-Model by Williams et al. [164] is noteworthy since it permits to estimate whether the performance of an algorithm is ultimately bounded by the memory interface or by its amount of computation. The model relies on the operational intensity of a computation, that is, on the number of operations per byte transferred between the memory and the processor. We will not expand further on this model.

To reduce the von-Neumann bottleneck's impact, a hierarchy of *caches* resides between processor and memory. The caches are memories with a short latency and comparatively small capacity. They store instructions and data that are to be reused soon intermediately. Figure 2.2 illustrates a typical cache hierarchy with two levels. The upper L1 level of very small capacity caches data and instructions separately. The lower L2 level does not separate these. The registers serve the CPU as an intermediate storage for single values. Typically, the lower levels are *inclusive*: any data that resides in the L1 cache also resides in the L2 cache. Since the size of caches is limited, a *replacement strategy* must be applied to evict old data from the cache to make space for new data. When data is loaded into the cache, not a single memory address is loaded from memory, but an entire row of data, a *cache line*. Applications can profit from this behavior if they do not access memory randomly, but in a linear forward pattern.

In general, to profit from caching, programs must exhibit good *data locality*. There are two kinds of data locality. The first is the aforementioned *spatial data locality*, which permits applications to profit from chunking data into cache lines. The second is *temporal data locality*: if two operations operate on the same memory address, they should be executed with few other operations in between. Thereby, when the second operation is executed, no data must be loaded from memory because a copy of the data is still cached.

Pipelining Processors apply several techniques to achieve instruction-level parallelism. One of these is *pipelining*. If complex instructions are split into several simple steps that are to be executed consecutively by different components of the processor, then it is possible to start executing the next command before the previous has terminated. Figure 2.3 illustrates this property: with pipelining, the execution time of three commands that each have four steps can be reduced to 1.5 times the execution time of a single command. If the pipeline is

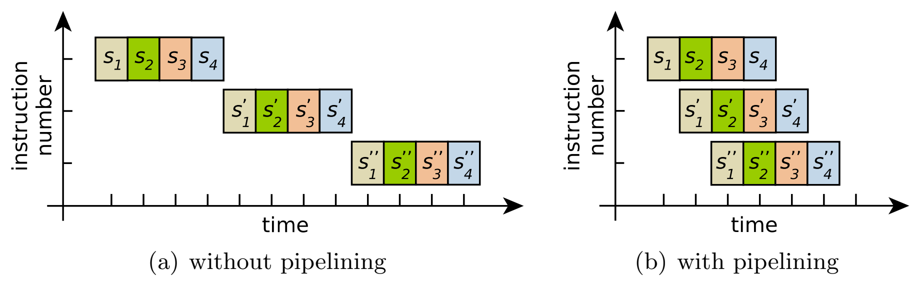


Figure 2.3: Pipelining increases instruction throughput.

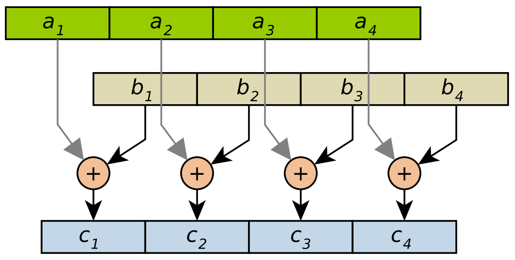


Figure 2.4: Vectorization allows to execute multiple additions $c = a + b$ simultaneously.

filled optimally pipelined execution leads to a throughput of one instruction per clock cycle.

Superscalarity Superscalar processors have another way of instruction-level parallelism. They can execute more than one instruction in parallel. Therefore, they are able to fetch and decode multiple instructions in parallel and provide more than one pipeline for integer and floating-point arithmetic. Also, caches must be fast enough to allow several reads and writes per clock cycle.

Vectorization SIMD instructions (*Single Instruction Multiple Data*) operate on an array of values of fixed length simultaneously. For instance, this allows to execute multiple integer additions simultaneously, as illustrated by Figure 2.4. This requires special registers and instructions to load or store values simultaneously. To enable compilers to generate vector instructions automatically and, thereby, speed up the execution of loops, programs must be shaped accordingly. Memory accesses in a loop nest's innermost loop must have a stride of 0 or 1. Furthermore, it must be possible to execute the innermost loop's iterations in parallel.

Simultaneous Multi-Threading (SMT) Duplicating registers permits to execute multiple instruction sequences (*threads*) in parallel on one processor. While execution units become idle if they are not needed in single-thread execution, multi-thread execution permits them to be used by other threads in the meantime. An important thing to be aware of is that the threads share the caches and, thus, may end up in a race, evicting each others' data from the cache memory. From the outside, a multi-threaded CPU appears as multiple logical CPUs.

2.1.2 Multicore Processors

To overcome the power dissipation dilemma mentioned, processor manufacturers switched from singlecore to multicore processors that combine multiple CPUs on a single processor socket.

Each CPU core has its own upper-level caches. Lower-level caches are typically shared among CPU cores. Figure 2.5 illustrates the schematic structure of a dual-core processor with separate L1 and L2 caches per core and a shared larger L3 cache. A sequential program

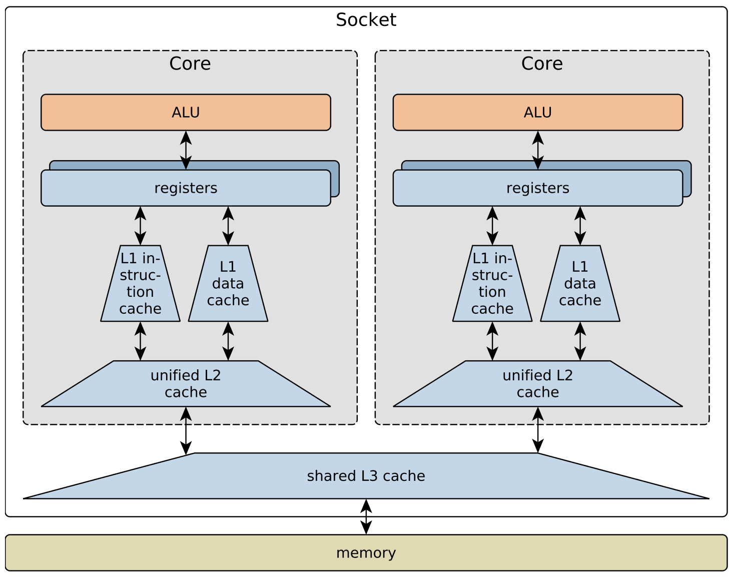


Figure 2.5: Diagram of a dual-core processor. Each core has its own L1 caches and unified L2 cache. Further, the cores share a larger L3 cache.

can make use of only one core and does not profit from the presence of multiple cores.

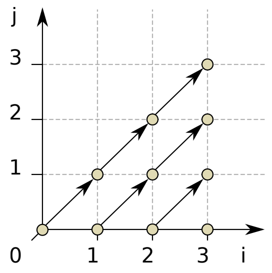
To exploit a multicore processor optimally, programs must be parallelized and execution threads should operate locally on a limited data set that fits, at best, into a core's local caches. Further, the workload should be distributed such that cores must not synchronize, i.e., wait for each other to exchange data, too often and the partition of the workload should be balanced. Otherwise, some cores may predominantly wait for other cores to perform most of the computation.

2.2 Polyhedron Model

The *polyhedron model* [54] is a mathematical abstraction of loop programs. It models programs as the union of polyhedra and relations between polyhedra. Figure 2.6 illustrates this representation.

```
for (int i = 0; i < 4; ++i) {
  for (int j = 0; j <= i; ++j)
    A[i + 1][j + 1] = A[i][j];
}
```

(a) A two-dimensional loop nest



(b) The model of the loop nest

Figure 2.6: A two-dimensional loop nest and its representation in the polyhedron model. The vector space's dimensionality corresponds to the loop nest's depth. The reader can verify that the polyhedron's bounds directly correspond to the loop bounds. The arrows denote reuse of data among loop iterations.

Multi-dimensional (linearly affine) functions represent program transformations. The particular advantage of the polyhedron model is its ability to encode an arbitrary sequence of many classical loop transformations, which allows to abstract from the atomic transformations steps in the sequence and to directly choose a complex transformation. Thereby, the model overcomes the difficult task of selecting the optimal sequence of compiler transformations for a given program and the optimal ordering of these transformations. Among the expressible loop transformations are loop distribution/fusion[1] [79], skewing[2] [166], tiling[3] [73], and index set splitting [64]. Transformations operate on the source model. New code can be generated from the result of the transformation, the target model.

Finally, the polyhedron model permits to model and sample systematically the search space of legal transformations of a program. Here, legality refers to the preservation of the program's semantics. Legally transformed code will compute the same as the original code, but the operations may be executed in a different order and, potentially, in parallel.

Typically, the polyhedron model serves to optimize promising program regions. Linear algebra kernels, dynamic programming, and stencil computations are frequent candidates for polyhedral optimization. The optimizations aim at the improvement of data locality and the parallelization of loops [1, 25, 52, 53].

In this section, we first introduce the polyhedron model's mathematical foundations to the extent at which they are relevant for our iterative polyhedral schedule optimization (Subsection 2.2.1). Subsequently, we introduce the model itself in Subsection 2.2.2.

2.2.1 Mathematical Foundation

Here, we present the mathematical foundation of the polyhedron model. For the work at hand, this theory is of particular importance. To be able to sample legal transformations for a given program, we must exploit many of the theory's properties. This subsection borrows mainly from Schrijver [138] and also from Pan [112].

2.2.1.1 Integer Sets and Polyhedra

The most basic structures within the polyhedron model are linearly affine functions and $\mathbb{Z}$-polyhedra. We start by defining them step by step. We assume that the reader is familiar with basic terms from linear algebra, such as vector spaces, vectors, matrices, and basic operations such as matrix-vector products.

Matrices $\mathbb{R}^{m \times n}, m, n \in \mathbb{N}$ denotes the set of all matrices with real elements contained in m rows and n columns. Let $A \in \mathbb{R}^{m \times n}$ be some matrix. By $A^{(i,j)}$, we denote cell j in row i of A. $A^{(i,\bullet)}$ addresses row i and $A^{(\bullet,j)}$ denotes column j. To select rows $i, i+1, ..., j$, we write $A^{(i..j,\bullet)}$. We address a submatrix by $A^{(i..j,k..l)}$.

Given a vector $\vec{\beta} \in \mathbb{R}^n$, we can address its ith component by writing $\vec{\beta}^{(i)}$. $A \cdot \vec{\beta}$ denotes the matrix-vector product. By $A\vec{\beta}$ we denote the horizontal coupling of $\vec{\beta} \in \mathbb{R}^n$ and A. The operation produces matrix $B \in \mathbb{R}^{m \times (n+1)}$ with $B^{(\bullet,1..n)} = A$ and $B^{(\bullet,n+1)} = \vec{\beta}$.

Example 2.2.1.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 5 \\ 3 & 4 & 6 \end{pmatrix}$$

◁

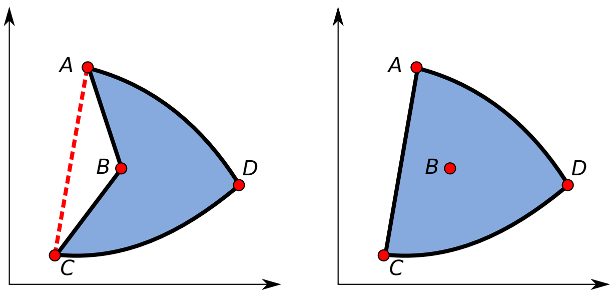
To simplify the notation of algorithms, we introduce the functions

$$\text{rows} : \mathbb{R}^{m \times n} \to \mathbb{N}, n, m \in \mathbb{N}$$
$$\text{cols} : \mathbb{R}^{m \times n} \to \mathbb{N}, n, m \in \mathbb{N},$$

[1] Computing iterations of different statements in one loop (fusion) fuses these statements. Computing each statement's iterations in a different loop distributes two statements.

[2] Skewing shifts the iterations of a loop by a multiple of the iteration variable of a loop that encases the skewed loop. This transformation can expose parallelism and enable tiling.

[3] Tiling changes the execution order of a loop nest by blocking the nest's loops and permuting the loops that enumerate the blocks to the outside of the nest. This can improve data locality and efficient parallelization.



(a) A non-convex set (b) A convex set

Figure 2.7: Examples of a non-convex set and a convex set in $\mathbb{R}^2$.

which return the number of rows and the number of columns of a given matrix.

Analogously to $\mathbb{R}^n$ and $\mathbb{R}^{m\times n}$, we define $\mathbb{Z}^n$ and $\mathbb{Q}^n$ to be the set of all n-dimensional vectors with integer components and rational components, respectively. $\mathbb{Z}^{m\times n}$ is the set of all $m \times n$ matrices with integer elements and $\mathbb{Q}^{m\times n}$ is the set of all $m \times n$ matrices with rational elements.

Given $S \subseteq \mathbb{R}^n$, we define $\dim(S) = n$ to be the dimensionality of S.

Linearly Affine Functions are very important within the polyhedron model since, as we will see later, they can be used to express many loop transformations and sequences of such transformations. Before defining linearly affine functions, let us recall linear functions.

Definition 2.2.1. Let $m, n \in \mathbb{N}$ and $f : \mathbb{R}^n \to R^m$ be a function. f is a *linear function* if matrix $A \in R^{m\times n}$ exists with

$$(\forall \vec{x} \in \mathbb{R}^n : f(\vec{x}) = A \cdot \vec{x}) .$$

Based on the definition of linearity, we can define affinity:

Definition 2.2.2. Let $m, n \in \mathbb{N}$ and $f : \mathbb{R}^n \to R^m$ be a function. f is a *linearly affine function* if a linear function $g : \mathbb{R}^n \to R^m$ defined by a matrix $A \in \mathbb{R}^{m\times n}$ and a vector $\beta \in \mathbb{R}^m$ exist with

$$\left(\forall \vec{x} \in \mathbb{R}^n : f(\vec{x}) = g(\vec{x}) + \vec{\beta} = \left(A\vec{\beta}\right) \cdot \begin{pmatrix} \vec{x} \\ 1 \end{pmatrix}\right) .$$

We define $\dim(f) = m$.

Convex Set An important property of polyhedra that we rely on is their convexity.

Definition 2.2.3. Let $C \subseteq \mathbb{R}^n, n \in \mathbb{N}$ be a set of n-dimensional vectors. C is a *convex set* if

$$(\forall \vec{x}, \vec{y} \in C : (\forall \alpha \in [0, 1] : \alpha \cdot \vec{x} + (1 - \alpha) \cdot \vec{y} \in C)) .$$

Intuitively, and as can be seen from Example 2.2.2, a set is convex iff, for any two points that are elements of the set, each point on the line segment between the two points is also an element of the set. This allows us to remain inside a set while we interpolate between any of its elements. The intersection of two convex sets of n-dimensional vectors is a convex set.

Example 2.2.2 (Convexity)**.** To demonstrate convexity, we illustrate a convex set and a non-convex set in $\mathbb{R}^2$ graphically. The set in Figure 2.7(a) is not a convex set since there are points on the line segment between A and C that are not elements of the set. On the other hand, Figure 2.7(b) illustrates a convex set since no points in the set can be connected by a line segment that crosses the boundaries of the set. ◁

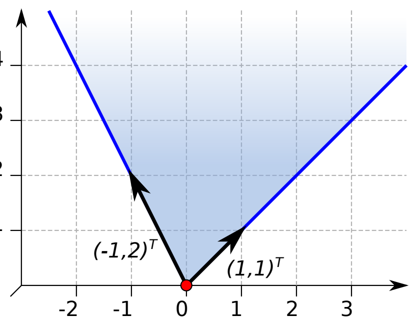


Figure 2.8: An illustration of the cone generated by the vectors $\{(-1,2)^T,(1,1)^T\}$.

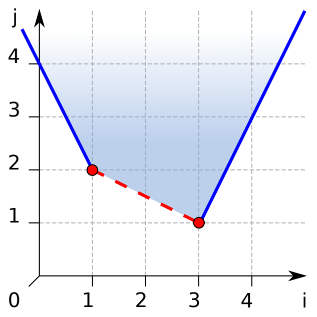


Figure 2.9: Illustration of a two-dimensional polyhedron. The polyhedron's vertices and their convex hull (dashed line) are marked.

Let $S = \{\vec{i} \mid \vec{i} \in \mathbb{R}^n \ \wedge \ (p_1(\vec{i}) \vee p_2(\vec{i}) \vee \ldots \vee p_m(\vec{i})\}, m, n \in \mathbb{N}$ be some set of n-dimensional vectors. We define $\Lambda(S) = \{\{\vec{i} \mid \vec{x} \in \mathbb{R}^n \ \wedge \ p_i(\vec{x})\} \mid i \in \{1,2,\ldots,m\}\}$. We will use Λ in the definition of algorithms that require an according decomposition of sets of vectors.

Convex Hull From the definition of a convex set, we can go on to the definition of the convex hull of a set of vectors $S \subseteq \mathbb{R}^n, n \in \mathbb{N}$. Before we introduce its convex hull, we look at the simpler definition of the linear hull of S:

Definition 2.2.4. The *linear hull* of S is defined as

$$\text{lin.hull}(S) = \left\{ \sum_{i=1}^{t} \gamma_i \cdot \vec{x_i} \mid t \in \mathbb{N} \ \wedge \ \vec{x_1}, \ldots, \vec{x_t} \in S \ \wedge \ \gamma_1, \ldots, \gamma_t \in \mathbb{R} \right\}.$$

The combinations generating lin.hull(S) are called *linear combinations*.

The convex hull of S is the smallest convex set that contains all elements of S. Formally:

Definition 2.2.5. Given set S as defined above, its *convex hull* is

$$\text{conv.hull}(S)$$
$$= \left\{ \sum_{i=1}^{t} \alpha_i \cdot \vec{x_i} \mid t \in \mathbb{N} \ \wedge \ \vec{x_1}, \ldots, \vec{x_t} \in S \ \wedge \ \alpha_1, \ldots, \alpha_t \geq 0 \ \wedge \ \sum_{i=1}^{t} \alpha_i = 1 \right\} \subseteq \text{lin.hull}(S).$$

The linear combinations that generate conv.hull(S) are named *convex combinations*.

Example 2.2.3 (Convex Hull)**.** The set pictured in Figure 2.7(b) is the convex hull of the set pictured in Figure 2.7(a). The set in Figure 2.7(b) is its own convex hull. ◁

Convex Cone Similar to the definition of the convex hull of a set of vectors is the definition of their convex cone:

Definition 2.2.6. Let $S \subseteq \mathbb{R}^n, n \in \mathbb{N}$ be a set of n-dimensional vectors. The *convex cone* generated by S is defined as follows:

$$\text{cone}(S) = \left\{ \sum_{i=1}^{t} \beta_i \cdot \vec{x_i} \mid t \in \mathbb{N} \ \wedge \ \vec{x_1}, \ldots, \vec{x_t} \in S \ \wedge \ \beta_1, \ldots, \beta_n \in \mathbb{R}_0^+ \right\}.$$

We call the linear combinations generating cone(S) *conical combinations*.

Example 2.2.4. Figure 2.8 illustrates the two-dimensional cone generated by the vectors $\{(-1,2)^T,(1,1)^T\}$. ◁

Polyhedron We can now recall the definition of a polyhedron.

Definition 2.2.7. Given a matrix $A \in \mathbb{R}^{m \times n}, m, n \in \mathbb{N}$ and a vector $\vec{\beta} \in \mathbb{R}^m$, a *polyhedron* is a convex set of n-dimensional vectors that is defined as follows:

$$\{\vec{x} \mid \vec{x} \in \mathbb{R}^n \ \wedge \ A \cdot \vec{x} \leq \vec{\beta}\}.$$

We call this definition *constraint representation* of the polyhedron. Each constraint that is encoded by A and $\vec{\beta}$ corresponds to a hyperplane that separates $\mathbb{R}^n$ into two half-spaces. Being the intersection of the remaining half-spaces, a polyhedron is a convex set bounded by hyperplanes.

Example 2.2.5. Figure 2.9 shows the polyhedron that is represented by the following constraints:

$$\begin{array}{lll} -2 \cdot i - j & \leq & -4 \\ \wedge \ -i - 2 \cdot j & \leq & -5 \\ \wedge \ 2 \cdot i - j & \leq & 5. \end{array}$$

◁

Decomposition Theorem for Polyhedra The constraint representation of polyhedra presented above is a representation that expresses polyhedra by enumerating their bounding hyperplanes. Now, we introduce a geometric and more tangible representation of polyhedra. We start by making several auxiliary definitions before recalling how polyhedra can be decomposed into their geometric base components.

Let $m, n \in \mathbb{N}$, $A \in \mathbb{R}^{m \times n}$, and $\vec{\beta} \in \mathbb{R}^m$ define polyhedron $P = \{\vec{x} \mid \vec{x} \in \mathbb{R}^n \ \wedge \ A \cdot \vec{x} \leq \vec{\beta}\}$.

Definition 2.2.8. Let $\vec{c} \in \mathbb{R}^n \setminus \vec{0}$ and $\delta = \max\{\vec{c} \cdot \vec{x} \mid \vec{x} \in \mathbb{R}^n \ \wedge \ A \cdot \vec{x} \leq \vec{\beta}\}$. The affine hyperplane $H = \{\vec{x} \mid \vec{x} \in \mathbb{R}^n \ \wedge \ \vec{c} \cdot \vec{x} = \delta\}$ is a *supporting hyperplane* of P. The set $H \cap P$ is a *face* of P.

Example 2.2.6. The faces of the polyhedron in Figure 2.9 are its vertices at $(1, 2)$ and $(3, 1)$, the line segment connecting the two vertices, the ray pointing away from $(1, 2)$ with direction vector $(-1, 2)$, and the ray pointing away from $(3, 1)$ with direction vector $(1, 2)$. ◁

Based on the definition of a polyhedron's faces, we can define minimal faces.

Definition 2.2.9. A *minimal face* of a polyhedron is a face that does not contain any other of the polyhedron's faces.

Example 2.2.7. The minimal faces in Example 2.2.6 are the two vertices. ◁

Definition 2.2.10. The *characteristic cone* of P is defined as

$$\text{char.cone}(P) = \{\vec{y} \mid \vec{y} \in \mathbb{R}^n \ \wedge \ (\forall \vec{x} \in P : \vec{x} + \vec{y} \in P)\}.$$

char.cone(P) contains the vectors pointing into directions in which P is unbounded.

From the definition of a characteristic cone it is easy to derive the notion of the lineality space of a polyhedron. If a polyhedron is unbounded in two opposing directions, the respective vectors are elements of its lineality space.

Definition 2.2.11. Formally, the *lineality space* of P can be defined as

$$\text{lin.space}(P) = \text{char.cone}(P) \cap \{-\vec{y} \mid \vec{y} \in \text{char.cone}(P)\}.$$

Definition 2.2.12. Given the characteristic cone of a polyhedron, the cone's one-dimensional faces are called its *extremal rays*.

Theorem 2.2.1 (Decomposition Theorem of Polyhedra)**.** *P can be generated as follows:*

$$P = \{\vec{v} + \vec{r} + \vec{l} \mid \vec{v} \in \text{conv.hull}(V) \ \wedge \ \vec{r} \in \text{cone}(R) \ \wedge \ \vec{l} \in \text{lin.hull}(L)\}.$$

V is a set of points, each chosen arbitrarily from a different minimal face of P. R is the set of extremal rays (or rays for short) of char.cone(P). Finally, L is an arbitrary set of vectors generating lin.space(P).

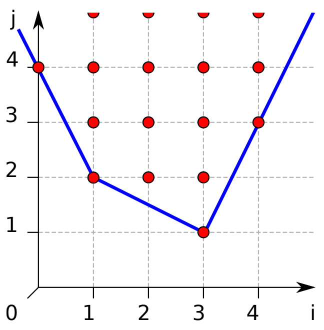


Figure 2.10: Illustration of the $\mathbb{Z}$-polyhedron encased by the polyhedron in Example 2.2.5.

We call the representation of a polyhedron that results from Theorem 2.2.1 the polyhedron's *geometric representation.* If P is *pointed*, that is, if lin.space(P) is zero-dimensional, V is the set of the polyhedron's vertices. It is possible to choose V, R, and L uniquely, apart from stretching of vectors. We will see later that, for our use case, it is advisable to choose the elements of R and L such that the vectors' components are integers that have small absolute values if that is possible.

Proof. The proof of Theorem 2.2.1 is due to Schrijver [138]. □

In case of $R = L = \emptyset$, P is a *polytope*, i.e., a polyhedron of finite volume.

Example 2.2.8. The polyhedron in Examples 2.2.5 has the following geometric representation:

$$V = \{(1, 2), (3, 1)\}; \quad R = \{(-1, 2), (1, 2)\}; \quad L = \emptyset.$$

◁

Integer Sets and $\mathbb{Z}$-Polyhedra In the polyhedron model, sets of iterations of statements are represented by points with integer coordinates within polyhedra. Such sets are called *$\mathbb{Z}$-polyhedra*:

Definition 2.2.13. An *integer set* S of dimensionality $n \in \mathbb{N}$ is a subset of $\mathbb{Z}^n$.

Definition 2.2.14. A *$\mathbb{Z}$-polyhedron* of dimensionality $n \in \mathbb{N}$ is the intersection of a polyhedron $P \subseteq \mathbb{R}^n$ and $\mathbb{Z}^n$. The definition of a *$\mathbb{Z}$-polytope* is analogous.

Example 2.2.9. Figure 2.10 shows the $\mathbb{Z}$-polyhedron encased by the polyhedron in Example 2.2.5.

◁

2.2.1.2 Chernikova's Algorithm

N.V. Chernikova [107, 108, 109] proposed an algorithm to compute the geometric representation of a polyhedron P from its constraint representation $\{\vec{x} \mid \vec{x} \in (\mathbb{R}_0^+)^n \ \wedge \ A \cdot \vec{x} \leq \vec{\beta}\}, A \in \mathbb{R}^{m \times n}$, $m, n \in \mathbb{N}, \beta \in \mathbb{R}^m$. This algorithm is known as "Chernikova's algorithm". By repeatedly applying a set of matrix transformations, the algorithm calculates the extremal rays of the polyhedral cone $C = \{(\vec{x}, \xi)^T \mid \vec{x} \in (\mathbb{R}_0^+)^n \ \wedge \ \xi \in \mathbb{R}_0^+ \ \wedge \ -A \cdot \vec{x} + \vec{\beta} \cdot \xi \geq \vec{0}\}$. From this solution, the extremal rays and vertices of P can be derived [97]. Following Fernández and Quinton [55], the same algorithm can be used to solve the reverse problem.

Being able to calculate the geometric representation of polyhedra allows us to sample vectors from inside polyhedra as their generators' linear combination in an easily controllable way. Of course, we must always adhere to the decomposition theorem of polyhedra. We use this technique to sample legal transformations for programs, which are represented as coefficient matrices of linearly affine functions.

Fernandéz and Quinton extended Chernikova's algorithm to be able to calculate the geometric representation of arbitrary polyhedra $\{\vec{x} \mid \vec{x} \in \mathbb{R}^n \ \wedge \ A \cdot \vec{x} \leq \vec{\beta}\}$. Le Verge [90] was able to further improve the extended algorithm and, thereby, reduce its average execution time.

Algorithm 2.1 performs the steps given by Matheiss and Rubin [97] to compute the geometric representation of a polyhedron $\{\vec{x} \mid \vec{x} \in (\mathbb{R}_0^+)^n \ \wedge\ A \cdot \vec{x} \leq \vec{\beta}\}$.

Algorithm 2.1: Chernikova's Original Algorithm

```
Input: $A \in \mathbb{R}^{m\times n}, m, n \in \mathbb{N}, \beta \in \mathbb{R}^m$ representing polyhedron $P$
Output: Set of vertices $V$ and set of rays $R$ with $P = \{\vec{v} + \vec{r} \mid \vec{v} \in \text{conv.hull}(V) \ \wedge\ \vec{r} \in \text{cone}(R)\}$
1  $U \leftarrow \left(-A\vec{\beta}\right)$ ;                                   ▷ Adjoin $-A$ and $\vec{\beta}$
2  $L \leftarrow I_{n+1}$ ;                                          ▷ Identity matrix of dimensionality $n+1$
3  $Y \leftarrow \begin{pmatrix} U \\ L \end{pmatrix}$
4  while $(\exists (i,j) \in \{1,...,m\} \times \{1,...,\text{cols}(U)\} : U^{(i,j)} < 0)$ do
5      if $(\exists i \in \{1,...,m\} : U^{(i,\bullet)} \in (\mathbb{R}^-)^{\text{cols}(U)})$ then
6          return $V = \emptyset, R = \{\vec{0}\}$
7      $r \leftarrow \min\{i \mid i \in \{1,...,m\} \ \wedge\ (\exists j \in \{1,...,\text{cols}(U)\} : U^{(i,j)} < 0)\}$
8      $R \leftarrow \{j \mid j \in \{1,...,\text{cols}(U)\} \ \wedge\ Y^{(r,j)} \geq 0\}$
9      $\overline{Y} \leftarrow 0_{\mathbb{R}^{\text{rows}(Y)\times 0}}$ ;             ▷ Zero-matrix with 0 columns and rows($Y$) rows
10     for $j \in R$ do
11         $\overline{Y} \leftarrow \overline{Y} Y^{(\bullet,j)}$ ;                        ▷ Adjoin $\overline{Y}$ and column $j$ of $Y$
12     if $((\text{rows}(Y) = 2) \ \wedge\ (Y^{(r,1)} \cdot Y^{(r,2)} < 0))$ then
13         $\vec{v} \leftarrow \left|Y^{(r,2)}\right| \cdot Y^{(\bullet,1)} + \left|Y^{(r,1)}\right| \cdot Y^{(\bullet,2)}$; $\overline{Y} \leftarrow \overline{Y}\vec{v}$
14     else
15         $S \leftarrow \{(s,t) \mid Y^{(r,s)} \cdot Y^{(r,t)} < 0 \ \wedge\ s < t\}$
16         $I_0 \leftarrow \{i \mid (i \in \{1,...,\text{rows}(Y)\} \ \wedge\ \forall j \in \{1,...,\text{cols}(Y)\} : Y^{(i,j)} \geq 0)\}$
17         for $(s,t) \in S$ do
18             $I_1 \leftarrow \{i \mid i \in I_0 \ \wedge\ Y^{(i,s)} = Y^{(i,t)} = 0\}$
19             if $\neg(\exists u \in \{1,...,\text{cols}(Y)\} : u \neq s \ \wedge\ u \neq t \ \wedge\ (\forall i \in I_1 : Y^{(i,u)} = 0))$ then
20                 $\alpha_1 \leftarrow \left|Y^{(r,t)}\right|$; $\alpha_2 \leftarrow \left|Y^{(r,s)}\right|$
21                 $\vec{v} \leftarrow Y^{(\bullet,s)} \cdot \alpha_1 + Y^{(\bullet,t)} \cdot \alpha_2$; $\overline{Y} \leftarrow \overline{Y}\vec{v}$
22     $Y \leftarrow \overline{Y}$; $U \leftarrow \overline{Y}^{(1..m,\bullet)}$; $L \leftarrow \overline{Y}^{(m+1..m+n+1,\bullet)}$
23 $V \leftarrow \{L^{(1..n,i)} \mid i \in \{1,...,\text{cols}(L)\} \ \wedge\ L^{(n+1,i)} > 0\}$; $R \leftarrow \{L^{(1..n,i)} \mid i \in \{1,...,\text{cols}(L)\} \ \wedge\ L^{(n+1,i)} = 0\}$
24 return $V, R$
```

The algorithm operates on a matrix Y that is initiated as follows:

$$Y = \begin{pmatrix} U \\ I_{n+1} \end{pmatrix}, U = -A\vec{\beta}.$$

It processes one row of U at a time. Depending on the elements of the current row, the algorithm adds columns to Y and replaces columns. The algorithm continues until the columns of matrix L match the extremal rays of the cone C.

As to the time and space complexity of Chernikova's algorithm, one may recall an example given by Feautrier [52]: a hypercube in a d-dimensional vector space ($d \in \mathbb{N}$) may be represented either by $2 \cdot d$ inequalities that correspond to its bounding hyperplanes or by enumerating its 2^d many vertices. Chernikova's algorithm has exponential execution time.

We give an example to illustrate Chernikova's algorithm:

Example 2.2.10. Consider polyhedron $P = \{\vec{x} \mid \vec{x} \in \mathbb{R}^2 \wedge A \cdot \vec{x} \leq \vec{\beta}\}$ with $A = \begin{pmatrix} 1 & -1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix}$ and $\vec{\beta} = (0, 0, -2)^T$. Figure 2.11 shows P.

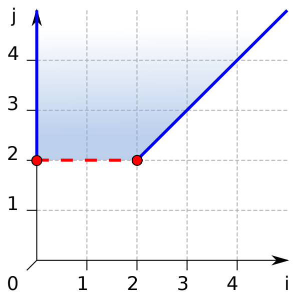


Figure 2.11: A simple polyhedron to illustrate Chernikova's algorithm.

We construct matrix Y as

$$Y = \begin{pmatrix} -A\beta \\ I_3 \end{pmatrix} = \left(\begin{array}{ccc} -1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & -2 \\ \hline 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right).$$

First Round

$$r = 1, R = \{2,3\}, \overline{Y} = \left(\begin{array}{cc} 1 & 0 \\ 0 & 0 \\ 1 & -2 \\ \hline 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{array}\right).$$

$$S = \{(1,2)\}.$$

Inspection of $(s,t) = (1,2)$:

$$I_1 = \{6\}, \alpha_1 = \alpha_2 = 1.$$

Adjoin $Y^{(\bullet,s)} \cdot \alpha_1 + Y^{(\bullet,t)} \cdot \alpha_2 = (0,1,1,1,1,0)^T$ to $\overline{Y}$

$$Y = \overline{Y} = \left(\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & -2 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right).$$

Second Round

$$r = 3, R = \{1,3\}, \overline{Y} = \left(\begin{array}{cc} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ \hline 0 & 1 \\ 1 & 1 \\ 0 & 0 \end{array}\right).$$

$$S = \{(1,2),(2,3)\}.$$

- Inspection of $(s,t) = (1,2)$:

  $$I_1 = \{2,4\}, \alpha_1 = 2, \alpha_2 = 1.$$

  Adjoin $Y_{(\bullet,s)} \cdot \alpha_1 + Y^{(\bullet,t)} \cdot \alpha_2 = (2,0,0,0,2,1)^T$ to $\overline{Y}$
- Inspection of $(s,t) = (2,3)$:

  $$I_1 = \{1\}, \alpha_1 = 1, \alpha_2 = 2.$$

  Adjoin $Y^{(\bullet,s)} \cdot \alpha_1 + Y^{(\bullet,t)} \cdot \alpha_2 = (0,2,0,2,2,1)^T$ to $\overline{Y}$

$$Y = \overline{Y} = \left(\begin{array}{cccc} 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 2 \\ 1 & 1 & 0 & 0 \\ \hline 0 & 1 & 0 & 2 \\ 1 & 1 & 2 & 2 \\ 0 & 0 & 1 & 1 \end{array}\right).$$

The algorithm terminates after the second round because all elements in the upper half of Y are non-negative. Consequently, $P = \text{conv.hull}(\{(0,2),(2,2)\}) + \text{cone}(\{(0,1),(1,1)\})$. $\triangleleft$

2.2.1.3 Parametric Integer Programming

The *Simplex Algorithm* by George B. Dantzig in 1947 [47] solves *linear programs*

$$\min\{\vec{c} \cdot \vec{x} \mid \vec{x} \in \mathbb{R}^n \ \wedge \ A \cdot \vec{x} \leq \vec{\beta}\}$$

with $n, m \in \mathbb{N}, A \in \mathbb{R}^{m \times n}, \beta \in \mathbb{R}^m, \vec{c} \in \mathbb{R}^n$. The maximal solution can be determined as well. From Section 2.2.1.1, we know that the set of valid solutions to the inequality system $A \cdot \vec{x} \leq \vec{\beta}$ is a polyhedron $P \subseteq \mathbb{R}^n$. At least one of the vertices of P is among the solutions for which the value of $\vec{c} \cdot \vec{x}$ is minimal. The Simplex Algorithm makes use of this property by greedily passing from one vertex of the polyhedral set of valid solutions to one of its adjacent vertices until it finds an optimal solution.

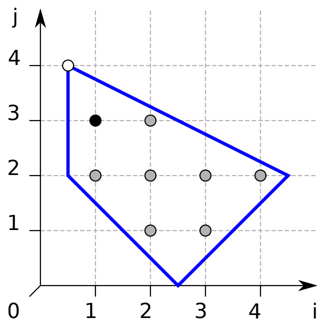


Figure 2.12: Corresponding polyhedron of the linear program in Example 2.2.11. While the optimal real solution is a vertex (white dot), the optimal integer solution (black dot) is an interior point.

Linear programs exist for which the Simplex Algorithm has exponential execution time [81]. *Integer linear programs* (ILPs, for short) search for integer solutions

$$\min\{\vec{c}\cdot\vec{x} \mid A\cdot\vec{x}\le\vec{\beta} \ \wedge\ \vec{x}\in\mathbb{Z}^n\}, A\in\mathbb{R}^{m\times n}, n,m\in\mathbb{N}, \vec{\beta}\in\mathbb{R}^m, \vec{c}\in\mathbb{R}^n \tag{2.1}$$

and are more difficult to solve: as illustrated by Example 2.2.11, the optimal solution is not located necessarily at a vertex of the corresponding polyhedron.

Example 2.2.11. Consider the system of inequalities

$$\begin{aligned}-2\cdot i-2\cdot j&\le -5\\ 2\cdot i-2\cdot j&\le 5\\ 2\cdot i+4\cdot j&\le 17\\ -2\cdot i&\le 1\end{aligned}$$

and the following objective function:

$$-2\cdot j-3\cdot i.$$

Figure 2.12 shows the corresponding polyhedron of this linear program. Both, the optimal real solution (white dot) and the optimal integer solution (black dot) are marked. While the optimal real solution is a vertex of the polyhedron, the optimal integer solution is the interior point $(1,3)$. The objective function's optimal value is -9. $\triangleleft$

Parametric integer programming is a method that was introduced by Feautrier [50]. It can solve ILPs that contain parametric lower and upper bounds for variables. The possibility to have parametric bounds for variables is handy in the context of polyhedral code optimization since parametric loop bounds lead to inequalities such as $0\le i<n$. Parametric integer programming does not search for the optimal solution vector according to an objective function. Instead, it finds the lexicographically minimal integer solution vector from the set of legal solutions. Definition 2.2.15 recalls the definition of lexicographic ordering:

Definition 2.2.15. Let $\vec{u},\vec{v}\in\mathbb{R}^n, n\in\mathbb{N}$. $\vec{u}$ is lexicographically smaller than $\vec{v}$, which we denote by $\vec{u}\prec\vec{v}$ if, and only if:

$$\Big(\exists i\in\{1,\dots,n\} : (\forall j\in\{1,\dots,i-1\} : \vec{u}^{(j)}=\vec{v}^{(j)}) \ \wedge\ \vec{u}^{(i)}<\vec{v}^{(i)}\Big).$$

An ILP, as defined in Equation 2.1, can be denoted as a parametric integer program (PIP)

$$\text{lex.min}\left\{\begin{pmatrix}x'\\ \vec{x}\end{pmatrix} \mid \begin{pmatrix}1 & -\vec{c}^{\,T}\\ \vec{0} & A\end{pmatrix}\cdot\begin{pmatrix}x'\\ \vec{x}\end{pmatrix}\begin{matrix}=\\ \le\end{matrix}\begin{pmatrix}0\\ \vec{\beta}\end{pmatrix} \wedge\ \vec{x}\in\mathbb{Z}^n \ \wedge\ x'\in\mathbb{Z}\right\}.$$

Examples can be found in work by Bondhugula et al. [25, 28].

Parametric integer programming makes use of Gomory Cuts to find integer solutions. The idea is to compute a real solution to a linear program and then add constraints to exclude one of the solution vector's non-integer components. The procedure continues until an integer solution has been found. The extra constraints that are introduced by the cuts make it necessary to use the dual simplex method [138].

The run-time complexity of Parametric integer programming is exponential [50]. In fact, integer linear programming is an NP-complete problem [63, 75].

2.2.1.4 Affine Form of Farkas' Lemma

The affine form of Farkas' Lemma has an important role in the polyhedral optimization framework. Given the model of a program it helps to mathematically express the set of legal transformations for that program.

Particularly, we refer to the variant of the affine form of Farkas' Lemma given by Feautrier [52], Darte et al. [48], and Pouchet et al. [122]:

Theorem 2.2.2 (The Affine Form of Farkas' Lemma (variant)).
Let polyhedron $P = \{\vec{x} \mid A \cdot \vec{x} \leq \vec{\beta} \ \wedge\ \vec{x} \in \mathbb{R}^n\}$ with $n, m \in \mathbb{N}, A \in \mathbb{R}^{m \times n}, \vec{\beta} \in \mathbb{R}^m$ and let $f : \mathbb{R}^n \to \mathbb{R}$ be a one-dimensional linearly affine function over $\mathbb{R}^n$. If P is non-empty and $(\forall \vec{x} \in P : f(\vec{x}) \geq 0)$ then f is a linearly affine combination of the linear inequalities bounding P:

$$\left(\exists \gamma_0, \gamma_1, \ldots, \gamma_m \in \mathbb{R}_0^+ : \left(\forall \vec{x} \in P : f(\vec{x}) = \gamma_0 + \sum_{i=1}^{m} \gamma_i \cdot (-A^{(i,\bullet)} \cdot \vec{x} + \vec{\beta}^{(i)})\right)\right).$$

2.2.1.5 Volume of $\mathbb{Z}$-polytopes

In the polyhedron model, being able to calculate the volume of $\mathbb{Z}$-polytopes, i.e., the number $\operatorname{card}(P)$ of points contained in a $\mathbb{Z}$-polytope P, is useful for many purposes. These include calculating the memory traffic of a statement as the number of different accesses to memory cells at different execution steps (compare Bastoul and Feautrier [18] although they did not calculate the precise volumes) or calculating the number of iterations of a loop nest as part of an estimate of the loop nests amount of computation. Verdoolaege et al. [157] mention further use cases.

Barvinok's original counting algorithm [15] permits to determine the cardinality of a d-dimensional $\mathbb{Z}$-polytope with constant bounds. The execution time is exponential in $\mathcal{O}(d)$.

Based on Barvinok's work, Verdoolaege et al. [157] proposed a method to express the cardinality of a d-dimensional $\mathbb{Z}$-polytope with parametric bounds. The outcome of their function is a piece-wise quasi-polynomial or step-polynomial [157] in the parameters of the $\mathbb{Z}$-polytope's bounds. The execution time remains exponential.

2.2.1.6 Relations of Integer Set

Let $n, m \in \mathbb{N}$ and let $S \subseteq \mathbb{Z}^n$ and $T \subseteq \mathbb{Z}^m$. Relations between S (the domain) and T (the range) are subsets of $S \times T$. The cross-product can be viewed as an integer set of dimension $m + n$, which we denote as follows:

$$S \times T = \left\{ \begin{pmatrix} \vec{x} \\ \vec{y} \end{pmatrix} \mid \vec{x} \in S \ \wedge\ \vec{y} \in T \right\}.$$

Relations of integer sets may be constrained by linearly affine (in-)equalities.

Further, we need the definition of the lexicographic maximum of a relation $R \subseteq S \times T$ [153]:

$$\operatorname{lex.max}(R) = \left\{ \begin{pmatrix} \vec{x} \\ \vec{y} \end{pmatrix} \mid \begin{pmatrix} \vec{x} \\ \vec{y} \end{pmatrix} \in R \ \wedge\ \neg \left(\exists \vec{z} : \begin{pmatrix} \vec{x} \\ \vec{z} \end{pmatrix} \in R \ \wedge\ \vec{y} \prec \vec{z} \right) \right\}.$$

The lexicographic minimum of a relation is defined analogously.

2.2.2 Polyhedral Optimization Framework

At the start of Section 2.2, we briefly illustrated how loop nests can be represented by $\mathbb{Z}$-polyhedra. We believe that this way of abstracting loops can well be considered as the core idea of the polyhedron model. The previous subsection expounds the mathematical foundation of the polyhedron model.

Listing 2.1: Computation of a symmetric rank-k update: $C = \alpha \cdot A \cdot A^T + \beta \cdot C$.

```
for (int i = 0; i < n; i++) {
 for (int j = 0; j <= i; j++)
   C[i][j] *= beta; // statement R
 for (int j = 0; j < m; j++)
   for (int k = 0; k <= i; k++)
     C[i][k] += alpha * A[i][j] * A[k][j]; // statement S
}
```

Based on that, we recall the model itself. We start by discussing its expressiveness and the limits of its applicability (Section 2.2.2.1). We continue by laying out the details of the representation of programs in the polyhedron model (Section 2.2.2.2) and the determination of legal program transformations within the model (Section 2.2.2.3). Subsequently, we cover the representation of parallelism and tiling in the polyhedron model (Section 2.2.2.4). Finally, we provide an overview of the widely known and used PLuTo scheduling algorithm by Bondhugula et al. [25] and its derivatives (Section 2.2.2.5). We use this algorithm as a baseline for our proposed iterative optimization technique. We do not discuss the detection of code regions that are relevant for polyhedral optimization and polyhedral code generation (i.e., the reverse transformation of the model to code) [16, 66, 131] since they are not relevant for this thesis.

All concepts presented in this subsection are commonly known and applied building blocks of the polyhedral optimization framework.

2.2.2.1 Expressiveness and Limitations

Code regions must be *static control parts* (*SCoPs*) to be representable in the polyhedron model.

All loops in a SCoP must be `for`-loops with a constant stride. The code must operate on linear data structures that must not alias (overlap) in memory. Any functions called in a SCoP must be free of side-effects. In many cases, it may be beneficial to inline called functions. This is particularly useful if the called function's body contains loops. The code region may contain conditional constructs (`if`s). Loop bounds and branch conditions must be linearly affine expressions in structure parameters and iteration variables of encasing loops. A *structure parameter* is an integer variable that exists in the context of the SCoP and whose value remains unchanged throughout the SCoP's execution. `A[n] + i <= m` is non-affine, irrespectively of whether `A[n]` changes during the execution of the SCoP. A SCoP must have a single entry point and a single exit point. This prohibits many use cases of `return` and `goto`, and `continue` and `break`.

The validity conditions must hold recursively in that each subregion of a SCoP must again be a SCoP.

Early definitions of static control require the stride of any loop to be 1 [51].

Wider definitions of static control and transformations to enable these exist: Benabderrahmane et al. [21] expand static control to `while`-loops with arbitrary bounds. They rely on overapproximations in SCoPs' models. Also, they are able to handle non-affine conditionals. This includes data-dependent conditionals. The Polyhedral Extraction Tool [155] supports data-dependent accesses, assignments, and conditionals up to certain restrictions.

Example 2.2.12. To illustrate static control, we show a slightly modified version of the benchmark program `syrk` of PolyBench 4.1 [121] in Listing 2.1. The example is a symmetric rank-k update: given two matrices $A \in \mathbb{R}^{n \times m}, n, m \in \mathbb{N}$ and $C \in \mathbb{R}^{n \times n}$ and values $\alpha, \beta \in \mathbb{R}$, the code computes $C = \alpha \cdot A \cdot A^T + \beta \cdot C$. In this example, the structure parameters of the SCoP are the variables `n` and `m`. This SCoP serves as a running example throughout the thesis. ◁

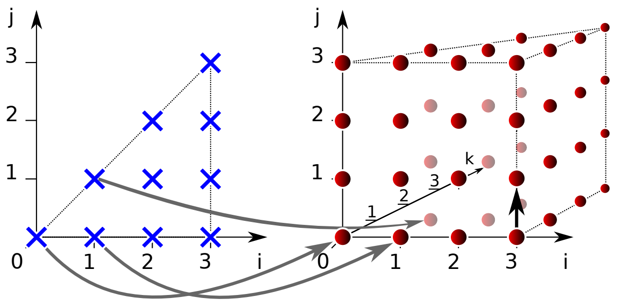


Figure 2.13: Iteration domains of the SCoP in Listing 2.1. The crosses on the left are the instances of statement R; the bullets on the right are the instances of S. The arrows represent data dependences. They connect source to target. For perceptibility, we show only exemplary instances of the dependences. The dependence arrows shown replicate throughout the iteration domain according to the patterns hinted.

2.2.2.2 Modeling Programs

A SCoP's model consists of three components: the statements' iteration domains, the schedule and the statements' memory access relations. The iteration domains define "what" will be computed. The schedule captures the execution order. A transformation of a SCoP will be, in effect, a replacement of its schedule. Finally, the memory access relations link iteration domains to areas of memory that will be accessed during program execution. Analysis of these relations reveals data dependences. The way of representing SCoPs in the polyhedron model that we present here has been proposed by Cohen et al. [43].

Iteration Domain A *statement instance* is a single execution of a statement within a single run of a SCoP. Consequently, statements that are encased by loops have multiple instances. The polyhedron model represents programs at the statement instance level. In this representation, a statement instance is represented by its *iteration vector*, that is, the corresponding values of the encasing loops' iterators. The order of the vector's components resembles the nesting order of the loops, from outer to inner. The single instance of a statement that is not surrounded by loops has a zero-dimensional iteration vector, i.e., an *iteration scalar*.

Given a statement S, the set of all of its instances' iteration vectors is called the statement's *iteration domain*, which we denote by I_S. Since loop iterators are integer variables, iteration domains are integer sets. Thanks to static control, these sets are expressible as unions of polyhedra. The linearly affine constraints that bound the polyhedra correspond to the loop bounds and to conditionals that guard a statement. If the concrete number of iterations of a loop is unknown statically, these constraints are parameterized. The dimensionality of the polyhedra that represent a statement's iteration domain corresponds to the number of loops that encase the statement.

Example 2.2.13. The iteration domains of the SCoP in Example 2.2.12 are represented by the following integer sets, which are both polyhedral:

$$I_R = \{(i, j)^T \mid i, j \in \mathbb{Z} \ \wedge \ 0 \leq i < n \ \wedge \ 0 \leq j \leq i\}$$
$$I_S = \{(i, j, k)^T \mid i, j, k \in \mathbb{Z} \ \wedge \ 0 \leq i < n \ \wedge \ 0 \leq j < m \ \wedge \ 0 \leq k \leq i\}.$$

$n, m \in \mathbb{Z}$ are parameters that correspond to the SCoP's structure parameters.

Figure 2.13 illustrates the two polyhedra for $n = m = 4$. ◁

Memory Access Functions Besides statements' iteration domains, the model must capture statements' memory accesses. Arrays and other linear data structures are represented as integer sets, which are called *memory locations*. Memory accesses are represented as relations between iteration domains and memory locations. One must distinguish between read and write accesses.

Example 2.2.14. Statement R of our running example has the following read access functions (the subscript denotes the accessed memory location):

$$read^R_C(i,j)^T = (i,j)^T,\ read^R_\beta(i,j)^T = (),\ write^R_C(i,j)^T = (i,j)^T.$$

The access functions of statement S are

$$\begin{aligned}&read^S_C(i,j,k)^T = (i,k)^T,\ write^S_C(i,j,k)^T = (i,k)^T,\\ &read^S_\alpha(i,j,k)^T = (),\\ &read^S_A(i,j,k)^T = (i,j)^T,\ read^S_A(i,j,k)^T = (k,j)^T.\end{aligned}$$

$\triangleleft$

Schedules Iteration domains describe the instances of a SCoP's statements, but they do not describe the statement instances' execution order. Execution order is represented by a multi-dimensional linearly affine function with integer coefficients. This function is called a *schedule*. A schedule can be represented by one function per statement. Each statement schedule is a linearly affine expression in the SCoP's structure parameters and the loop iterators of the statement's surrounding loops. The coefficients should be integers. Without loss of generality, we may assume that the ranges of all statement schedules of a SCoP have the same dimensionality. Execution order is given by the transformed iteration vectors' lexicographic order.

Given a statement S, we denote its schedule by Θ_S. In case of a multi-dimensional schedule, we call its one-dimensional components *schedule dimensions*. We enumerate them from outermost to innermost. To address dimension d of Θ_S, we write $\Theta_S^{(d)}$. Similarly, we address the schedule dimensions d to e ($d < e$) by $\Theta_S^{(d..e)}$.

Program transformation works by replacing the schedule in a SCoP's model.

Example 2.2.15. The running example has the following schedule:

$$\Theta_R(i,j)^T = (i,0,j,0)^T,\ \Theta_S(i,j,k)^T = (i,1,j,k)^T.$$

$\triangleleft$

2.2.2.3 Dependence Analysis and Legality of Schedules

Dependences From memory accesses result data dependences. A *data dependence* from one statement instance, the source, to another, the target, exists if both instances access the same memory cell and the schedule prescribes that the former statement instance is executed before the latter. In the polyhedron model, one encodes data dependences by dependence polyhedra. A *dependence polyhedron* is a relation between the iteration domains of a source statement and a target statement. It captures a set of pairs of dependent statement instances that can be expressed using linearly affine inequalities. In some cases, these polyhedra overapproximate the true set of dependent statement instances. For a dependence polyhedron that involves source statement O (origin) and target statement T (target), we write $D_{O,T}$.

Depending on the type of each of the memory accesses, one distinguishes the following kinds of data dependences:

Flow Dependence The source access is a write and the target access is a read.

Anti Dependence The source access is a read and the target access is a write.

Output Dependence Both accesses are write accesses.

Input Dependence Both accesses are read accesses.

Legality of Schedules The first three kinds of dependences affect the legality of schedules. These are the dependences that involve at least one write access. Any schedule that would assign an execution step to the target statement instance of a legality-affecting data dependence that is prior to the dependence's source statement instance must be considered illegal since it may yield a different computation result of the transformed program. In the case of flow and anti dependences, one is only interested in those dependences, for which no other write to the same memory cell occurs in between. In the case of output dependences, one can omit those for which an intermediate write or read to the same memory cell exists. The original exact data flow analysis algorithm, which omits transitive dependences as just described, has been proposed by Feautrier [51]. We use the approximative method described by Verdoolaege [153].

Example 2.2.16. Our running example `syrk` has the following dependence polyhedra (excluding input dependences):

$$\begin{aligned} D_{R,S} &= \{(i,j,i,0,j)^T \mid i < n \ \wedge \ 0 \le j \le i \ \wedge \ (i,j)^T \in I_R \ \wedge \ (i,0,j)^T \in I_S\} \\ D_{S,S} &= \{(i,j,k,i,j+1,k)^T \mid 0 \le i < n \ \wedge \ 0 \le j \le m-2 \ \wedge \ 0 \le k \le i \\ &\qquad \wedge \ (i,j,k)^T \in I_S \ \wedge \ (i,j+1,k)^T \in I_S\}. \end{aligned}$$

◁

A schedule Θ is *legal* iff $\big(\forall (\vec{i}\ \vec{j})^T \in D_{O,T} : \Theta_O(\vec{i}) \prec \Theta_T(\vec{j})\big)$ holds for all dependence polyhedra $D_{O,T}$ [53] that affect legality. A dependence that affects legality and for which this condition does not hold with respect to Θ is *violated* by Θ. With respect to one-dimensional schedules, the terms weak and strong satisfaction are common. Dimension d of a schedule Θ *satisfies* dependence polyhedron $D_{O,T}$ *weakly* if $\big(\forall (\vec{i}\ \vec{j})^T \in D_{O,T} : \Theta_O^{(d)}(\vec{i}) \le \Theta_T(\vec{j})\big)$ holds. The dependence polyhedron is *satisfied strongly* under the following condition: $\big(\forall (\vec{i}\ \vec{j})^T \in D_{O,T} : \Theta_O^{(d)}(\vec{i}) < \Theta_T(\vec{j})\big)$. The first dimension of a schedule that satisfies a dependence polyhedron strongly, *carries* the dependence polyhedron. Analogously, a multi-dimensional schedule carries $D_{O,T}$ if $\big(\forall (\vec{i}\ \vec{j})^T \in D_{O,T} : \Theta_O^{(d)}(\vec{i}) \prec \Theta_T(\vec{j})\big)$. Replacing $\prec$ by $\preceq$ yields the definition of weak satisfaction.

A concept that is related to weak and strong satisfaction of dependence polyhedra is a dependence polyhedron's direction with respect to a schedule dimension d. The direction of a dependence polyhedron with respect to d is defined if the value of $\mathrm{sgn}(\Theta_T^{(d)}(\vec{j}) - \Theta_T^{(d)}(\vec{i}))$ is the same for all $(\vec{i}\ \vec{j})^T \in D_{O,T}$: If the value is always 1, the dependence goes forward. If the value is 0, the dependence is orthogonal to d. -1 is backward. The direction of a dependence polyhedron may not always be defined: for instance, some elements of the polyhedron may point forward, while others point backward.

These concepts also apply to dependences between single pairs of statement instances.

Computing Legal Schedules Using Farkas' Lemma, it is possible to compute the set of all one-dimensional schedules that satisfy a dependence polyhedron $D_{O,T}$ weakly or strongly [53, 122, 124]. Let $\vec{p}$ be the vector of the SCoP's structure parameters, $\vec{i_O}$ the vector of the iteration variables of the loops encasing statement O, and $\vec{i_T}$ the respective vector for statement T. Thus, one can write:

$$D_{O,T} = \left\{ \begin{pmatrix} \vec{i_O} \\ \vec{i_T} \end{pmatrix} \mid M_{O,T} \cdot \begin{pmatrix} \vec{i_O} \\ \vec{i_T} \\ \vec{p} \\ 1 \end{pmatrix} \le \vec{\beta_{O,T}} \right\}.$$

One-dimensional statement schedules can be written as follows:

$$\Theta_S(\vec{i_S}) = \begin{pmatrix} \vec{\lambda_S} \\ \vec{\mu_S} \end{pmatrix} \cdot \begin{pmatrix} \vec{i_S} \\ \vec{p} \end{pmatrix} + \nu_S.$$

$\vec{\lambda_S}$ is the coefficient vector for $\vec{i_S}$, and $\vec{\mu_S}$ is the coefficient vector for $\vec{p}$.

For $D_{O,T}$ to be satisfied weakly, $\Theta_T(\vec{i_T}) - \Theta_O(\vec{i_O}) \geq 0$ must hold for any $(\vec{i_O}, \vec{i_T})^T \in D_{O,T}$. Thus, Farkas' Lemma is applicable:

$$\begin{aligned}\Theta_T(\vec{i_T}) - \Theta_O(\vec{i_O}) &= \underbrace{\begin{pmatrix} \vec{\lambda_T} \\ \vec{\mu_T} \end{pmatrix} \cdot \begin{pmatrix} \vec{i_T} \\ \vec{p} \end{pmatrix} + \nu_T - \begin{pmatrix} \vec{\lambda_O} \\ \vec{\mu_O} \end{pmatrix} \cdot \begin{pmatrix} \vec{i_O} \\ \vec{p} \end{pmatrix} - \nu_O}_{(1)} \\ &= \underbrace{\gamma_0 + \sum_{i=1}^{\mathrm{rows}(M_{O,T})} \left(\gamma_i \cdot \left(-M_{O,T}{}^{(i,\bullet)}\right) \cdot \begin{pmatrix} \vec{i_O} \\ \vec{i_T} \\ \vec{p} \\ 1 \end{pmatrix} + \vec{\beta_{O,T}}^{(i)} \right)}_{(2)};\end{aligned}$$

$$\gamma_0, \ldots, \gamma_{\mathrm{rows}(M_{O,T})} \in \mathbb{R}_0^+.$$

After factoring out each iteration variable and structure parameter in (2), the counterpart of each iteration variable's coefficient and each structure parameter's coefficients in (1) can be identified in (2). The same holds for ν_O and ν_T. Together with the positiveness of each γ_i the resulting equation system represents the set of coefficient vectors of all schedule functions that satisfy $D_{O,T}$ weakly. This set is a polyhedral cone.

2.2.2.4 Tiling and Parallelization

Tiling [73, 74] is a non-affine loop transformation that can optimize data locality. The transformation is beneficial if statement instances that are close together in a spatial sense operate on the same memory addresses or nearby memory addresses. In such a case, the CPU caches can be exploited by executing these statement instances closely together, both, in a temporal and spatial sense (i.e., on the same CPU core, or the same compute unit of a GPU). In addition, tiling can help to improve the effectiveness of coarse-grained loop parallelization by controlling the workload per thread and, thereby, steering the synchronization and communication overhead [73].

To tile a loop nest, one blocks each loop and then permutes the loops that enumerate the blocks (the *tile loops*) to the outside and the loops that iterate inside the blocks (the *point loops*) to the inside.

Per schedule dimension that corresponds to the loop nest to be tiled, tiling adds an additional schedule dimension. A sufficient criterion for the legality of tiling an (imperfect) loop nest's schedule $\Theta^{(d,e)}, d < e$, is the criterion given by Bondhugula et al. [28]: for any data dependence from an instance with iteration vector $\vec{i}$ of a statement O to an instance with iteration vector $\vec{j}$ of statement T that is uncarried by schedule dimensions $1, \ldots d-1$ we must have

$$\left(\forall k \in \{d, \ldots, e\} : \Theta_T^{(k)}(\vec{j}) - \Theta_O^{(k)}(\vec{i}) \geq 0\right).$$

This criterion guarantees that dependences are orthogonal or point forward with respect to all dimensions of Θ. A sequence of schedule dimensions to which tiling is applicable is a *tilable band.*

There are two ways to encode parallelism in a schedule. The first way is assigning the same execution step to multiple statements. The second way are schedule dimensions $\Theta^{(d)}$ that encode loops and to which all dependences that are uncarried by $\Theta^{(1..d-1)}$ are orthogonal.

2.2.2.5 The PLuTo Scheduling Algorithm and its Derivatives

The PLuTo algorithm [25] is a model-based approach to polyhedral schedule optimization based on Farkas' Lemma and parametric integer programming. Employed by many polyhedral optimizers, among them LLVM's Polly[65], GCC's[4] Graphite [147], PLuTo [26], PoCC [126] and PPCG [158], it is probably the most widely used contemporary polyhedral scheduling algorithm. PLuTo optimizes for tiling and data locality. Using an objective function, the algorithm minimizes the reuse distance of data dependences. The PLuTo algorithm starts from a dependence graph (V, E) of a SCoP. V contains the SCoP's statement instances and E contains all data dependences between these instances, except input dependences. The algorithm computes a sequence of schedule dimensions that satisfy all dependences in E weakly. Per statement, each schedule dimension is linearly independent in the coefficients of the statement's iteration variables to all previous schedule dimensions. Also, the trivial solution in which all schedule coefficients are zero is avoided. The linear independence of the produced schedule dimensions ensures that each of the dimensions yields a loop in the transformed program. Thus, the algorithm computes a tilable band.

If no additional schedule dimension can be appended to the current band, the algorithm removes the dependences from E that are carried by the current band. Then, the algorithm analyzes the dependence graph associated to E and determines its strongly connected components. It determines a legal execution order for the strongly connected components and appends a schedule dimension that encodes the determined order. The algorithm continues recursively per strongly connected component of the dependence graph until all dependences are carried and each statement schedule encodes a number of loops that equals the dimensionality of the statement's iteration domain.

The original PLuTo algorithm makes a tradeoff to avoid a combinatorial explosion during the determination of program transformations. It requires that, per statement and schedule dimension, the sum of the iteration variables' coefficients must be positive. This excludes some loop transformations that require negative iterator coefficients [28]. For instance, a loop reversal that is not combined with another loop transformation will be impossible, while a loop reversal combined with positive skewing (e.g., $-j + 2 \cdot i$) will be allowed.

The scheduling algorithm implemented in the Integer Set Library (isl) by Verdoolaege [152, 153] (the "isl scheduler" [156]) and PLUTO+ [28] are extensions of the PLuTo algorithm that remedy this limitation.

In the schedules constructed by the original PLuTo algorithm, parallelism occurs by coincidence as a side-effect of optimizing temporal data locality. The isl scheduler differs in this aspect since it can optionally try to force each band to have an outer parallel dimension. If the algorithm fails to force such a dimension, it uses the scheduling algorithm by Feautrier [52, 53] to compute the dimension. In this case, the resulting dimension carries as many of the remaining dependences as possible and, thus, they can be removed from E.

Zinenko et al. [171] identified a conflict in the optimization for parallelization on the one hand and the maximization of temporal and spatial locality on the other hand. They propose an extension of the isl scheduler with the required tradeoff in mind. The extended algorithm is suitable for scheduling for both CPU and GPU targets.

In our work, we use performance yielded by the schedules found by the original isl scheduler as a baseline for the schedules found by our iterative optimization technique.

2.3 Machine Learning and Iterative Optimization Techniques

We move the focus to techniques of iterative (or search-based) optimization and machine learning that we rely on. We start by introducing genetic algorithms which are a way of searching for a good solution in a search space of candidate solutions to a given optimization

[4]`https://gcc.gnu.org/`

problem (refer to Section 2.3.1). In the absence of the ability to construct the optimal solution directly, genetic algorithms can improve over random search by traversing the search space with some guidance.

Section 2.3.2 introduces decision trees and random forests, which are techniques of supervised machine learning. We use them to learn performance models of program transformations from training sets of transformations whose profitability is known.

2.3.1 Genetic Algorithms

In the absence of an efficient algorithm to construct the optimal solution to a given optimization problem, *genetic algorithms* [42, 68, 71, 100] may be able to find solutions that are acceptably profitable. For a genetic algorithm to be successfully applicable, two preconditions must hold [42]: candidate solutions should be cheap to generate and the profitability or *fitness* of a solution should be computationally cheap to determine. Further, genetic algorithms will have difficulties to improve the quality of a solution if profitable solutions are only located in sharply defined, small regions of the search space.

The principle of genetic algorithms [71] is inspired by natural reproduction and selection. Starting from an initial *population* of solutions, or *chromosomes*, a genetic algorithm determines each solution's profitability. Solutions will then be crossed to produce a new population as an offspring of the old population. To introduce new genetic material into the population, the offspring will further be mutated. The likeliness of a solution to reproduce increases with its profitability. Instead of population we also use the term *generation*. The procedure terminates as soon as some termination criterion holds.

We use two extensions to the general schema of genetic algorithms: *Elitism* [12] lets the most profitable solutions in a generation survive to the next generation unmodified. Thereby, the maximum fitness in a population cannot be smaller than in the previous population. Pouchet et al. [124] combine genetic algorithms with the idea of *simulated annealing* [68]: the higher the number of already produced generations, the closer one is to the discoverable optimum, and the more local the search has to become. This can be achieved by scaling the intensity of mutations by a factor that is inverse to the number of produced generations.

Pouchet et al. [124] proved that genetic algorithms can be used successfully for schedule optimization in the polyhedron model.

2.3.2 Supervised Machine Learning

Supervised machine learning subsumes techniques that serve to learn a predictor function $f : A \to B$. A is a set of vectors of values for independent variables or *features*. B is the value range of a *dependent variable*. f is supposed to predict the dependent variable's value for vectors in A. To derive f, a machine-learning algorithm analyzes a subset of $A \times B$, the *training set*. For this to work, the independent variables and the dependent variable must correlate, but the correlation need not be known. The training set must be representative for $A \times B$ to allow for an accurate predictor.

2.3.2.1 Decision Trees

A *classification and regression tree* (*CART*) [32] is one technique of supervised machine learning. A CART is the result of recursively splitting the training set into two subsets until each subset is homogeneous in terms of the dependent variables' values or a maximum splitting criterion, such as the minimum size of the set, is reached. Each split yields a node in a binary decision tree that stores the predicate on the independent variable according to which the split was made. Subsets of the training set that do not get split further during training are represented by the leaf nodes of the decision tree. Each leaf stores the majority values of the dependent variable for the corresponding subset of the training set.

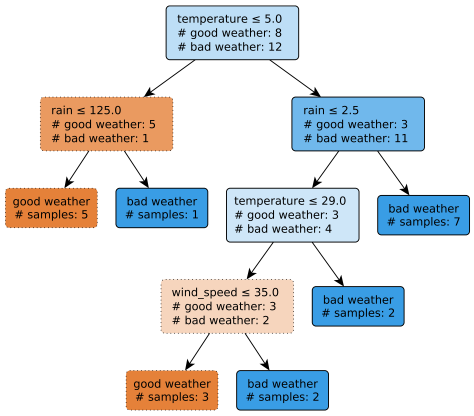


Figure 2.14: A decision tree that can distinguish good and bad weather conditions. Per node, we show the number of corresponding training samples grouped by weather condition.

To determine optimal splitting criteria, one uses measures of impurity such as the Gini[5] index of diversity [32].

To make a prediction for an element $a \in A$, one evaluates the decision tree's root node's predicate for a and recursively continues with the subtree indicated by the evaluation of the predicate. Having reached a leaf node, the prediction is the value stored by the leaf. Example 2.3.1 illustrates the technique.

Example 2.3.1. Figure 2.14 shows a decision tree classifier to assess the weather condition based on temperature ($°C$), wind speed (km/h), humidity ($\%$), and rain fall (mm).

From Figure 2.14 one can verify that a temperature of 20 $°C$, 10 km/h of wind speed, an air humidity of 50% and rain fall of $0mm$ means that the weather is good. $\triangleleft$

Decision trees are susceptible to overfitting. An *overfitted* classifier is one that resembles its training set closely, but, in contrast, has a poor predictive power with respect to the correlation between A and B that ought to be detected. Among the techniques to reduce overfitting are pruning of subtrees and not splitting the training into subsets with a size below a given limit. Especially pruning is known to be effective.

2.3.2.2 Random Forests

Random forests [31] are another approach to reduce the overfitting effect of CART. Instead of one CART, one learns several. Each tree is learned from a subset of the training set and its nodes' predicates cover only a subset of the independent variables. A random forest's prediction is the average (or majority) prediction of its individual trees.

[5]named after the Italian statistician Corrado Gini

3 Related Work

We propose an iterative (or search-based) technique to schedule optimization in the polyhedron model. Furthermore, we propose to reduce the benchmarking effort that must be spent to optimize programs iteratively by using surrogate performance models. The models are learned from schedules that result from previous iterative optimizations and the schedules' respective measured execution times. Our models characterize schedules using a set of structural features and features that are related to particular performance aspects. We do not use program features and, therefore, do not distinguish between programs of different structure, which would allow to classify programs and would allow to apply a different schedule performance model for each program class. The use of program features would also require a comprehensive training set of programs, though. Our models are classifiers that can distinguish likely profitable from likely unprofitable schedules.

This chapter presents work that is related to ours. Primarily, we cover polyhedral schedule optimization techniques that are either iterative, model-based, or hybrid. Hybrid techniques combine iterative search and model-based optimization. Model-based techniques may either use a static heuristics, which, in the case of polyhedral schedule optimization, is frequently encoded in a linear program or a series of linear programs, or can rely on machine-learned performance models. We describe work from the three categories named in the respective Sections 3.1, 3.2, and 3.3.

Polyhedral features of schedules and programs can serve other purposes than machine learning and iterative optimization. We describe work that uses polyhedral features or, more generally, characterizations of schedules or programs for other purposes than machine learning in Section 3.4.

In Section 3.5, we present approaches to iterative compilation and uses of machine learning in compilation that do not rely on the polyhedron model.

Ashouri et al. [9] present an extensive survey of compiler autotuning using machine learning. The survey also covers iterative approaches that do not rely on machine learning. It comprises approaches that are in the context of the polyhedron model.

3.1 Iterative Polyhedral Scheduling Algorithms

Iterative polyhedral optimizers do not rely (entirely) on static or learned performance models. Instead, to optimize a program, they test multiple samples from a search space of potential schedules and assess each schedule's profitability, for instance, in terms of speedup in execution time. The search space traversal can be at random, it may be directed by a traversal strategy, which can rely on a performance model, it can be an enumeration that optionally employs a pruning strategy to avoid the exploration of likely futile subsets of the search space, or it may be feedback-driven such as a genetic algorithm.

Some early approaches to iterative polyhedral schedule optimization differ strongly from ours because they put legality constraints aside and consequently suffer from many illegal schedules. All are based on the Unified Transformation Framework (UTF) [78]. GAPS [104] uses a genetic algorithm to find polyhedral schedule transformations. GAPS starts from a random population of potentially illegal transformations that is seeded by one legal transformation. The fitness function either measures execution time of transformed programs or predicts loop and synchronization overhead. In the experiments of Nisbet et al., at most 5.5% of 20,000 schedules generated were legal. ICE [105] is built on top of GAPS. It enables the genetic algorithm to apply more transformations, including tiling. ICE has new mutation operators in addition to the ones that exist in GAPS. The genetic operators can account for profiling data. The fitness function measures execution time. Long and O'Boyle [93] optimize JAVA programs adaptively. They machine-learn the performance impact of loop transformations that are expressible in UTF. To match unknown programs with learned strategies, program similarity is expressed by a set of features. The feedback from the execution of each program optimized refines the model learned. While not evaluating multiple schedules during the compilation of a single program, this approach is similar to ours because it learns models from information gathered during the optimization and subsequent execution of programs optimized and uses these models to optimize unseen programs. Long and Fursin [91, 92] also optimize JAVA code. They separate the search of mappings of the iteration variables (tiling, skewing) and the exploration of the constant part of schedules (loop distribution, fusion). Run-time feedback directs the search. We do not make this distinction of loop schedule and constant part of the schedule in POLYITE.

We build on the approach to iterative polyhedral schedule optimization by Pouchet et al. [122, 124], which was designed to optimize the sequential execution time of programs in the absence of tiling iteratively. The approach's particularity is that it restricts its search to schedules that retain program semantics. Thereby, no illegal schedules need to be identified and purged after their construction as is necessary in the early approaches mentioned. The primary exploration strategy by Pouchet et al. relies on the enumeration of the schedules in a subspace of the search space and a completion strategy for the remaining search space dimensions. To overcome the search space's size, Pouchet et al. propose a decoupling heuristics that prohibits the exploration of subsets of the search space that are likely unprofitable and a genetic algorithm with tailored genetic operators under which their search space of legal schedules is closed. Notable is their thorough search space sensitivity analysis [123]. To optimize for tiling and parallelization in a purely iterative manner, we go beyond the approach of Pouchet et al. We avoid search space restrictions that are viable only for sequential execution and propose new sampling strategies for schedules. Instead of their decoupling heuristics, we use non-uniform random sampling as our basic sampling technique for schedules. Also, we propose new genetic operators that are more suitable for the traversal of our much less constrained search space.

Trifunovic et al. [146] propose a static cost model for loop vectorization and propose a search-based scheduling approach that finds the schedule with the least vectorization cost in a search space that comprises all combinations of loop interchanges for a SCoP.

Park et al. [114, 116] compose schedules from sequences of high-level polyhedral transformations such as loop fusion, tiling, or prevectorization. Each high-level transformation needs an enabling transformation that permits its semantics-preserving encoding into a schedule. They characterize programs using performance counters and use several models to predict the speedup that can be expected from the application of a certain sequence of high-level optimizations to a given program. Their approach may either output a single optimization sequence that the trained performance model considers to be optimal for the program to be optimized, or it may output a set of optimization sequences. In the latter case, the most profitable sequence must be identified by benchmarking. It is never necessary to evaluate more than six program versions. The search space explored by Park et al. is less finely grained than ours. However, in contrast to us, they have the ability to learn models that are applicable to a wider range of programs as they can characterize and classify programs.

Ruvinskiy and van Beek [134] build on the approach by Park et al. [114, 116]. Other than Park et al., who rely on regression, Ruvinskiy and van Beek learn a classifier to predict which of two sequences of primitive high-level transformations is better for a given program. They are able to choose a single optimal schedule directly instead of having to benchmark several schedules.

Kronawitter and Lengauer [87] use polyhedral schedule features to prune schedules in a schedule search space exploration for the data locality optimization, parallelization, and vectorization of stencil codes. They describe the schedule search space that they explore as practically equal to the search space constructed by PLUTO+ [28]. Therein lies a major difference to our approach. Some of their filters are applicable early in the exploration process and, thereby, not only avoid the schedules' evaluation by benchmarking but also reduce the time needed for the exploration of the search space in general. Like us, Kronawitter and Lengauer rely on Chernikova's algorithm (refer to Section 2.2.1.2) to sample their schedule search space. Instead of performing a non-uniform random exploration like us, they enumerate deterministically by combining the generators of the polyhedra that represent their search space.

Vasilache et al. [151] propose TENSOR COMPREHENSIONS (TC), which is a domain-specific language in the context of deep neural networks. They propose a polyhedral just-in-time (JiT) compiler for TC that leverages the ISL scheduler [156]. The JiT compiler uses a compilation cache that is populated with CUDA kernels generated by an autotuning of the polyhedral program optimization's configuration. The autotuner uses a genetic algorithm. Among the options tuned are tile, block, and grid sizes, loop unrolling bounds, choices for loop fusion and distribution, and several lower-level GPU specific options. In contrast to us, Vasilache et al. use the static PLUTO scheduling algorithm and autotune only the optimization's configuration but not the schedule itself.

Sato et al. [136] autotune tile sizes and rely on LLVM [89] and POLLY [65]. This work is orthogonal to ours. It autotunes tile sizes but relies on the static PLUTO scheduling algorithm. Conversely, we use fixed tile sizes but search for an optimal program schedule.

3.2 Model-Based Polyhedral Scheduling Algorithms

Frequently used scheduling algorithms in contemporary polyhedral compilers are model-driven. A parametric integer programming algorithm [50] (refer to Section 2.2.1.3) is used to select lexicographically minimal solution vectors from a polyhedron. Most notable are the algorithm by Feautrier [52, 53] and PLUTO [25]. Feautrier's algorithm reduces latency. Feautrier [52] states the generator representation of a polyhedron together with Chernikova's algorithm as an alternative way of solving the constraint system of his algorithm. PLUTO (refer to Section 2.2.2.5) optimizes for tiling and data locality. The Integer Set Library (ISL) by Verdoolaege [152] comprises a generalization of PLUTO, the ISL scheduler [156]. Another generalization is PLUTO+ [28]. Both generalizations have fewer practical limitations than the original PLUTO algorithm. Particularly, they allow for arbitrary transformations that require negative coefficients for iteration variables. The ISL scheduler has the ability to force each band to have an outer parallel dimension. If the algorithm fails to force such a dimension, it uses the scheduling algorithm by Feautrier [52, 53] to compute the dimension and, thereby, obtain a schedule dimension that carries as many data dependences as possible.

Advances over the original PLUTO algorithm exist. They target the balance between loop fusion and distribution and the conflicting demands of parallelism and data locality. Loop distribution enables parallelism, but it reduces data locality. Loop fusion does the opposite. Bondhugula et al. [27] proposed a loop fusion model that is expressible as an ILP. Zinenko et al. [171] improve the ISL scheduler [156] to take spatial proximity of memory accesses into account explicitly. Live-range reordering [11] improves the applicability of tiling.

Pradelle et al. [128] generate multiple parallelized versions of loop nests using the polyhedron model. At run time, they select the best version of each loop nest, per invocation. The selection is based on an execution time prediction that relies on a loop iteration count and offline benchmarking of training programs with different numbers of threads. To count loop iterations efficiently at run time, Pradelle et al. rely on Ehrhart polynomials [157].

In an approach to implement deep neural networks on FPGAs, Zhang et al. [169] leverage the PLuTo algorithm and an analytical performance model that allows them to tune variable design aspects. They use ILP for an efficient design space exploration.

3.3 Hybrid Polyhedral Scheduling Algorithms

Pouchet et al. [126, 127] combine iterative and model-driven optimization in the polyhedron model. The result is an automatic parallelization framework in which tiling, parallelization, and vectorization can be applied. They iteratively find a fusion structure (statement interleaving) and use the PLuTo scheduling algorithm to complete the schedule and achieve the applicability of tiling, parallelism, and data locality. They show that it is theoretically possible to express the set of all m-dimensional schedules for a program as one convex set. Thereby, if we limited our search to at most m-dimensional schedules, we could abolish the division of the search space into search space regions. On the other hand, Pouchet et al. state that this unified search space's high dimensionality makes its handling infeasible. Specifically, algorithms such as parametric integer programming, Chernikova's algorithm, and Fourier-Motzkin variable elimination [138] have exponential run-time complexity, which likely makes their application on the resulting schedule search space representation infeasible. The approach's iterative part is limited to finding the optimal multi-dimensional statement interleaving, whereas the schedules are completed by the static PLuTo scheduling algorithm. We see this approach as orthogonal to ours: both approaches target data locality and, consequently, may enable tiling. Also, both approaches enable parallelism. The primary difference lies in the fact that Pouchet et al. optimize only partly by search space exploration, whereas our technique is purely search-based.

3.4 Other Use Cases of Polyhedral Features of Schedules and Programs

Polyhedral features and, more generally, characterizations of programs or schedules can serve purposes other than machine learning.

Bao et al. [13] build on the observation that careful reductions in processor frequency do not reduce execution time significantly but can save energy. They employ polyhedral features that predict speedup from parallelization and operational intensity to statically categorize program regions and then select a frequency.

Zinenko et al. [170] propose a technique to transform schedule trees [66], which are a polyhedral hybrid representation of schedules that borrows from linearly affine schedule functions on the one hand and abstract syntax trees (ASTs) on the other hand. We use schedule trees in Polyite and recall their fundamental concept in Section 5.2. Zinenko et al. enable the specification of patterns ("matchers") to characterize declaratively parts of a schedule tree that are to be selected for modification. Further, a transformer can be specified that will be applied to all of the tree's parts that match the pattern. Their approach can be expected to ease the implementation of schedule transformations and analyses, such as our feature extraction, in polyhedral compilers.

3.5 Other Approaches to Machine Learning and Iterative Optimization in Compilation

This section describes work on search-based compilation (Section 3.5.1), compilation that combines iterative search and machine learning (Section 3.5.2), and compilation that uses machine-learned models without exploring a search space (Section 3.5.3) that is outside the context of the polyhedron model.

3.5.1 Iterative Approaches

Knijnenburg et al. [83] use several variants of iterative optimization to select loop unrolling factors and tile sizes. The techniques used include a genetic algorithm and simulated annealing. Furthermore, Knijnenburg et al. [84] propose to reduce the number of program versions that need to be tested in the iterative compilation with an approximative static cache model.

Cooper et al. [44] use virtual execution to generate feedback for an adaptive optimization process to find an optimal compiler phase sequence. To initialize the virtual execution, one initial profiling run of the program to optimize is required.

Chen et al. [39, 40] analyze in which circumstances iterative optimization for compiler option tuning works. Their research question is "[...] *if one selects a combination of optimizations based on runs over one or a few data sets, will that combination still be the best for other data sets?*" [39]. Chen et al. observe that across a set of 32 programs for each of which they generate 1000 data sets it was possible to find at least one combination of compiler optimizations that yields, per data set, at least 83% percent [39] (or 86% [40]) of the maximum speedup that was reachable for the data set. Furthermore, they research the question about why iterative optimization works [40].

Kelefouras [77] optimizes loop nests iteratively. The considered set of possible loop transformations comprises two levels of loop nest tiling with different tile sizes in each case, scalar replacement, register allocation, loop unrolling with different unroll factors, different data array layouts, and loop interchange. The approach is limited to loop nests that operate on arrays. All array accesses must use linear subscript functions. By taking characteristics of hardware and code into account, the size of the search space can be reduced drastically.

Beaugnon et al. [19] propose TELAMON, which optimizes CUDA kernels with static control in an iterative manner for an execution on GPUs. Within its search space for a kernel, TELAMON has the ability to efficiently find the kernel's version that has the globally minimal execution time. TELAMON's search space is represented by an internal representation (IR) of the kernel that can encode unspecified implementation choices. Thus, one instance of the IR can represent multiple programs. Conceptually, the search space is a decision tree that TELAMON explores with a branch-and-bound algorithm. When the exploration visits a leaf, i.e., a fully specified version of the kernel, Beaugnon et al. execute the kernel and measure its execution time. If necessary, they update the minimum execution time $t_{\min}$ known so far subsequently. Their analytical performance model can predict a lower bound for all kernel versions in a subtree of the decision tree. If, for a subtree, the predicted bound is greater than $t_{\min}$, the subtree will be pruned. Like POLYITE, TELAMON is designed to optimize programs without (fully) relying on static cost models. In contrast to us, Beaugnon et al. do not represent programs and program transformations in the polyhedron model. They do not machine-learn performance models but rely on an analytical performance model. Still, the optimization is directed by feedback from the execution of program versions.

3.5.2 Combined Machine-Learning and Iterative Approaches

Park et al. [113] evaluate different modeling techniques for an iterative search for a good compiler phase sequence. They rely on performance counters to characterize programs.

Ashouri et al. [8] also rely on dynamic features to optimize the order of compiler phases. That is, from profiling during a single program execution and a set of hand-crafted rules, they deduce the impact on execution time by a specific sequence.

Other approaches rely on static program features. Stock et al. [144] use machine learning to improve the automatic vectorization of tensor contraction codes and rely on features of assembly code. Their models rank different program variants according to their relative performance. The best can then be chosen. Agakov et al. [2] use predictive modeling to focus iterative compiler compilation on areas of the search space that are likely to give the best performance. The search space consists of sequences of several program transformations. They use program features to find correlations with training programs evaluated iteratively offline and select the model for the program class that fits the new program best. Agakov et al. evaluate two techniques to representing the profitable areas of the search space and use both random exploration and a genetic algorithm for the actual search space exploration. Ashouri et al. [7] use Bayesian networks to prune their exploration space in the compiler autotuning framework COBAYN and rely on static and dynamic program features and features that characterize the control flow.

Given a new program to be compiled, Thomson et al. [145] can decide whether the already present training data is suitable to optimize the program or additional search space exploration, i.e., retraining would be beneficial.

Chen et al. [38] optimize tensor programs using learned domain-specific statistical cost models and an exploration of a search space that consists of possible transformations from tensor operators to low-level code. During the search space exploration, they update the cost model continuously. The cost models are transferable between different tensor operators.

3.5.3 Machine-Learning Approaches

Monsifrot et al. [101] learn a decision tree for estimating how beneficial loop unrolling is for a given loop. They use static loop features to characterize loops. To obtain training data for a decision tree, each loop in a training set of loops is executed twice: once unrolled and once as the original loop. A loop can be in one of four classes that characterize the impact of loop unrolling on the loop's execution time. Before they learn the decision tree, Monsifrot et al. aggregate the training data by clustering loops with equal feature vectors and representing each cluster basically by its majority class.

Cavazos et al. [34] optimize a compiler optimization sequence. They sample optimization sequences randomly offline and apply them to a training set of programs. They characterize the program versions using the values of performance counters. Thereby, they obtain training data from which they learn a model using logistic regression [24]. For a program to be optimized, the model learned can predict for each compiler optimization the probability that its application will be useful. The optimizations' relative order of application is fixed and, thus, Cavazos et al. do not tackle the compiler phase ordering problem.

Using the "reverse k-nearest neighbor" model, Long and Zhu [94, 95] decide whether learned performance models are applicable to a given program.

Fursin et al. [59] propose MILEPOST GCC, which is a compiler that uses machine learning. MILEPOST GCC adapts its internal optimization heuristics to optimize previously unseen programs on a given architecture. Fursin et al. classify programs using static program features and predict a combination of optimizations for the given program based on similar programs in the training set. They also propose a model that extends the vector of program features by the selected optimizations and allows to estimate whether a given combination of optimizations is profitable for a given program. MILEPOST GCC relies on GCC.

Park et al. [115] rely on features that they extract from programs' intermediate graph-based program representation to predict a compiler optimization sequence that is likely beneficial for a given program. Alternatively, a small number of optimization sequences can be predicted. Thus, the approach can also be used in an iterative manner. In addition to the graph-based features, Park et al. rely on performance counters, the impact of each optimization sequence from a particular set of sequences on the program, and source code features. The optimizations considered are loop unrolling with different unrolling factors, different loop fusion strategies, tiling, parallelization, and vectorization. The compiler used is the polyhedral source-to-source compiler PoCC [126]. We classify the approach as not polyhedral because, although it uses a polyhedral compiler, the technique is not inherently connected to the polyhedron model.

For the purpose of predictive modeling, Park et al. [117] propose to extract static features of loops and loop nests using the pattern-driven code transformation system HERCULES [76] that allows to analyze and transform programs by specifying patterns. HERCULES relies on the declarative language PROLOG [133] for deductive inference.

Wang et al. [160] use a combination of static analysis and profiling to detect parallelizable loops and employ a machine-learned support vector machine (SVM) classifier that relies on static and dynamic program features to decide which loops to parallelize.

Ashouri et al. [6] train linear regression models from variants of training programs offline to be able to predict a good sequence of program optimizations for a given program. They use LLVM as their compiler. Ashouri et al. use dynamic architecture-independent program features. An optimization sequence is grown step by step. In each step, the application speedup, which results from applying a certain optimization subsequent to the ones selected already, is predicted.

Cosenza et al. [45] use SVMs to learn models that allow them to speed up the autotuning of program transformation parameters for stencils. They rely on the PATUS DSL source-to-source stencil compiler [41].

4 Sampling the Search Space of Legal Schedules

In this chapter, we describe the random sampling of the search space of legal linearly affine schedules for a given SCoP.

Our sampling technique is partly derived from the technique that Pouchet et al. [124] proposed for their iterative schedule optimizer LeTSeE [120]. This technique, in turn, builds on the fundamental scheduling algorithm by Feautrier [52, 53].

Pouchet et al. optimized only for sequential execution and did not take additional schedule optimization, such as tiling, into account. They narrowed the search space of legal schedules to a specific subset in a way that is reasonable for their optimization goal. This choice's advantage is that it avoids the extremely large number of schedules that would have had to be taken into account otherwise. Since they did not post-process (for instance by tiling or strip mining) their schedules, it was unnecessary for Pouchet et al. to produce injective schedules. In fact, entire loops of the transformed program may not be encoded in their schedules. Pouchet et al. continue to generate schedule dimensions only until all dependence polyhedra are carried. In contrast, polyhedral code generators such as isl [66] complete schedules to injective functions implicitly.

Pouchet's sampling technique for multi-dimensional schedules comprises an algorithm for search space construction and both a decoupling technique and a genetic algorithm to traverse the constructed search space. The traversal techniques use statistical knowledge that has resulted from a performance-sensitivity analysis of schedule coefficients [123, 124].

The restrictions imposed on the search space by Pouchet et al. likely prohibit tiling and parallelization. To meet our goals, we had to eliminate these restrictions. Further, to be able to apply tiling to schedules, we must encode all loops of the transformed program in the schedule explicitly. In the following, we propose a generalization of their technique that allows us to explore, theoretically, a SCoP's entire search space of legal schedules. Our sampling technique will, for reasons of practicality, reach some schedules with a much higher probability than others. It does not rely on statistical knowledge, but solely on the search space's structure and the static configuration of Polyite.

The proposed technique is highly configurable: the exploration can be biased to explore primarily schedules that satisfy certain criteria, such as sparseness of a schedule function's coefficient matrix or coarse-grained parallelism.

Taking the complete search space of legal schedules for a given program into account causes a dilemma: on the one hand, we must remove the restrictions that prohibit tiling and parallelization; on the other hand, we cannot model the entire search space in a tractable way that permits sampling. To overcome this dilemma, we divide the search space into subsets, which we call *search space regions*, and sample the set of search space regions. Each search space region is represented by a list of polyhedra. For this purpose, we propose a modification to Pouchet's algorithm for the construction of the search space. Note that if one bounds the maximum dimensionality of the schedules to $m \in \mathbb{N}$, it is theoretically possible to express the set of all legal at most m-dimensional schedules for a given program as one convex set, but the dimensionality of the resulting search space likely poses the computational infeasibility of handling it [127].

In order to achieve a good coverage of the search space or, in other words, a strong diversity of the schedules sampled, we sample the search space by choosing repeatedly a random region, selecting randomly one or two schedules from this region and continuing with another region.

In this section, we begin by stating the objectives of a schedule sampling algorithm. We continue by motivating our decision to use the same schedule representation for the sampling of schedules as Pouchet [124] (Section 4.2). We recall Pouchet's motivation to limit the search to legal schedules (Section 4.3). Next, we discuss the size and structure of the generalized search space that we would like to explore (Section 4.4). Then, we recall Pouchet's algorithm for search space construction (Section 4.5.1) and introduce our extensions that allow to sample different subsets of the schedule search space (Section 4.5.2). In Section 4.6, potential techniques to sample schedules from subsets of the schedule search space are presented and discussed. In the remainder of the chapter, necessary steps to prepare schedules for the application of tiling and strip mining (Section 4.7) and a way to adapt the characteristics of Pouchet's approach in Polyite (Section 4.8) are described.

Sections 4.2 to 4.5.1 primarily recall Pouchet's search space construction for the purpose of presenting the complete schedule sampling technique proposed.

4.1 Objectives of a Sampling Algorithm

We have identified the following to be the main objectives of an effective sampling algorithm for polyhedral schedules.

Observed Computational Complexity Depending on the size and complexity of the SCoP to be optimized, iterative optimization may require a large number of schedules to be generated and evaluated. Thus, an acceptably low observed computational complexity of the sampling algorithm for schedules is crucial for efficiency.

Coverage of the Schedule Search Space As mentioned already, the schedule search space is divided into a, theoretically infinite, number of subsets. The subsets are not necessarily pairwise disjoint. Each of the subsets contains a, theoretically infinite, number of schedules. Most of the subsets and most of the schedules in each subset are redundant as they correspond to the same execution order as other schedules, or are equivalent to other subsets. The redundancy has four main causes, which we illustrate in Example 4.1.1:

1. Statements' and statement instances' textual order can be encoded in various ways.
2. Multiplying a row of a schedule matrix yields a schedule that is equivalent to the original schedule.
3. The absolute value of some schedule coefficients may be too large, or can even be reduced to 0 without altering the schedule.Vasilache addresses the second cause and also the shift of an iteration domain by a constant or multiples of structure parameters, which is subsumed by the third cause.
4. Schedule functions can contain redundant dimensions: these do not contribute to the lexicographic ordering of statement instances.

Many other schedules are not redundant, but still useless because they contain unreasonably large coefficients. A discussion of the equivalence of schedules can also be found in Nicholas Vasilache's PhD thesis [149]. Vasilache addresses the second cause and also the shift of an iteration domain by a constant or multiples of structure parameters, which is subsumed by the third cause.

An effective schedule sampling algorithm need not be able to reach every schedule in the search space, but it must cover a reasonable subset.

Example 4.1.1. We recall iteration domains and schedules of our example `syrk`:

$$\begin{aligned}
I_R = {} & \{(i,j)^T \mid i,j \in \mathbb{Z} \ \wedge\ 0 \le i < n \ \wedge\ 0 \le j \le i\} \\
I_S = {} & \{(i,j,k)^T \mid i,j,k \in \mathbb{Z} \ \wedge\ 0 \le i < n \ \wedge\ 0 \le j < m \ \wedge\ 0 \le k \le i\}, n, m \in \mathbb{Z} \\
& \Theta_R(i,j)^T = (i,0,j,0)^T,\ \Theta_S(i,j,k)^T = (i,1,j,k)^T.
\end{aligned}$$

The following two statement schedules are equivalent to the original schedule:

$$\Theta'_R(i,j)^T = (42 \cdot i, j, 0, 0, 1)^T,\ \Theta'_S(i,j,k)^T = (42 \cdot i + 21, j, k, k, 0)^T.$$

Schedule dimensions four and five are dispensable, which leads to

$$\Theta'_R(i,j)^T = (42 \cdot i, j, 0)^T,\ \Theta'_S(i,j,k)^T = (42 \cdot i + 21, j, k)^T.$$

We can then divide the first schedule dimension by its coefficients' greatest common divisor, which yields

$$\Theta'_R(i,j)^T = (2 \cdot i, j, 0)^T,\ \Theta'_S(i,j,k)^T = (2 \cdot i + 1, j, k)^T$$

We notice that the first schedule dimension corresponds to a loop with stride two. In each iteration, we first execute an instance of statement R and then an instance of statement S. This execution order matches the first two dimensions of schedule Θ. The larger iteration variable coefficients would have an effect if one encoded tiling in the original and the simplified schedule with the same tile sizes in both cases. They influence the number of statement instances that are executed per tile. Yet, this number should be determined primarily by the chosen tile sizes and not the values of iteration variable coefficients. $\triangleleft$

Biasing the Exploration An iterative optimization for parallelization with OpenMP requires schedules that enable coarse-grained parallelism. To avoid evaluating an unnecessarily large number of schedules, it is likely beneficial to encounter primarily schedules with this property. Yet, one may wish not to exclude other schedules entirely from the search space. A widely held conjecture is that schedules with a dense coefficient matrix are likely unprofitable as they yield unnecessarily complex index computations in the transformed program.

As discussed above, schedules with large coefficients are likely equivalent to schedules with smaller coefficients. Further, too large coefficients of iteration variables require the adaptation of tile sizes and may yield integer overflows during the execution of the transformed program. Consequently, it must be possible to bound the absolute value of schedule coefficients.

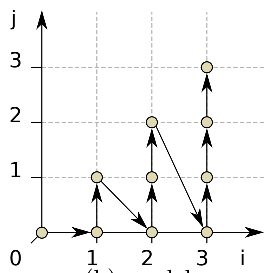
In summary, we envision a sampling algorithm that can be configured to explore primarily schedules that exhibit certain characteristics.

4.2 The Optimal Schedule Representation for Sampling

We need a schedule representation that permits us to uniformly represent and sample the schedule search space that results from a SCoP's legality-affecting data dependencies (refer to Section 2.2.2.3). Section 2.2.2.3 recalls how legality constraints on linearly affine schedule functions are derived from dependence polyhedra. While the use of one-dimensional schedule functions suffices in some situations, supporting multi-dimensional schedule functions is inevitable for two reasons. The first has been pointed out by Feautrier [53]: some static control parts have data dependences that prevent the representation of a legal execution order by a one-dimensional linearly affine schedule. Figure 4.1(a) reproduces an example of such a SCoP given by Feautrier. From the model (refer to Figure 4.1(b)) that corresponds to the example's code, readers may convince themselves that any linearly affine one-dimensional schedule either does not carry or violates one of two sets of data dependences. One set are

```
for (int i = 0; i <= n; i++)
  for (int j = 0; j <= i; ++j)
    s = s + A[i][j]
```

(a) code



(b) model

Figure 4.1: An example of a SCoP for which no legal one-dimensional linearly affine schedule exists. The example is due to Feautrier [53].

the dependences that correspond to the arrows that point forward in the direction of the i-dimension in Figure 4.1(b). The other set are the dependences that correspond to the arrows pointing upward in Figure 4.1(b).

In other cases, like in Figure 2.6, a legal one-dimensional linearly affine schedule does exist. While the loop nest in the figure is two-dimensional, one schedule dimension suffices to carry the data dependences. The missing inner loop need not be encoded explicitly since polyhedral code generators, such as ISL [66], can choose a possible execution order and generate the remaining code accordingly. Yet, for analysis and transformation of schedules, such as the application of tiling, it may be necessary to have an explicit encoding of all of the transformed program's loops in the schedule. Again, this requires the use of multi-dimensional schedules, in the case of a loop nesting-level that is larger than 1.

We follow Pouchet et al. [124] and use schedule coefficient matrices (*schedule matrix* for short) to represent schedules. This choice is natural: schedules are linearly affine functions, which are fully characterized by their coefficient matrix (refer to Definitions 2.2.1 and 2.2.2). Applying Farkas' Lemma to a dependence polyhedron $D_{O,T}$ and the prototype of a one-dimensional schedule function yields the set of all schedule coefficient vectors $(\vec{\lambda}_O \vec{\mu}_O, \nu_O)^T$, $(\vec{\lambda}_T \vec{\mu}_T, \nu_T)^T$ such that $D_{O,T}$ is weakly or strongly satisfied; $\vec{\lambda}_T$ contains one coefficient per variable in $\vec{i}_T$, $\vec{\mu}_T$ contains one coefficient per structure parameter of the SCoP, and ν_T is the constant part of the one-dimensional schedule. Refer to Section 2.2.2.3 for details. Thus, an n-dimensional ($n \in \mathbb{N}$) schedule Θ_S of statement S is represented by a matrix

$$M_{\Theta_S} = \begin{pmatrix} \vec{\lambda}_S^1 & \vec{\mu}_S^1, & \nu_S^1 \\ \vec{\lambda}_S^2 & \vec{\mu}_S^2, & \nu_S^2 \\ \vdots & \vdots & \vdots \\ \vec{\lambda}_S^n & \vec{\mu}_S^n, & \nu_S^n \end{pmatrix}.$$

Since data dependences may exist between statements that are different from each other, it is impossible to choose statement schedules independently and at the same time ensure the legality of the schedule. Thus, Pouchet et al. unify all schedule matrices in one matrix. Say, we have an n-dimensional schedule Θ for a SCoP with m statements ($m, n \in \mathbb{N}$). Θ can be encoded as follows:

$$M_\Theta = \begin{pmatrix} \vec{\lambda}_{S_1}^1 & \vec{\lambda}_{S_2}^1 & \dots & \vec{\lambda}_{S_m}^1 & \vec{\mu}_{S_1}^1 & \vec{\mu}_{S_2}^1 & \dots & \vec{\mu}_{S_m}^1, & \nu_{S_1}^1, & \nu_{S_2}^1, & \dots, & \nu_{S_m}^1 \\ \vec{\lambda}_{S_1}^2 & \vec{\lambda}_{S_2}^2 & \dots & \vec{\lambda}_{S_m}^2 & \vec{\mu}_{S_1}^2 & \vec{\mu}_{S_2}^2 & \dots & \vec{\mu}_{S_m}^2, & \nu_{S_1}^2, & \nu_{S_2}^2, & \dots, & \nu_{S_m}^2 \\ \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \dots & \vdots \\ \vec{\lambda}_{S_1}^n & \vec{\lambda}_{S_2}^n & \dots & \vec{\lambda}_{S_m}^n & \vec{\mu}_{S_1}^n & \vec{\mu}_{S_2}^n & \dots & \vec{\mu}_{S_m}^n, & \nu_{S_1}^n, & \nu_{S_2}^n, & \dots, & \nu_{S_m}^n \end{pmatrix}.$$

Other than Pouchet et al., who use schedule matrices with integer coefficients, we must use rational schedule coefficients due to the nature of our primary sampling strategy (refer

Table 4.1: Schedule matrix of the schedule in Example 2.2.15.

Θ_R					Θ_S					
i	j	n	m	1	i	j	k	n	m	1
1	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0

to Section 4.6.5) and some of our genetic operators (refer to Chapter 6). Rational schedule matrices are unsuitable for standard polyhedral code generation algorithms, but can be transformed to integer schedule matrices as described in Section 4.6.5. We call the rows of a schedule matrix *schedule coefficient vectors.*

Example 4.2.1. The schedule in Example 2.2.15 is represented by the schedule matrix shown in Table 4.1. $\triangleleft$

4.3 Restriction to Legal Schedules

Pouchet et al. constrain the schedule search space to legal schedules, i.e., schedules that do not violate any legality-affecting dependences. While this choice complicates the sampling of schedules, it is strongly motivated by the fact that the number of illegal and redundant schedules grows exponentially faster with the program size than the number of legal schedules [122]. Nisbet [104] et al. did not restrict the search space of their genetic algorithm to legal schedules. They observed a share of at most 5.5% legal schedules among 20,000 schedules generated. This underlines the necessity to restrict the schedule search space to legal schedules. Therefore, we have followed the choice made by Pouchet et al.

To represent the legality-affecting data dependences of a SCoP by a set of dependence polyhedra we use an approach by Kronawitter [86]. We describe this technique in the following.

Kronawitter starts with the approximate data flow analysis algorithm described by Verdoolaege [153] to calculate the set T of all pairs $\begin{pmatrix}\vec{x}\\ \vec{y}\end{pmatrix}$ of statement instances in the SCoP such that $\vec{y}$ depends on $\vec{x}$. Next, a set of dependence polyhedra whose union equals or overapproximates T must be calculated. In a first step, T is split by the pair of source statement O and target statement T. This yields per pair of statements O and T with iteration domains I_O and I_T the following relation:

$$DS_{O,T} = \left\{ \begin{pmatrix}\vec{x}\\ \vec{y}\end{pmatrix} \mid \vec{x} \in I_O \ \wedge \ \vec{y} \in I_T \ \wedge \ \vec{y} \text{ depends on } \vec{x} \right\}.$$

To be able to construct a large portion of the set of legal schedules in the sampling process, the sets $DS_{O,T}$ must be split at a finer grain level. Algorithm 4.1 serves this purpose. It repeatedly calculates the lexicographic minimum $DS_{O,T_{\min}}$ (refer to Section 2.2.1.6) of $DS_{O,T}$ and then splits $DS_{O,T_{\min}}$ along any disjunctions in its definition. Finally, $DS_{O,T_{\min}}$ is subtracted from $DS_{O,T}$. The iteration continues until $DS_{O,T}$ is empty. Example 4.3.1 illustrates a case in which $DS_{O,T}$ can be split into an infinite number of sets $DS_{O,T_{\min}}$. Endless cycling is prevented by applying an upper bound on the number of splits. If the limit is reached, $DS_{O,T}$ is split along any disjunction in its definition. This approach to splitting the sets $DS_{O,T}$ is coarser grained than the default strategy and it is non-deterministic.

Finally, since the resulting sets of dependence instances may not be polyhedral, we calculate their convex hulls.

Algorithm 4.1: Partition a set of dependent pairs of statement instances into polyhedra

```
Input: Set $DS_{O,T} \subseteq \left\{ \begin{pmatrix} \vec{x} \\ \vec{y} \end{pmatrix} \mid \vec{x} \in I_O \ \wedge \ \vec{y} \in I_T \right\}$
Output: Set $DP_{O,T}$ of polyhedra such that $DS_{O,T} \subseteq \bigcup_{P \in DP_{O,T}} P$
Parameters: split_max is the maximum number of polyhedra into which $DS_{O,T}$ ought to be split.
1  n_split ← 0
2  $DP_{O,T} \leftarrow \emptyset$
3  while n_split < split_max $\wedge$ $DS_{O,T} \neq \emptyset$ do
4      $DS_{O,T_{\min}} \leftarrow \text{lex.min}(DS_{O,T})$
5      for $P \in \Lambda(DS_{O,T_{min}})$ do
6          $DP_{O,T} \leftarrow DP_{O,T} \cup \{P\}$
7          n_split ← n_split + 1
8          if n_split > split_max then
9              break
10 if n_split > split_max then
11     $DP_{O,T} \leftarrow \Lambda(DS_{O,T})$
12 $DP_{O,T} \leftarrow$ map(conv.hull, $DP_{O,T}$)
13 return $DP_{O,T}$
```

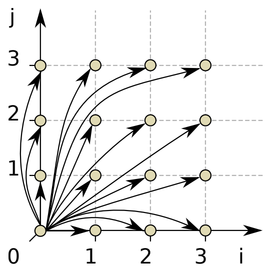


Figure 4.2: Iteration domain and dependences of the SCoP in Example 4.3.1. All dependences originate at the statement instance at $(0, 0)$.

Example 4.3.1.
Let S be a statement with $I_S = \{(i,j)^T \mid i,j \in \mathbb{Z} \ \wedge \ 0 \leq i \leq n \ \wedge \ 0 \leq j \leq n\}, n \in \mathbb{Z}$. Let the statement's schedule be $\Theta_S(i,j)^T = (i,j)^T$ and let there be the following dependence polyhedron:

$$D_{S,S} = \{(0,0,i,j)^T \mid (0,0)^T \in I_S \ \wedge \ (i,j)^T \in I_S\}.$$

We illustrate $D_{S,S}$ in Figure 4.2 for $n = 3$. All dependences in $D_{S,S}$ originate at the statement instance at $(0, 0)$. Attempting to split the relation by slicing off repeatedly its lexicographic minimum yields a set per element of $D_{S,S}$. The algorithm does not split $D_{S,S}$ further. $\triangleleft$

Per dependence polyhedron $D_{O,T}$, Pouchet et al. define two polyhedral sets of schedule coefficient vectors. Set $W_{O,T}$ represents all one-dimensional schedules that satisfy $D_{O,T}$ at least weakly. Set $S_{O,T}$ represents all one-dimensional schedules that satisfy $D_{O,T}$ strongly. To satisfy multiple dependence polyhedra, one must intersect the respective polyhedra $W_{O,T}$ and $S_{O,T}$.

4.4 The Size and Structure of the Search Space

A dependence polyhedron can be carried by any schedule dimension, assuming that all preceding dimensions satisfy it weakly. Depending on the choices made, the number of schedule dimensions and the legality constraints of the dimensions vary. Restrictions originate only from the interplay of dependences, as illustrated by Figure 4.1(b). In the illustrated case, there are two dependence polyhedra and it is impossible to carry both in the first schedule dimension.

The full search space is a set of lists of polyhedra of diverse lengths. Each list represents one search space region. Per list, the vectors contained in the dth polyhedron are the possible schedule coefficient vectors for schedule dimension d of the schedules contained in the respective search space region. Given a dependence polyhedron $D_{O,T}$ and a legal schedule for the respective SCoP, the number of schedule dimensions that precede the dimension that carries $D_{O,T}$ may be infinitely large. Thus, the maximum length of the lists of polyhedra that represent the search space regions is unbounded and the number of search space regions is infinite.

Pouchet et al. considered neither arbitrary schedules with outer parallelism nor injective schedules. Consequently, in their case, the maximum number of schedule dimensions is equal to the number of dependence polyhedra and the total number of search space regions is at most exponential in the number of dependence polyhedra.

To reduce the size of the search space, Pouchet et al. limited their search to one specific search space region. This region is determined by processing dependence polyhedra in a specific order: they are sorted by the memory traffic of the dependent statements and the degree at which they interfere with other dependence polyhedra. Two dependence polyhedra $D_{O,T}$ and $D_{O',T'}$ *interfere* if $S_{O,T} \cap S_{O',T'} = \emptyset$.

4.5 Sampling Search Space Regions

Our algorithm to sample the set of search space regions for a given SCoP is a generalization of the algorithm for search space construction by Pouchet et al. [124]. We start by laying out their algorithm and present our amendments subsequently. Algorithm 4.2 is the amended procedure. The parts that we have modified or added are highlighted.

Algorithm 4.2: Sampling of Search Space Regions

Input: G: Set of dependence polyhedra for dependences between pairs of statements O and T
U: (Universe) A polyhedron that represents the set of schedule coefficients that will be considered despite restrictions by legality constraints that originate from data dependences

Output: A search space region's model. The model is a list of polyhedra. The d^{th} polyhedron contains the allowed coefficient vectors for dimension d of the schedules in the modeled search space region.

Parameters: `p` At schedule dimension d, all schedules contained in the constructed search space region will satisfy strongly at least one of the dependence polyhedra that are not carried by all schedule in one of the dimensions $1, \ldots, d-1$ with probability `p`.

```
1  $d \leftarrow 0$; $P \leftarrow \langle\rangle$
2  while $G \neq \emptyset$ do
3      $d \leftarrow d + 1$; $P_d \leftarrow U$;
4      foreach $D_{O,T} \in G$ do
5          $W_{O,T} \leftarrow \{\vec{m}_\Theta \mid \vec{m}_\Theta \in U \wedge \vec{m}_\Theta$ weakly satisfies $D_{O,T}\}$;
6          $P_d \leftarrow P_d \cap W_{O,T}$;
7      $G_d \leftarrow \emptyset$; $q \leftarrow$rand$([0,1])$
8      if $q \leq$ p then
9          $G_d \leftarrow$ rand$(2^G \setminus \{\emptyset\})$;
10     foreach $D_{O,T} \in G_d$ do
11         $S_{O,T} \leftarrow \{\vec{m}_\Theta \mid \vec{m}_\Theta \in U \wedge \vec{m}_\Theta$ strongly satisfies $D_{O,T}\}$;
12         if $P_d \cap S_{O,T} \neq \emptyset$ then
13             $P_d \leftarrow P_d \cap S_{O,T}$;
14     $G \leftarrow G \setminus \{D_{O,T} \mid D_{O,T} \in G \wedge \forall \vec{m}^d_\Theta \in P_d : \vec{m}^d_\Theta$ strongly satisfies $D_{O,T}\}$;
15     $P \leftarrow P_d :: P$
16 return reverse$(P)$
```

Function `rand` selects randomly an element from a given set. Elements are chosen with uniform probability. Operator :: prepends an element to a list. Function `reverse` reverses a list.
The parts of the algorithm that differ from Pouchet's original search space construction are highlighted.

4.5.1 Search Space Construction by Louis-Noël Pouchet

The algorithm starts from a list G of all data dependence polyhedra in the SCoP. G is sorted in ascending order according to the criterion described in Section 4.4. Pouchet's algorithm constructs a model of a specific set of schedules. In our terminology, this set corresponds to one search space region. It is represented by a list of polyhedra $P_1, ..., P_n$, with P_d containing all possible schedule coefficient vectors for schedule dimension d. As long as G is not empty, another schedule dimension, i.e., a polyhedron, will be appended to the list. P_d is initially a hypercube such that each schedule coefficient is constrained to the interval $[-1, 1]$. In Algorithm 4.2, these constraints would already have to be present in U. Subsequently, P_d is being intersected with the sets $W_{O,T}$ of all $D_{O,T}$ in G. After this point, all one-dimensional schedules encoded by the vectors in P_d satisfy the dependences in G weakly. In an additional iteration across G, the algorithm tests, per $D_{O,T} \in G$, whether $P_d \cap S_{O,T}$ is non-empty. If so, P_d is updated to the intersection's result and $D_{O,T}$ is removed from G. The order of the remaining dependences in G must now be updated with respect to their mutual interference.

From the algorithm described, it is apparent that, in the search space considered by Pouchet et al. [124], each dimension of each schedule carries at least one data dependence polyhedron. This yields, primarily, inner parallelism.

4.5.2 Our Generalization

As mentioned, we cannot model the complete search space of legal schedules for a program. Its size is infinite and its constraint representation in disjunctive normal form would contain an infinite number of conjunctions. Bounding the schedules' maximum dimensionality by a fixed number $m \in \mathbb{N}$ would allow to represent the set of all m-dimensional schedules as one polyhedron. Yet, applying algorithms with exponential run-time complexity to this polyhedron is most likely infeasible [127]. Therefore, we divide the search space along the disjunctions into search space regions and modify Pouchet's algorithm for search space construction such that we can use it to sample these search space regions. In order to allow for arbitrary outer parallelism, we must permit outer schedule dimensions that do not carry dependences. Further, with parallelization and tiling in mind, skewing is an important enabling loop transformation. Skewing shifts the iterations of a loop by the value of an encasing loop's iteration variable. It does not change the execution order of statement instances, but it modifies the direction of dependences with respect to the shifted schedule dimension. This can enable tiling (refer to Section 2.2.2.4). The restriction of schedule coefficients to $[-1, 1]$ prohibits non-unit skewing. In consequence, we do not bound schedule coefficients beyond legality constraints.

In particular, we make the following amendments: G is no longer an ordered list of dependence polyhedra, but a set. U is not a hypercube such that each dimension of the schedule coefficient space is bounded to $[-1, 1]$, but a set that covers the entire schedule coefficient space. When we constrain P_d to schedules that carry dependence polyhedra, we consider only dependences from a randomly chosen subset $G_d \subseteq G$. Subsequently, we remove all dependence polyhedra from G that are carried by all one-dimensional schedules encoded in P_d. Only with a configurable probability $\mathsf{p} \in]0, 1]$, G_d is not empty. It is important to chose p neither too large, as that would prevent outer parallelism, nor too low because that would increase the dimensionality of schedules unreasonably. Leaving G_d empty allows P_d to contain fully parallel one-dimensional schedules.

The non-determinism in Algorithm 4.2 causes it to model different search space regions on different invocations. Example 4.5.1 demonstrates Algorithm 4.2.

Example 4.5.1. Recall the running example `syrk`. Let $\mathsf{p} = 0.4$. We have two statements S and R with $\vec{i}_S = (i,j)^T$ and $\vec{i}_R = (i,j,k)^T$ and $\vec{p} = (n,m)^T$. The set of dependence polyhedra is:

$$\begin{aligned}
G \; = \; \{ & \\
& D_{R,S} \; = \; \{(i,j,i,0,j)^T \mid i < n \;\wedge\; 0 \le j \le i \;\wedge\; (i,j)^T \in I_R \;\wedge\; (i,0,j)^T \in I_S\}, \\
& D_{S,S} \; = \; \{(i,j,k,i,j+1,k)^T \mid 0 \le i < n \;\wedge\; 0 \le j \le m-2 \;\wedge\; 0 \le k \le i \\
& \quad \wedge\; (i,j,k)^T \in I_S \;\wedge\; (i,j+1,k)^T \in I_S\} \\
\}. &
\end{aligned}$$

From these dependence polyhedra result the following sets of schedule coefficient vectors:

$$\begin{aligned}
W_{R,S} &= \{(\vec{\lambda}_S \; \vec{\lambda}_R \; \vec{\mu}_S \; \vec{\mu}_R, \nu_S, \nu_R)^T \mid \vec{\mu}_S^{(2)} \ge \vec{\mu}_R^{(2)} \;\wedge\; \vec{\mu}_S^{(1)} \ge \vec{\mu}_R^{(1)} \\
&\quad \wedge\; \vec{\lambda}_S^{(1)} + \vec{\lambda}_S^{(3)} + \vec{\mu}_S^{(1)} \ge \vec{\lambda}_R^{(1)} + \vec{\lambda}_R^{(2)} + \vec{\mu}_R^{(1)} \\
&\quad \wedge\; \vec{\mu}_S^{(1)} + \vec{\mu}_R^{(2)} + \nu_S \ge \vec{\mu}_R^{(1)} + \vec{\mu}_R^{(2)} + \nu_R\} \\
S_{R,S} &= \{(\vec{\lambda}_S \; \vec{\lambda}_R \; \vec{\mu}_S \; \vec{\mu}_R, \nu_S, \nu_R)^T \mid \vec{\mu}_S^{(2)} \ge \vec{\mu}_R^{(2)} \;\wedge\; \vec{\mu}_S^{(1)} \ge \vec{\mu}_R^{(1)} \\
&\quad \wedge\; \vec{\lambda}_S^{(1)} + \vec{\lambda}_S^{(3)} + \vec{\mu}_S^{(1)} \ge \vec{\lambda}_R^{(1)} + \vec{\lambda}_R^{(2)} + \vec{\mu}_R^{(1)} \\
&\quad \wedge\; \vec{\mu}_{S\,(1)} + \vec{\mu}_R^{(2)} + \nu_S \ge \vec{\mu}_R^{(1)} + \vec{\mu}_R^{(2)} + \nu_R + 1\} \\
W_{S,S} &= \{(\vec{\lambda}_S \; \vec{\lambda}_R \; \vec{\mu}_S \; \vec{\mu}_R, \nu_S, \nu_R)^T \mid \vec{\lambda}_S^{(2)} \ge 0\} \\
S_{S,S} &= \{(\vec{\lambda}_S \; \vec{\lambda}_R \; \vec{\mu}_S \; \vec{\mu}_R, \nu_S, \nu_R)^T \mid \vec{\lambda}_S^{(2)} \ge 1\}.
\end{aligned}$$

Round 1 $d = 1$.

Let $q > \mathsf{p} = 0.4$. Consequently, $G_1 = \emptyset$ and therefore

$$\begin{aligned}
P_1 = W_{R,S} \cap W_{S,S} &= \{(\vec{\lambda}_S \; \vec{\lambda}_R \; \vec{\mu}_S \; \vec{\mu}_R, \nu_S, \nu_R)^T \mid \vec{\lambda}_S^{(2)} \ge 0\} \\
&\quad \cap \; \{(\vec{\lambda}_S \; \vec{\lambda}_R \; \vec{\mu}_S \; \vec{\mu}_R, \nu_S, \nu_R)^T \mid \vec{\mu}_S^{(2)} \ge \vec{\mu}_R^{(2)} \;\wedge\; \vec{\mu}_S^{(1)} \ge \vec{\mu}_R^{(1)} \\
&\qquad \wedge\; \vec{\lambda}_S^{(1)} + \vec{\lambda}_S^{(3)} + \vec{\mu}_S^{(1)} \ge \vec{\lambda}_R^{(1)} + \vec{\lambda}_R^{(2)} + \vec{\mu}_R^{(1)} \\
&\qquad \wedge\; \vec{\mu}_S^{(1)} + \vec{\mu}_R^{(2)} + \nu_S \ge \vec{\mu}_R^{(1)} + \vec{\mu}_R^{(2)} + \nu_R\} \\
&= \{(\vec{\lambda}_S \; \vec{\lambda}_R \; \vec{\mu}_S \; \vec{\mu}_R, \nu_S, \nu_R)^T \mid \vec{\mu}_S^{(2)} \ge \vec{\mu}_R^{(2)} \;\wedge\; \vec{\mu}_S^{(1)} \ge \vec{\mu}_R^{(1)} \\
&\qquad \wedge\; \vec{\lambda}_S^{(1)} + \vec{\lambda}_S^{(3)} + \vec{\mu}_S^{(1)} \ge \vec{\lambda}_R^{(1)} + \vec{\lambda}_R^{(2)} + \vec{\mu}_R^{(1)} \\
&\qquad \wedge\; \vec{\mu}_S^{(1)} + \vec{\mu}_R^{(2)} + \nu_S \ge \vec{\mu}_R^{(1)} + \vec{\mu}_R^{(2)} + \nu_R \\
&\qquad \wedge\; \vec{\lambda}_S^{(2)} \ge 0\}.
\end{aligned}$$

It remains that $G = \{D_{R,S}, D_{S,S}\}$.

Round 2 $d = 2$.

Again, let $q > \mathsf{p}$. Consequently, $P_2 = P_1$ and $G = \{D_{R,S}, D_{S,S}\}$.

Listing 4.1: The program in Listing 2.1 (page 18) transformed by the schedule in Example 4.5.1.

```
#pragma omp parallel for
for (int c0 = 0; c0 < n; c0++)
 for (int c1 = 0; c1 <= c0; c1++) {
   C[c0][c1] *= beta; // statement R
   for (int c2 = 0; c2 < m; c2++)
     C[c0][c1] += alpha * A[c0][c2] * A[c1][c2]; // statement S
 }
```

Round 3 $d = 3$.

Let $q < \mathsf{p}$ and $G_3 = \{D_{R,S}\}$. We obtain

$$\begin{aligned}P_3 &= W_{R,S} \cap W_{S,S} \cap S_{R,S} \underset{\{S_{R,S} \subset W_{R,S}\}}{=} W_{S,S} \cap S_{R,S}\\ &= \{(\vec{\lambda}_S\ \vec{\lambda}_R\ \vec{\mu}_S\ \vec{\mu}_R, \nu_S, \nu_R)^T \mid \vec{\lambda}_S^{(2)} \geq 0\}\\ &\quad \cap \{(\vec{\lambda}_S\ \vec{\lambda}_R\ \vec{\mu}_S\ \vec{\mu}_R, \nu_S, \nu_R)^T \mid \vec{\mu}_S^{(2)} \geq \vec{\mu}_R^{(2)} \ \wedge\ \vec{\mu}_S^{(1)} \geq \vec{\mu}_R^{(1)}\\ &\qquad\quad \wedge\ \vec{\lambda}_S^{(1)} + \vec{\lambda}_S^{(3)} + \vec{\mu}_S^{(1)} \geq \vec{\lambda}_R^{(1)} + \vec{\lambda}_R^{(2)} + \vec{\mu}_R^{(1)}\\ &\qquad\quad \wedge\ \vec{\mu}_S^{(1)} + \vec{\mu}_R^{(2)} + \nu_S \geq \vec{\mu}_R^{(1)} + \vec{\mu}_R^{(2)} + \nu_R + 1\}\\ &= \{(\vec{\lambda}_S\ \vec{\lambda}_R\ \vec{\mu}_S\ \vec{\mu}_R, \nu_S, \nu_R)^T \mid \vec{\mu}_S^{(2)} \geq \vec{\mu}_R^{(2)} \ \wedge\ \vec{\mu}_S^{(1)} \geq \vec{\mu}_R^{(1)}\\ &\qquad\quad \wedge\ \vec{\lambda}_S^{(1)} + \vec{\lambda}_S^{(3)} + \vec{\mu}_S^{(1)} \geq \vec{\lambda}_R^{(1)} + \vec{\lambda}_R^{(2)} + \vec{\mu}_R^{(1)}\\ &\qquad\quad \wedge\ \vec{\mu}_S^{(1)} + \vec{\mu}_R^{(2)} + \nu_S \geq \vec{\mu}_R^{(1)} + \vec{\mu}_R^{(2)} + \nu_R + 1\\ &\qquad\quad \wedge\ \vec{\lambda}_S^{(2)} \geq 0\}\end{aligned}$$

and $G = \{D_{S,S}\}$.

Round 4 $d = 4$.

Let $q < \mathsf{p}$ and $G_4 = \{D_{S,S}\}$. We obtain $P_4 = W_{S,S} \cap S_{S,S} \underset{\{S_{S,S} \subset W_{S,S}\}}{=} S_{S,S}$.

One schedule contained in the search space region that is represented by $\langle P_1, P_2, P_3, P_4 \rangle$ is

$$\Theta_R(i,j)^T = (i,j,0,0)^T,\ \Theta_S(i,j,k)^T = (i,k,0,j)^T.$$

It corresponds to the parallelized code in Listing 4.1. ◁

4.5.3 Termination

Pouchet et al. [124] split the set of a SCoP's legality-affecting data dependences into dependence polyhedra such that all dependences in a polyhedron are carried by the same dimension of the SCoP's original schedule. Thus, it is always possible to construct the search space such that it contains the original schedule and the algorithm terminates for the same reason as the scheduling algorithm by Feautrier [53].

We use a coarser-grained splitting of the set of data dependences into dependence polyhedra (refer to Section 4.3). While this reduces the size of the already large search space, it can exclude the SCoP's original schedule matrix from the search space, such as in Example 4.3.1 or in Example 4.5.2. In Example 4.5.2, the dependences modeled in one dependence polyhedron are not only carried by different dimensions of the original schedule, but some of the dependences carried by an outer dimension have a negative direction with respect to an inner dimension that carries some of the remaining dependences. For both examples, Algorithm 4.1 models all dependences in one dependence polyhedron. Therefore, the SCoPs'

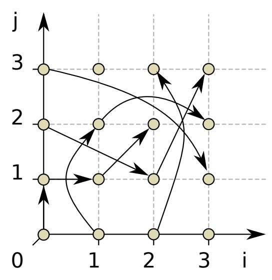


Figure 4.3: Iteration domain and dependences of the SCoP in Example 4.5.2.

original schedule matrices are not contained in the respective search spaces. Cases such as Example 4.3.1, in which the elements of the dependence polyhedron that are carried by outer dimensions do not point backwards in the direction of the inner dimensions that carry the remaining dependences, can obviously be resolved by skewing with a positive skewing factor. The remaining cases, such as Example 4.5.2, result from a dependence's target statement instance's coordinate in an inner dimension of the iteration domain being determined by the source statement instance's coordinate in an outer dimension of the iteration domain. Again, these cases can be resolved by skewing with a positive skewing factor. Recall that such skewing is always legal [149]. In the case of Example 4.3.1 and in similar cases, we do allow for the construction of the original schedule with additional schedule dimensions appended. These additional dimensions must be skewed to carry otherwise uncarryable dependence polyhedra. Since the preceding schedule dimensions already carry all data dependences, the additional schedule dimensions are artificial and can be eliminated in a subsequent schedule simplification procedure.

Example 4.5.2.
Consider a statement S with $I_S = \{(i,j)^T \mid i,j \in \mathbb{Z} \wedge 0 \leq i < n \wedge 0 \leq j < n\}, n \in \mathbb{Z}$ and $\Theta_S(i,j)^T = (i,j)^T$. The set of data dependences

$$D_{S,S} = \{(i,j,i+j,i+1)^T \mid i,j \in \mathbb{Z} \wedge (i,j)^T \in I_S \wedge i+j < n \wedge i+1 < n\} \subseteq I_S \times I_S, n \in \mathbb{Z}$$

is a polyhedral set that would not be split by Algorithm 4.1. Therefore, Algorithm 4.2 would have to construct a schedule dimension that carries $D_{S,S}$. Figure 4.3 shows that neither dimension of Θ_S carries $D_{S,S}$. Θ_S is not part of the search space that we consider. $\triangleleft$

4.5.4 Summary

In this section, we have explained why we cannot model the entire set of legal schedules for a given program as one in a way that facilitates sampling. Subsequently, we have presented a way of sampling subsets of the set of all legal schedules for a given SCoP. Each subset corresponds to a set of schedules of equal dimensionality and a specific mapping of dependence polyhedra to schedule dimensions that are guaranteed to satisfy them strongly. To sample schedules from the set of legal schedules and achieve a good coverage of the search space, one must apply Algorithm 4.2 repeatedly and choose randomly a small number of schedules from each of the obtained search space regions.

4.5.5 Discussion

Pouchet's sampling algorithm, and ours, treat data dependences at the granularity level of dependence polyhedra. If a schedule dimension carries some, but not all of a dependence polyhedron's elements, the carried dependences need not be taken into account during the remaining schedule dimensions' construction and, therefore, could be removed from the dependence polyhedron. Instead, we keep satisfying them weakly, until we have constructed a schedule dimension that satisfies the entire dependence polyhedron strongly. In other words: the granularity at which we partition the set of pairs of instances of two statements that are in dependence into dependence polyhedra influences the completeness of the search. The avoided computational effort justifies the choice to use dependence polyhedra. By splitting the set of dependences into dependence polyhedra only once, we can precompute the sets $W_{O,T}$ and $S_{O,T}$ and only have to apply Farkas' Lemma twice per dependence polyhedron. Otherwise, after scraping away already carried dependences, we may end up, first, having to split the remaining set into convex subsets, and then having to reapply Farkas' Lemma. Moreover, as described in Section 4.5.3, in situation such as the ones described, the search space contains schedules that are equivalent to the ones excluded. The difference lies in additional inner schedule dimensions that can later be eliminated by a simplification procedure.

4.6 Sampling Schedules from Search Space Regions

Regions of the search space of legal schedules for a SCoP are represented by lists of polyhedra. Consequently, in order to sample schedules from search space regions, one must sample points from polyhedra. Algorithm 4.3 illustrates the general procedure to sample schedules from search space regions.

Algorithm 4.3: Sampling of Schedules from Search Space Regions

```
  Input: $n$-dimensional search space region represented by the list $\langle P_1, P_2, \ldots, P_n \rangle$ of polyhedra
  Output: schedule matrix $M_\Theta$ with $n$ rows
1 $M_\Theta \leftarrow 0_{\mathbb{Q}^{n \times \dim(P_1)}}$
2 for $d \leftarrow 0$ to $n$ do
3     $M_\Theta^{(d,\bullet)} \leftarrow$ sample($P_d$)
4 return $M_\Theta$
```

The choice of the schedule coefficient vector $\vec{m}$ is independent for each dimension.

In the following, we describe and discuss six ways to realize the function `sample` in Algorithm 4.3. The first four techniques were selected and assessed for their suitability for random sampling with uniform distribution from single search space regions by Danner [46].

4.6.1 Enumeration of Schedules

A simple way to sample points from a $\mathbb{Z}$-polytope in $\mathbb{Z}^n$ with uniform distribution, is to enumerate all elements of the $\mathbb{Z}$-polytope first, and then select randomly a point from the obtained sequence. Danner [46] did not describe a concrete algorithm for enumeration.

One technique that is based on projecting out dimensions of the schedule coefficient vector space using an adaptation of Fourier-Motzkin variable elimination is due to Pouchet [119]. Fourier-Motzkin variable elimination can project variables out of a system of inequalities. Generally, the polyhedra that model search space regions are of infinite volume. Therefore, it is necessary to bound them before using enumeration sampling. A possible way to bound the polyhedra in a search space region's model is to intersect them with a hypercube such that each schedule coefficient is bound to small integers. A reasonable hypercube could be $[-3, 3]^n$ to allow for non-unit skewing, but to also avoid overly large schedule coefficients. If the polytope that results from the intersection is empty, the bounds must be widened.

Fourier-Motzkin variable elimination [138] can yield a solution set that may be bounded by a super-exponential number of constraints [119, 161]. Weispfenning [161] names

$$(n+1) \cdot \left(\frac{|\mathcal{S}|}{2}\right)^{2^n}$$

as a bound for the number of constraints in the case that $|\mathcal{S}| \geq 2$. $n \in \mathbb{N}$ is the number of variables in the input constraint system and $\mathcal{S}$ is the index set of the input constraint

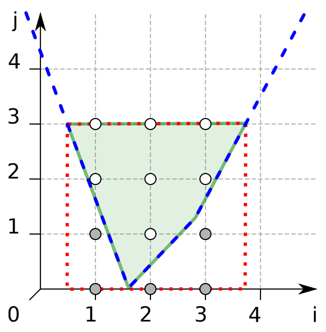


Figure 4.4: Illustration of acceptance-rejection sampling for $\mathbb{Z}$-polytopes.

system (consequently, $|S|$ is the number of input constraints.). Yet, redundancy elimination such as the one proposed by Pouchet [119] improves the practical scalability.

4.6.2 Acceptance-Rejection Sampling

Acceptance-rejection sampling [96] is a method to sample random values according to a (discrete) probability distribution D that is difficult to simulate directly (e.g., by the use of the inversion method). Acceptance-rejection sampling will generate samples according to a suitable probability distribution D' that is easier to simulate than D and then decide whether to accept the generated sample according to D. The method's effectiveness depends on the rejection rate, i.e., the number of unaccepted samples.

In our case, D is the uniform distribution on a $\mathbb{Z}$-polytope $P \subseteq \mathbb{Z}^n$ and D' is the uniform distribution on the unrotated $\mathbb{Z}$-hypercube H with minimal volume that contains P. Sampling elements of H with uniform distribution is simple: per dimension $d \in \{1, \dots, n\}$, H has a lower bound lb_d and an upper bound ub_d. To sample a point $\vec{x} = (x_1, x_2, \dots, x_n)^T \in H$, one can choose randomly $x_d \in [lb_d, ub_d] \cap \mathbb{Z}$ independent of $x_1, \dots, x_{d-1}, x_{d+1}, \dots, x_n$. Then one must check whether $P \cap \{x\}$ is empty. Only in the case of non-emptiness, we accept p.

Similarly to enumeration sampling (refer to Section 4.6.1), acceptance-rejection sampling needs a polytope to operate on. Thus, we must intersect the polyhedra that represent a search space region with a bounding hypercube.

Figure 4.4 illustrates acceptance-rejection sampling. The original $\mathbb{Z}$-polyhedron is the one bounded by the dashed lines. Intersecting it with $[-3, 3]^2$ yields the $\mathbb{Z}$-polytope P that corresponds to the filled area. D is the uniform distribution on P and D' is the uniform distribution on the hypercube that is bounded by the dotted line. Any of the filled circles will be rejected and any of the white circles will be accepted. This technique is also described by Mete and Zabinsky [99]. They find that, while it allows exact uniform sampling of P, it can be very inefficient if n is large.

4.6.3 Pattern Hit-and-Run Sampling

Pattern hit-and-run sampling (PHR sampling) [99] is a Markov chain Monte Carlo technique [33] to sample points from a discrete polytope according to a given probability distribution P^*. $\mathbb{Z}$-polytopes are a kind of discrete polytopes. A *Markov chain* [33] is a sequence of states such that a transition from one state to another occurs with a fixed probability. The sum of the transition probabilities from one state to all other states is 1. Starting at state S_i, the choice for its successor S_{i+1} depends only on S_i: given a sequence of previous states $S_1, S_2, \dots, S_i$ we have $P(S_{i+1} \mid S_1, S_2, \dots, S_i) = P(S_{i+1} \mid S_i)$.

While Danner [46] provides a wider overview of hit-and-run sampling of polytopes, we focus on one specific algorithm that was also referenced and described by Danner.

In our case, PHR starts from a random valid point $\vec{x}$ in a discrete polytope $Q \in \mathbb{Z}^n$. It performs a random walk that originates at $\vec{x}$ and points surrounding $\vec{x}$. The latter are chosen according to a random pattern. The random walk visits a finite number of points that are reachable via positive and negative multiples of a randomly chosen directional vector $\vec{v} \in \mathbb{Z}^n$. A random point $\vec{z}$ of the points on the generated random walk that are elements of Q is chosen. Next, PHR tests whether $\vec{z}$ should be accepted as the next sample. This test involves P^*.

Being a variant of the Metropolis-Hastings algorithm [69], PHR generates a sequence of samples that converges to P^*. The more samples, the better the approximation of P^*.

4.6.4 Geometric Divide-and-Conquer Sampling

Danner [46] proposed a sampling algorithm for $\mathbb{Z}$-polytopes $P \subseteq \mathbb{Z}^n$ that is based on an approach by Pak [111]. Pak's algorithm finds a hyperplane H that splits P in two halves of approximatively equal cardinality. It continues recursively with either one of the halves or $P \cap H$. The recursion stops as soon as the remaining $\mathbb{Z}$-polytope contains a single point. This point is the next sample.

Pak uses Barvinok's algorithm (refer to Section 2.2.1.5) to determine a $\mathbb{Z}$-polytope's cardinality.

Pak's algorithm has a time complexity in $\mathcal{O}(n^2 \cdot \log(L) \cdot L^{\mathcal{O}(n)})$. L is the bit-size of the input constraint system that describes P. The first two factors are the number of calls to Barvinok's counting algorithm and $L^{\mathcal{O}(n)}$ the time complexity of Barvinok's algorithm itself. The authors point out that the number of invocations of Barvinok's algorithm can be reduced to $n \cdot \log(L)$.

Danner modified the algorithm by switching to enumeration sampling (refer to Section 4.6.1) if the remaining $\mathbb{Z}$-polytope's cardinality is below a constant threshold. Algorithm 4.4 is Danner's adapted procedure. The probability of recursing into $H_+ \cap P$, $H_- \cap P$, or $H \cap P$ depends on the respective polyhedron's volume. Thus, the sampling is uniform.

Algorithm 4.4: Combined Divide-and-Conquer and Enumeration Sampling

```
Input: $\mathbb{Z}$-Polytope P $\subseteq \mathbb{Z}^n$
Output: Uniformly sampled point $\vec{p} \in P$
Parameters: t  Threshold for switching to enumeration sampling
1  Procedure divideAndConquerSample(P)
2      if |P| $\leq$ t then
3          return rand(P)
4      Find hyperplane $H$ such that $|H_+ \cap P| / |P| < 0.5 \wedge |H_- \cap P| / |P| < 0.5$ with $H_+$ and $H_-$ being
         the half-spaces of $\mathbb{R}^n \setminus H$
5      $\alpha \leftarrow |H_+ \cap P| / |P|$ ; $\beta \leftarrow |H_- \cap P| / |P|$
6      $p \leftarrow$ rand([0, 1])
7      if $p \leq \alpha$ then
8          return divideAndConquerSample($H_+ \cap P$)
9      if $p \leq \alpha + \beta$ then
10         return divideAndConquerSample($H_- \cap P$)
11     return divideAndConquerSample($H \cap P$)
```

4.6.5 Geometric Approach Based on the Decomposition Theorem for Polyhedra ("Chernikova Sampling")

According to the Decomposition Theorem for Polyhedra (Theorem 2.2.1), any polyhedron $P \in \mathbb{R}^n$, can be represented geometrically by a set V of points, one from each of P's minimal faces, a set R of unidirectional rays, and a set L of bidirectional lines. Per point $\vec{p} \in P$, sets of coefficients $\alpha_1, ..., \alpha_{|V|} \in \mathbb{R}_0^+, \sum_{i=1}^{|V|} \alpha_i = 1$, $\beta_1, ..., \beta_{|R|} \in \mathbb{R}_0^+$, $\gamma_1, ..., \gamma_{|L|} \in \mathbb{R}$ exist such that

$$\vec{p} = \sum_{\vec{v}_i \in V} \alpha_i \cdot \vec{v}_i + \sum_{\vec{r}_i \in R} \beta_i \cdot \vec{r}_i + \sum_{\vec{l}_i \in L} \gamma_i \cdot \vec{l}_i.$$

Chernikova's algorithm (refer to Section 2.2.1.2) allows us to compute the geometric representation from the constraint representation. Thus, we can sample a polyhedron by

Table 4.2: Characteristics of the PolyBench 4.1 benchmarks' SCoPs. The SCoPs have been modeled by the version of Polly that we used for the empirical evaluation[6].

benchmark	# statements	# deps	max. loop depth	# structure parameters	% unit generators	avg. $\lvert L\rvert$	avg. $\lvert R\rvert$	avg. $\lvert V\rvert$	% rational points
2mm	4	6	3	7	99.83%	9	19	1	0.00%
3mm	6	10	3	9	99.89%	10	31	1	0.00%
adi	9	64	3	3	83.79%	4	27	4	1.10%
atax	3	4	2	3	99.88%	4	11	1	0.00%
bicg	2	4	2	3	100.00 %	4	6	1	0.00%
cholesky	4	8	3	2	88.76%	3	27	4	0.00%
correlation	13	19	3	3	97.18%	17	37	1	0.00%
covariance	7	12	3	3	99.49%	4	26	1	0.00%
deriche	11	25	2	3	91.54%	4	70	26	0.00%
doitgen	3	8	4	5	99.08%	6	1	1	0.00%
durbin	10	43	2	1	79.12%	2	28	4	0.76%
fdtd-2d	4	24	3	4	91.58%	5	32	3	0.00%
floyd-warshall	1	18	3	2	100.00 %	3	1	1	0.00%
gemm	2	2	3	5	100.00 %	8	7	1	0.00%
gemver	4	6	2	2	99.73%	3	15	1	0.00%
gesummv	3	3	2	2	100.00 %	4	7	1	0.00%
gramschmidt	9	23	3	3	85.47%	4	45	4	0.17%
heat-3d	2	171	4	2	25.80%	3	19	2	43.40%
jacobi-1d	2	16	2	2	76.94%	3	4	1	10.79%
jacobi-2d	2	56	3	3	69.57%	4	6	1	7.77%
lu	3	8	3	2	90.25%	3	23	3	1.82%
ludcmp	20	89	3	2	77.57%	3	66	10	0.68%
mvt	2	2	2	2	100.00 %	8	2	1	0.00%
nussinov	8	24	3	3	97.79%	13	3	1	0.43%
seidel-2d	1	59	3	3	85.56%	4	3	1	0.47%
symm	5	21	3	4	92.22%	5	1	1	0.01%
syr2k	2	2	3	4	100.00 %	7	7	1	0.00%
syrk	2	2	3	4	100.00 %	7	6	1	0.00%
trisolv	3	5	2	2	97.63%	3	8	1	0.00%
trmm	2	4	3	4	84.12%	6	9	1	0.00%

[6]We used Polly in the version after commit `2b618e01` (Jan. 27, 2016) of `http://llvm.org/git/polly.git`. This was the most recent version when we started carrying out experiments for this thesis. Note that newer versions of Polly model SCoPs using fewer structure parameters.

choosing coefficients for its generators (i.e., the elements of the sets V, R, and L) that adhere to the constraints imposed by the Decomposition Theorem for Polyhedra.

The number of vertices of a polytope in $\mathbb{R}^n$ can be as much as 2^n [52], which makes their enumeration computationally infeasible. On the other hand, for Chernikova sampling, we do not need to bound the value range of the schedule coefficients besides legality constraints. This reduces the number of generators. For the evaluation of Polyite, we used the PolyBench 4.1[121] benchmark set. Table 4.2 is a statistics regarding the average number and characterization of the generators of 1000 search space regions chosen randomly and independently per benchmark program in the set. For the average number of rays, lines, and points, we analyzed P_1 of each search space regions's representation since this polyhedron is among the most constrained polyhedra in each region's representation. Across PolyBench 4.1 the number of generators is acceptably low if one does not bound schedule coefficients further than necessary for the preservation of legality. Further, we analyzed the share of points with rational coordinates in search space regions' geometric representation and the share of generators with components from $\{-1, 0, 1\}$.

By the design of Chernikova sampling, we allow for rational schedule coefficients. Mathematically, this is not problematic since a schedule coefficient vector in $\mathbb{Q}^n$ can be transformed into a schedule coefficient vector in $\mathbb{Z}^n$ by multiplying it with its lowest common denominator (LCD). Both schedule coefficient vectors encode schedules that impose the same relative execution on all statement instances. Yet, a large LCD can lead to an integer vector with very large components. This may trigger integer overflows in the generated code. Therefore, such vectors are to be avoided and we use integer coefficients for the generators.

The points in V remain as a source for rational components in schedule coefficient vectors. Their coefficients must be from $[0, 1]$ and the coefficients' sum must be 1. Even for points with integer coordinates, this makes schedule coefficient vectors with integer components unlikely if more than one point has a non-zero coefficient. Therefore, we use single randomly chosen points as the basis for schedule coefficient vectors. Table 4.2 shows that points with rational coordinates are rare. We allow an arbitrary number of rays and lines to have non-zero coefficients.

As we start the construction of a schedule coefficient vector from a single point in V and because we use only integer coefficients for the generators some schedules may be unreachable. Example 4.6.1 demonstrates such a case.

Example 4.6.1. The polyhedron $P = \{(i, j)^T \mid -i + j + 1 \geq 0 \ \land \ 2 \cdot i + j - 5 \geq 0\}$ is spanned by point $(2, 1)^T$ and rays $(-1, 2)^T, (1, 1)^T$. Point $(2, 2)^T$ corresponds to the linear combination $(2, 1)^T + \frac{1}{3} \cdot (-1, 2)^T + \frac{1}{3} \cdot (1, 1)^T$. By construction of P as an element of a schedule search space region's representation, $(\forall \alpha \in \mathbb{R} : \alpha \cdot (2, 2)^T \in P)$ holds. Yet, one can verify that the equation $\alpha \cdot (2, 2)^T = \beta \cdot (-1, 2)^T + \gamma \cdot (1, 1)^T$ is unsolvable as an ILP. Therefore, in the context of scheduling, neither $(2, 2)^T$ nor any vector that is obviously equivalent to it can be constructed from $P's$ generators and integer coefficients. $\triangleleft$

Some of the genetic operators that Chapter 6 describes are still able to introduce rational generator coefficients to reach otherwise unreachable schedules.

Chernikova sampling permits to bias the exploration towards schedules with certain properties easily. The number of rays and lines with non-zero coefficients can be bounded in order to avoid a too high density of schedule matrices. Per invocation of the sampling strategy on a polyhedron P, we select a random subset of P's rays and lines that has non-zero coefficients. The coefficients for the chosen rays are sampled randomly from $\{1, \ldots, \mathsf{rRange}\}, \mathsf{rRange} \in \mathbb{N}$. The coefficients for the chosen lines are sampled randomly from $\{-\mathsf{lRange}, \ldots, \mathsf{lRange}\} \setminus \{0\}, \mathsf{lRange} \in \mathbb{N}$. If we allowed arbitrarily large schedule coefficients, we would, as already mentioned, obtain erroneous programs. On the other hand, too small coefficients would prohibit opportunities for loop distribution and non-unit skewing. From the statistics in Table 4.2, we know that most of the polyhedra's generators' components are from $\{-1, 0, 1\}$. This allows us to mostly control the schedule coefficients' value range by adjusting rRange and lRange. We set $\mathsf{rRange} = \mathsf{lRange} = 3$.

4.6.5.1 Corner Cases

Rarely, we observe points in V with rational coordinates that have large denominators. It is then necessary to replace these by the closest point with integer coordinates that is inside the polyhedron. Also, rays with extremely large components must be purged from R.

4.6.5.2 Bounding Schedule Coefficients

Chernikova sampling cannot sample points from the interior of polytopes. It is therefore not advisable to start the search space construction (Algorithm 4.2) from a hypercube U that is bounded to the coefficients' value range desired and then use Chernikova sampling to sample the resulting polyhedra P_d. Instead, one must drop the constraints that bound the schedule coefficients' value range to $\{-1, 0, 1\}$ but are not required for the schedules' legality from P_d before the application of Chernikova's algorithm. Subsequently, one combines Chernikova sampling with acceptance-rejection sampling: any schedule coefficient vector produced that is not in U will be rejected.

4.6.6 Sampling by Projection

Projection sampling from a $\mathbb{Z}$-polyhedron $P \in \mathbb{Z}^n$ constructs a sample point by iterating over the dimensions of $\mathbb{Z}^n$ and choosing a coordinate per dimension d such that the resulting point belongs to P. To obtain the set of allowed values for the coordinate in dimension d, the algorithm projects P onto d. A polyhedron's projection onto a dimension of its encasing vector space is not necessarily a convex set (i.e., an interval) [123]. The projection's "holes" must be beard in mind.

Pouchet [119] samples by projection to enumerate the points contained by an n-dimensional $\mathbb{Z}$-polytope P. It is possible to explore only the projection of P onto a subspace of $\mathbb{Z}^n$. We adapt this technique to sample points from arbitrary $\mathbb{Z}$-polyhedra without having to use enumeration. Also, we add the ability to choose coefficient values according to other discrete probability distributions than uniform distribution. Algorithm 4.5 is the complete procedure. We define $-\infty$ to be the minimum and ∞ to be the maximum of a set that covers $\mathbb{Z}$.

Algorithm 4.5: Projection Sampling

```
Input: Polyhedron $P \in \mathbb{Z}^n$
Output: Point $\vec{p} \in P$
Parameters: coeffsMin $\in \mathbb{Z}$: Preferred minimum value for schedule coefficients
            coeffsMax $\in \mathbb{Z}$: Preferred maximum value for schedule coefficients
            f : $\mathbb{Z} \to [0, 1]$  Probability function of the discrete prob. distribution for schedule coefficients.
1  $\vec{p} \leftarrow \vec{0}$
2  $Dims \leftarrow [1, \dim(P)] \cap \mathbb{N}$
3  while $Dims \neq \emptyset$ do
4      $d \leftarrow$ rand($Dims$); $Dims \leftarrow Dims \setminus \{d\}$
5      $P' \leftarrow$ project($P$, $d$)
       ▷ Determine the minimum and maximum value for the corrdinate of $\vec{p}$ in dimension $d$. ◁
6      $d_{\min} = \min\{(\max\{\text{coeffsMin}, \min P'\}), (\max P')\}$
7      $d_{\max} = \max\{(\min\{\text{coeffsMax}, \max P'\}), (\min P')\}$
8      $P' \leftarrow P' \cap [d_{\min}, d_{\max}]$; $c \leftarrow 0$
9      do
10         $c \leftarrow$ rand($P'$)
11     while ¬accept($c$, f);
12     $\vec{p}^{(d)} \leftarrow c$
13 return $\vec{p}$
```

Function `project` projects a given polyhedron P onto the given dimension d of P's surrounding vector space. Function `accept` tests whether a given uniformly sampled value c should be accepted according to a given probability function in the context of acceptance-rejection sampling.

Unless forced to do so by constraints or choices made for other coefficients, the algorithm does not choose coordinate values that lie outside a configurable set $[\text{coeffsMin}, \text{coeffsMax}] \cap \mathbb{Z}$. Thereby, we ensure schedule coefficients with reasonably small absolute values.

4.6.7 Discussion

We have presented six techniques to sample schedule coefficient vectors from polyhedra. In the following, we assess each of them for their compliance with the objectives named in Section 4.1.

Observed Computational Complexity To sample a single point from a $\mathbb{Z}$-polytope P, enumeration sampling (refer to Section 4.6.1) must enumerate the elements of P and select one randomly. This makes enumeration sampling computationally infeasible for $\mathbb{Z}$-polytopes with a large volume [46]. The schedule coefficient space can have hundreds of dimensions. In the absence of data dependences, the polytope to sample from is a hypercube $H = [lb, ub]^n \cap \mathbb{Z}^n, lb, ub \in \mathbb{Z}, n \in \mathbb{N}$. Since $|H| = (ub - lb + 1)^n$, enumeration sampling is computationally infeasible.

Acceptance-rejection sampling (refer to Section 4.6.2) samples points from a polytope's surrounding minimal bounding box uniformly. It keeps ("accepts") those points that belong to the polytope. Acceptance-rejection sampling is efficient if the acceptance rate is high [46]. In the context of scheduling, it is possible to construct search space regions such that the acceptance rate after bounding the schedule coefficients is below 2% (refer to Example 4.6.2) if we only consider schedule matrices with integer coefficients. This makes the technique inefficient.

Example 4.6.2. Sampling the set of all weakly satisfying one-dimensional schedules with schedule coefficients from $[-3, 3] \cap \mathbb{Z}$ for the program shown in Listing 4.2 with acceptance-rejection sampling results in an acceptance rate of $\approx$ 1.30%. The schedule coefficient space has eight dimensions.

◁

Listing 4.2: A small program with an extraordinarily low acceptance rate of acceptance-rejection sampling if the schedule coefficients are bounded to $[-3, 3] \cap \mathbb{Z}$.

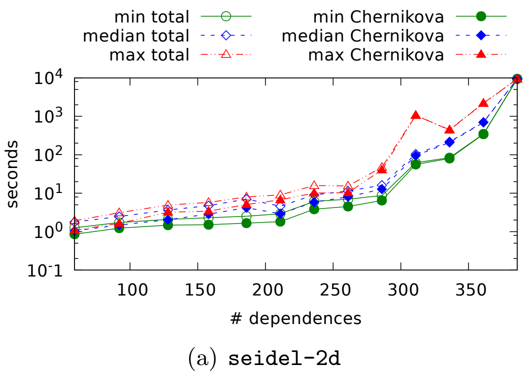
```
for (i = 0; i < 4; ++i) {
  A[i] = 42;
  A[i + 1] = A[i];
}

for (i = 0; i < 4; ++i) {
  A[i] = 41;
  A[i + 1] = A[i];
}
```

Pattern hit-and-run sampling (refer to Section 4.6.3) must generate a high number of points to achieve a uniform distribution on a $\mathbb{Z}$-polytope. For a particular class of n-dimensional $\mathbb{Z}$-polytopes, Mete and Zabinsky [99] show that this number is $n^{5.5}$. Generally, the polytopes that result from bounding the polyhedra in search space regions' models are not in this class [46]. Recall that, in order to obtain a good coverage of a SCoP's schedule search space, we must first sample the set of search space regions by invoking Algorithm 4.2 repeatedly and then sample a small number of schedules from each region. Having to generate a very large number of schedules from each search space regions to keep just a few of them is inefficient. PHR-sampling could be useful to obtain an approximately uniform distribution from the sampling of one specific search space region.

Seeking a technique to sample schedule from a single search space region with uniform distribution, Danner [46] resorted to geometric divide-and-conquer sampling. We determined the empirical run-time complexity of geometric divide-and-conquer sampling (refer to Section 4.6.4), Chernikova sampling (refer to Section 4.6.5), and projection sampling (refer to Section 4.6.6). For this purpose, we had to scale the number of statements, schedule coefficient space dimensions, and data dependence polyhedra of a SCoP. This can be done in a semantics-preserving way by partially unrolling the SCoP's loops [148]. We experimented with the stencil programs `seidel-2d` and `jacobi-2d` of PolyBench 4.1. The algorithms have been implemented in Scala 2.11 [110] and were executed on OpenJDK 8 on an Intel® Xeon® E5-2650 v2 @ 2.60GHz CPU. The operating system was Debian 9 with Linux 4.9. We used isl (commit `cfebc0c6` (Dec. 11, 2015) of `https://repo.or.cz/isl.git`) [152] and libbarvinok (commit `91ba8f18` (May 26, 2018) of `http://repo.or.cz/barvinok.git`)[157], which implements Barvinok's counting algorithm (refer to Section 2.2.1.5) to represent polyhedra. We used Polly in the version of commit 2b618e01 (Jan. 27, 2016) of `http://llvm.org/git/polly.git` to model the unrolled loop nests as SCoPs.

The duration of converting search space regions' constraint representation to their geometric representation dominates the run time of Chernikova sampling. Thus, we measured the duration of sampling a search space region with Algorithm 4.2 (sampling of search space regions) and converting its representation. Here, we do not take the additional schedule dimensions and respective polyhedra into account that result from schedule completion (refer to Section 4.7). Figure 4.5 shows per SCoP-size the minimum, maximum and average duration in seconds for five invocations of Algorithm 4.2 and the conversion of the respective result. Besides the total duration, we show the duration of all invocations of Chernikova's algorithm separately. The time complexity of Chernikova's algorithm dominates the time complexity of Algorithm 4.2.

In the case of geometric divide-and-conquer sampling and projection sampling, the main cost lies within the sampling of schedules from search space regions. Thus, we measured the minimum, average and maximum duration in seconds of five invocations of Algorithm 4.2 and the subsequent sampling of one schedule from each resulting search space region. Figure 4.7 shows the results for geometric divide-and-conquer sampling. We can show results for `seidel-2d` only, as in the case of `jacobi-2d` the duration of sampling one schedule was



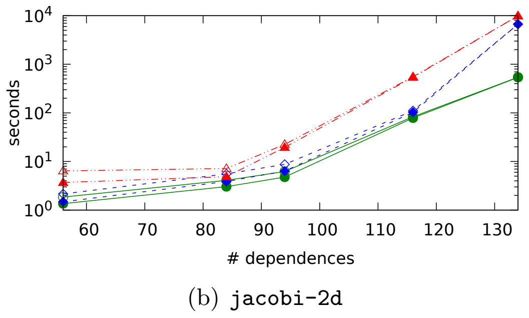


Figure 4.5: Mininum, median, and maximum duration in seconds of constructing a search space region and converting it to its geometric representation relative to the number of data dependence polyhedra. We show the total duration and the duration of the conversion with Chernikova's algorithm only.[7]

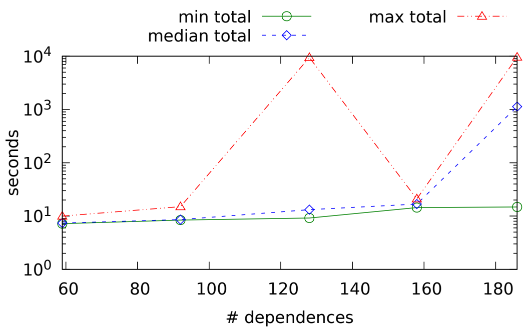
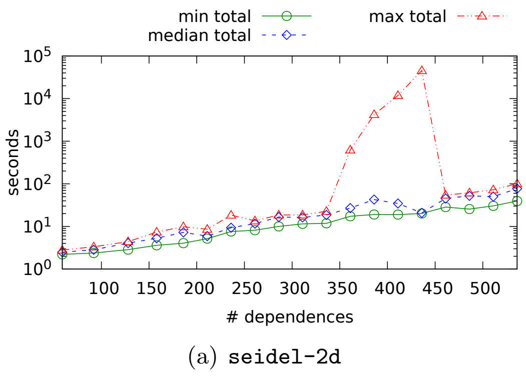


Figure 4.6: Minimum, median, and maximum duration in seconds of constructing a search space region and sampling a schedule from it with geometric divide-and-conquer sampling. The duration is relative to the number of data dependence polyhedra. The benchmark is `seidel-2d`.[7]



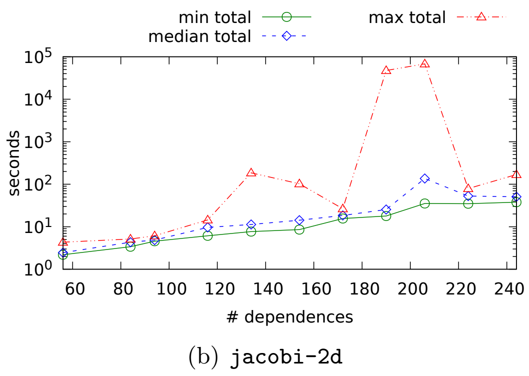


Figure 4.7: Minimum, median, and maximum duration in seconds of constructing a search space region and sampling a schedule from it with projection sampling. The duration is relative to the number of data dependence polyhedra.[7]

[7] Only the points represent measured samples. The lines were drawn to improve perceivability.

longer than one day with a single unrolled iteration per loop at the second loop level. Figure 4.6 shows the results for projection sampling.

From the experiment described, geometric divide-and-conquer sampling is impractical for the sampling of large numbers of schedules. The current implementation by Danner [46] is the unoptimized algorithm that runs in $\mathcal{O}(n^2 \cdot \log(L) \cdot L^{\mathcal{O}(n)})$. An optimized version of the algorithm has a reduced run-time complexity in $\mathcal{O}(n \cdot \log(L) \cdot L^{\mathcal{O}(n)})$. Instead of performing $n^2 \cdot \log(L)$ many invocations of Barvinok's algorithm, their number can be reduced to $n \cdot \log(L)$. Yet, the exponential run-time complexity of Barvinok's algorithm remains. At median projection sampling is very efficient. Yet, cases exist in which the duration of projection sampling exceeds its median observed duration by far.

Based on these observations, we exclude all sampling strategies except Chernikova sampling and projection sampling from the discussion.

Coverage of the Schedule Search Space Projection sampling can reach any schedule in the search space. Chernikova sampling will not reach some schedules for practical reasons, but some of the genetic operators proposed in Chapter 6 mitigate this limitation. A particular advantage of Chernikova sampling is that, if one stores together with each row of a schedule matrix the linear combination of generators that formed the row, one obtains an opportunity for fine grained schedule mutation in a genetic algorithm.

Biasing the Exploration Both Chernikova sampling and projection sampling have parameters that permit to bias the exploration of search space regions towards schedules with particular properties.

4.7 Schedule Completion

The combination of Algorithms 4.2 (Sampling of Search Space Regions) and 4.3 (Sampling of Schedules from Search Space Regions) does not yield schedules in which every loop of the transformed program is encoded explicitly. As mentioned, polyhedral code generators will choose one possible execution order and generate code accordingly. Yet, without encoding the inner loop levels explicitly, the iterative optimization does not have full control over them and further transformation of schedules by applying tiling or strip mining would be impossible. Consequently, each randomly generated schedule matrix undergoes a process that we call schedule completion. *Schedule completion* appends rows to a schedule matrix until we cannot find, for any statement, another row that is linearly independent in the iteration variable coefficients.

Given a set $S \subseteq \mathbb{Z}^n$, we construct its set of linearly independent vectors as the complement of the linearly dependent vectors [86]:

$$\text{lin.indep}(S) = \mathbb{Z}^n \setminus \left\{ \sum_{\vec{v}_i \in S} \alpha_i \cdot \vec{v}_i \mid \alpha_1, \ldots, \alpha_{|S|} \in \mathbb{R} \right\}.$$

Due to the exclusion of $\vec{0}$, lin.indep(S) is not necessarily convex but a union of several polyhedra. To sample a vector from lin.indep(S), we choose one of the polyhedra randomly and apply the selected sampling technique for polyhedra.

Be aware that this procedure deviates from the schedule completion described by Pouchet et al. [124]. Their schedule completion initializes previously undefined coefficients of a schedule matrix and corrects coefficients in the schedule matrix to place it inside their schedule search space.

4.8 Adapting Pouchet's Approach in Polyite

To be able to endorse the assumption that parallelization requires (a) the consideration of a less restricted search space than the one proposed by Pouchet et al. [124] and (b) schedule completion as described in Section 4.7, we devised an algorithm for search space construction following Pouchet et al. [124] and combined it with our approach to splitting the set of data dependences into dependence polyhedra and the variant of Chernikova sampling that can bound the values of schedule coefficients (refer to Section 4.6.5.2).

A direct use of their implementation LeTSeE [120] was not possible, due to the incompatibility of its file format with Polly. Also, LeTSeE does not output schedule trees, which Polly requires as input to its schedule transformations, such as tiling and strip mining. Moreover, and to the best of our knowledge, statement iteration domains that can be expressed only as the union of multiple polyhedra cannot be represented with LeTSeE's input format. Polly and Polyite support these. Finally, we do not rely on the original sampling strategy, since it is based on assumptions that may not hold in the context of tiling and parallel execution.

Pouchet et al. only estimated the order of magnitude of statements' memory traffic using a simplification of an estimating formula by Bastoul and Feautrier [18]. Let S be a statement and A be an n-dimensional array that S accesses. Let m_d be the size of dimension d of A. Pouchet et al. propose to approximate the traffic from S to dimension d of A as $m_d^{r_{A,d}}$. $r_{A,d}$ is the rank of the concatenation of all subscript matrices of all accesses from S to dimension d of A. In our adaptation, we aggregate these values by summation. Thus, we approximate the traffic $T_{S,A}$ from S to A as

$$T_{S,A} = \sum_{d=1}^{n} m_d^{r_{A,d}}.$$

Thanks to the availability of libbarvinok, we can compute a more precise estimate of a statement's traffic. This estimate is also inspired by Bastoul and Feautrier [18]. In this estimation, we do not account for the presence of the cache hierarchy between the main memory and the processor. Let S be a statement, M be a memory location, and $A_{S,M}$ bet the set of all memory access functions from S to M. We can estimate the memory traffic from S to M as

$$\left| \bigcup_{f \in A_{S,M}} \{ (\Theta_S(\vec{i}), f(\vec{i})) \mid \vec{i} \in I_S \} \right|.$$

The difference to the estimate by Bastoul and Feautrier [18] is, that in their case Θ_S is not an injective schedule, but a *chunking function*, which partitions the iteration domain of S into chunks such that the data on which a chunk operates fits into the cache.

Bastoul and Feautrier [18] and Pouchet were interested only in the order of magnitude of statements' memory traffic. Thus, in the adapted algorithm. we calculate the memory traffic in the number of accessed array cells. A more precise implementation should multiply the calculated data volume per statement and array by the bit width of the array's data type for better accuracy. Yet, for the PolyBench 4.1 benchmark set, which we use in our empirical evaluation, this improvement has almost no effect. With the exception of the program `nussinov`, which operates on a `char` array of size $n \in \mathbb{N}$ and an array of `int` values of size n^2, all programs in PolyBench 4.1 use only one data type for the arrays on which they operate.

Optionally, we can combine this algorithm with schedule completion (refer to Section 4.7) to study the impact of the latter.

5 Schedule Simplification and Analysis

We propose a transformation of schedule matrices, as they result from the sampling technique described in Chapter 4, to a representation that is more amenable to an analysis and further transformation of schedules than the matrix representation itself. Whenever we need to analyze a schedule's structure, or pass it to POLLY, which is the polyhedral program optimizer that we use to model SCoPs and apply schedules to SCoPs, we transform the schedule in the way that we describe in the following.

Schedule matrices (refer to Section 4.2) are a convenient representation for the sampling of schedules. They have been used by Pouchet et al. [124] for their approach to iterative schedule optimization and we have followed their choice. The particular advantage of schedule matrices for sampling is their uniformity in the representation of all linearly affine schedule functions for a given SCoP: they can all be represented by matrices with the same number $m \in \mathbb{N}$ of columns and differing numbers of rows. Given a schedule matrix M_Θ that is incomplete in that not all data dependence polyhedra of the SCoP are carried by schedule Θ, or not all loops in the transformed program are encoded explicitly in Θ, to build row $M_\Theta^{\text{rows}(M_\Theta)+1}$, one must choose a vector from $\mathbb{Z}^m$.

In Section 5.1, we motivate why schedule matrices are less suitable for a further transformation, for instance by tiling, or an analysis of schedules. Such an analysis could be the extraction of structural schedule features for the purpose of classification (refer to Chapter 7).

A schedule representation that serves transformation and analysis of schedules much better is the schedule tree [66]. A schedule tree is a data structure that is related closely to the structure of the PLuTo-algorithm's recursion tree (refer to Section 2.2.2.5). We recall schedules trees to the extent necessary for this thesis in Section 5.2.

The result of a schedule optimization with ISL's variant of the PLuTo algorithm is a schedule tree. While ISL can transform schedule trees to schedule matrices, it does not provide a sophisticated reverse transformation. With ISL, the only option is to wrap the coefficient matrix in a schedule tree that essentially consists of a single node [154]. Since the expressiveness of the schedule representation does not profit from this, we propose a more meaningful transformation of schedule matrices to schedule trees (refer to Sections 5.3) and a subsequent simplification of the resulting schedule trees (refer to Section 5.4). Particularly in the case of feature extraction from schedules, it is desirable that schedules that yield the same transformed program also yield the same feature values. A simplification of schedules trees eases the achievement of this goal. The simplification preserves the semantics of the schedule before the application of tiling. Further, it preserves and enlarges tilable bands.

In Section 5.5, we recall how we can determine the equivalence of two schedules. Further, we describe the detection of parts of schedule trees that correspond to loops and the detection of tilable bands and parallelism. In Section 5.6, we discuss the presented transformation of schedule matrices to schedule trees.

5.1 Motivation

To apply tiling to a schedule Θ or to identify parallelism, one must identify the loop nests of the program after transformation by Θ. In the polyhedron model, this requires the identification of the textual order induced by each schedule dimension.The *textual order* is the ordering of statement instances that is independent of loops, but is induced merely by the statements' arrangement in the program code. Schedule matrices offer no explicit and easily traceable way of encoding the textual order.

Further, since our schedule matrices are randomly generated, they contain non-zero schedule coefficients that neither influence the execution order of statement instances nor skew loops, which could enable tiling. Even entire rows of schedule matrices may be spare. Despite being uninfluential, such coefficients complicate the identification of schedules' structural properties such as tilable bands and should therefore be set to 0. Spare rows should be removed. The identification of spare schedule coefficients would also profit from a schedule representation in which textual execution order is encoded in a unique and directly traceable way. Finally, schedule coefficients' absolute values can be needlessly large. Too large coefficients can trigger errors in transformed programs. Particularly, they may cause integer overflows.

Example 5.1.1 illustrates the degrees of freedom in schedule matrices described.

Example 5.1.1. A schedule that encodes the execution order of statement instances in our running example `syrk` (Listing 2.1 on page 18) and that matches the code's execution order in a natural way is:

$$\Theta_R(i,j)^T = (i,0,j,0)^T,\ \Theta_S(i,j,k)^T = (i,1,j,k)^T.$$

Another schedule that encodes the same execution order is

$$\Theta_R(i,j)^T = (2\cdot i, 42\cdot j + m, i, 0)^T,\ \Theta_S(i,j,k)^T = (2\cdot i+1, 0, 21\cdot j - m, k)^T.$$

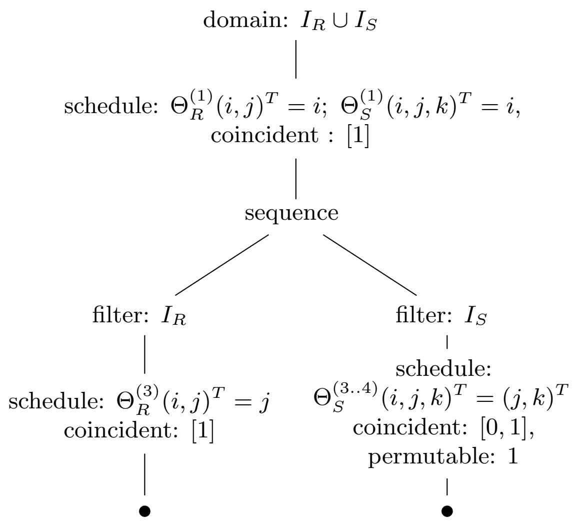
Let us use the latter schedule as a running example to illustrate the schedule tree transformation and the steps of the schedule tree simplification. ◁

5.2 Schedule Trees

We find that schedule trees [66] can be described as a polyhedral hybrid representation that borrows from linearly affine schedule functions on the one hand and abstract syntax trees (ASTs) on the other hand. As mentioned, a schedule function can essentially be represented as a schedule tree by merely encapsulating it in a single schedule tree node. The result is a valid schedule tree. That is, a schedule tree can serve as a simple wrapper for an arbitrary schedule function. Yet, it appears that the intended use of schedule trees is to rely on schedule functions to express loops and to utilize the tree's topology to express textual order.

We recall the concept, to the extent necessary, on the basis of our running example `syrk`.

Example 5.2.1. Figure 5.2.1 shows the schedule tree that encodes the execution order of the statement instances in the source code of our running example `syrk` (Example 2.2.12 on page 18). The tree's root node is its domain node. A *domain node* specifies the SCoP's iteration domain. Underneath the domain node follows a one-dimensional *band node*. Band nodes contain partial schedule functions and typically serve the purpose of encoding loops. In our case, the band node underneath the root node encodes the SCoP's outermost loop, which encases both of its statements. The dimensions of a band node's partial schedule function can be marked *coincident* (parallel). The band node's child is a sequence node. A *Sequence node* specifies that its children must be executed in the order given. It serves the particular purpose of encoding textual ordering. Each of a sequence node's subtrees has a filter node at its root. A *filter node* has a similar purpose as the domain node at the root of a complete schedule tree: it specifies the iteration domain of the subtree. Therefore, a filter node contains an integer set that must be a subset of its parent sequence node's iteration domain. Together, the sets contained in the filter nodes underneath a sequence node must form a partition of the sequence node's iteration domain. Otherwise, some statement instances would be computed twice, and statement instances that are undeclared by the



domain: $I_R \cup I_S$
schedule: $\Theta_R^{(1)}(i,j)^T = i$; $\Theta_S^{(1)}(i,j,k)^T = i$,
coincident : [1]
sequence
filter: I_R
filter: I_S
schedule: $\Theta_R^{(3)}(i,j)^T = j$
coincident: [1]
schedule:
$\Theta_S^{(3..4)}(i,j,k)^T = (j,k)^T$
coincident: [0, 1],
permutable: 1

Figure 5.1: The schedule tree of our running example `syrk` (Example 2.2.12).

domain node would be scheduled, or statement instances would not be scheduled at all. In the example, all statement instances of statement R are scheduled in the first subtree and all statement instances of statement S are scheduled in the second subtree. The sequence node carries the dependence polyhedron $D_{R,S}$ of our running example. The schedule function of the band node in the right subtree is two-dimensional. It encodes the j- and the k-loop that encase statement S. The band node is marked *permutable*, which enables it for tiling. Branches of schedule trees are terminated by *leaf nodes*. The construction of a schedule tree starts from the bottom up. That is, one starts with the leaf nodes. Per branch, its leaf node specifies the iteration domain. Filter nodes and the domain node result implicitly from unifying the iteration domains of the leaf nodes in their subtrees. $\triangleleft$

Note that we imply in graphical representations of schedule trees and in our algorithms that the instances of different statements are distinguishable by their statement affiliation.

5.3 Schedule Tree Transformation

This section describes the transformation of a schedule matrix to an unsimplified schedule tree. The algorithm presented in the following starts from a set of statements $S_1, \ldots, S_m$ with respective iteration domains $I_{S_1}, \ldots, I_{S_m}$ and an n-dimensional schedule matrix M_Θ. M_Θ can be decomposed into the statement schedules $\Theta_{S_1}, \ldots, \Theta_{S_m}$. The algorithm detects schedule dimensions that encode the textual ordering of statements and partitions recursively the set of statement instances according to textual order. The detected splits are recorded in a schedule tree.

Schedule dimension k *textually orders* two statements X and Y if it prescribes that all instances of X are executed before all instances of Y. Formally, we obtain the following partial order $<^k$ of the statements' iteration domain:

$$I_X <^k I_Y \Leftrightarrow \Big(\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : \Theta_X^{(1..k-1)}(\vec{i}_X) = \Theta_Y^{(1..k-1)}(\vec{i}_Y) \;\Rightarrow\; \Theta_X^{(k)}(\vec{i}_X) < \Theta_Y^{(k)}(\vec{i}_Y)\Big).$$

The ISL code generator [66] also determines the textual ordering of statements. All polyhedral code generators must determine textual ordering, but they must operate at a finer grain level and split statements' iteration domains [16, 66, 150]. Schedule trees would allow us to do the same and, thereby, we could obtain a stronger simplification of schedules. We would also be able to obtain a schedule tree that resembles even more closely the code that results from the application of a schedule. In Section 5.6, we elaborate on the advantages, but also the disadvantages that would result from this extension.

Another way of encoding the textual ordering of statements in schedule dimension k has been described by Bastoul [17], Grosser et al. [65], and Danner [46]. Let schedule dimension k encode a loop with stride $\alpha_k \in \mathbb{N}$ that encases the statements $S_1, ...S_m$. Let each statement schedule $\Theta_x, x \in \{1, ..., m\}$ be of the form $\alpha_k \cdot i + \beta_{k_x}$ in dimension k with $\beta_{k_1}, ..., \beta_{k_m} \in ([0, ..., \alpha^k[\ \cap\ \mathbb{N})$. i is an iteration variable in the original program. We can represent schedule dimension k by a band node that is followed by a sequence node. The band stores, per statement, the one-dimensional schedule $\alpha \cdot i$. The sequence node enumerates the statements according to the values $\beta_{k_1}, ..., \beta_{k_m}$. If, for two statements S_x and S_y, $\beta_{k_x} = \beta_{k_y}$, they must be placed into the same subtree of the sequence node. Following Bastoul and Grosser et al., we could replace α by 1 in the band node constructed, but one of the simplification steps that we describe in Section 5.4 will take care of this. We denote the partial order on statement iteration domains induced be the schema described by $\tilde{<}^k$. Grosser et al. refer to the technique of detecting statements' textual order encoded in the way described as "shifted stride detection".

Algorithm 5.1 illustrates the recursive construction of a schedule tree starting from a set I of statement iteration domains and, per statement, an n-dimensional schedule matrix.

Algorithm 5.1: Basic Schedule Tree Construction (`constructTree`)

```
Input:  $I = \{I_{S_1}, ..., I_{S_m}\}$: Set of statement iteration domains
        $\Theta_{S_1}, ..., \Theta_{S_m}$: $n$-dimensional statement schedules
        $k$: current dimension, $n$: total number of schedule dimensions
Output: The constructed schedule tree.
1  Procedure Partition($J$, $\otimes$)                ▷ Set of iteration domains, order predicate
2      return $\langle P_1, ..., P_l \subseteq J \mid (\forall i, j \in \{1, ..., l\} : (i < j) \Rightarrow (P_i \cap P_j = \emptyset \wedge (\forall (I_X, I_Y) \in P_i \times P_j : I_X \otimes I_Y))) \wedge (\forall i \in \{1, ..., l\} : \neg(\exists Q \subset P_i : Q \neq \emptyset \wedge (\forall I_X \in Q : (\forall I_Y \in (P_i \setminus Q) : (\exists (\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : \Theta_X^{(1..k-1)}(\vec{i}_X) = \Theta_Y^{(1..k-1)}(\vec{i}_Y)) \wedge I_X \otimes I_Y)))) \rangle$
3  if $k > n$ then return LeafNode($I$);
4  if $\Theta_{S_1}, ..., \Theta_{S_m} == 0$ then return constructTree($I$, $\Theta_{S_1}, ..., \Theta_{S_m}$, $k+1$, $n$) ;
5  children ← ⟨⟩; $\langle P_1, ..., P_l \rangle$ ← Partition($I, <^k$)
6  if $l > 1$ then
7      for $i \in [1, l]$ do
8          $S \leftarrow \{\Theta_X | I_X \in P_i\}$; children.append(FilterNode($P_i$, constructTree($P_i$, $S$, $k$, $n$)))
9      return SeqNode(children)
10 $\langle P'_1, ..., P'_{l'} \rangle$ ← Partition($I, \tilde{<}^k$)
11 if $l' > 1$ then
12     for $i \in [1, l']$ do
13         $S \leftarrow \{\Theta_X | I_X \in P'_i\}$; children.append(FilterNode($P'_i$, constructTree($P'_i$, $S$, $k$, $n$)))
14     return BandNode($\Theta^k_{S_1} - \beta_{k_1}, ..., \Theta^k_{S_m} - \beta_{k_m}$, SeqNode(children))
15 return BandNode($\Theta^k_{S_1}, ..., \Theta^k_{S_m}$, constructTree($I$, $\Theta_{S_1}, ..., \Theta_{S_m}$, $k+1$, $n$))
```

The algorithm iterates over the schedule dimensions from the outermost to the innermost. At schedule dimension k, we topologically partition I according to either $<^k$ or $\tilde{<}^k$. In both cases, the result is a partitioning with a total order of its elements according to $<^k$ or $\tilde{<}^k$, respectively. The preference is always for $<^k$. If the resulting partitioning according to $<^k$ has more than one element, we construct a sequence node with one subtree per element of the partitioning. Otherwise, if the partitioning according to $\tilde{<}^k$ has more than one element, we construct a band node followed by a sequence node. In both cases, the sequence node's subtrees result from processing recursively schedule dimension k and the respective partition of I. If both the inspection with $>^k$ and with $\tilde{<}^k$ yield a singleton partitioning we construct a band node that contains schedule dimension k and proceed recursively for schedule dimension $k+1$ and I. If $k > n$, the algorithm constructs a leaf node and terminates. Due to the condition on line 2, the algorithm is heuristic and, in particular, procedure `Partition` does not split the given set of statements into an ordered partition of maximum size necessarily. We present an improved algorithm in Section 5.6. Example 5.3.1 demonstrates Algorithm 5.1.

Example 5.3.1. We start from the second of the schedules for `syrk` in Example 5.1.1:

$$\Theta_R(i,j)^T = (2 \cdot i, 42 \cdot j + m, i, 0)^T,\ \Theta_S(i,j,k)^T = (2 \cdot i + 1, 0, 21 \cdot j - m, k)^T.$$

Initial invocation: `constructTree(`$I = \{I_R, I_S\}, S = \{\Theta_R, \Theta_S\}, k = 1, n = 4$`)`

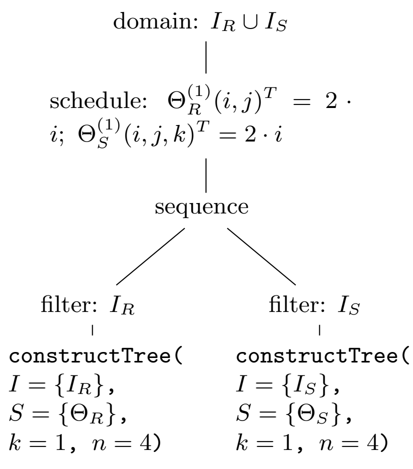
We need to inspect the first schedule dimension:

$$\Theta_R^{(1)}(i,j)^T = 2 \cdot i,\ \Theta_S^{(1)}(i,j,k)^T = 2 \cdot i + 1.$$

From the first schedule dimension and the statements' iteration domains

$$I_R = \{(i,j)^T \mid i,j \in \mathbb{Z} \ \wedge\ 0 \le i < n \ \wedge\ 0 \le j \le i\}$$
$$I_S = \{(i,j,k)^T \mid i,j,k \in \mathbb{Z} \ \wedge\ 0 \le i < n \ \wedge\ 0 \le j < m \ \wedge\ 0 \le k \le i\}$$

schedule dimension 1 prescribes an interleaved execution of R and S. Thus, the statements are not textually ordered according to $<^k$. The topological partitioning according to $<^k$ is the singleton list $\langle\{I_R, I_S\}\rangle$. Looking at the first schedule dimension, we can see that the schedule allows a partitioning according to $\tilde{<}^k$: we have $\alpha_1 = 2$, $\beta_{1_R} = 0, \beta_{1_S} = 1$. This leads to the topological partitioning $\langle\{I_R\}, \{I_S\}\rangle$ and we construct the following prefix of a schedule tree:


domain: $I_R \cup I_S$
schedule: $\Theta_R^{(1)}(i,j)^T = 2 \cdot i$; $\Theta_S^{(1)}(i,j,k)^T = 2 \cdot i$
sequence
filter: I_R
filter: I_S
constructTree($I = \{I_R\}$, $S = \{\Theta_R\}$, $k = 1$, $n = 4$)
constructTree($I = \{I_S\}$, $S = \{\Theta_S\}$, $k = 1$, $n = 4$)

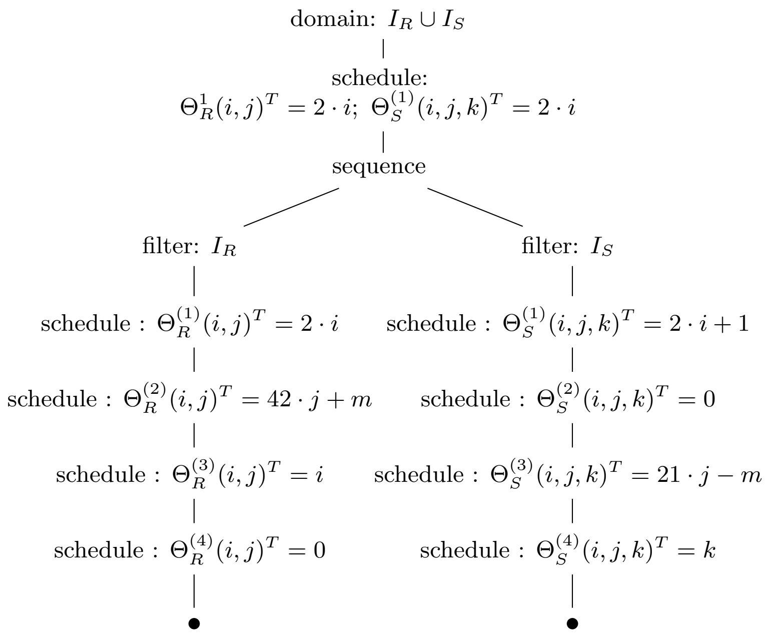
The recursive calls appended at the schedule tree's bottom both operate on a single statement's iteration domain. Thus, neither of both will produce another sequence node. Both calls yield a sequence of one-dimensional band nodes that contain the dimensions of Θ_S and, respectively, Θ_R in their given order. Figure 5.2 shows the resulting schedule tree. $\triangleleft$

5.4 Schedule Tree Simplification

The transformation of a schedule matrix to a schedule tree is followed by a sequence of schedule tree simplification steps. A simplification step is a recursive function on schedule trees that transforms a schedule tree's subtrees from the bottom up.

Using these definitions, we define two criteria that a simplifying transformation that transforms an original schedule Θ to a simplified schedule Θ' must comply with to be semantics-preserving in the context of polyhedral schedules. Both schedules are composed

Figure 5.2: A schedule tree as it results from the application of Algorithm 5.1 in Example 5.3.1.


domain: $I_R \cup I_S$
schedule:
$\Theta^1_R(i,j)^T = 2 \cdot i$; $\Theta^{(1)}_S(i,j,k)^T = 2 \cdot i$
sequence
filter: I_R
filter: I_S
schedule : $\Theta^{(1)}_R(i,j)^T = 2 \cdot i$
schedule : $\Theta^{(1)}_S(i,j,k)^T = 2 \cdot i + 1$
schedule : $\Theta^{(2)}_R(i,j)^T = 42 \cdot j + m$
schedule : $\Theta^{(2)}_S(i,j,k)^T = 0$
schedule : $\Theta^{(3)}_R(i,j)^T = i$
schedule : $\Theta^{(3)}_S(i,j,k)^T = 21 \cdot j - m$
schedule : $\Theta^{(4)}_R(i,j)^T = 0$
schedule : $\Theta^{(4)}_S(i,j,k)^T = k$

of a set of statement schedule functions. For a statement S, we address the old statement schedule by Θ_S and the new statement schedule by Θ'_S, respectively. Let I be a set of statement iteration domains, whose union is the iteration domain of the node that we are processing currently. In proofing the correctness of a schedule tree simplification step, we need not take statement instances into account that are not in I. Since the execution order is a lexicographic order, a change of the current schedule node or its subtrees cannot affect statement instances that are not contained in I.

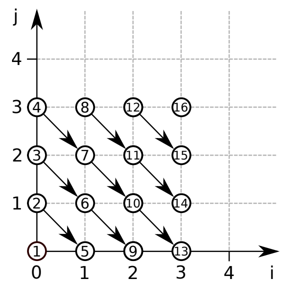
Execution Order A simplification step must not change the execution order of any pair of statement instances according to Θ. Formally, Θ and Θ' must be equivalent in the following sense:

$$\Big(\forall I_X, I_Y \in I : \big(\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : \Theta_X(\vec{i}_X) \prec \Theta_Y(\vec{i}_Y) \Leftrightarrow \Theta'_X(\vec{i}_X) \prec \Theta'_Y(\vec{i}_Y)\big)\Big). \quad (5.1)$$

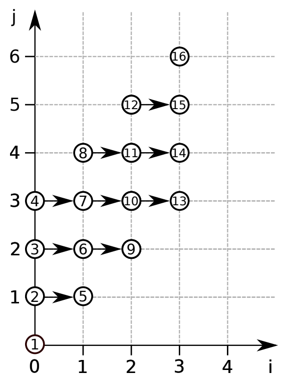
We simplify schedules before the application of tiling. Skewing is a code transformation that can enable rectangular tiling. Example 5.4.1 shows that a transformation on schedules that preserves execution order in the sense above may transform a schedule that skews a loop nest to a schedule that does not skew.

Example 5.4.1.
Let S be a statement with iteration domain $I_S = \{(i,j)^T \mid 0 \le i,j < n \ \wedge \ i,j \in \mathbb{N}\}, n \in \mathbb{N}$. Let $D_{S,S} = \{(i,j,i+1,j-1)^T \mid (i,j)^T \in I_S \ \wedge \ (i+1,j-1)^T \in I_S\}$ be a dependence polyhedron that affects schedules' legality. Let $\Theta(i,j) = (i,j)$ and $\Theta'(i,j) = (i,j+i)$ be two schedules. Figure 5.3 illustrates the iteration domain's images $\{\Theta(\vec{i}) \mid \vec{i} \in I_S\}$ (Figure 5.3(a)) and $\{\Theta'(\vec{i}) \mid \vec{i} \in I_S\}$ (Figure 5.3(b)) for $n = 4$. Other than Θ, Θ' enables tiling by skewing the iteration domain. Yet, as can be verified easily from Figure 5.3, both schedules impose the same execution order on the statement instances. $\triangleleft$

Thus, a second criterion to limit the set of semantics-preserving schedule simplification steps is required. This criterion must be sensitive to skewing.



(a) No rectangular tiling without skewing.



(b) Skewing enables rectangular tiling.

Figure 5.3: The schedule illustrated in Figure 5.3(a) does not skew the iteration domain, while the schedule illustrated in Figure 5.3(b) skews and, thereby, enables rectangular tiling. The statement instances are numbered in execution order. Both schedules prescribe the same order.

Direction of Data Dependences Let us assume that, after schedule simplification, $\Theta^{(k)}$ has become $\Theta'^{(k')}$. In the case that a simplification has deleted dimension k of Θ entirely from the schedule, we do not need to consider it further. The direction of any legality-affecting data dependence from an instance of a statement X with iteration vector $\vec{i}_X$ to an instance of a statement Y with iteration vector $\vec{i}_Y$ with respect to $\Theta'^{(k')}$ must equal its direction with respect to $\Theta^{(k)}$:

$$\operatorname{sgn}(\Theta_Y^{(k)}(\vec{i}_Y) - \Theta_X^{(k)}(\vec{i}_X)) = \operatorname{sgn}(\Theta_Y^{(k')}(\vec{i}_Y) - \Theta_X^{(k')}(\vec{i}_X)). \tag{5.2}$$

In other words, if we project the SCoP's transformed iteration domains onto vector space dimensions k and k', the corresponding dependence arrow's direction must persist.

By this second criterion, we do not require the preservation of skewing in general, but in all cases in which skewing may enable tiling.

Since we simplify the schedule trees before the encoding of tiling in the schedule, we preserve the semantics of the schedule before tiling. In addition, we preserve and enlarge sequences of schedule dimensions that correspond to loop nests that are tilable in the transformation proposed. The simplification does eliminate information that may affect the execution order of a SCoP's statement instances after tiling. In some cases, the effect depends on the tile sizes chosen. In Section 5.6, we show exemplary cases in which the execution order after tiling differs before and after the schedule's simplification.

Danner [46] also contributed proofs of correctness for the schedule tree simplification steps that we propose, but used an insufficient condition in place of Condition 5.2. In particular, Danner's condition does not guarantee the preservation of skewing.

In the remainder of this section, we present the steps that we apply in their given order to simplify schedule trees. We prove each step's compliance with Conditions 5.1 and 5.2.

5.4.1 Remove Statements' Common Offset

Translating all of a SCoP's statement instances by the same vector does not change their execution order and, therefore, does not change the SCoP's schedule [149]. Figure 5.4 illustrates this fact. Algorithm 5.2 derives a vector $\vec{v}$ that is parametric in that it contains multiples of the SCoP's structure parameters. We construct $\vec{v}$ such that, for any statement

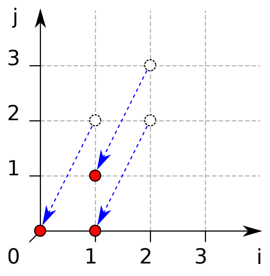


Figure 5.4: Moving all three points by the same directional vector and distance preserves their lexicographic order.

iteration domain $I_X \in I$, the points in $\{\Theta(\vec{i}) + \vec{v} \mid \vec{i} \in I_X\}$ are located closer to $\vec{0}$ than the points in $\Theta_X(I_X)$:

$$\left(\forall I_X \in I : \left(\forall \vec{i} \in I_X : \left(\forall d \in \{1, \ldots, \dim(\vec{i})\} : \left|\left(\Theta_X(\vec{i}) + \vec{v}\right)^{(d)}\right| \leq \left|\Theta_X(\vec{i})^{(d)}\right|\right)\right)\right).$$

The algorithm operates on each band node's partial schedule and determines, per band node, suitable coefficients for the structure parameters. The resulting linear combination is the offset by which all statement instances in the band node's iteration domain will be shifted. If, after the transformation, all schedule coefficients of the schedule are zero, we can remove the band node from the schedule tree by pulling up its child node.

Correctness Let $\vec{p}$ be the vector of the SCoP's structure parameters and $M \in \mathbb{Z}^{\dim(\Theta)\times(\dim(\vec{p})+1)}$. Let $\vec{v} = M \cdot \begin{pmatrix}\vec{p}\\1\end{pmatrix}$. From the simplification step's definition the following holds for each $I_X \in I$:

$$\left(\forall \vec{i} \in I_X : \Theta'_X(\vec{i}) = \Theta_X(\vec{i}) + \vec{v}\right). \tag{5.3}$$

Adherence to Condition 5.1.

$$\begin{array}{lcl}
\Big(\forall I_X, I_Y \in I : (\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : & & \\
\Theta_X(\vec{i}_X) \prec \Theta_Y(\vec{i}_Y) & \Leftrightarrow & \triangleright \text{ add } \vec{v} \text{ on either side} \\
\Theta_X(\vec{i}_X) - \vec{v} \prec \Theta_Y(\vec{i}_Y) + \vec{v} & \Leftrightarrow & \triangleright \text{ Equation 5.3} \\
\Theta'_X(\vec{i}_X) \prec \Theta'_Y(\vec{i}_Y)\Big) & &
\end{array}$$

Adherence to Condition 5.2.

$$\begin{array}{lcl}
\Big(\forall I_X, I_Y \in I : (\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : (\forall k \in \{1, \ldots, \dim(\Theta)\} : (\vec{i}_Y \text{ depends on } \vec{i}_X) \Rightarrow (& & \\
\operatorname{sgn}(\Theta_Y^{(k)}(\vec{i}_Y) - \Theta_X^{(k)}(\vec{i}_X)) & = & \triangleright \text{ arithmetic} \\
\operatorname{sgn}(\Theta_Y^{(k)}(\vec{i}_Y) - \Theta_X^{(k)}(\vec{i}_X) + 0) & = & \triangleright \text{ arithmetic} \\
\operatorname{sgn}(\Theta_Y^{(k)}(\vec{i}_Y) - \Theta_X^{(k)}(\vec{i}_X) + (\vec{v}^{(k)} - v^{(k)})) & = & \triangleright \text{ associativity and distributivity} \\
\operatorname{sgn}(\Theta_Y^{(k)}(\vec{i}_Y) + \vec{v}^{(k)} - \Theta_X^{(k)}(\vec{i}_X) - \vec{v}^{(k)}) & = & \triangleright \text{ associativity} \\
\operatorname{sgn}((\Theta_Y^{(k)}(\vec{i}_Y) + \vec{v}^{(k)}) - (\Theta_X^{(k)}(\vec{i}_X) + \vec{v}^{(k)})) & = & \triangleright \text{ Equation 5.3} \\
\operatorname{sgn}(\Theta'^{(k)}_Y(\vec{i}_Y) - \Theta'^{(k)}_X(\vec{i}_X))))\Big) & &
\end{array}$$

5.4.2 Overly Large Schedule Coefficients

This simplification step reduces schedule coefficients' absolute values. Coefficients with overly large absolute values are impractical mainly for three reasons: they correlate with numerical overflows in index computations, extreme skewing, and the difficulty to choose a proper tile size. Figure 5.5 illustrates the interplay of loop strides and tile sizes. The larger the loop strides, the fewer statement instances are computed per tile for the same tile size.

Algorithm 5.2: Remove Statement's Common Offset from Band Nodes' Schedules (`RemoveCommonOffset`)

```
Input: A schedule (sub)-tree represented by its root node
Output: A simplified schedule (sub)-tree represented by its root node
Procedure ReduceOffsets(schedules)
    for $d = 1$ to $\dim(\varepsilon\Theta_X(\Theta_X \in \text{schedules}))$ do
        if IsCstDim($d$) $\vee$ IsParamDim($d$) then
            coeffs $\leftarrow \emptyset$
            foreach $\Theta_X \in$ schedules do
                coeffs $\leftarrow$ coeffs $\cup$ {getCoeff($\Theta_X$, $d$)}
            $\delta \leftarrow 0$
            if $\min(\text{coeffs}) < 0 \wedge \max(\text{coeffs}) < 0$ then
                $\delta \leftarrow -(\max \text{coeffs})$
            else if $\min(\text{coeffs}) > 0 \wedge \max(\text{coeffs}) > 0$ then
                $\delta \leftarrow -(\min \text{coeffs})$
            foreach $\Theta_X \in$ schedules do
                schedules $\leftarrow$ (schedules $\setminus \{\Theta_X\}) \cup$ {setCoeff($\Theta_X$, $d$, getCoeff($\Theta_X$, $d$) $+ \delta$)}
    return schedules
Procedure RemoveCommonOffset($n$)
    switch $n$.type do
        case BandNode do
            return BandNode(ReduceOffsets($n$.schedules), RemoveCommonOffset($n$.child))
        case SeqNode do
            return SeqNode ($n$.children.map(RemoveCommonOffset))
        case FilterNode do
            return FilterNode($n$.domain, RemoveCommonOffset($n$.child))
        case LeafNode do
            return $n$
```

$\varepsilon x(P(x))$ Selects a value x for which $P(x)$ is true [10].

`IsParamDim` : $\mathbb{N} \to \mathbb{B}$ Tests whether a given natural number is the index of a structure parameter's coefficient in a schedule's coefficient space.

`IsCstDim` : $\mathbb{N} \to \mathbb{B}$ Tests whether a given natural number is the index of a coefficient for the constant 1 in a schedule's coefficient space.

`getCoeff` Given a one-dimensional linearly affine function f and an index d, `getCoeff` returns the dth coefficient in f.

`setCoeff` Given a one-dimensional linearly affine function f, an index d, and $\alpha \in \mathbb{Z}$ `setCoeff` returns a copy of f with its dth coefficient set to α.

Multiplying rows of a schedule matrix by a strictly positive rational is legal [149]. For the reasons listed above, we seek coefficients in band nodes' partial schedules as close to zero as possible. To find a suitable factor for the simplification of a band node, Algorithm 5.3 determines the greatest common divisor (GCD) of the schedule coefficients per one-dimensional partial schedule. Dividing by the GCD guarantees that the resulting schedule coefficients are integers.

Similarly, polyhedral code generators modify loop strides to avoid conditional statements in loop bodies. For instance, Bastoul [17] determines the optimal loop stride as the GCD of the strides required by each of the statements in a loop's body and expressions that correspond to the required lower loop bound.

Correctness Given a schedule Θ, let us assume that we have rescaled $\Theta^{(k)}$ by the factor $1/\alpha_k, k \in \{1, ..., \dim(\Theta)\}, \alpha_k \in \mathbb{Z}$. Thus, we obtain

$$\Big(\forall I_X \in I : \big(\forall k \in \{1, ..., \dim(\Theta)\} : (\forall \vec{i} \in I_X : \Theta'^{(k)}(\vec{i}) = \frac{1}{\alpha_k} \cdot \Theta^{(k)}(\vec{i}))\big)\Big). \tag{5.4}$$

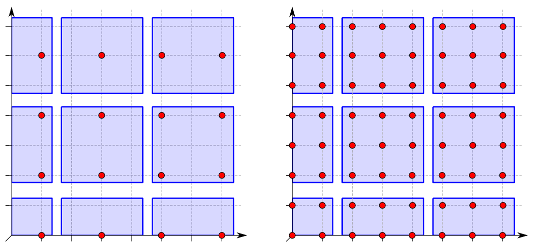


Figure 5.5: Both figures show a section of a tiled two-dimensional iteration domain. While the tile size is the same in both cases, the loop stride is bigger on the left side. This reduces the maximum number of statement instances computed per tile.

Algorithm 5.3: Divide Schedule Coefficients by Their GCD (`ReduceSchedCoeffs`)

```
Input: A schedule (sub)-tree represented by its root node
Output: A simplified schedule (sub)-tree represented by its root node
1  Procedure DivideByGCD(schedules)
2      foreach $\Theta_X \in$ schedules do
3          for $d = 1$ to $\dim(\Theta_X)$ do
4              coeffs ← coeffs ∪ {getCoeff($\Theta_X$, $d$)}
5      foreach $s \in$ schedules do
6          for $d = 1$ to $\dim(\Theta_X)$ do
7              schedules ← (schedules \ {$\Theta_X$}) ∪ {setCoeff($\Theta_X$, $d$, getCoeff($\Theta_X$, $d$) / gcd(coeffs))}
8      return schedules
9  Procedure ReduceSchedCoeffs($n$)
10     switch $n$.type do
11         case BandNode do
12             return BandNode(DivideByGCD($n$.schedules), ReduceSchedCoeffs($n$.child))
13         case SeqNode do
14             return SeqNode ($n$.children.map(ReduceSchedCoeffs))
15         case FilterNode do
16             return FilterNode($n$.domain, ReduceSchedCoeffs($n$.child))
17         case LeafNode do
18             return $n$
```

Adherence to Condition 5.1.

$$\Big(\forall I_X, I_Y \in I : \big(\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y :$$

$$\Theta_X(\vec{i}_X) \prec \Theta_Y(\vec{i}_Y)$$

$$\Leftrightarrow \qquad \triangleright \text{ definition of lexicographic ordering}$$

$$(\exists d \in \{1, \dots, \dim(\Theta)\} : (\forall d' \in 1, \dots, d-1 : \Theta_X^{(d')}(\vec{i}_X) = \Theta_Y^{(d')}(\vec{i}_Y)) \wedge \Theta_X^{(d')}(\vec{i}_X) < \Theta_Y^{(d')}(\vec{i}_Y))$$

$$\Leftrightarrow \qquad \triangleright \text{ linearity}$$

$$(\exists d \in \{1, \dots, \dim(\Theta)\} : (\forall d' \in 1, \dots, d-1 : \frac{1}{\alpha_{d'}} \cdot \Theta_X^{(d')}(\vec{i}_X) = \frac{1}{\alpha_{d'}} \cdot \Theta_Y^{(d')}(\vec{i}_Y)) \wedge \frac{1}{\alpha_d} \cdot \Theta_X^{(d)}(\vec{i}_X) < \frac{1}{\alpha_d} \cdot \Theta_Y^{(d)}(\vec{i}_Y))$$

$$\Leftrightarrow \qquad \triangleright \text{ Equation 5.4}$$

$$(\exists d \in \{1, \dots, \dim(\Theta)\} : (\forall d' \in 1, \dots, d-1 : \Theta'^{(d')}_X(\vec{i}_X) = \Theta'^{(d')}_Y(\vec{i}_Y)) \wedge \Theta'^{(d')}_X(\vec{i}_X) < \Theta'^{(d')}_Y(\vec{i}_Y))$$

$$\Leftrightarrow \qquad \triangleright \text{ definition of lexicographic ordering}$$

$$\Theta'_X(\vec{i}_X) \prec \Theta'_Y(\vec{i}_Y)\big)\Big)$$

Adherence to Condition 5.2.

$$
\begin{aligned}
\Big(\forall I_X, I_Y \in I : \big(\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : (\forall k \in \{1, \ldots, \dim(\Theta)\} : (\vec{i}_Y \text{ depends on } \vec{i}_X) \Rightarrow \big(\\
\operatorname{sgn}(\Theta_Y^{(k)}(\vec{i}_Y) - \Theta_X^{(k)}(\vec{i}_X)) \quad &= \quad \triangleright\ \tfrac{1}{\alpha_k} > 0 \\
\operatorname{sgn}\big(\tfrac{1}{\alpha_k} \cdot (\Theta_Y^{(k)}(\vec{i}_Y) - \Theta_X^{(k)}(\vec{i}_X))\big) \quad &= \quad \triangleright\ \text{distributivity} \\
\operatorname{sgn}\big(\tfrac{1}{\alpha_k} \cdot \Theta_Y^{(k)}(\vec{i}_Y) - \tfrac{1}{\alpha_k} \cdot \Theta_X^{(k)}(\vec{i}_X)\big) \quad &= \quad \triangleright\ \text{Equation 5.4} \\
\operatorname{sgn}\big(\Theta'^{(k)}_Y(\vec{i}_Y) - \Theta'^{(k)}_X(\vec{i}_X)\big)\big)\big)\big)\Big)
\end{aligned}
$$

5.4.3 Elimination of Superfluous Subtrees

Adding dimensions to an injective schedule does not change the schedule. Thus, given an n-dimensional schedule Θ, if we can determine $d < n$ such that $\Theta^{(1..d)}$ is injective, we may safely delete schedule dimensions $d+1, \ldots, n$. Analogously, Algorithm 5.4 replaces a schedule node and its children by a leaf node if the restriction of the schedule represented by the node's ancestors to the node's iteration domain is an injective function.

Algorithm 5.4: Remove Superfluous Subtrees (`RemoveSuperfluousSubtrees`)

```
Input:  A schedule (sub)-tree represented by its root node
Output:  A simplified schedule (sub)-tree represented by its root node
1  Procedure RemoveSuperfluousSubtrees($n$)
2      if ($n$.parentSchedule$_{|n.\text{domain}}$ is injective) ∧ $n$.type ≠ FilterNode then
3          return LeafNode($n$.domain)
4      switch $n$.type do
5          case BandNode do
6              return BandNode($n$.schedules, RemoveSuperfluousSubtrees($n$.child))
7          case SeqNode do
8              return SeqNode ($n$.children.map(RemoveSuperfluousSubtrees))
9          case FilterNode do
10             return FilterNode($n$.domain, RemoveSuperfluousSubtrees($n$.child))
11         case LeafNode do
12             return $n$
```

Correctness Let us assume

$$\Big(\forall I_X \in I : \Big(\forall \vec{i} \in I_X : \Theta'_X(\vec{i}) = \Theta_X^{(1..d)}(\vec{i})\Big)\Big) . \tag{5.5}$$

Consequently, the following must be true:

$$\Big(\forall I_X, I_Y \in I : \Big(\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : (X \neq Y \ \vee\ \vec{i}_X \neq \vec{i}_Y) \ \Rightarrow\ \Theta_X^{(1..d)}(\vec{i}_X) \neq \Theta_Y^{(1..d)}(\vec{i}_Y)\Big)\Big) . \tag{5.6}$$

Adherence to Condition 5.1.

$$
\begin{aligned}
\big(\forall I_X, I_Y \in I : (\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : \\
\Theta_X(\vec{i}_X) \prec \Theta_Y(\vec{i}_Y) \quad &\Leftrightarrow \quad \triangleright\ \text{Equation 5.6} \\
\Theta_X^{(1..d)}(\vec{i}_X) \prec \Theta_Y^{(1..d)}(\vec{i}_Y) \quad &\Leftrightarrow \quad \triangleright\ \text{Equation 5.5} \\
\Theta'_X(\vec{i}_X) \prec \Theta'_Y(\vec{i}_Y))\big)
\end{aligned}
$$

Adherence to Condition 5.2. We do not need to consider schedule dimensions $d+1, \ldots, \dim(\Theta)$ because they have been removed.

$$
\begin{aligned}
\Big(\forall I_X, I_Y \in I : \Big(\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : \big(\forall k \in \{1, \ldots, d\} : (\vec{i}_Y \text{ depends on } \vec{i}_X) \Rightarrow \big(\\
\operatorname{sgn}(\Theta_Y^{(k)}(\vec{i}_Y) - \Theta_X^{(k)}(\vec{i}_X)) \quad &= \quad \triangleright\ \text{Equation 5.5} \\
\operatorname{sgn}(\Theta'^{(k)}_Y(\vec{i}_Y) - \Theta'^{(k)}_X(\vec{i}_X))\big)\big)\Big)\Big)
\end{aligned}
$$

5.4.4 Elimination of Degenerate Loops

We call a loop *degenerate* if it performs only a single iteration. Obviously, such loops are unnecessary and should be eliminated. Their loop bodies must be left in place. Given a schedule Θ, we can test whether schedule dimension $d, 1 < d < \dim(\Theta)$, yields no other code than a degenerate loop by testing the following condition:

$$\Big(\forall I_X, I_Y \in I : \Big(\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : \big(\Theta_X^{(1..d-1)}(\vec{i}_X) = \Theta_Y^{(1..d-1)}(\vec{i}_Y)\big) \Rightarrow \big(\Theta_X^{(d)}(\vec{i}_X) = \Theta_Y^{(d)}(\vec{i}_Y)\big)\Big)\Big). \tag{5.7}$$

Any band node whose partial schedule adheres to Condition 5.7 can be deleted from a schedule tree by pulling up its child node. Algorithm 5.5 describes the procedure.

Algorithm 5.5: Eliminate Superfluous Band Nodes (`RemoveSuperfluousBandNodes`)

```
Input: A schedule (sub)-tree represented by its root node
Output: A simplified schedule (sub)-tree represented by its root node
1  Procedure RemoveSuperfluousBandNodes($n$)
2      if $(n.\mathsf{type} = \mathsf{BandNode}) \wedge (\forall I_X, I_Y \in n.\mathsf{domain} : (\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : (n.\mathsf{parentSchedule}_{|I_X}(\vec{i}_X) =$
         $n.\mathsf{parentSchedule}_{|I_Y}(\vec{i}_Y)) \Rightarrow (n.\mathsf{schedule}_{|I_X}(\vec{i}_X) = n.\mathsf{schedule}_{|I_Y}(\vec{i}_Y))))$ then
3          return RemoveSuperfluousBandNodes($n$.child)
4      switch $n$.type do
5          case BandNode do
6              return BandNode($n$.schedules, RemoveSuperfluousBandNodes($n$.child))
7          case SeqNode do
8              return SeqNode ($n$.children.map(RemoveSuperfluousBandNodes))
9          case FilterNode do
10             return FilterNode($n$.domain, RemoveSuperfluousBandNodes($n$.child))
11         case LeafNode do
12             return $n$
```

Correctness Let us assume that, after the transformation of schedule Θ by Algorithm 5.5, schedule dimension k has become dimension k' of Θ'. Then, we know the following about $\Theta'^{(k'+1)}$ for the statement instances in I:

$$\Big(\forall I_X \in I : \Big(\forall \vec{i} \in I_X :
\Theta'^{(k'+1)}_X = \begin{cases} \Theta_X^{(k+2)}(\vec{i}) & \text{if } \big((k+2) \leq \dim(\Theta_X)\big) \\ & \quad \wedge \big(\forall I_X, I_Y \in I : (\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : \big(\Theta_X^{(1..k)}(\vec{i}_X) = \Theta_Y^{(1..k)}(\vec{i}_Y)\big) \\ & \qquad \Rightarrow \big(\Theta_X^{(k+1)}(\vec{i}_X) = \Theta_Y^{(k+1)}(\vec{i}_Y)\big)\big)\big) \\ \Theta_X^{(k+1)}(\vec{i}) & \text{if } \big(\exists I_X, I_Y \in I : \big(\exists(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : \\ & \quad \Theta_X^{(1..k)}(\vec{i}_X) = \Theta_Y^{(1..k)}(\vec{i}_Y) \ \wedge \ \Theta_X^{k+1}(\vec{i}_X) \neq \Theta_Y^{k+1}(\vec{i}_Y)\big)\big) \\ \perp & \text{if } \Theta^{(k+1)} \text{ has been eliminated and } \Theta^{(k+2)} \text{ does not exist.} \end{cases}
\Big)\Big) \tag{5.8}$$

The first case of Equation 5.8 corresponds to the case that Θ^{k+1} has been eliminated by Algorithm 5.5. In the second case, Θ^{k+1} has not been eliminated and we set $\Theta'^{k'+1} = \Theta^{k+1}$. In the third case, we eliminated the innermost dimension of Θ and $\Theta'^{(k')}$ equals the last dimension of Θ that has not been eliminated. Thus, dimension $k'+1$ of Θ' does not exist.

Adherence to Condition 5.1.

$$\begin{aligned}
&\Big(\forall I_X, I_Y \in I : \big(\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : \\
&\Theta_X(\vec{i}_X) \prec \Theta_Y(\vec{i}_Y) \\
&\Leftrightarrow \quad \triangleright \text{ definition of lexicographic ordering} \\
&(\exists d \in \{1, \ldots, \dim(\Theta)\} : (\forall d' \in 1, \ldots, d-1 : \Theta_X^{(d')}(\vec{i}_X) = \Theta_Y^{(d')}(\vec{i}_Y)) \wedge \Theta_X^{(d')}(\vec{i}_X) < \Theta_Y^{(d')}(\vec{i}_Y)) \\
&\Leftrightarrow \quad \triangleright \text{ From Equation 5.8 schedule dimension } d \text{ cannot have been eliminated and the schedule dimensions are not shuffled.} \\
&(\exists d \in \{1, \ldots, \dim(\Theta')\} : (\forall d' \in 1, \ldots, d-1 : \Theta'^{(d')}_X(\vec{i}_X) = \Theta'^{(d')}_Y(\vec{i}_Y)) \wedge \Theta'^{(d')}_X(\vec{i}_X) < \Theta'^{(d')}_Y(\vec{i}_Y)) \\
&\Leftrightarrow \quad \triangleright \text{ definition of lexicographic ordering} \\
&\Theta'_X(\vec{i}_X) \prec \Theta'_Y(\vec{i}_Y))\Big)
\end{aligned}$$

Adherence to Condition 5.2.

$$\begin{aligned}
&\Big(\forall I_X, I_Y \in I : \Big(\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : (\vec{i}_Y \text{ depends on } \vec{i}_X) \\
&\Rightarrow \Big(\forall k \in \{1, \ldots, \dim(\Theta)\} : \\
&\big(\exists k' \in \{1, \ldots, \dim(\Theta')\} : \Theta^{(k)} \text{has become } \Theta'^{(k')} \\
&\Rightarrow \quad \triangleright \text{ we can ignore eliminated schedule dimensions} \\
&\big(\operatorname{sgn}(\Theta_Y^{(k)}(\vec{i}_Y) - \Theta_X^{(k)}(\vec{i}_X)) \\
&= \quad \triangleright \text{ From Equation 5.8, } \Theta^{(k)} \text{ and } \Theta'^{(k')} \text{ are equal} \\
&\operatorname{sgn}(\Theta'^{(k')}_Y(\vec{i}_Y) - \Theta'^{(k')}_X(\vec{i}_X))))\Big)\Big)\Big)
\end{aligned}$$

5.4.5 Further Normalization of Band Nodes

The band nodes' one-dimensional partial schedules can be simplified beyond the removal of statements' common constant offset in the schedule's codomain (refer to Section 5.4.1) and the division of the schedule coefficients by their GCD (refer to Section 5.4.2). Given a band nodes one-dimensional partial schedule $\Theta^{(k)}, k \in \{1, \ldots, \dim(\Theta)\}$, we can compute the set of all schedules that are equivalent in the sense of Conditions 5.1 and 5.2:

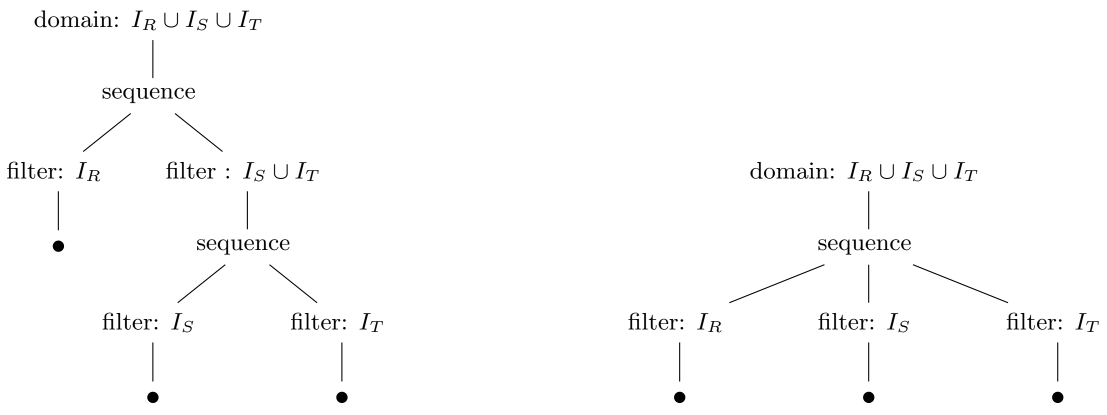
$$\begin{aligned}
\Big\{\Theta^* \mid \Big(&\forall I_X, I_Y \in I : \Theta_X^* : I_X \to \mathbb{Z} \ \wedge \ \Theta_Y^* : I_Y \to \mathbb{Z} \\
\wedge \Big(&\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : \big((\Theta_X^{(k)}(\vec{i}_X) = \Theta_Y^{(k)}(\vec{i}_Y)) \Rightarrow (\Theta_X^*(\vec{i}_X) = \Theta_Y^*(\vec{i}_Y))\big) \\
&\wedge \big((\Theta_X^{(k)}(\vec{i}_X) > \Theta_Y^{(k)}(\vec{i}_Y)) \Rightarrow (\Theta_X^*(\vec{i}_X) > \Theta_Y^*(\vec{i}_Y))\big)\Big)\Big)\Big\}
\end{aligned}$$

We use a technique proposed by Bondhugula et al. [28] to select the vector with minimal absolute values of its components from a $\mathbb{Z}$-polyhedron and Farka's Lemma. Thereby, we can map all equivalent one-dimensional schedules to a canonical representation.

By empirical evaluation, we found that this simplification is computationally too expensive for practical use.

5.4.6 Collapsing Cascades of Sequence Nodes

A simplification that is not implemented in Polyite, which internally uses its own representation of schedule trees, is the collapsing of cascades of sequence nodes. This is a transformation that isl [152] performs during the construction of an isl schedule tree. Figure 5.6(a) shows a schedule tree that consists essentially of a cascade of sequence nodes. The schedule tree's inner nodes can be combined to a single sequence node whose children are filter nodes to which the original leaf nodes are attached. Figure 5.6(b) shows the simplification's result.



(a) the original schedule tree

(b) the simplified schedule tree

Figure 5.6: Cascades of sequence nodes that can be collapsed. Figure 5.6(a) shows a schedule tree consists of a cascade of sequence nodes. Figure 5.6(b) shows an equivalent simplified schedule tree.

5.5 Schedule Tree Analysis

Based on the transformation of schedule matrices to schedule trees, we present techniques to expose fundamental structural properties of schedules. We adjust schedule trees to persist the information gathered. The structures that we identify are parts of band nodes' partial schedules that correspond to loops in the generated code, permutable bands, which allow for tiling, and band nodes whose partial schedules do not carry legality-affecting data dependences and therefore are parallel. We demonstrate that our detection of tilable bands is heuristic. This kind of information is relevant for the preparation of schedule trees for further transformation, such as tiling or strip mining, and for the characterization of schedules without the need to generate an AST by polyhedral code generation.

We conclude the section by recalling the identification of equivalent schedules.

5.5.1 Detection of Loop-Generating Schedule Dimensions

Danner [46] enabled Polyite to identify dimensions of a statement's schedule that correspond to a loop that encases the statement in the transformed program. Given a statement S and its n-dimensional schedule Θ_S, recall that $\Theta_S : I_S \to \mathbb{Z}^n, n \in \mathbb{N}$, can be represented as follows:

$$\Theta_S(\vec{i}) = \begin{pmatrix} \vec{\lambda}_S^1 & \vec{\mu}_S^1 & \nu_S^1 \\ \vec{\lambda}_S^2 & \vec{\mu}_S^2 & \nu_S^2 \\ \vdots & \vdots & \vdots \\ \vec{\lambda}_S^n & \vec{\mu}_S^n & \nu_S^n \end{pmatrix} \cdot \begin{pmatrix} \vec{i} \\ \vec{p} \\ 1 \end{pmatrix}.$$

$\Theta_S^{(d)}, d \in \{1, \ldots, n\}$ corresponds to a loop iff $\vec{\lambda}_S^d \neq \vec{0} \ \wedge \ \vec{\lambda}_S^d \in \text{lin.indep}(\{\vec{\lambda}_S^1, \vec{\lambda}_S^2, \ldots, \vec{\lambda}_S^{d-1}\})$. By definition, only band nodes can contain partial schedules that correspond to loops. Danner extended the schedule tree data type to express, per dimension of a band node's partial schedule and statement S, whether the schedule dimension corresponds to a loop that encases S.

5.5.2 Detection of Permutable Schedule Bands

To be able to tile loop nests of the transformed program, we must identify contiguous sequences of band nodes in schedule trees whose partial schedules' are permutable. We must then replace any such sequence by a single band node whose partial schedule represents the schedule encoded by the former sequence of band nodes. We must further mark the new band node as permutable. Thereby, the information that the band node's schedule can be tiled becomes persistent. A contiguous sequence of band nodes $n_1, n_2, \dots, n_m$ storing the one-dimensional partial schedules $\Theta^{(d)}, \Theta^{(d+1)}, \dots, \Theta^{(d+m-1)}$ for the statement iteration domains in a set I can be merged into a single permutable band node if the following condition holds:

$$\Big(\forall D_{O,T} \in G : \Big((I_O, I_T \in I) \ \wedge\ \big(\forall e \in \{1, \dots, d-1\} : \Theta^{(e)} \text{ does not carry } D_{O,T}\big)\Big) \\ \Rightarrow \Big(\forall e \in \{d, \dots, d+m-1\} : \Theta^{(e)} \text{ weakly satisfies } D_{O,T}\Big)\Big).$$

G is the set of a SCoP's legality-affecting dependence polyhedra as defined in Chapter 4. Starting from any band node that is a child of a schedule tree's domain node or a child of a filter node, we grow sequences of band nodes of maximal length for which the above condition holds.

This identification of tilable bands is heuristic for two reasons. First, we treat dependences at the granularity level of dependence polyhedra. It may not be necessary to consider the dependences in $D \subseteq D_{O,T}, D_{O,T} \in G$ if $\exists d' \in \mathbb{N} : (1 \le d' < d) \ \wedge\ \Theta^{d'}$ carries D. Second, with this technique, we may detect a sequence of band nodes $n_1, n_2, \dots, n_m$ that can be grouped, whereas we could have detected $n_2, \dots, n_{m+o}$ with $o > 1$. This situation arises if a dependence polyhedron $D_{O,T}$ exists that is weakly satisfied by the partial schedules of $n_1, \dots, n_m$, but not all of the partial schedules of $n_{m+1}, \dots, n_{m+o}$ and the partial schedules of $n_{m+1}, \dots, n_{m+o}$ weakly satisfy all dependence polyhedra that are weakly satisfied by n_2.

While the suggested improvements increase the applicability of tiling and thereby increase data locality, they also increase the computational cost of identifying tilable bands.

5.5.3 Detection of Parallelism

For the detection of parallelism, we must process dependences at the statement instance level. From Section 2.2.2.4, we know that dimension $\Theta^{(d)}$ of a band node's partial schedule is parallel if all legality-affecting dependences that are uncarried by $\Theta^{(1..d-1)}$ are orthogonal to $\Theta^{(d)}$. Θ refers to the schedule of the band node's iteration domain. Together with the identification of parts of schedules that correspond to loops, this detection of parallelism allows for the identification of parallel loops without the necessity to generate an AST.

Example 5.5.1. Figure 5.7 shows the schedule tree that results from Example 5.3.1 before and after the tree's simplification and the identification of tilable bands and parallelism. $\triangleleft$

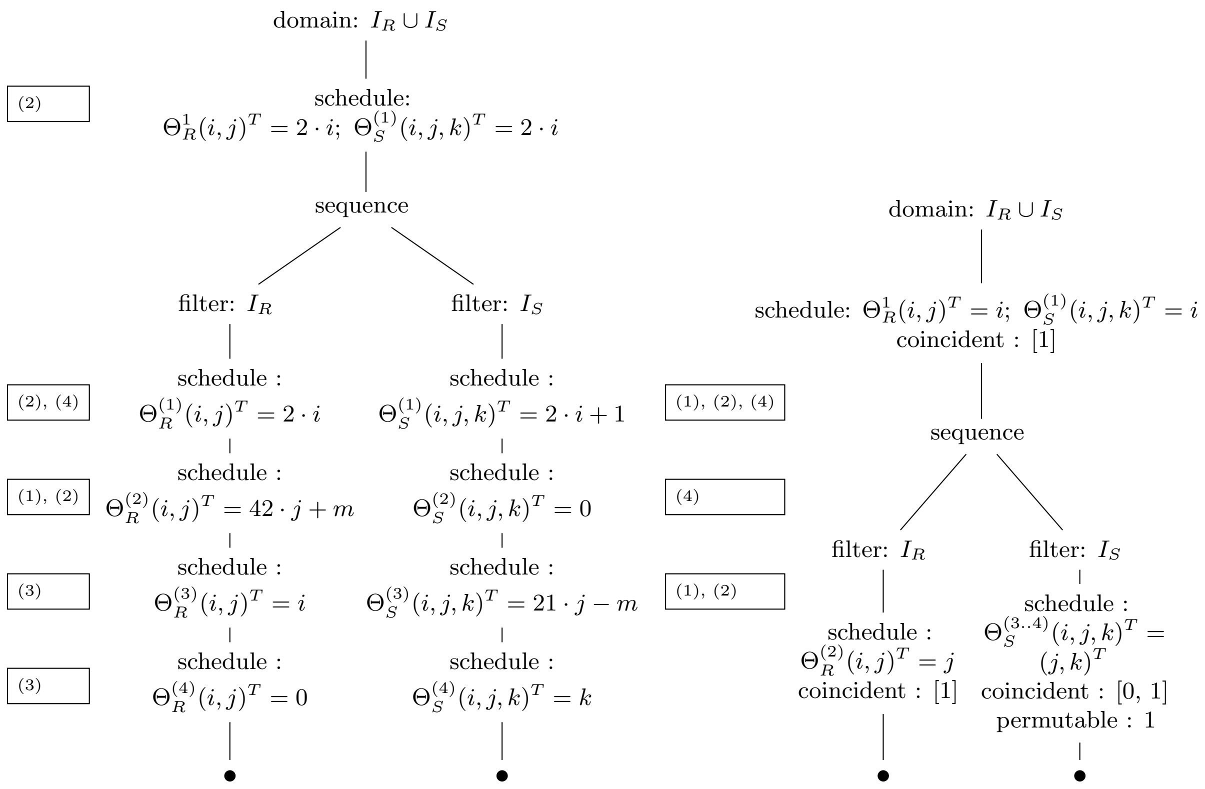
5.5.4 Detecting Equivalence Classes

For the analysis of a schedule space exploration technique or to prune schedules from a set of schedules that are redundant in that they yield the same transformed code as another schedule, it may be relevant to identify equivalence classes of schedules. Following Vasilache [149], two schedules Θ, Θ' for a SCoP S with a set of statement iteration domains I are equivalent if they execute any pair of statement instances of S in the same order:

$$\Big(\forall I_X, I_Y \in I : \Big(\forall (\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : \big(\Theta(\vec{i}_X) \prec \Theta(\vec{i}_Y)\big) \Leftrightarrow \big(\Theta'(\vec{i}_X) \prec \Theta'(\vec{i}_Y)\big)\Big)\Big). \qquad (5.9)$$

From Example 5.4.1, we know that two schedules that are equivalent according to Equation 5.9 may no longer be equivalent after the detection of tilable bands in these schedules and the application of tiling to the schedules. To decide whether two schedules yield the same execution order of the statement instances in the transformed program, we must apply all schedule (tree) transformations (such as simplification, tiling, and strip mining) and then test the equivalence of their outcomes according to the condition in Equation 5.9.

(1)	Removal of statements' common offset
(2)	Reduction of overly large schedule coefficients
(3)	Elimination of superfluous subtrees
(4)	Elimination of degenerate loops


domain: $I_R \cup I_S$
(2)
schedule: $\Theta_R^1(i,j)^T = 2 \cdot i$; $\Theta_S^{(1)}(i,j,k)^T = 2 \cdot i$
sequence
filter: I_R
filter: I_S
(2), (4)
schedule : $\Theta_R^{(1)}(i,j)^T = 2 \cdot i$
schedule : $\Theta_S^{(1)}(i,j,k)^T = 2 \cdot i + 1$
(1), (2), (4)
(1), (2)
schedule : $\Theta_R^{(2)}(i,j)^T = 42 \cdot j + m$
schedule : $\Theta_S^{(2)}(i,j,k)^T = 0$
(4)
(3)
schedule : $\Theta_R^{(3)}(i,j)^T = i$
schedule : $\Theta_S^{(3)}(i,j,k)^T = 21 \cdot j - m$
(1), (2)
(3)
schedule : $\Theta_R^{(4)}(i,j)^T = 0$
schedule : $\Theta_S^{(4)}(i,j,k)^T = k$
domain: $I_R \cup I_S$
schedule: $\Theta_R^1(i,j)^T = i$; $\Theta_S^{(1)}(i,j,k)^T = i$
coincident : [1]
sequence
filter: I_R
filter: I_S
schedule : $\Theta_R^{(2)}(i,j)^T = j$
coincident : [1]
schedule : $\Theta_S^{(3..4)}(i,j,k)^T = (j,k)^T$
coincident : [0, 1]
permutable : 1

(a) The unsimplified schedule tree. Its nodes are annotated by references to the simplification steps by which they are modified during the tree's simplification.

(b) The simplified schedule tree after the identification of permutable bands and parallel schedule dimensions.

Figure 5.7: The schedule tree that results from Example 5.3.1 before and after its simplification and the identification of tilable bands and parallel (coincident) dimensions of partial schedules.

5.6 Discussion

Listing 5.1: Program with partially fused loop nests.

```
for (int i = 0; i < n; i += 1) {
  for (int j = 0; j < m; j += 1)
    #pragma omp parallel for
    for (int k = 0; k < n; k += 1) {
      if (n >= j + 1)
        S(i, j, k);
      T(i, j, k);
    }
  #pragma omp parallel for
  for (int j = max(0, m); j < n; j += 1)
    for (int k = 0; k < n; k += 1)
      S(i, j, k);
}
```

In this section, we presented a transformation of a schedule matrix to a schedule tree, and a subsequent schedule tree simplification. We can modify the simplified schedule trees and attribute their nodes in order to expose information regarding tilable bands and parallelism. Extraction and particularly persistence of this information would have been less straightforward on the original schedule matrix. Further, we can identify the parts of a schedule that encode the loops in the transformed program.

The transformation of a schedule matrix to a schedule tree is heuristic as it recognizes the textual order only of entire statement iteration domains, but not of subsets of iteration domains. Example 5.1 illustrates that this can affect the outcome of a schedule tree's analysis.

Example 5.6.1. Let S, T be two statements with iteration domains

$$I_S = \{(i,j,k)^T \mid i,j,k \in \mathbb{N} \ \wedge \ 0 \leq i,j,k < n\}$$
$$I_T = \{(i,j,k)^T \mid i,j,k \in \mathbb{N} \ \wedge \ 0 \leq i,k < n \ \wedge \ 0 \leq j < m\}$$

and schedules

$$\Theta_S(i,j,k)^T = (i,j,k)^T, \quad \Theta_T(i,j,k)^T = (i,j,k)^T.$$

Let $n, m \in \mathbb{N}$. Consider the following two dependence polyhedra

$$D_{T,T} = \{(i,j,k,i,j+1,k)^T \mid (i,j,k)^T \in I_T \ \wedge \ (i,j+1,k)^T \in I_T\}$$
$$D'_{T,T} = \{(i,j,k,i+1,j,k)^T \mid (i,j,k)^T \in I_T \ \wedge \ (i+1,j,k)^T \in I_T\}$$

that we consider to be legality-affecting. Listing 5.1 shows the corresponding program. The i-loop, which encases S and T, cannot be parallelized because it carries $D'_{T,T}$. The j-loop is only fused partially since it iterates from 0 to $m-1$ for T and from 0 to $n-1$ for S. The second of the two loop nests encased by the i-loop performs the remaining iterations of S in the case that $n > m$. In the first of the two loop nests, the j-loop must be executed sequentially because it carries $D_{T,T}$. In the second loop nest, the j-loop can be executed in parallel. With the technique described in this chapter, we cannot recognize this parallelism because, if we modeled the textual order of the two loop nests with a sequence node, S would occur in both of the sequence node's subtrees. While a more sophisticated schedule tree construction would allow us to identify the parallelism in the second loop nest, using this information would be difficult as its relevance depends on the, generally unavailable, information that $n > m$. Moreover, a more finely grained representation of textual order in schedule trees would reduce the applicability of tiling in the present case: without inserting a sequence node underneath the band node that encodes the i-loop, we can fully tile this imperfect loop nest. In the presence of the sequence node, the outermost i-loop could not be part of a tilable band. Note that polyhedral compilers, such as POLLY, will still be able to detect all potential parallelism. $\triangleleft$

If SCoP extraction could provide useful information regarding structure parameters, such as strict positiveness, we could use this information to improve the recall of the identification of textual ordering of statements according to $<^k$ (refer to Equation 5.3). Let two statement instances be given. If one instance's execution step depends on a structure parameter's value and we cannot tell whether the parameter is strictly positive or negative, we may fail to decide which of the two instances will be executed first.

Unfortunately, the version of POLLY that we used for the evaluation does not provide such information to POLYITE.

Moreover, the heuristic nature of Algorithm 5.1 can prevent the recognition of two statement's textual order. The algorithm can be improved by replacing Procedure `Partition` on line 2 by the procedure shown in Algorithm 5.6.

Algorithm 5.6: An Improved Procedure `Partition` for Algorithm 5.1

```
Procedure Partition($J$, $\otimes$)                    ▷ Set of iteration domains, order predicate
    return $\langle P_1, ..., P_l \subseteq J \mid (\forall i,j \in \{1,...,l\} : (i<j) \Rightarrow (P_i \cap P_j = \emptyset \ \wedge \ (\forall (I_X, I_Y) \in P_i \times P_j : I_X \otimes I_Y))) \wedge (\forall i \in \{1,...,l\} : \neg(\exists Q \subset P_i : Q \neq \emptyset \wedge (\forall I_X \in Q : (\forall I_Y \in (P_i \setminus Q) : I_X \otimes I_Y))))\rangle$
```

We proposed this improved algorithm for schedule tree construction earlier [62], but it deviated from the implementation.

To this end, we have recalled that a more sophisticated identification of textual order is possible and that such a representation would enhance the analysis of schedules. Beyond the improvement that results from the use of Algorithm 5.6, a realization would be analogous to the identification of textual ordering by polyhedral code generators. Also, more information regarding structure parameters would have to be taken into account. Unfortunately, we must assume that such information is unavailable in general. We have decided to trade the possibility of improved schedule analysis for the better optimization of schedule trees constructed heuristically in situations like Example 5.1.

When preparing schedules for code generation, one of the conditions for textual order should be disabled. Otherwise, the possibility to find schedules that enable tiling of imperfect loop nests would be reduced. We decided to disable the detection of shifted strides ($\tilde{<}^k$). A preparation of schedule trees that determines the applicability of tiling must reflect this choice. For other analyses, like the detection of parallelism, the more finely grained identification of textual order is preferable. Alternatively, we could detect textual whenever this is possible, and re-convert parts of a schedule tree underneath a sequence node that has been yielded by $\tilde{<}^k$ to a band node after simplification. This would allow for a stronger simplification of the schedule trees and we would potentially allow for an even more frequent applicability of tiling to imperfect loop nests.

As we pointed out in the introduction of Section 5.4, the schedule tree simplification presented preserves the semantics of a schedule before the application of tiling. Moreover, it preserves, or even enlarges sequences of schedule dimensions that encode loop nests that are tilable. If one encodes tiling into a schedule tree before and after the tree's simplification, the outcome may differ, even if the simplification did not result in an enlarged tilable band. Schedule dimensions that have no effect on the corresponding transformed code without tiling can become influential after tiling. In some cases, this effect depends on the tile sizes chosen. Moreover, an analysis of a schedule before tiling may yield different results than the analysis of the same schedule after tiling. Again, the specific effect may dependent on the choice of the tile sizes. We present three exemplary cases in Examples 5.6.2, 5.6.3, and 5.6.4. In all three examples, we assume that the entire schedule is stored in one permutable band.

In Example 5.6.2 we illustrates a case in which two statements change their textual order after the application of tiling. We do not account for such a situation in the analysis of the schedule tree that represents the transformed program before the application of tiling.

Example 5.6.3 shows a case in which two statements are distributed before the application of tiling but, after tiling, the execution of the tiles is interleaved.

Finally, we illustrate in Example 5.6.4, how tiling can change the schedule dimension that corresponds to a loop and how it can reverse the direction of a loop.

For the following examples, the computation of the sample programs is irrelevant. Therefore, we only sketch statements using a notation similar to a function call in our code snippets.

Example 5.6.2. Let us assume two statements X, Y with iteration domains

$$I_X = \{(i,j)^T : i,j \in \mathbb{Z} \ \land \ 0 \leq i < n \ \land \ 0 \leq j < n\},\ I_Y = \{i : i \in \mathbb{Z} \ \land \ 0 \leq i < n\}, n \in \mathbb{N}$$

and schedules

$$\Theta_X(i,j)^T = (i,0,j,0)^T,\ \Theta_Y(i) = (i,1,0,-1)^T.$$

The corresponding code is shown in Listing 5.2. In an iteration of the outermost loop, Y will be executed after X.

If we tile the schedule using tile size 32, we obtain the code shown in Listing 5.3. The iterations of Y are restricted to the first iteration of the inner tile loop by a guard and they are being computed before the iterations of X. The reason is that in the new second

Listing 5.2: The program from Example 5.6.2 before tiling.

```
for (int c0 = 0; c0 < n; c0 += 1) {
  for (int c2 = 0; c2 < n; c2 += 1)
    X(c0, c2);
  Y(c0);
}
```

Listing 5.3: The program from Example 5.6.2 after tiling. The statements' textual order has changed.

```
for (int c0 = 0; c0 < n; c0 += 32)
  for (int c2 = 0; c2 < n; c2 += 32) {
    if (c2 == 0)
      for (int c4 = 0; c4 <= min(31, n − c0 − 1); c4 += 1)
        Y(c0 + c4);
    for (int c4 = 0; c4 <= min(31, n − c0 − 1); c4 += 1)
      for (int c6 = 0; c6 <= min(31, n − c2 − 1); c6 += 1)
        X(c0 + c4, c2 + c6);
  }
```

Listing 5.4: The program from Example 5.6.3 before tiling.

```
for (int c1 = 0; c1 < n; c1 += 1)
  for (int c2 = 0; c2 < n; c2 += 1)
    X(c1, c2);
for (int c1 = 0; c1 < n; c1 += 1)
  for (int c2 = 0; c2 < n; c2 += 1)
    Y(c1, c2);
```

Listing 5.5: The program from Example 5.6.3 after tiling. While the statements were fully distributed before tiling, the statements' tiles are executed in an alternating and, thus, fused pattern after tiling.

```
for (int c1 = 0; c1 < n; c1 += 32)
  for (int c2 = 0; c2 < n; c2 += 32) {
    for (int c4 = 0; c4 <= min(31, n − c1 − 1); c4 += 1)
      for (int c5 = 0; c5 <= min(31, n − c2 − 1); c5 += 1)
        X(c1 + c4, c2 + c5);
    for (int c4 = 0; c4 <= min(31, n − c1 − 1); c4 += 1)
      for (int c5 = 0; c5 <= min(31, n − c2 − 1); c5 += 1)
        Y(c1 + c4, c2 + c5);
  }
```

dimension of the tiled schedule, which corresponds to $\Theta^{(2)}$ both statements are assigned to execution step 0. Schedule dimension 4 has no effect on the execution order before tiling and, in consequence, it would be eliminated by schedule simplification. ◁

Example 5.6.3. Consider the statements X, Y with iteration domains

$$I_X = \{(i,j)^T : i,j \in \mathbb{Z} \ \wedge \ 0 \le i < n \ \wedge \ 0 \le j < n\}$$
$$I_Y = \{(i,j)^T : i,j \in \mathbb{Z} \ \wedge \ 0 \le i < n \ \wedge \ 0 \le j < n\}, n \in \mathbb{N}$$

and schedules

$$\Theta_X(i,j)^T = (0,i,j)^T,\ \Theta_Y(i,j)^T = (1,i,j)^T.$$

Listing 5.4 shows the corresponding code before the application of tiling. The statements are fully distributed. After tiling (refer to Listing 5.5) the execution of the tiles is fused and the statements are distributed only inside each tile. ◁

Example 5.6.4. Consider statements X, Y with iteration domains

$$I_X = \{(i,j)^T : i,j \in \mathbb{Z} \ \wedge \ 0 \le i < n \ \wedge \ 0 \le j < n\},\ I_Y = \{i : i \in \mathbb{Z} \ \wedge \ 0 \le i < n\}, n \in \mathbb{N}$$

and schedules

$$\Theta_X(i,j)^T = (0,i,j)^T,\ \Theta_Y(i) = (4,i,-4 \cdot i)^T.$$

◁

Listing 5.6 shows the code of the example before tiling and Listing 5.7 shows the code after tiling with tile size 4. The innermost loop that encases statement Y is reversed after tiling. The reason is that, while $\Theta_Y^{(4)}$, which contains a negative coefficient for iteration variable i, had no influence on the execution order before tiling, it corresponds to the innermost loop that encases Y after tiling.

Listing 5.6: The program from Example 5.6.4 before tiling.

```
for (int c1 = 0; c1 < n; c1 += 1)
  for (int c2 = 0; c2 < n; c2 += 1)
    X(c1, c2);
for (int c1 = 0; c1 < n; c1 += 1)
  Y(c1);
```

Listing 5.7: The program from Example 5.6.4 after tiling. After tiling, the innermost loop that encases statement Y is reversed.

```
for (int c1 = 0; c1 < n; c1 += 4)
  for (int c2 = 0; c2 < n; c2 += 4)
    for (int c4 = 0; c4 <= min(3, n − c1 − 1); c4 += 1)
      for (int c5 = 0; c5 <= min(3, n − c2 − 1); c5 += 1)
        X(c1 + c4, c2 + c5);
for (int c1 = 0; c1 < n; c1 += 4)
  for (int c2 = max(−n + 1, −c1 − 3); c2 <= −c1; c2 += 1)
    Y(−c2);
```

Despite these differences, it may be preferable to analyze schedules before tiling because the analyses' computational cost and algorithmic complexity are reduced. One must weigh up (run-time) complexity and analytical precision.

6 Validity-Preserving Genetic Operators

In Chapter 4, we proposed a technique to sample randomly schedules from the set of legal schedules for a given SCoP. It is possible to bias the search such that some subsets of the search space are explored more thoroughly than others. Still, random sampling produces schedules independent of each other and does not take knowledge that it could have gathered from previously evaluated schedules into account. A genetic algorithm (refer to Section 2.3.1) improves the search with respect to this aspect: it evaluates its current population of candidate solutions and derives new solutions from the existing ones. The chance for a schedule to replicate by being mutated or being crossed with another schedule depends on its fitness, which, in our case, is the yielded speedup in execution time. Thereby, a genetic algorithm performs a directed random walk and, for the same number of evaluated schedules, a genetic algorithm may find a better schedule in terms of yielded speedup in execution time than random exploration.

Pouchet et al. [124] proposed a genetic algorithm with custom genetic operators under which their search space is closed. Their results show that their genetic algorithm converges to a higher speedup in execution time of the transformed program than their random exploration. Pouchet's random exploration can be described as lexicographic enumeration of a subset of the schedule coefficient vector space's dimensions combined with pruning and completion of the missing coefficients and schedule matrix rows while preserving legality. They describe their random exploration as a uniform traversal of the search space, while the genetic algorithm is designed to perform a non-uniform traversal.

Pouchet et al. [124] demonstrated that their schedule search space has little locality in a sense that a very small change, like replacing a single coefficient of an already profitable schedule, is unlikely to produce a better schedule. It is more likely that the resulting schedule yields a similar or even worse speedup. They point out that this makes the non-uniformity of an efficient search space traversal particularly important.

While the random exploration that we propose performs a non-uniform exploration of the search space itself, it is still relevant to investigate whether a genetic algorithm can outperform random exploration in terms of speedup of the transformed program and also duration of the exploration. The supposed advantage of a genetic algorithm over a non-uniform random exploration lies in its ability to switch to a more finely grained search in later generations after a start from a diverse set of schedules as its initial population and strong mutations of the schedules during early generations.

This chapter describes the schema of our a genetic algorithm for polyhedral schedule optimization, which is similar to that of Pouchet et al. (refer to Section 6.1). We continue by presenting novel genetic operators that have the capability to traverse our wider schedule search space while preserving legality (refer to Section 6.2). To retain the diversity of the genetic algorithm's population, one must avoid introducing duplicate schedules into the population. The equivalence of schedules can be defined in many ways. We discuss some alternatives in Section 6.3. In Section 6.4, we give an outlook at a distributed genetic algorithm for polyhedral schedule optimization, which Felix Bernasch proposed in his Bachelor thesis [23].

6.1 A Genetic Algorithm for Polyhedral Schedule Optimization

The schema of our genetic algorithm reflects mostly that of Pouchet et al. [124]. We start from a set of schedules generated randomly. To ensure strong diversity in that set, we call Algorithm 4.2 (sampling of search space regions) repeatedly and, thereby, obtain the representations of search space regions chosen randomly. Using Algorithm 4.3 (sampling of schedules from search space regions) and one of the sampling techniques presented in Section 4.6, we sample a small number of schedule matrices from each of the search space regions (at most two per region). We add a schedule matrix to the population if it is not equivalent to any of the already present schedule matrices according to the chosen schedule equivalence relation (refer to Section 6.3).

We determine the fitness of the schedules in the current population by applying each schedule to the SCoP, generating code, and measuring the generated code's execution time. Next, we select the population's fitter half and use it as a basis for the next population. The retention of the best schedules known is commonly referred to as elitism [12]. To reach full population size, we create mutations and crossings of the schedules in the basis. In each iteration of this reproduction loop, a mutation and a crossover can occur with the same probability. After making this choice, we must choose a particular mutation operator or crossover operator, respectively. Again, the choice of the particular mutation or crossover operator is uniform. We extend the schema by adding a very small number of schedules generated randomly. This decreases the probability to become trapped at local optima.

We also adapt Pouchet's idea to combine the genetic algorithm with simulated annealing [42, 68]. At the start of the optimization process, one mutates the schedules strongly and gradually reduces the size of the changes as the maximum reachable speedup is approached. Thereby, in our case, we cover many different search space regions at the beginning and localize the search when the maximum speedup yielded by the schedules evaluated starts to converge. Algorithm 6.1 illustrates the procedure.

Algorithm 6.1: Schema of POLYITE's Genetic Algorithm for Schedule Optimization

```
Input:  S: The SCoP to be optimized
        n ∈ ℕ: The regular population size
        r ∈ [0, 1]: The share of randomly generated schedules to be added to each generation
             of the population
Output: The final population: A set of schedules together with the results of their fitness
        evaluation.
1  g ← 0 ;                                                  ▷ Index of the current generation
2  P ← sampleRandomly(n, S) ;      ▷ Sample randomly n schedules using Algorithms 4.2 and 4.3
3  P ← evaluate(P)
4  while ¬terminate(g, P) do
5      B ← select(P)
6      P ← B
7      P ← P ∪ sampleRandomly(⌊r · n⌋, S)
8      P ← P ∪ reproduce(B, |P|, n)
9      P ← evaluate(P)
10     g ← g + 1
11 return P
```

Function `evaluate` labels schedules with their fitness.
Function `select` elects schedules that form the next generation's basis.
Function `reproduce` grows P to full population size by mutating and crossing.

Our genetic algorithm may use one of two termination criteria. It can be configured to terminate either after a fixed number of generations or after an analysis of a window that contains the most recent $k \in \mathbb{N}$ generations has revealed that the maximum achieved speedup has not changed significantly over these generations. For the latter, we analyze all pairs of subsequent generations inside the window. For both generations, we determine the minimum execution time of the program to be optimized that is yielded by any schedule in the population. If the two minima do not differ by more than a configurable maximum ratio for all pair of subsequent generations, we terminate. The number of generations can still be bounded by a configurable maximum.

Finally, we follow the idea of not constructing genetic operators that aim for specific loop transformations. This would be against the philosophy of polyhedral optimization to map each statement instance individually to an execution step, rather than applying a complex sequence of individual loop transformations. Our genetic operators are designed to allow a move from one search space region to another, which permits us to reach schedules that are not contained in search space regions visited before. The set of legal schedules remains closed under the genetic operators.

Many aspects of the genetic algorithm schema reflect the finding by Pouchet et al. [124] that the polyhedral schedule search space contains mostly schedules that are no improvement. The schedules that perform well are reported to be scarce. In particular, we refer to the strong diversity in the initial population, the strong mutations at the start of the exploration, and the elitism.

6.2 Schedule Mutation and Crossover

We continue by describing our novel mutation and crossover operators. The design of suitable genetic operators is an essential part of constructing an effective genetic algorithm [42]. In our case, it is particularly relevant to be able to traverse our schedule search space that consists of the union of different search space regions. Further, an offspring schedule that is generated from one or two previous schedules should retain some properties of its ancestors but, at the same time, it should differ in other properties. In the case of the mutation operators, an annealing factor controls the amount of similarity. Not all of the sampling techniques described in Chapter 4 are capable of reaching every schedule in the search space considered. In particular, Chernikova sampling (refer to Section 4.6.5) is not. Consequently, two genetic operators are designed specifically to compensate for this drawback.

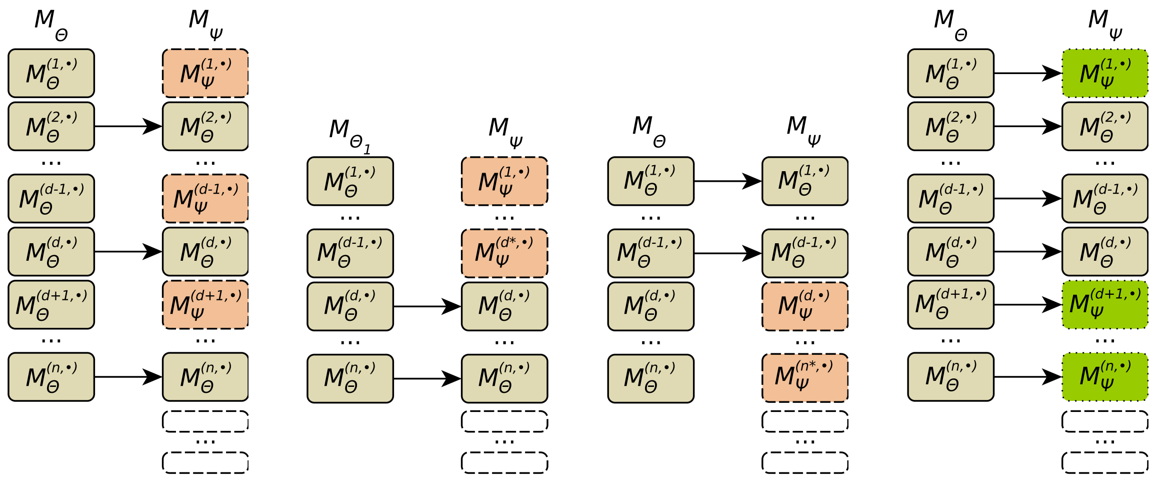
Like random sampling, the genetic algorithm operates on schedule matrices with rational schedule coefficients. The motivation is the same as for random sampling, which Section 4.2 describes. Also, this choice enables us to reuse the algorithms described in Sections 4.5 and 4.6 to replace rows of given schedule matrices. Similarly to random sampling, every schedule that results from mutation and crossing is refined further by schedule completion (refer to Section 4.7). Thereby, we retain the schedule's completeness in that every loop of a transformed program is encoded explicitly in the respective schedule.

Let us look at each genetic operator in detail. In the operators' descriptions, we write G for the set of legality-affecting dependence polyhedra. $C_{\Theta}^{1..d}$ is the set of dependence polyhedra carried by schedule dimensions $1, \ldots, d, d \in \mathbb{N}$, of schedule Θ. Analogously, we write $U_{\Theta}^{1..d}$ for the set of dependence polyhedra not carried by dimensions $1, \ldots, d$. Finally, we refer to the set of dependence polyhedra satisfied strongly by $\Theta^{(d)}$ as S_{Θ}^{d}.

Section 6.2.3 provides a brief overview of the genetic operators presented.

6.2.1 Mutation Operators

Figure 6.1 illustrates our schedule mutation operators. Each operator takes a schedule matrix M_{Θ} as its input and returns a new schedule matrix M_{Ψ}. The mutation of a schedule is a non-deterministic procedure. As we apply schedule completion subsequently to all mutation operators, the modifications of the schedules go beyond the mutations themselves.



(a) Dimension Replacement (b) Prefix Replacement (c) Suffix Replacement (d) Generator Coefficient Replacement

Figure 6.1: Illustrations of our schedule mutation operators. Each box symbolizes a row of a schedule matrix. The rows represented by the narrow blank boxes at the bottom of the offspring matrices result from schedule completion.

6.2.1.1 Dimension Replacement

The idea of dimension replacement is to remove ineffective rows from a schedule matrix. The operator constructs M_Ψ by replacing randomly chosen rows in a copy of M_Θ.

The algorithm for the construction of M_Ψ iterates over the rows of M_Θ starting from the top-most row. At row $M_\Theta^{(d,\bullet)}$, we either set $M_\Psi^{(d,\bullet)} = M_\Theta^{(d,\bullet)}$, or we construct $M_\Psi^{(d,\bullet)}$ to represent a one-dimensional partial schedule that satisfies all dependence polyhedra in $U_\Psi^{1..d-1}$ weakly. Further, $M_\Psi^{(d,\bullet)}$ is constructed such that schedule dimension $\Psi^{(d)}$ carries all dependence polyhedra in $C_\Theta^d \setminus C_\Psi^{1..d-1}$. To obtain $M_\Psi^{(d,\bullet)}$, we start by running an iteration of Algorithm 4.2 (sampling of search space regions). We initialize G to the set of dependence polyhedra not carried by $\Psi^{(1..d-1)}$ and G_d to the set of dependences carried by $\Theta^{(d)}$ and not carried by $\Psi^{(1..d-1)}$. The result of this first step is a polyhedron P containing legal vectors to which we may initialize $M_\Psi^{(d,\bullet)}$. To sample a vector from P, any algorithm for the sampling of vectors with rational components from polyhedra can be used (refer to Section 4.6). Should G_d be empty, we generate $M_\Psi^{(d,\bullet)}$ in the fashion of schedule completion (refer to Section 4.7). Thereby, we avoid generating a schedule dimension that might be eliminated entirely by schedule tree simplification (refer to Section 5.4). Simulated annealing reduces the number of replaced schedule dimensions gradually.

Figure 6.1(a) illustrates dimension replacement. To show that the set of legal schedules is closed under dimension replacement, we prove Theorem 6.2.1.

Theorem 6.2.1. Let S be a SCoP and let G be a set of dependence polyhedra that represent the set of legality-affecting data dependences in S. Let $n \in \mathbb{N}$ and let M_Θ be the matrix of a legal n-dimensional schedule for S. Let M_Ψ be a schedule matrix that results from mutating M_Θ by dimension replacement. The following holds:

$$\left(\forall d \in \{1, \ldots, n\} : C_\Theta^{1..d} \subseteq C_\Psi^{1..d}\right). \tag{6.1}$$

$C_\Theta^{1..n} = G$ and Condition 6.1 imply the legality of Ψ.

Proof of Theorem 6.2.1. We use mathematical induction.

Base Case: $d = 1$

Case 1: $M_\Psi^{(1,\bullet)} = M_\Theta^{(1,\bullet)}$

$C_\Theta^1 \subseteq C_\Psi^1$ holds because the two schedule prefixes are equal.

Case 2: The algorithm generates a new row $M_\Psi^{(1,\bullet)}$.

By the definition of dimension replacement, $\Psi^{(1)}$ satisfies the dependence polyhedra in $G \setminus C_\Psi^{1..0} = G \setminus \emptyset = G$ weakly and the dependence polyhedra in C_Θ^1 strongly. It follows immediately that $C_\Theta^1 \subseteq C_\Psi^1$.

Induction Hypothesis: Let d' be a value chosen arbitrarily but fixed from $\{1, \dots, n-1\}$. It holds that

$$C_\Theta^{1..d'} \subseteq C_\Psi^{1..d'}.$$

Step Case: $d = d' + 1$

Case 1: $M_\Psi^{(d,\bullet)} = M_\Theta^{(d,\bullet)}$

$$\begin{array}{ccl}
C_\Psi^{1..d} & = & \rhd \text{ definitions in Section 2.2.2.3, } \Psi^{(d)} = \Theta(d)\text{, and } d = d' + 1 \\
C_\Psi^{1..d'} \cup S_\Theta^d & \supseteq & \rhd \text{ induction hypothesis} \\
C_\Theta^{1..d'} \cup S_\Theta^d & = & \rhd \text{ definitions in Section 2.2.2.3 and } d = d' + 1 \\
C_\Theta^{1..d} & &
\end{array}$$

Case 2: The algorithm generates a new row $M_\Psi^{(d,\bullet)}$. By the definition of dimension replacement, we know that $C_\Psi^d = C_\Theta^d \setminus C_\Psi^{1..d-1}$.

$$\begin{array}{ccl}
C_\Psi^{1..d} & = & \rhd\ d = d' + 1 \\
C_\Psi^{1..d'+1} & = & \rhd \text{ definitions in Section 2.2.2.3} \\
C_\Psi^{1..d'} \cup C_\Psi^{d'+1} & = & \rhd \text{ definition of dimension replacement} \\
C_\Psi^{1..d'} \cup (C_\Theta^{d'+1} \setminus C_\Psi^{1..d'}) & = & \rhd \text{ set-theoretical simplification} \\
C_\Psi^{1..d'} \cup C_\Theta^{d'+1} & \supseteq & \rhd \text{ induction hypothesis} \\
C_\Theta^{1..d'} \cup C_\Theta^{d'+1} & = & \rhd \text{ definitions in Section 2.2.2.3} \\
C_\Theta^{d'+1} & = & \rhd\ d = d' + 1 \\
C_\Theta^d & &
\end{array}$$

□

6.2.1.2 Schedule Prefix Replacement

The operator changes the prefix of a schedule matrix. That is, it replaces a number of rows at the top of the schedule matrix. Figure 6.1(b) illustrates this type of mutation. The number of rows to be replaced is chosen randomly, but the annealing factor limits the maximum length of the prefix. The new prefix carries at least the dependence polyhedra carried by the old prefix.

Let us assume that we would like to replace $M_\Theta^{(1..e,\bullet)}, e \in \{1, \dots, \text{rows}(M_\Theta)\}$. The length of the prefix of M_Ψ can be any number $e' \in \mathbb{N} \setminus \{0\}$. To produce $M_\Psi^{(1..e',\bullet)}$, we start by running a modification of Algorithm 4.2 (sampling of search space regions). The modified algorithm chooses the sets G_d randomly as subsets of $C_\Theta^{1..e}$ and terminates as soon as all dependences in $C_\Theta^{1..e}$ are carried. The result of Algorithm 4.2 is a list of polyhedra. To sample the rows of $M_\Psi^{(1..e',\bullet)}$ from this list, we rely on Algorithm 4.3 (sampling of schedules from search space regions) and any of the sampling techniques for polyhedra that are described in Section 4.6.

By coincidence, $\Psi^{(1..e')}$ may carry more dependence polyhedra than $\Theta^{(1..e)}$. Also, Ψ may be located in a different search space region than Θ. Altering the outer schedule dimensions influences specifically the fusion and distribution of statements in the transformed program. It is legal to append $M_\Theta^{(e+1..\text{rows}(M_\Theta),\bullet)}$ to $M_\Psi^{(1..e',\bullet)}$ because, by the definition of schedule prefix replacement, we have $C_\Theta^{1..e} \subseteq C_\Theta^{1..e'}$.

6.2.1.3 Schedule Suffix Replacement

Other than schedule prefix replacement, this operator replaces a number of rows chosen randomly at the bottom of a schedule matrix. We illustrate this kind of mutation in Figure 6.1(c).

The number of rows replaced is random, but is bounded from above by the annealing factor. Let $n = \text{rows}(M_\Theta)$ and let $m \in \mathbb{N}, 0 < m < n$, be the number of rows to be replaced. We set $M_\Psi^{(1..n-m,\bullet)} = M_\Theta^{(1..n-m,\bullet)}$. To complete M_Ψ and preserve legality, we generate a schedule that carries the dependence polyhedra in $G \setminus C_\Theta^{1..n-m}$. We use Algorithms 4.2 and 4.3 for this purpose. In the case that $C_\Theta^{1..n-m} = G$, the new schedule's inner dimensions result entirely from schedule completion.

The inner dimensions of a schedule have a particular influence on spatial data locality and the applicability of loop vectorization and tiling.

6.2.1.4 Generator Coefficient Replacement

This mutation operator is specific to the search space exploration with Chernikova sampling (refer to Section 4.6.5). We showed that by its design Chernikova sampling cannot reach certain schedules (refer to Example 4.6.1). Generator coefficient replacement fills this gap.

As illustrated in Figure 6.1(d), generator coefficient replacement is similar to dimension replacement (refer to Section 6.2.1.1). It replaces some rows of a schedule matrix while leaving others intact. Yet, generator coefficient replacement is more finely grained. The new rows have a bigger similarity to the rows that they replace than with dimension replacement.

When we sample a schedule matrix M_Θ, Algorithm 4.3 picks each row $M_\Theta^{(d,\bullet)}$, with $d = 1, ..., \text{rows}(M_\Theta)$, from a polyhedron P_d that represents one dimension of a search space region. $M_\Theta^{(d,\bullet)}$ is a linear combination of a subset of the generators of P_d. P_d's generators are divided into a set V_d of points, one from each of P_d's minimal faces, a set R_d of rays, and a set L_d of lines. When using Chernikova sampling, we store this combination along with each row of a schedule matrix:

$$M_\Theta^{(d,\bullet)} = \sum_{\vec{v}_i \in V_\Theta^d} \alpha_i \cdot \vec{v}_i + \sum_{\vec{r}_i \in R_\Theta^d} \beta_i \cdot \vec{r}_i + \sum_{\vec{l}_i \in L_\Theta^d} \gamma_i \cdot \vec{l}_i, V_\Theta^d \subseteq V_d \ \wedge \ V_\Theta^d \neq \emptyset, R^d \subseteq R_\Theta^d, L_\Theta^d \subseteq L_d.$$

The coefficients adhere to the following constraints:

$$\alpha_1, ..., \alpha_{|V_\Theta^d|} \in \mathbb{Q}^+ \ \wedge \ \sum_{i=1}^{|V_\Theta^d|} \alpha_i = 1, \beta_1, ..., \beta_{|R_\Theta^d|} \in \mathbb{Q}^+, \gamma_1, ..., \gamma_{|L_\Theta^d|} \in \mathbb{Q} \setminus \{0\}.$$

In schedule matrices produced by Algorithm 4.3 (sampling of schedules from search space regions) and Chernikova sampling, we can alter the generators' coefficients of any row in a way that obeys the above constraints and preserve schedules' legality. Recall from Section 4.6

that we can choose a coefficient vector from each of the polyhedra P_d that represent a search space region independently and obtain a legal schedule matrix. This allows for a finely grained mutation operator. While Chernikova sampling uses only integer coefficients for the generators, generator coefficient replacement may introduce rational coefficients with small denominators. Thereby, it solves cases such as Example 4.6.1.

Many of the genetic operators presented in Section 6.2 exploit situations in which schedule dimensions $\Theta^{(d)}$ satisfy dependence polyhedra $D_{O,T}$ weakly or strongly not by the definition of the polyhedron P_d from which they had been sampled originally, but by coincidence. In such cases, $P_d \setminus W_{O,T} = \emptyset$ and $P_d \setminus S_{O,T} = \emptyset$ may not hold. Modifying the generator coefficients of a schedule dimension may cause dimension d or a subsequent one to violate dependence polyhedra. Alternatively, a dependence polyhedron may not be carried by the resulting schedule M_Ψ. In these cases, we use Algorithms 4.2 (sampling of search space regions) and 4.3 (sampling of schedules from search space regions) to replace the schedule matrix suffix starting from the first illegal row or to complete M_Ψ such that all dependence polyhedra are carried.

Simulated annealing controls the number of rows mutated and, per mutated row, the share of generator coefficients mutated. It may be necessary to alter additional generator coefficients to preserve legality.

6.2.1.5 Outlook: Partial Schedule Replacement

In addition to the mutation operators that were implemented in Polyite at the time of the evaluation, we propose another mutation operator that allows the search to move between search space regions and retain much of the parent schedule's properties at the same time.

Replacing blocks of subsequent schedule dimensions (or partial multi-dimensional schedules), as illustrated in Figure 6.2, increases particularly the genetic algorithm's ability to push the schedule dimension that carries a dependence polyhedron inward or outward. At the same time, the other schedule dimensions are retained. Simulated annealing steers the dimensionality of the partial schedule that is to be replaced.

The proof of partial schedule replacement's legality is largely analogous to the proof of dimension replacement's legality.

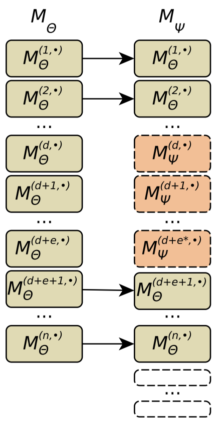


Figure 6.2: Partial Schedule Replacement.

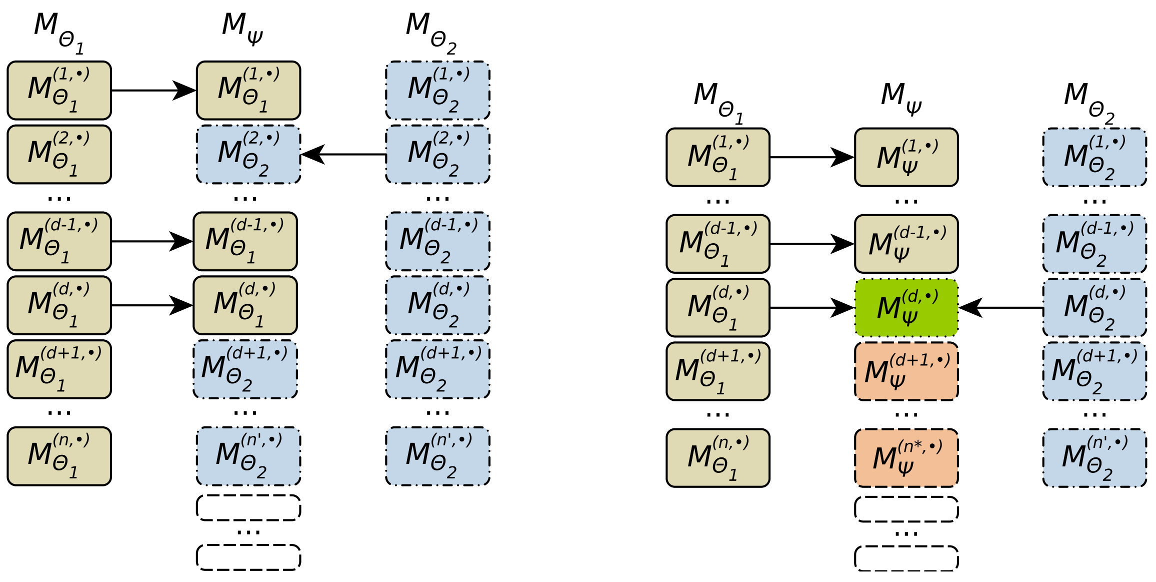
6.2.2 Crossover Operators

We continue with the crossover operators. A crossover operator takes two schedule matrices M_{Θ_1} and M_{Θ_2} as input and combines them to an offspring schedule matrix Θ_Ψ. Figure 6.3 illustrates our tailored legality-preserving crossover operators for schedule optimization.

6.2.2.1 Row-Based Crossover

As illustrated by Figure 6.3(a), row-based crossover takes as input two schedule matrices M_{Θ_1} and M_{Θ_2} and recombines their rows to a new schedule matrix M_Ψ. Per schedule dimension d, the operator may set either $M_\Psi^{(d)} = M_{\Theta_1}^{(d)}$ or $M_\Psi^{(d)} = M_{\Theta_2}^{(d)}$.

Pouchet et al. [124] proposed a row-based crossover for their genetic algorithm. Since their exploration is limited to a single search space region, it is legal to assign $M_{\Theta_1}^{(d,\bullet)}$ or $M_{\Theta_2}^{(d,\bullet)}$ to $M_\Psi^{(d,\bullet)}$ for any $d \in \{1, \dots, n\}$ with n being the dimensionality of the search space region. Thus, their row-based crossover is uniform [100] at row level. Pouchet et al. see the advantage of a row-based crossover in its ability to combine profitable dimensions of two schedules Θ_1 and Θ_2 to a new schedule Θ_Ψ, and, thereby, eliminate other, less profitable dimensions of both Θ_1 and Θ_2. They state that, in the absence of a row-based crossover, Θ_Ψ could still originate from a mutation of Θ_1 or Θ_2, but with smaller probability.


M_{Θ_1}
M_{Ψ}
M_{Θ_2}
$M_{\Theta_1}^{(1,\bullet)}$
$M_{\Theta_1}^{(2,\bullet)}$
$M_{\Theta_1}^{(d-1,\bullet)}$
$M_{\Theta_1}^{(d,\bullet)}$
$M_{\Theta_1}^{(d+1,\bullet)}$
$M_{\Theta_1}^{(n,\bullet)}$
$M_{\Theta_2}^{(1,\bullet)}$
$M_{\Theta_2}^{(2,\bullet)}$
$M_{\Theta_2}^{(d-1,\bullet)}$
$M_{\Theta_2}^{(d,\bullet)}$
$M_{\Theta_2}^{(d+1,\bullet)}$
$M_{\Theta_2}^{(n',\bullet)}$
$M_{\Psi}^{(1,\bullet)}$
$M_{\Psi}^{(d-1,\bullet)}$
$M_{\Psi}^{(d,\bullet)}$
$M_{\Psi}^{(d+1,\bullet)}$
$M_{\Psi}^{(n^*,\bullet)}$

(a) Row-Based Crossover (b) Geometric Crossover

Figure 6.3: Illustrations of our schedule crossover operators. Each box symbolizes a row of a schedule matrix. The rows represented by the narrow blank boxes at the bottom of the offspring matrices M_Ψ resemble the rows appended by schedule completion.

The wider range of our search space complicates the construction of a row-based crossover under which the set of legal schedules is closed. Two schedules can originate from different search space regions. Therefore, they can have different numbers of dimensions and may carry a given legality-affecting dependence polyhedron at different dimensions. Combining their schedule matrices' rows arbitrarily in the way described does not necessarily lead to a legal offspring schedule matrix. Instead, we must construct M_Ψ recursively, starting from the first row and continuing with the next. Each recursive step appends one row. The recursion stops as soon as Ψ carries all dependences in G. If we end up in one of the following situations, while choosing row d of M_Ψ, we must backtrack and change our choice of a previous row:

- We have tested both choices for row d unsuccessfully.
- We find that, to retain legality, we would have to use both $M_{\Theta_1}^{(d,\bullet)}$ and $M_{\Theta_2}^{(d,\bullet)}$ for $M_\Psi^{(d,\bullet)}$ because, otherwise, either one or the other dependence polyhedron in G is not carried by Ψ.
- Any choice that we can make for $M_\Psi^{(d,\bullet)}$ immediately leads to a schedule that violates some dependence polyhedron in G.

By coincidence, or due to our choice of M_{Θ_1} or M_{Θ_2}, we may end up reconstructing M_{Θ_1} or M_{Θ_2}. Algorithm 6.2 is the complete procedure. The algorithm avoids searching subtrees of the search tree that it explores as soon as it detects that none of the solutions in a subtree is a legal schedule. We show that Theorem 6.2.2 holds.

Theorem 6.2.2. All schedules constructed by Algorithm 6.2 satisfy all dependence polyhedra in G weakly. The schedule represented by M'_Ψ in line 21 is certain to be a prefix of a legal schedule.

The algorithm always returns a legal schedule. We show that Theorem 6.2.3 holds.

Theorem 6.2.3. Algorithm 6.2 always returns a schedule that carries all dependence polyhedra in G.

Algorithm 6.2: Row-Based Crossover (`rowCross`)

```
Input:      $M_{\Theta_1}, M_{\Theta_2}$: The pair of schedules to be crossed
            $d \in \mathbb{N}$: The index of the current schedule dimension
            $M_\Psi$: The already constructed prefix of the offspring matrix
Output:     The matrix of a legal schedule that carries all dependences in $G$ or $\perp$ if we could not complete
            $M_\Psi$ accordingly
Parameters: $G$: Set of legality-affecting dependence polyhedra
1  $U \leftarrow G \setminus C_\Psi^{1..d-1}$ ;                    ▷ dependence polyhedra that must be carried
   ▷ $M_\Psi$ is complete. ◁
2  if $U = \emptyset$ then
3      return $M_\Psi$
   ▷ We will not be able to carry all dependences in $G$. ◁
4  if $\text{rows}(M_{\Theta_1}) < d \;\wedge\; \text{rows}(M_{\Theta_2}) < d$ then
5      return $\perp$
6  $\mathsf{mustUse}_{\Theta_1} \leftarrow \text{rows}(M_{\Theta_2}) < d$
7  $\mathsf{mustUse}_{\Theta_2} \leftarrow \text{rows}(M_{\Theta_1}) < d$
   ▷ Are we forced to choose $M_{\Theta_1}^{(d,\bullet)}$ or $M_{\Theta_2}^{(d,\bullet)}$ to carry some dependences? ◁
8  $\mathsf{mustUse}_{\Theta_1} \leftarrow (\exists D_{O,T} \in U : M_{\Theta_1}^{(d,\bullet)} \in S_{O,T} \;\wedge\; (\forall e \in \{d+1, \ldots, \text{rows}(M_{\Theta_1}) : M_{\Theta_1}^{(e,\bullet)} \notin S_{O,T}\}) \;\wedge\; (\forall e \in$
       $\{d, \ldots, \text{rows}(M_{\Theta_2}) : M_{\Theta_2}^{(e,\bullet)} \notin S_{O,T}\}))$
9  $\mathsf{mustUse}_{\Theta_2} \leftarrow (\exists D_{O,T} \in U : M_{\Theta_2}^{(d,\bullet)} \in S_{O,T} \;\wedge\; (\forall e \in \{d+1, \ldots, \text{rows}(M_{\Theta_2}) : M_{\Theta_2}^{(e,\bullet)} \notin S_{O,T}\}) \;\wedge\; (\forall e \in$
       $\{d, \ldots, \text{rows}(M_{\Theta_1}) : M_{\Theta_1}^{(e,\bullet)} \notin S_{O,T}\}))$
   ▷ We cannot use $M_{\Theta_1}^{(d,\bullet)}$ and $M_{\Theta_2}^{(d,\bullet)}$ $\rightarrow$ Backtrack ◁
10 if $\mathsf{mustUse}_{\Theta_1} \;\wedge\; \mathsf{mustUse}_{\Theta_2}$ then
11     return $\perp$
   ▷ Select potential assignments for $M_\Psi^{(d,\bullet)}$ ◁
12 if $\mathsf{mustUse}_{\Theta_1}$ then
13     $C \leftarrow \{M_{\Theta_1}^{(d,\bullet)}\}$
14 else if $\mathsf{mustUse}_{\Theta_2}$ then
15     $C \leftarrow \{M_{\Theta_2}^{(d,\bullet)}\}$
16 else
17     $C \leftarrow \{M_{\Theta_1}^{(d,\bullet)}, M_{\Theta_2}^{(d,\bullet)}\}$
   ▷ Test each potential assignment for $M_\Psi^{(d,\bullet)}$ ◁
18 $M''_\Psi \leftarrow \perp$
19 foreach $\vec{m} \in C$ do
20     if $(M''_\Psi = \perp) \;\wedge\; (\forall D_{O,T} \in U : \vec{m} \in W_{O,T})$ then
21         $M'_\Psi \leftarrow \begin{pmatrix} M_\Psi \\ \vec{m} \end{pmatrix}$
22         $M''_\Psi \leftarrow$ rowCross($M_{\Theta_1}, M_{\Theta_2}, d+1, M'_\Psi$)
23         if $M''_\Psi \neq \perp$ then
24             return $M''_\Psi$
25 return $M''_\Psi$
```

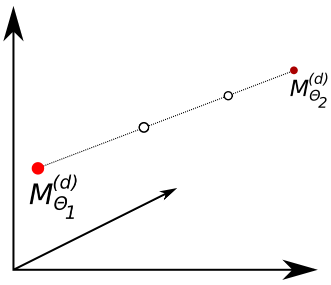


Figure 6.4: The idea of geometric crossover is to create a schedule coefficient vector by interpolating two existing schedule coefficient vectors. The white points on the line segment are potential assignments for $M^{(d)}_{\Psi}$.

Proof of Theorem 6.2.2. We show that, for each $n \in \mathbb{N} \setminus \{0\}$, all schedule matrices with n rows that are constructed by Algorithm 6.2 satisfy all dependence polyhedra in G weakly.

Base Case: $n = 1$.
From $\text{rows}(M'_\Psi) = n = 1$ follows that M_Ψ is an empty matrix. Furthermore, since Θ_1 and Θ_2 are legal schedules, neither $\Theta^{(1)}_1$ nor $\Theta^{(1)}_2$ violate a dependence polyhedron in G. Therefore, $M^{(1,\bullet)}_{\Theta_1}$ and $M^{(1,\bullet)}_{\Theta_2}$ are both legal assignments for ${M'_\Psi}^{(1,\bullet)}$ and M'_Ψ satisfies all dependence polyhedra in G weakly.

Induction Hypothesis: Let $n' \in \mathbb{N} \setminus \{0\}$ be chosen arbitrarily but fixed. Any schedule matrix M_Ψ constructed by Algorithm 6.2 with $\text{rows}(M_\Psi) = n'$ satisfies all dependence polyhedra in G weakly.

Step Case: $n = n' + 1$
To construct coefficient matrix M'_Ψ with n rows, Algorithm 6.2 appends a row to a given coefficient matrix M_Ψ with n' rows. From the induction hypothesis, we know that the schedule represented by M_Ψ weakly satisfies all dependence polyhedra in G. The guard in line 20 prevents us from appending schedule coefficient vectors to M_Ψ that violate dependence polyhedra in $U = G \setminus C^{1..n'}_\Psi$. Thus, all schedule matrices M'_Ψ with n rows that we construct in line 21 satisfy all dependence polyhedra in G weakly. □

Proof of Theorem 6.2.3. The result of the recursive case of Algorithm 6.2 (the return on line 25) is entirely the result of a recursive call. Thus, to show that the algorithm returns only legal schedules, it is sufficient to inspect the non-recursive cases of Algorithm 6.2, which are the returns on lines 3, 5, and 11. In all but the case on line 3, the algorithm returns $\perp$. In line 3, M_Ψ will be returned under the condition that Ψ carries all dependences in G. Thus, Algorithm 6.2 can return only legal schedules or $\perp$.

The algorithm always returns a legal schedule, as it can always reconstruct M_{Θ_1} or M_{Θ_2} (up to the last schedule dimension that carries a dependence polyhedron), which are both legal schedules. □

6.2.2.2 Geometric Crossover

The idea of geometric crossover is to interpolate between the rows of two schedule matrices. This allows us to reach schedules that we cannot reach using Chernikova sampling. Figure 6.4 illustrates the idea.

Let $d \in \{1, \ldots, \min\{\text{rows}(M_{\Theta_1}), \text{rows}(M_{\Theta_2})\}\}$. We construct row $M^{(d,\bullet)}_\Psi$ as a convex combination of $M^{(d,\bullet)}_{\Theta_1}$ and $M^{(d,\bullet)}_{\Theta_2}$. Recall that with Chernikova sampling row d of a schedule matrix is a linear combination of the generators of the polyhedron P_d that represents the schedule's corresponding search space region's d^{th} dimension. Chernikova sampling cannot reach points in P_d that are reachable only via a convex combination of multiple points, of which each belongs to a different of P_d's minimal faces. Furthermore, Chernikova sampling uses integer coefficients for the generators. Geometric crossover lifts both limitations.

We cross M_{Θ_1} and M_{Θ_2} in dimension d if $U_{\Theta_1}^{1..d-1} \subseteq U_{\Theta_2}^{1..d-1}$ or $U_{\Theta_2}^{1..d-1} \subseteq U_{\Theta_1}^{1..d-1}$. Without loss of generality, let us assume $U_{\Theta_1}^{1..d-1} \subseteq U_{\Theta_2}^{1..d-1}$. We show that Theorem 6.2.4 holds.

Theorem 6.2.4.
Let Θ_1 and Θ_2 be two legal schedules for a SCoP. Let $d \in \{1, \ldots, \min\{\dim(\Theta_1), \dim(\Theta_2)\}\}$ and $U_{\Theta_1}^{1..d-1} \subseteq U_{\Theta_2}^{1..d-1}$. All one-dimensional schedules represented by the coefficient vectors in conv.hull$(M_{\Theta_1}^{(d,\bullet)}, M_{\Theta_2}^{(d,\bullet)})$ satisfy the dependences in $U_{\Theta_1}^{1..d-1}$ weakly.

Proof of Theorem 6.2.4. The preconditions tell us that

$$\left(\forall D_{O,T} \in U_{\Theta_1}^{1..d-1} : \left(\forall \begin{pmatrix} \vec{i}_O \\ \vec{i}_T \end{pmatrix} \in D_{O,T} : {\Theta_1}_T^{(d)}(\vec{i}_T) - {\Theta_1}_O^{(d)}(\vec{i}_O) \geq 0\right)\right) \tag{6.2}$$

and

$$\left(\forall D_{O,T} \in U_{\Theta_1}^{1..d-1} : \left(\forall \begin{pmatrix} \vec{i}_O \\ \vec{i}_T \end{pmatrix} \in D_{O,T} : {\Theta_2}_T^{(d)}(\vec{i}_T) - {\Theta_2}_O^{(d)}(\vec{i}_O) \geq 0\right)\right). \tag{6.3}$$

Let $\alpha \in [0, 1]$. Further, let $D_{O,T} \in U_{\Theta_1}^{1..d-1}$ and $\begin{pmatrix} \vec{i}_O \\ \vec{i}_T \end{pmatrix} \in D_{O,T}$. By definition

$$\Theta_X^{(d)}(\vec{i}_X) = M_{\Theta_X}^{(d)} \cdot \begin{pmatrix} \vec{i}_X \\ \vec{p} \\ 1 \end{pmatrix}$$

holds for any statement X, statement instance $\vec{i}_X \in I_X$ and schedule dimension d. $\vec{p}$ is the vector of the SCoP's structure parameters.

Let us show that $M_\Psi^{(d,\bullet)} = \alpha \cdot M_{\Theta_1}^{(d,\bullet)} + (1-\alpha) \cdot M_{\Theta_2}^{(d,\bullet)}$ represents a one-dimensional schedule that satisfies the dependence between $\vec{i}_O$ and $\vec{i}_T$ weakly. From $\alpha \in [0, 1]$ we know that $M_\Psi^{(d,\bullet)} \in$ conv.hull$(M_{\Theta_1}^{(d,\bullet)}, M_{\Theta_2}^{(d,\bullet)})$.

$$\begin{gathered}
\Psi_T^{(d)}(\vec{i}_T) - \Psi_O^{(d)}(\vec{i}_O) \\
= \qquad \triangleright \text{ Definition of } \Psi \\
\left(\alpha \cdot {M_{\Theta_1}}_T^{(d,\bullet)} + (1-\alpha) \cdot {M_{\Theta_2}}_T^{(d,\bullet)}\right) \cdot \begin{pmatrix} \vec{i}_T \\ \vec{p} \\ 1 \end{pmatrix} - \left(\alpha \cdot {M_{\Theta_1}}_O^{(d,\bullet)} + (1-\alpha) \cdot {M_{\Theta_2}}_O^{(d,\bullet)}\right) \cdot \begin{pmatrix} \vec{i}_O \\ \vec{p} \\ 1 \end{pmatrix} \\
= \qquad \triangleright \text{ distributivity} \\
\alpha \cdot {M_{\Theta_1}}_T^{(d,\bullet)} \cdot \begin{pmatrix} \vec{i}_T \\ \vec{p} \\ 1 \end{pmatrix} + (1-\alpha) \cdot {M_{\Theta_2}}_T^{(d,\bullet)} \cdot \begin{pmatrix} \vec{i}_T \\ \vec{p} \\ 1 \end{pmatrix} - \alpha \cdot {M_{\Theta_1}}_O^{(d,\bullet)} \cdot \begin{pmatrix} \vec{i}_O \\ \vec{p} \\ 1 \end{pmatrix} - (1-\alpha) \cdot {M_{\Theta_2}}_O^{(d,\bullet)} \cdot \begin{pmatrix} \vec{i}_O \\ \vec{p} \\ 1 \end{pmatrix} \\
= \qquad \triangleright \text{ commutativity} \\
\alpha \cdot {M_{\Theta_1}}_T^{(d,\bullet)} \cdot \begin{pmatrix} \vec{i}_T \\ \vec{p} \\ 1 \end{pmatrix} - \alpha \cdot {M_{\Theta_1}}_O^{(d,\bullet)} \cdot \begin{pmatrix} \vec{i}_O \\ \vec{p} \\ 1 \end{pmatrix} + (1-\alpha) \cdot {M_{\Theta_2}}_T^{(d,\bullet)} \cdot \begin{pmatrix} \vec{i}_T \\ \vec{p} \\ 1 \end{pmatrix} - (1-\alpha) \cdot {M_{\Theta_2}}_O^{(d,\bullet)} \cdot \begin{pmatrix} \vec{i}_O \\ \vec{p} \\ 1 \end{pmatrix} \\
= \qquad \triangleright \text{ definitions of } \Theta_1 \text{ and } \Theta_2 \\
\alpha \cdot {\Theta_1}_T^{(d)}(\vec{i}_T) - \alpha \cdot {\Theta_1}_O^{(d)}(\vec{i}_O) + (1-\alpha) \cdot {\Theta_2}_T^{(d)}(\vec{i}_T) - (1-\alpha) \cdot {\Theta_2}_O^{(d)}(\vec{i}_O) \\
= \qquad \triangleright \text{ distributivity} \\
\alpha \cdot \left({\Theta_1}_T^{(d)}(\vec{i}_T) - {\Theta_1}_O^{(d)}(\vec{i}_O)\right) + (1-\alpha) \cdot \left({\Theta_2}_T^{(d)}(\vec{i}_T) - {\Theta_2}_O^{(d)}(\vec{i}_O)\right) \\
\geq \qquad \triangleright \text{ Equation 6.2 and } \alpha \geq 0 \\
0 + (1-\alpha) \cdot \left({\Theta_2}_T^{(d)}(\vec{i}_T) - {\Theta_2}_O^{(d)}(\vec{i}_O)\right) \\
\geq \qquad \triangleright \text{ Equation 6.3 and } \alpha \in [0,1] \Rightarrow (1-\alpha) \geq 0 \\
0
\end{gathered}$$

We have shown that, by its construction, $M_\Psi^{(d,\bullet)}$ carries all dependences in $U_{\Theta_1}^{1..d-1}$. To construct a legal schedule, we set $M_\Psi^{(1..d-1,\bullet)} = M_{(\Theta_1}^{(1..d-1,\bullet)}$ and complete M_Ψ by generating

Table 6.1: Overview of the mutation and crossover operators. All operators are designed such that they cannot derive an illegal schedule from legal schedules.

Genetic Operator	**Description**	**Illustration**
Dimension Replacement	Replaces randomly chosen rows of a schedule matrix.	Figure 6.1(a)
Schedule Prefix Replacement	Replaces the first rows of a schedule matrix.	Figure 6.1(b)
Schedule Suffix Replacement	Replaces the last rows of a schedule matrix	Figure 6.1(c)
Partial Schedule Replacement (currently not implemented)	Replaces randomly chosen blocks of subsequent schedule matrix rows. The blocks' lengths are also random.	Figure 6.2
Generator Coefficient Replacement	Replaces schedule matrix rows chosen randomly by schedule coefficient vectors constructed from the same generators as the original rows.	Figure 6.1(d)
Row-Based Crossover	Combines the rows of two schedule matrices to a new schedule matrix.	Figure 6.3(a)
Geometric Crossover	Interpolates between rows of two given schedule matrices.	Figure 6.3(b)

a new suffix using Algorithms 4.2 (sampling of search space regions) and 4.3 (sampling of schedules from search space regions). Figure 6.3(b) illustrates the composition of M_Ψ. When using Chernikova sampling, the set of generators of $M_\Psi^{(d,\bullet)}$ is the union of the generators of $M_{\Theta_1}^{(d,\bullet)}$ and $M_{\Theta_2}^{(d,\bullet)}$. We have to derive their new coefficients from the original coefficients and α. □

6.2.3 Summary

In total, we have presented seven different genetic operators. Five are mutation operators and two are crossover operators. The operators serve different purposes and vary in the degree of difference between an offspring schedule matrix and its parent schedule matrices. While the crossover operators and some of the mutation operators have the ability to construct an offspring schedule matrix that lies in a different search space region than its parents, there are also mutation operators that operate at a more finely grained level.

Table 6.1 provides an overview of our mutation and crossover operators.

6.3 Equivalence of Schedules

For a genetic algorithm, it can be important to avoid adding duplicate solutions to its population or even solutions that are equivalent to already present ones with respect to some predicate. With respect to our genetic algorithm for polyhedral schedule optimization, at least four different equivalence relations are candidates:

1. Equivalent schedules share the same rational coefficient matrix. If constructed via Chernikova sampling (refer to Section 4.6.5), they must also have the same set of generators per matrix row.

2. Equivalent schedules share the same integer coefficient matrix after the conversion from rational to integer coefficients (refer to Section 4.6.5).

3. Equivalent schedules correspond to the same simplified schedule tree.

4. Equivalent schedules yield the same execution order after the application of tiling and potential other schedule tree optimizations such as strip mining. This criterion is equivalent to Equation 5.9 in Section 5.5.4.

Especially the first two criteria, but to some extent also the third, permit the presence of different schedule matrices in the genetic algorithm's population that prescribe the same execution order of the SCoP's statement instances. Yet, some schedule matrices may be more effective than others: for instance, there may be schedule matrices that contain extremely large schedule coefficients or the original rational schedule matrix could contain coefficients with high denominators. We must consider such matrices to be degraded as they are likely to yield integer overflows, which can cause miscompilation and wrong computation results. Introducing extrem coefficients into more schedules by mutation and crossing is likely counterproductive. Therefore, evicting all representations of a generally effective schedule, except the most degraded one, may cause the population of the genetic algorithm to degenerate. Currently, we use the first criterion as the primary criterion for schedule equivalence in POLYITE.

6.4 Felix Bernasch's Bachelor Thesis: A Distributed Genetic Algorithm for Polyite

Felix Bernasch proposes an approach to turn POLYITE's genetic algorithm into a distributed evolutionary algorithm in his Bachelor thesis [23]. A *distributed evolutionary algorithm* [3] maintains a population of candidate solutions that is split into subpopulations. Each of the subpopulations develops separately. Occasionally, the subpopulations exchange small numbers of solutions. The subpopulations are said to reside on islands, which may correspond to different compute nodes, processes or sockets. A migration policy determines the frequency of the exchange of solutions between the islands, the selection of the solutions that will migrate, the solutions that will be replaced by the migrants, and the arrangement of the islands. The latter determines the pairs of islands that can exchange migrants.

Alba and Tomassini [3] describe that the advantage of distributed evolutionary algorithms over standard evolutionary algorithms, which maintain a single large population, lies not only in an efficient parallelization across several sockets or the nodes of a compute cluster. Splitting the population into several subpopulations that communicate only occasionally is likely to decrease the numbers of solutions that need to be tested in order to find an acceptably profitable solution. The reason is that a distributed evolutionary algorithm tends to keep a stronger diversity of its complete population and is therefore more likely to explore different regions of the search space simultaneously than standard evolutionary algorithms. This also reduces the probability of premature convergence [4].

Bernasch proposes to use either of two given topologies for POLYITE's distributed evolutionary algorithm. The first is a ring-based topology to which Bernasch refers as the Neighbor Strategy. We illustrate this strategy in Figure 6.5(a). The islands are arranged in a circle and each can exchange migrating schedules with its left and its right neighbor. The Neighbor Strategy is inspired by the ring topology described by Belkadi et al. [20] and by the stepping stone model, which Hiroyasu et al. [70] describe. Further, Bernasch describes what he calls the Neighborhood Strategy. We illustrate it in Figure 6.5(b). The Neighborhood Strategy arranges the islands on a rectangular grid. Each island can exchange migrating schedules with any of its four neighbors (top, left, bottom, and right). Islands on the edge of the grid exchange schedules with the respective islands on the grid's opposite side. Belkadi et al. [20] describe this strategy and refer to it as a grid topology with two dimensions. It can also be interpreted as a grid that is projected onto the surface of a torus.

The migration takes place every $n \in \mathbb{N}$ generations. During a migration phase, an island sends schedules to one of each of its potential migration partners. The order in which an

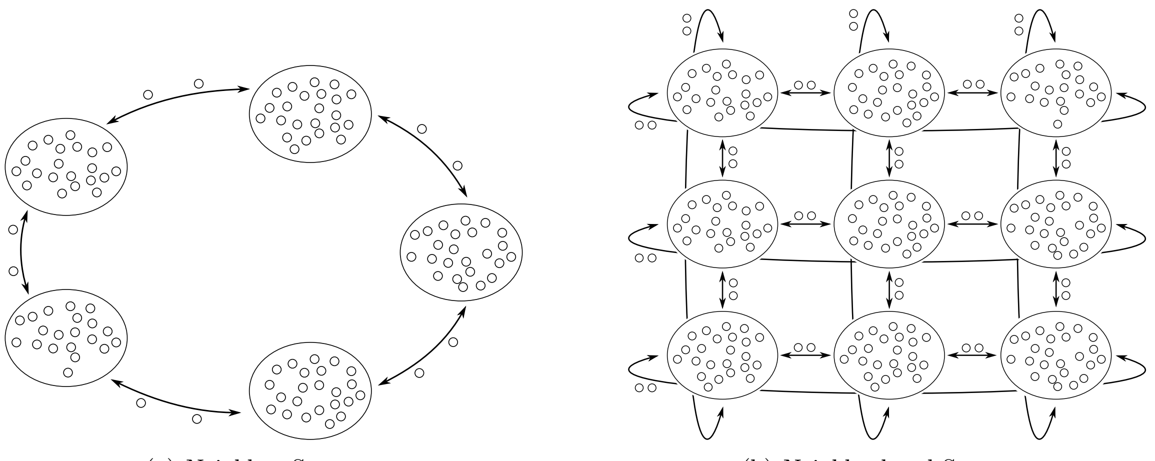

(a) Neighbor Strategy (b) Neighborhood Strategy

Figure 6.5: Illustrations of POLYITE's distributed evolutionary algorithms. The islands are represented by the bigger circles. The small circles represent individual schedules. Islands that can exchange migrating schedules are connected by arrows.

island serves its migration partners is fixed. The migration frequency n and the number of schedules that migrate from one island to another are configurable. Any schedule in the basis of an island's next local population is an equal candidate for migration.

7 Classification of Schedules

The genetic algorithm for schedule optimization presented in Chapter 6 determines a schedule's profitability by benchmarking. Given a program to be optimized and a schedule that has not been evaluated yet, Polyite applies the schedule to the program and measures the transformed program's execution time. Having evaluated all schedules in the current population, the genetic algorithm selects the fitter half of the population and uses it as the basis for the subsequent population. Figure 7.1 illustrates this procedure.

While benchmarking of the transformed program is the most precise method to determine a schedule's profitability, it is also time-consuming and hardware-demanding. Moreover, cross-compilation is impossible since the target hardware must be available to determine the execution time of a program variant. One could reduce the optimization time by benchmarking with very small data sets, but this would bias the optimization result since it may reduce the effect of data locality optimizations and coarse-grained parallelization. Instead, we propose to replace benchmarking to the extent possible by a less precise, but cheaper-to-evaluate performance predictor. This predictor is based on a surrogate performance model learned from a set of training data using supervised machine learning. The training data must originate from iterative optimizations using the genetic algorithm based on benchmarking, which we call GA_B in the following, as shown in Figure 7.1, or random exploration, or a combination of both. Thus, we still need the target hardware for generating training data for a surrogate performance model, but then we can use the trained model to optimize in an iterative, but mostly hardware-independent manner.

We train the surrogate performance models on structural features of schedules and the execution time of the corresponding transformed code. To make the approach practical, schedule features that can be extracted at low cost must be available. The features must be designed such that the learned model is transferable: the model must be able to predict the profitability of schedules for unseen programs. We use features that characterize schedules, rather than characterizing SCoPs or programs. Thereby, we give up on the possibility to learn a widely applicable performance model from a carefully chosen and comprehensive set of training programs. We gain the ability to train a model that targets a specific domain of programs with similar performance characteristics. Such a domain could be matrix multiplication. By abstaining from features that characterize programs (SCoPs in our case, i.e., program regions with static control), the ability to transfer a program model will not be hampered by differences between programs that have no influence on the choice of an effective program optimization in the specific case.

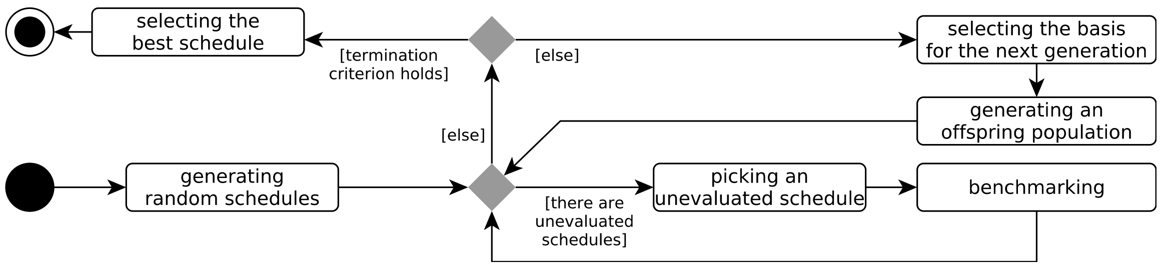
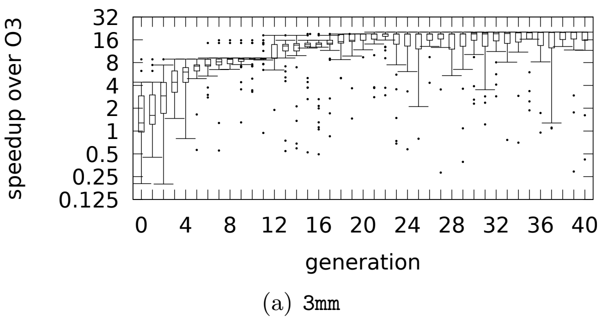

selecting the best schedule
[termination criterion holds]
[else]
selecting the basis for the next generation
generating an offspring population
[else]
generating random schedules
picking an unevaluated schedule
benchmarking
[there are unevaluated schedules]

Figure 7.1: Workflow of our genetic algorithm for schedule optimization, as described in Chapter 6. The schedule evaluation relies on execution time measurement.



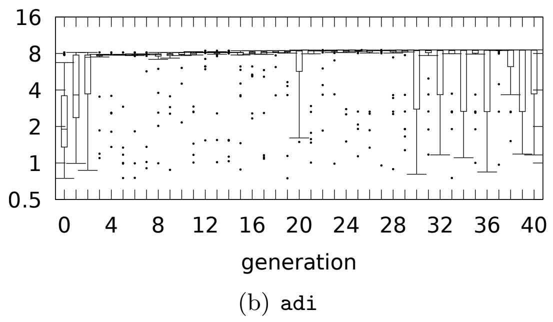


Figure 7.2: Distribution of speedups yielded by the schedules in the GA's generations for `3mm` and `adi`.

Danner [46] attempted to predict the speedup in execution time over the execution time of the original sequential code yielded by a schedule in his Master thesis. Danner used regression models and a set of schedule tree features that are strongly related to different performance aspects. Even though Danner's study covered only a set of simple programs (namely `cholesky`, `gemm`, `seidel-2d`, `syrk`, `syr2k`, and `trmm` of PolyBench 4.1 [121]), the results were disheartening. Normalizing the speedup values in the training data to $[0, 1]$ had not been tested, and may improve the prediction accuracy there. Another problem of Danner's approach is the high run-time complexity of the included processor cache hit rate approximation. This appears to be a general drawback of contemporary polyhedral cache models [14, 37]. Finally, Danner's study was performed on an early stage of Polyite and may be biased by issues that were resolved later on.

From this experience, we moved to another approach that is based on the classification of schedules and comparatively simple structural features of schedule trees [66] (refer to Section 5.2). An analysis of GA_B revealed that the variance of the speedups in execution time yielded by the schedules in the genetic algorithm's populations decreases fast. Also, the maximum speedup reachable is approached quickly. We attribute these findings primarily to GA_B's elitism and to the chosen configuration of the sampling strategy that makes profitable schedules likely to occur. In particular, coarse-grained parallelism is likely to occur and we avoid dense schedule matrices. Exemplarily, we illustrate the two findings in Figure 7.2 for the PolyBench 4.1 benchmarks `3mm` and `adi`.

We show, per benchmark and generation of the genetic algorithm, the distribution of the speedups yielded by the schedules in the population. The speedups are calculated over the execution time of the original sequential program. The genetic algorithm was executed for 40 generations, excluding the initial population, with a population size of 30. 630 schedules were tested in total. In both cases, the variance of speedups decreases fast. For `adi`, the maximum speedup is reached almost in the initial population. In the case of `3mm`, 90% of the maximum speedup are reached after eleven generations, which corresponds to 195 schedules tested. This and all other execution times of transformed program versions that we present in Chapter 7 were produced on an Intel Xeon E5-2650 v2 CPU @ 2.6GHz with eight physical cores and 20MB of L3 cache. The operating system was Debian 9 with Linux 4.9. To apply the schedules generated by Polyite, we relied on LLVM in the version of commit `bf8415a8` (Jan. 10, 2016), clang[8] in the version of commit `9909f323` (Jan. 27, 2016), and Polly in the version of commit `2b618e01` (Jan. 27, 2016)[9]. We modified Polly such that we could import schedule trees and apply Polly's schedule tree transformations, like tiling,

[8]LLVM's front end for languages in the C language family (C, C++, Objective C, etc. (`http://clang.llvm.org`)

[9]The commit hashes refer to commits in `http://llvm.org/git/llvm.git`, `http://llvm.org/git/clang.git`, and `http://llvm.org/git/polly.git`, respectively.

Table 7.1: The symbols that we use in the features' definitions.

$\dim(I_S)$	The dimensionality of the iteration domain of statement S.
I	The set of the SCoP's statements' iteration domains
$\dim(\Theta)$	Dimensionality of schedule Θ
$\mathcal{S}$	The number of sequence nodes in a given schedule tree.
$\mathcal{L}$	The number of leaves of a given schedule tree.
$\mathcal{A}$	The total number of memory access relations in the SCoP. We distinguish between read and write accesses.
Δ_{S_i}	The number of loops surrounding statement S_i, or 1 (instead of 0) in the absence of loops. As a simplification we use the value of $\dim(I_{S_i})$.

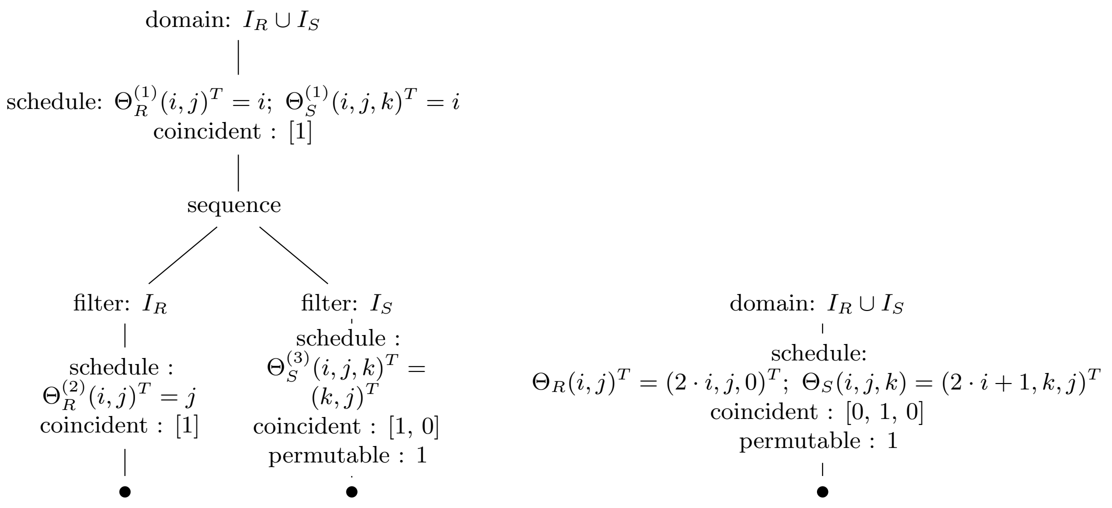
to imported schedules trees. We configured Polly to tile with a default tile size of 64, kept prevectorization (strip-mining) disabled, and enabled parallelization. Each program version was further optimized by `-O3`. The findings motivated us to use a classifier that identifies likely profitable schedules and to evaluate whether such a classifier suffices to guide our genetic algorithm.

In Section 7.1, we present a set of structural schedule features that can be extracted from schedule trees. A discussion of these features can be found in Section 7.1.4. We continue by describing how we can learn a surrogate performance model for a schedule classifier using supervised machine learning. Also, we propose two ways of combining the classifier with our genetic algorithm (refer to Section 7.2).

7.1 Schedule Features

Our schedule features are functions that map schedule trees to real numbers. We compute the feature values from simplified schedule trees, as described in Chapter 5. Schedule features are either structural, such as the depth of the schedule tree, or performance-related, such as coarse-grained parallelism. Note that the features' design relies on the fact that we use them to characterize linearly affine schedule functions. Further, we exploit the property that each statement occurs in exactly one branch of any schedule tree. The latter is due to the design of Algorithm 5.1 for schedule tree construction. Some features may not be meaningful for schedule trees that do not have these properties. In Table 7.1, we list the symbols that we use in the features' definitions. We assume a SCoP with $m \in \mathbb{N} \setminus \{0\}$ statements. Thus, we have $I = \{I_{S_1}, ..., I_{S_m}\}$.

As described in Section 5.6, to prepare a schedule for code generation, we transform it to a schedule tree, but do not represent every case of detectable textual execution order of statements using sequence nodes. In particular, we do not use shifted stride detection. Thereby, we extended the applicability of tiling to imperfect loop nests. In the design of our schedule features, we must reflect this decision: per feature, we describe whether we extract it from the schedule tree prepared for code generation or from a schedule tree that always represents detectable textual order using sequence nodes and allows for a stronger simplification. We refer to the schedule tree prepared for code generation by T_C and to the schedule tree during whose construction we also used shifted stride detection by T_S. As discussed in Section 5.6, extracting features from the schedules before tiling, which Polyite does not perform by itself but delegates to the polyhedral compiler that applies the schedule to the program to optimize, is a tradeoff between run-time complexity and algorithmic complexity on the one side, and accuracy on the other side.


domain: $I_R \cup I_S$
schedule: $\Theta_R^{(1)}(i,j)^T = i$; $\Theta_S^{(1)}(i,j,k)^T = i$
coincident : [1]
sequence
filter: I_R
schedule : $\Theta_R^{(2)}(i,j)^T = j$
coincident : [1]
filter: I_S
schedule : $\Theta_S^{(3)}(i,j,k)^T = (k,j)^T$
coincident : [1, 0]
permutable : 1
domain: $I_R \cup I_S$
schedule: $\Theta_R(i,j)^T = (2 \cdot i, j, 0)^T$; $\Theta_S(i,j,k) = (2 \cdot i + 1, k, j)^T$
coincident : [0, 1, 0]
permutable : 1

(a) T_S (b) T_C

Figure 7.3: Simplified schedule tree of the schedule in Equation 7.1. Shifted stride detection was enabled in the construction of the tree in Figure 7.3(a). Shifted stride detection was disabled in the construction of the tree in Figure 7.3(b).

7.1.1 Requirements for a Set of Schedule Features

Before we define the features, we list the requirements for schedule features that are necessary for the learned surrogate performance models to be transferable.

- A schedule feature must be independent of the SCoP size or the SCoP's complexity. Otherwise, performance models are not be transferable between SCoPs of different size since the affected features' value ranges depend on the size of the SCoPs: the same feature value has a different meaning for a small SCoP than for a large SCoP. As an example, an unsuitable feature is the number of statements in the program.
- It must be possible to normalize feature values to the interval $[0, 1]$ (or any other interval). Otherwise, we have a similar problem as with features that reflect SCoP size: we cannot transfer learned performance model between programs whose feature values' ranges differ. Good candidates for features that can be normalized are those for which we can determine a maximum feature value for a given SCoP. To normalize a feature's value, we may simply divide by the feature's maximum value for the current SCoP.
- Feature calculation must be fast. The duration must not exceed the time needed to evaluate a program version's execution time by benchmarking.
- Finally, the feature set should cover the most relevant performance aspects of schedules.

Example 7.1.1. We continue the running example `syrk`. We demonstrate feature extraction using the following schedule:

$$\Theta_R(i,j)^T = (2 \cdot i, j, 0)^T,\ \Theta_S(i,j,k)^T = (2 \cdot i + 1, k, j)^T. \tag{7.1}$$

From schedule tree construction and simplification, we obtain the two schedule trees T_S and T_C shown in Figure 7.3.

From Figure 7.3 it is apparent that, while T_C exposes a large tilable band, it does not expose much of the schedule's other structural properties. Conversely, T_S has smaller tilable bands, but the schedule tree is simplified more strongly and exposes parallelism and the textual order of the statements. ◁

In Section 7.1.4, we assess the set of features that we propose in the following for their compliance with the requirements enumerated above.

7.1.2 Structural Features of Schedule Trees

The following features characterize the structure of a schedule tree. They are not obviously related to a performance aspect. We discuss how they may relate to performance. Some of our structural features refer to loop distribution and fusion and are consequently not meaningful for SCoPs that consist of a single statement. A transfer of a performance model from a SCoP with a single statement to a SCoP with multiple statements is still possible, but the set of features used will have to be reduced to those that are meaningful for both kinds of SCoP.

7.1.2.1 Number of Leaves

The number of leaves of a schedule tree grows with loop distribution and shrinks with (partial) loop fusion (and cases in which we cannot decide about the actual order of two statements, as illustrated in Section 5.6).

Bondhugula et al. [27] proposed an approach to account for the impact of loop fusion and distribution in an automatic polyhedral parallelizing compiler. An approach by Pouchet et al. [125] determines a suitable balance of loop fusion and distribution by an iterative search and, at the same time, accomplishes parallelism, the minimization of communication and synchronization, and the possibility to tile the resulting loop nests in a model-driven way. Both approaches are motivated by the perception that maximal loop fusion and maximal loop distribution can act orthogonally in some situations. Fusing maximally can avert parallelization and vectorization, and it may interfere with hardware prefetching. Conversely, distributing maximally may reduce data locality. We illustrate the tradeoff between data locality and parallelism in Figure 7.4.

In our case, each statement corresponds to exactly one branch of a schedule tree. Consequently, the maximum number of leaves of a schedule tree is $|I|$. When Polyite fuses two statements that are not in a dependence relation, it tends to leave their textual order unspecified. We can leverage these facts to use the schedule tree's number of leaves as a feature that relates to (partial) loop fusion and loop distribution. The normalized feature is

$$F_{\text{Leaves}} = \frac{\mathcal{L}}{|I|}.$$

We extract the feature from T_S since it provides the most accurate information about textual ordering that is available.

Example 7.1.2. We continue Example 7.1.1. The SCoP has two statements and T_S has two leaf nodes (refer to Figure 7.3(a)). Consequently, we have

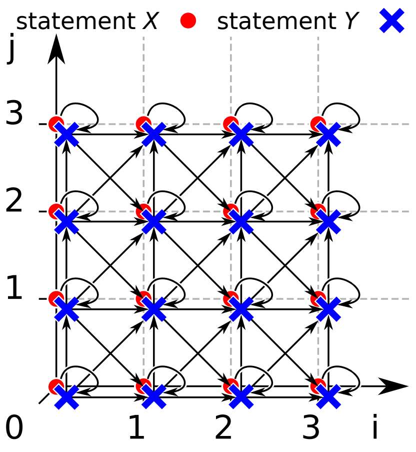
$$F_{\text{Leaves}} = \frac{\mathcal{L}}{|I|} = \frac{2}{2} = 1.$$

◁

7.1.2.2 Number of Sequence Nodes

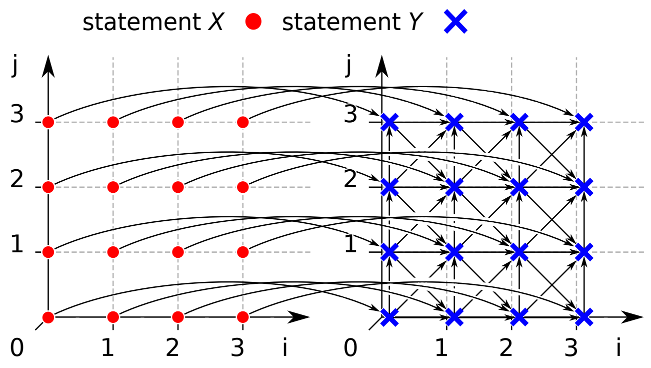
The number of sequence nodes is another schedule tree feature that corresponds to loop distribution and loop fusion. While F_{Leaves} is sensitive to a (partial) fusion of statements that are not in a dependence relation, a feature that expresses the number of sequence nodes of a schedule tree also incorporates the number of schedule dimensions at which the textual ordering of statements is encoded in a given schedule.

```
for (int i = 0; i < n; ++i)
  for (int i = 0; i < n; ++i) {
    A[i][j] = i; // statement X
    A[i + 1][j + 1] = A[i + 1][j] + A[i][j] +
        A[i][j + 1] + A[i][j + 2] // statement Y
  }
```



(a) Loop Fusion

```
# pragma omp parallel for
for (int i = 0; i < n; ++i)
  for (int i = 0; i < n; ++i)
    A[i][j] = i; // statement X
for (int i = 0; i < n; ++i)
  for (int i = 0; i < n; ++i)
    A[i + 1][j + 1] = A[i + 1][j] + A[i][j] +
        A[i][j + 1] + A[i][j + 2] // statement Y
```



(b) Loop Distribution

Figure 7.4: Comparison of loop fusion and loop distribution. While loop fusion improves data locality by enabling the immediate reuse of `A[i][j]`, it prohibits parallelization. In the case of loop distribution, we lose data locality, but the first loop nest can be parallelized.

The normalization of the feature's value relies on the maximum number of sequence nodes that can occur in a schedule tree in which every statement corresponds to exactly one branch. This number is reached in a schedule tree that is binary and in which every leaf corresponds to one statement. In such a tree, the number of leaves is $|I|$ and the number of sequence nodes is $|I| - 1$, which is a well known property of binary trees. Let $\mathcal{S}$ be a given schedule tree's number of sequence nodes. The feature is then defined as follows:

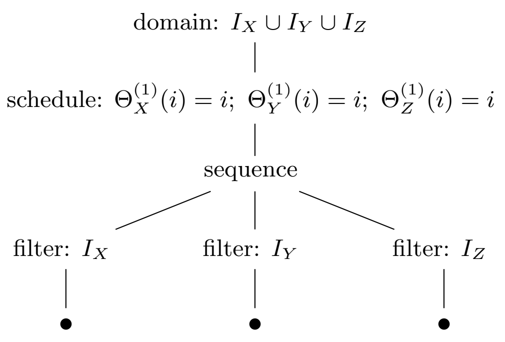
$$F_{\text{Seq}} = \frac{\mathcal{S}}{|I| - 1}.$$

Since the feature relates to the textual ordering of statements, we extract its value from T_S. By Example 7.1.3 we motivate the increased expressiveness of the combination of F_{Leaves} and F_{Seq} compared to only F_{Leaves}.

Example 7.1.3. Consider the following statement iteration domains:

$$I_X = I_Y = I_Z = \{i \mid i \in \mathbb{N} \ \wedge \ 0 \leq i < n\}, n \in \mathbb{N}.$$

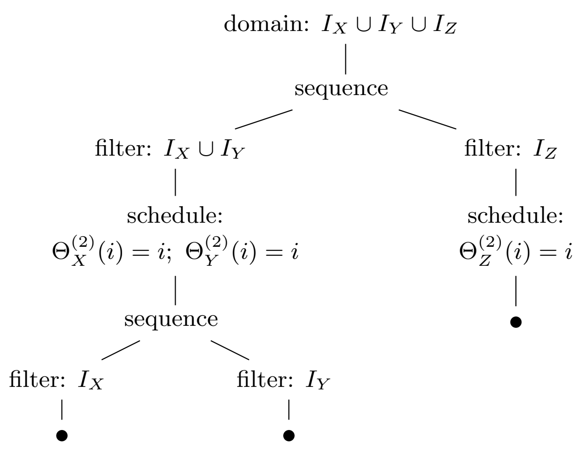
Figure 7.5 shows the trees of two schedules Θ_1 and Θ_2 for these iteration domains and the corresponding code. For this example, the computation of the sample program is irrelevant. Therefore, we only sketch statements using a notation similar to a function call in our code snippets. Although Θ_1 fuses all three statements in one loop while Θ_2 partially distributes the statements, we have $F_{\text{Leaves}} = 1$ for both schedules. In contrast, F_{Seq} exposes this difference: the feature's value is 0.5 for Θ_1 and 1 for Θ_2. ◁



domain: $I_X \cup I_Y \cup I_Z$
schedule: $\Theta^{(1)}_X(i) = i;\ \Theta^{(1)}_Y(i) = i;\ \Theta^{(1)}_Z(i) = i$
sequence
filter: I_X
filter: I_Y
filter: I_Z

```
for (int i = 0; i < n; ++i) {
  X(i);
  Y(i);
  Z(i);
```

(a) Θ_1: maximum loop fusion with a total ordering of the fused statements. $F_{\text{Leaves}} = 1, F_{\text{Seq}} = 1$



```
for (int i = 0; i < n; ++i) {
  X(i);
  Y(i);
}
for (int i = 0; i < n; i += 1)
  Z(i);
```

(b) Θ_2: partial loop distribution with a total ordering of the fused statements. $F_{\text{Leaves}} = 1, F_{\text{Seq}} = 0.5$.

Figure 7.5: Two schedules for a set of statement iteration domains. The schedules differ with respect to loop fusion and distribution. Only the combination of F_{Leaves} and F_{Seq} can expose this difference. Per schedule, we show its tree and the corresponding code.

Example 7.1.4. We continue Example 7.1.1. The schedule tree in Figure 7.3(a) has one sequence node and the SCoP has two statements. We get

$$F_{\text{Seq}} = \frac{\mathcal{S}}{|I| - 1} = \frac{1}{2 - 1} = 1.$$

$\triangleleft$

7.1.2.3 Schedule Tree Depth

Schedule trees can be characterized by their depth, i.e., by the length of the longest path from the domain node at the tree's root to any leaf.

The depth of a schedule tree increases with distribution of statements at different schedule dimensions and the absence of forward-only data communication in band nodes' partial schedules, which prevents the gathering of their partial schedules in single band nodes that are marked permutable. The height of a schedule tree is inversely proportional to the number of partial schedules that can be grouped into permutable band nodes and the frequency of (partial) loop fusion.

For a normalized schedule tree depth feature, we must determine the depth that a schedule tree for a given SCoP can have. Without counting domain nodes and filter nodes, the true maximum depth is

$$\Gamma = |I| + \sum_{i=1}^{|I|} \dim(I_{S_i}).$$

We observed that, when we sample schedules with Chernikova sampling (refer to Section 4.6.5) in a configuration that yields schedule matrices with a high sparsity, we mostly obtain schedules that corresponds to schedule trees that have a depth lower than the following, smaller bound:

$$\Gamma' = |I| + \max_{i=1}^{|I|} \dim(I_{S_i}) + 1.$$

With γ denoting the actual depth of a schedule tree, the normalized schedule tree depth feature is

$$F_{\text{Depth}} = \frac{\gamma}{\Gamma'}.$$

Normalizing the schedule tree depth by Γ', on the one hand, allows a few schedules to have a normalized schedule tree depth larger than 1 but, on the other hand, allows for a larger variance of the feature's value across different schedules. We extract F_{Depth} from T_S. Thereby, we primarily classify the maximally simplified schedule tree, but not so much the actual transformed program. Conversely, extracting the feature from T_C means accepting a different bias that is due to the reduced detection of textual ordering and the weaker simplification. In summary, directly relating F_{Depth} to performance is difficult.

The theoretical maximum depth Γ corresponds to the case that every statement corresponds to a different branch of the schedule tree, every sequence node separates only one statement from the remaining, not yet separated, statements. The depth of this cascade of sequence nodes and leaves is $|I|$. Below the schedule tree's domain node and above the first sequence node is a sequence of band nodes. Each band node encodes exactly one loop around one statement and none of them can be grouped into a permutable band node. The length of this sequence is $\sum_{i=0}^{|I|} \dim(S_i)$. In Example 7.1.5, we show a schedule tree that has depth Γ.

Example 7.1.5. Let us consider a SCoP with iteration domains

$$I_X = \{i \mid i \in \mathbb{N} \ \wedge \ 0 \leq i < n\}, \ I_Y = \{i \mid i \in \mathbb{N} \ \wedge \ 0 \leq i < n\}, n \in \mathbb{N}$$

and dependence polyhedron

$$D_{X,X} = \{(i, i+1)^T \mid i \in I_X \ \wedge \ (i+1) \in I_X\}.$$

The schedule

$$\Theta_X(i) = (i, -i, 0)^T, \ \Theta_Y(i) = (0, i, 1)^T$$

corresponds to the schedule tree in Figure 7.6, which has depth

$$\Gamma = |I| + \sum_{I_S \in I} \dim(I_S) = 2 + \dim(I_X) + \dim(I_Y) = 4.$$

Due to the negative direction of $D_{X,X}$ with respect to $\Theta_X^{(2)}$, the two one-dimensional band nodes cannot be combined to one band node with a two-dimensional partial schedule that is marked permutable. ◁

Example 7.1.6. Having shown a case in which a schedule tree has exactly depth Γ, let us illustrate a case in which a schedule tree has exactly depth Γ'. Given are the following statement iteration domains:

$$I_X = I_Y = I_Z = \{i \mid i \in \mathbb{N} \ \wedge \ 0 \leq i < n\}, n \in \mathbb{N}.$$

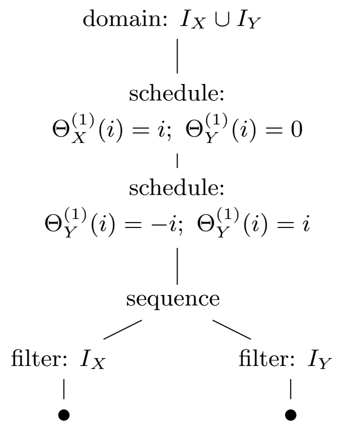
Assuming the absence of data dependences, the schedule

$$\Theta_X(i) = (i, 0, 0, 0, 0)^T, \ \Theta_Y(i) = (0, i, 1, 0, 0)^T, \ \Theta_Z(i) = (0, 0, 1, i, 1)^T$$

corresponds to a simplified schedule tree with depth

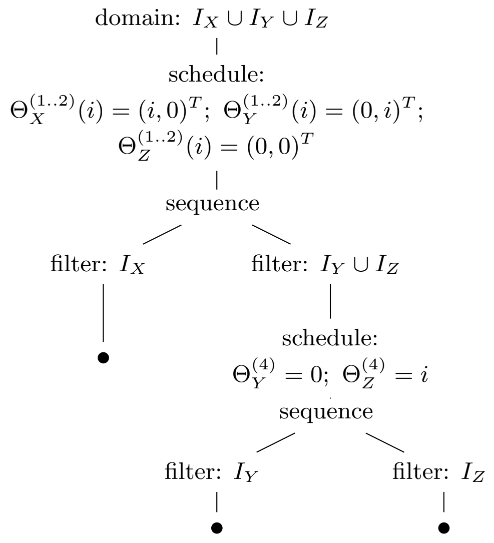
$$\Gamma' = |I| + (\max_{I_S \in I} \dim(I_S)) + 1 = 3 + \max\{\dim(I_X), \dim(I_Y), \dim(I_Z)\} + 1 = 3 + 1 + 1 = 5.$$

We show the schedule tree in Figure 7.7. ◁



domain: $I_X \cup I_Y$
schedule:
$\Theta_X^{(1)}(i) = i$; $\Theta_Y^{(1)}(i) = 0$
schedule:
$\Theta_Y^{(1)}(i) = -i$; $\Theta_Y^{(1)}(i) = i$
sequence
filter: I_X
filter: I_Y

Figure 7.6: Example of a schedule tree that has the maximum depth possible.



domain: $I_X \cup I_Y \cup I_Z$
schedule:
$\Theta_X^{(1..2)}(i) = (i, 0)^T$; $\Theta_Y^{(1..2)}(i) = (0, i)^T$;
$\Theta_Z^{(1..2)}(i) = (0, 0)^T$
sequence
filter: I_X
filter: $I_Y \cup I_Z$
schedule:
$\Theta_Y^{(4)} = 0$; $\Theta_Z^{(4)} = i$
sequence
filter: I_Y
filter: I_Z

Figure 7.7: A schedule tree that has exactly depth Γ'.

Example 7.1.7. We continue Example 7.1.1 to demonstrate F_{Depth}. The schedule tree in Figure 7.3(a) has depth $\gamma = 4$, the SCoP contains two statements and the maximum dimensionality of a statement iteration domain in the SCoP of `syrk` is $|I_S| = 3$. Thus,

$$F_{\text{Depth}} = \frac{\gamma}{\Gamma'} = \frac{\gamma}{|I| + \max\limits_{i=1}^{|I|} \dim(I_{S_i}) + 1} = \frac{4}{2 + \max\{\dim(I_R), \dim(I_S)\} + 1}$$
$$= \frac{4}{2 + \max\{2, 3\} + 1} = \frac{4}{6} = \frac{2}{3}.$$

◁

7.1.2.4 Sparsity of Iteration Variable Coefficients

We could substantiate the widely held conjecture, that the sparsity of schedule matrices sampled by Polyite using Chernikova sampling (compare Section 4.6.5) correlates to the probability of finding an effective schedule [61]. When sampling schedule coefficient vectors from polyhedra with Chernikova sampling, we construct each vector as a linear combination of a set of generators (points, rays, and lines). We can control the sparsity of the schedule matrix by controlling the number of generators with non-zero coefficients.

A schedule matrix that contains a large share of non-zero iteration variable coefficients (the $\vec{\lambda}$ parts in the statement schedule matrix rows) skews many loops. While skewing can enable parallelization and tiling, it also complicates the expressions in memory access functions and loop boundaries. Thus, needless skewing is detrimental to good performance. Also, non-zero coefficients of structure parameters may complicate the detection of statements' textual order.

For feature F_{SpIter}, we analyze the iteration variables coefficients in all band nodes' partial schedules. We determine the number of coefficients that are zero and divide this number by the total number of iteration variable coefficients. We extract the feature from T_S. Thereby, we avoid to incorporate as many of the coefficients that have no influence on the transformed code as possible, but also miss some coefficients that only become relevant after tiling. We could go a step further and incorporate only the iteration variable coefficients that occur in those parts of partial schedules that correspond to loops in the transformed program. On the other hand, we would neglect additional iteration variable coefficients that could become relevant after tiling.

Example 7.1.8. The total number of iteration variable coefficients in schedule tree T_S (refer to Figure 7.3(a)) of our running example is 13. Eight of them are zero. Thus, we have

$$F_{\text{SpIter}} = \frac{8}{13}.$$

$\triangleleft$

7.1.2.5 Sparsity of Structure Parameter Coefficients and the Constant

Analogously to the sparsity of the iteration variable coefficients, we can determine the sparsity of structural parameters and the constant parts of the schedules. That is, per dimension of a band node's partial schedule, we must examine the coefficients of the structure parameters and the constant (the $\vec{\mu}$ and ν parts in the statement schedule matrix rows). Feature F_{SpP} relates to partial fusion of statements, loop distribution that we cannot detect with the heuristic detection of statements' textual ordering in Section 5.3, and also loop peeling [143]. Loop peeling refers to splitting off a small number of first or last iterations of a loop. This can move the potentially costly identification and treatment of special cases that occur only in the computations at the border of an iteration space out of the loop. Loop peeling is a special case of loop splitting [64].

Example 7.1.9. The schedule tree of the schedule in Example 7.1.1 (refer to Figure 7.3(a)) does not contain any non-zero coefficients of structure parameters and the constant 1. Therefore, the features value is 1. $\triangleleft$

7.1.3 Performance-Related Features of Schedule Trees

The features introduced in Section 7.1.2 represent primarily the structure of a schedule tree, but have no immediate relation to specific performance aspects. Yet, we could relate most of them to certain aspects that do affect performance loosely. In the following, we present four schedule features that have an immediate relation to performance. They refer directly to coarse-grained parallelism, applicability of tiling, and temporal and spatial data locality. For the design of these features, it is important to know precisely how the polyhedral code generator that will be used to apply schedules to the program to be optimized, will process the schedules further, for instance by tiling, and how it will optimize the transformed code further, for example by parallelizing loops. The code generator on which POLYITE relies is POLLY in the version of commit `2b618e01` of `http://llvm.org/git/polly.git` from Jan. 27, 2016.

We start by describing one of the features that rely on the internal behavior of POLLY. The feature relates to coarse-grained parallelism.

7.1.3.1 Parallel Loops

Loop parallelization exploits the ability of multicore CPUs and SMT processors to run multiple execution threads in parallel. Creating and synchronizing threads both have a cost. This cost can be reduced by reducing the number of synchronization points between threads that run in parallel and by ensuring that the amount of computation performed by the threads justifies their creation. Thus, parallelizing an outer loop of a loop nest is likely to be more effective than parallelizing an inner loop. POLLY conforms to this argumentation: the code generator will parallelize the outermost loop in each loop nest that does not carry a data dependence. Yet, a second preconditions to an effective loop parallelization exists: single-thread locality, which means that each execution thread mostly operates on its own small enough set of data. Single-thread locality is important for parallelization on multicore

Listing 7.1: One loop with a guard for X. The loop cannot be parallelized.

```
for (int i = 0; i < 2 * n; i += 1) {
  if (n >= i + 1)
    X(i);
  Y(i);
}
```

Listing 7.2: Two loops without guards. The second loop can be parallelized.

```
for (int i = 0; i < n; i += 1) {
  X(i);
  Y(i);
}

#pragma omp parallel for
for (int i = n; i < 2 * n; ++i)
  Y(i);
```

processors to scale well [27]. If the second precondition is not met, simultaneously running threads will compete for the memory bus and for levels in the processor's cache hierarchy that are shared among the processor's cores (refer to Section 2.1.1). Specifically, the threads will mutually cancel each other's local data set from the shared cache.

Other than the parallelism feature proposed by Danner [46], our feature does not take the overhead yielded by parallelization, nor the interaction of parallelization with tiling into account. Tiling of a loop nest changes the effect of parallelizing the outermost loop in the nest that does not carry data dependences as it changes the amount of computation carried out between thread synchronization points. Also, we do not account for the interaction with data locality. We cover both tiling and data locality by separate, dedicated features.

In the design of our parallelism feature, we rely on the detection of schedule dimensions that do not carry data dependences in band nodes' partial schedules described in Section 5.5.3. Also, we use the detection of parts of band nodes' partial schedules that encode loops (refer to Section 5.5.1). Due to the heuristic nature of the parallelism detection, which we discuss in Section 5.6 our parallelism feature is approximative. Specifically, it will not be sensitive to cases of parallelism in the presence of partially fused loops, in which one of two neighboring loops is parallel and the other must be executed sequentially. Similarly, the fact that we are unable to determine statements' textual order in some situations prevents us from detecting parallelism in some cases. Furthermore, it cannot account for parallel loops that were not present before tiling. Attempting to detect all present parallelism would make the parallelism feature's value dependent on the values of structure parameters and loop iterators of encasing loops. Also, it would become dependent on the choice of tile sizes. Without the knowledge of all structure parameters' values, the knowledge of the tile sizes, and the ability to precisely predict the code generator's behavior, a precise feature value cannot be calculated. We illustrate the latter problem with Example 7.1.10.

Example 7.1.10. Consider the statement iteration domains

$$I_X = \{i \mid i, j \in \mathbb{N} \ \wedge \ 0 \le i < n\}$$
$$I_Y = \{i \mid i, j \in \mathbb{N} \ \wedge \ 0 \le i < 2 \cdot n\}, n \in \mathbb{N}$$

and the following data dependence polyhedron:

$$D_{X,X} = \{(i, i+1)^T \mid i \in I_X \ \wedge \ (i+1) \in I_X\}.$$

From the iteration domains, the dependence polyhedron and the schedule

$$\Theta_X(i) = (i, 0)^T, \ \Theta_Y(i) = (i, 1)^T.$$

A polyhedral code generator may generate the code in Listing 7.1[10] or the code in Listing 7.2. While we cannot parallelize in the case of Listing 7.1, half of the iterations of Y in Listing 7.2 can be executed in parallel. Such details of a code generator's behavior must be considered volatile and are therefore difficult to account for by an external optimizer. ◁

[10]This is the code produced by ISL, commit `cfebc0c6` (Dec. 11, 2015), with the default options applied.

Per statement, we number the dimensions of its schedule that correspond to loops in the generated code. We start from zero at the outermost such dimension. We then weigh each statement by the index of the outermost parallel and loop generating dimension in its schedule. In future, the feature may be improved by weighing each statement by its amount of computation. An estimation of the amount of computation has to take the statement's number of iterations into account, which requires knowledge of structure parameters' values, but also the number and kind of operations performed by the statement, and, potentially, the number and pattern of memory accesses. Weighing the statements in such a way may increase the feature's expressiveness.

Given statement S_i, we define δ_{S_i} to be the index of the outermost fully parallel loop-generating dimension in Θ_{S_i} and ω_{S_i} to be some weight that expresses the statement's amount of computation. The normalized feature is:

$$F_{\text{Par}} = \frac{\sum_{i=1}^{|I|} \omega_{S_i} \cdot \dfrac{\dim(I_{S_i}) - \delta_{S_i}}{\Delta_{S_i}}}{\sum_{i=1}^{|I|} \omega_{S_i}}.$$

Δ_{S_i} is either the dimensionality of the iteration domain of S_i or 1 in the absence of loops. By the latter, we avoid dividing by zero. Generally, Δ_{S_i} is an upper bound for the number of loops that encase S_i in the transformed program, since a schedule may eliminate some loops. In the current implementation of Polyite, we do not weigh statements by their amount of computation and, therefore, set $\omega_{S_i} = 1$ for all $i \in \{1, \ldots, |I|\}$. We extract the feature from T_S because the parallelism exposed by that schedule tree is maximal.

Example 7.1.11. The outermost band node of the schedule tree in Figure 7.1.1 encodes a parallel loop around both statements of our running example `syrk`. Thus, we have $\delta_R = \delta_S = 0$ and

$$F_{\text{Par}} = \frac{\sum_{i=1}^{|I|} \omega_{S_i} \cdot \dfrac{\dim(I_{S_i}) - \delta_{S_i}}{\Delta_{S_i}}}{\sum_{i=1}^{|I|} \omega_{S_i}} = \qquad \triangleright\ \omega_R = \omega_S = 1$$

$$\frac{\sum_{i=1}^{|I|} \dfrac{\dim(I_{S_i}) - \delta_{S_i}}{\Delta_{S_i}}}{\sum_{i=1}^{|I|} 1} = \frac{\dfrac{\dim(I_R) - \delta_R}{\Delta_R} + \dfrac{\dim(I_S) - \delta_S}{\Delta_S}}{|\{I_R, I_S\}|} = \frac{\dfrac{2-0}{2} + \dfrac{3-0}{3}}{2} = \frac{1+1}{2} = 1.$$

$\triangleleft$

An improved definition of F_{Par} would not take statements into account that are not encased by loops (i.e., statements that have a singleton iteration domain).

7.1.3.2 Tilable Schedule Bands

Tiling is a transformation of loop nests that can improve data locality and increase the effectiveness of coarse-grained loop parallelism [73]. As recalled in Section 2.2.2.4, rectangular tiling changes the execution order of loop nests. Tiling blocks the loop in the nest with a given block size, thereby doubling the number of loops- Subsequently, the loops are interchanged such that the loops that enumerate the blocks become the outer loops (the *tile loops*) of the nest while the loops that iterate inside the blocks become the inner loops (the *point loops*). The result is a partition of the loop nest's iteration space into blocks. The tile loops enumerate the tiles while the point loops iterate inside the tiles. The concept is generalizable to imperfect loop nests [171]. This transformation is applicable iff the schedule dimensions to be tiled form a permutable band: the dimensions must be interchangeable without violating data dependences. There are situations in which tiling is detrimental to performance [141]. Generally, the success of tiling depends on the proper choice of tile sizes, that is the block size per tiled loop. Tile sizes can be determined by empirical search [163], analytically [127], or by the combination of an analysis and limited empirical search [139].

Section 5.5.2, describes how we identify sequences of band nodes in schedule trees whose partial schedules correspond to sequences of permutable loops. In this process, we replace such sequences of band nodes by single band nodes, whose partial schedule represents the partial schedules of the replaced nodes. We mark the new nodes as representing permutable bands. We must now identify those permutable band nodes that will be tiled by POLLY. POLLY's tiling transformation operates on schedule trees. In the version of POLLY that we use, the transformation tiles permutable band nodes whose children are leaves. From version 5.0 (of July 2017) on, Polly tiles also permutable band nodes whose children are sequence nodes that have only leaves as their children. This increases the number of cases in which tiling will be applied after loop fusion. POLYITE supports both tiling criteria. We can partially avoid this shortcoming of older versions of POLLY by encoding textual ordering in loop bodies with shifted strides (refer to Section 5.3).

We rely on the detection of loop-generating parts of band nodes' partial schedules that is described in Section 5.5.2 and weigh each statement S_i by the number λ_{S_i} of loops that encase S_i and that are encoded in the tilable band that is associated to S_i. The normalized tiling feature is defined as follows:

$$F_{\text{Tile}} = \frac{\sum_{i=1}^{|I|} \frac{\lambda_{S_i}}{\Delta_{S_i}}}{|I|}.$$

We extract the feature from T_C. Thereby, we operate on the same schedule tree as POLLY's tiling transformation.

Danner's tiling feature [46] differs from ours in the weighing of the statements and the normalization of the feature value. Like us, Danner does not take the computational overhead into account that results from the tiling of a loop nest.

Example 7.1.12. From the schedule tree T_C in Figure 7.3(b), we know that all dimensions of the schedule in our running example can be gathered in one permutable band node. Thus, we have $\lambda_R = \dim(I_R) = 2$ and $\lambda_S = \dim(I_S) = 3$ and, therefore,

$$F_{\text{Tile}} = \frac{\sum_{i=1}^{|I|} \frac{\lambda_{S_i}}{\Delta_{S_i}}}{|I|} = \frac{\frac{\lambda_R}{\Delta_R} + \frac{\lambda_S}{\Delta_S}}{|\{I_R, I_S\}|} = \frac{\frac{2}{2} + \frac{3}{3}}{2} = \frac{1+1}{2} = 1.$$

$\lhd$

F_{Tile} can be improved by not taking statements into account that are encased by less than two loops (i.e., $|I_S| < 2$ for a statement S).

7.1.3.3 An Approximative Data Locality Feature

Processors access memory through a hierarchy of cache levels (refer to Section 2.1.1). Also, reading data from a memory address does not result in a single value being transferred to the CPU's cache, but an entire cache line (that is, a block of data). Since cache size is limited, a replacement strategy must make space for new cache lines. Thus, to profit from caching, cache lines must be re-accessed with few accesses to other cache lines in between.

For a data locality feature, an exact model of cache behavior would be ideal. However, the computations involved in such a model seem too complex for practical use [14, 37, 46]. It is noteworthy that recent work on analytical modeling of caches by Bao et al. [14] can handle multi-level cache hierarchies and shows a promising tendency in observed time complexity.

In consequence, we propose a computationally cheaper feature. The idea is that, the larger the volume of data communicated by the dependences in a dependence polyhedron $D_{O,T}$ is in terms of the number of involved memory cells (i.e., cache footprint), the more we can profit from executing the pairs of dependent statement instances modeled by $D_{O,T}$ with few other statement instances being executed in between. Consequently, the larger the volume of data communicated by $D_{O,T}$, the more deeply $D_{O,T}$ should be carried in the loop nest. This is relevant not only in the context of sequential execution, but particularly if we can execute an outer loop of a nest in parallel: we reduce the probability of execution threads mutually evicting data from shared cache levels that they would reuse otherwise. In this context, read-after-read dependences are relevant, too. The set of read-after-read dependences is specific to a schedule and, therefore, cannot be precomputed. To reduce the cost of computing the read-after-read dependences for a given schedule, we do not eliminate transitive dependences during their computation. Let O and T be two statements in a SCoP and let R_O, R_T be the respective sets of reading memory access functions. We calculate the set of dependence polyhedra that correspond to read-after-read dependences between two statements O and T and a schedule Θ as follows:

$$\Big\{\text{conv.hull}(D) \mid D \in \Lambda\Big(\Big\{\begin{pmatrix}\vec{i}_O\\ \vec{i}_T\end{pmatrix} \mid \vec{i}_O \in I_O \ \wedge\ \vec{i}_T \in I_T \ \wedge\ \Theta_O(\vec{i}_O) \prec \Theta_T(\vec{i}_T) \ \wedge$$
$$\big(\exists r_O \in R_O, r_T \in R_T : r_O \in (I_R \to A) \ \wedge\ r_T \in (I_T \to B) \ \wedge\ A = B \ \wedge\ r_O(\vec{i}_O) = r_T(\vec{i}_T)\big)\Big\}\Big)\Big\}.$$

We miss out on read-after-read dependences if a schedule leaves the textual ordering of two statements undetermined between which no legality-affecting data dependence exists. Furthermore, we cannot account for read-after-read dependences that were not present before tiling or for the case that the order of two reads changes due to tiling.

To estimate the communicated data volume of a dependence polyhedron, we evaluate, per array or scalar accessed (a *scalar* is a zero-dimensional array), the number of array cells that are accessed by both the dependence polyhedron's source and target statement instances. We distinguish between reads and writes. Analogously to R_S as the set of all reading memory access functions of statement S, let W_S be the set of all writing memory access functions of S. Let M be the set of all arrays and scalars in the SCoP. The estimated communicated data volume of a dependence polyhedron $D_{O,T}$ is:

$$\eta_{O,T} = v(R_O, R_T) + v(W_O, R_T) + v(R_O, W_T) + v(W_O, W_T)$$

with

$$v(A_O, A_T) = \sum_{L \in M} \left| \left(\bigcup_{f \in A_O} \Big\{ f(\vec{i}) \mid \vec{i} \in I_O \ \wedge\ f \in (O \to L) \ \wedge\ (\exists \vec{j} : \begin{pmatrix}\vec{i}\\ \vec{j}\end{pmatrix}) \in D_{O,T} \Big\} \right) \right.$$
$$\left. \cap \left(\bigcup_{f \in A_T} \Big\{ f(\vec{i}) \mid \vec{i} \in I_T \ \wedge\ f \in (O \to L) \ \wedge\ (\exists \vec{j} : \begin{pmatrix}\vec{j}\\ \vec{i}\end{pmatrix}) \in D_{O,T} \Big\} \right) \right|.$$

We calculate an upper bound for the number of values that would have to be cached at any time for fast reuse among all memory accesses of O and T that are involved in $D_{O,T}$. The distinction between reads and writes overexaggerates the feature's value if reads and writes occur simultaneously. Thereby, the feature also expresses whether there are read and write accesses to an array or scalar or only one kind of access. At the same time, the order of magnitude of the estimated communicated data volume remains unaffected. To calculate $\eta_{O,T}$, we rely on Barvinok's counting algorithm [157] (refer to Section 2.2.1.5). For the determination of these values, the value ranges of all memory access functions' arguments must be known (it suffices to know the values of all structure parameters that occur in loop bounds.). These values can be determined by instrumenting the memory accesses in the SCoP to be optimized and executing the instrumented program with a workload that is representative for the intended use case.

Further, we determine, per dependence polyhedron $D_{O,T}$, the innermost schedule dimension $d_{O,T}$ that carries dependences in $D_{O,T}$. Alternatively, we could determine the outermost, or the average dimension that carries dependences in $D_{O,T}$. Another option, which bears a higher computational cost, is to partition dependence polyhedra by the dimension that carries a dependence. Subsequently, each set in the partition would be associated with exactly one schedule dimension and we would obtain a more precise data locality feature.

Assuming that the set of all data dependences yielded by the given SCoP and schedule is modeled by k dependence polyhedra $D_{O_i,T_i}, i \in \{1, ..., k\}$, we define the following normalized data locality feature:

$$F_{\text{DataLoc}} = \frac{\sum\limits_{i=1}^{k} (\eta_{O_i,T_i} \cdot d_{O_i,T_i})}{\dim(\Theta) \cdot \left(\sum\limits_{i=1}^{k} \eta_{O_i,T_i}\right)}.$$

Because dependence polyhedra can be overapproximations of originally non-convex sets (refer to Sections 2.2.2.3 and 4.3) the computed numbers of accessed memory cells may exceed the true number of accesses.

We could increase the feature's expressiveness by multiplying the calculated numbers of array cells by the bit width of the respective arrays' data types. In the case of the benchmark set PolyBench 4.1, which we used in our experiments, this improvement has only marginal influence: `nussinov` is the only one of 30 programs in PolyBench 4.1 that uses arrays of more than one type. `nussinov` operates on a `char` array of size $n \in \mathbb{N}$ and an array of `int` values of size n^2.

We extract the feature from a schedule matrix. This schedule matrix results from retransforming T_S to the matrix representation.

Example 7.1.13. Setting the structure parameters in our running example `syrk` to $n = 2600$ and $m = 2000$, which corresponds to the use of the extra large data set configuration of PolyBench 4.1, yields $F_{\text{DataLoc}} \simeq 0.62$ for the schedule from Example 7.1.1 when using the feature's current implementation. ◁

7.1.3.4 Memory Access Pattern

Section 7.1.3.3 discusses that a cache miss during a memory access does not result in an individual memory cell's value being transferred to the processor's caches, but in the loading of an entire cache line. This yields another type of data locality, namely spatial locality: an efficient program does not access memory in random patterns but, ideally, in a pattern along the innermost dimensions of the arrays accessed. In addition, the memory accesses should have a small stride. Ideally, this stride is positive: in this case, the program may profit from Intel's data cache unit (or streaming) prefetcher that prefetches to the L1 data cache [72]. A program that has these properties can profit from the concept of cache lines: a cache miss will be followed by a series of cache hits that ends only when an access goes beyond the boundary of a recently loaded cache line. Hence, CPU caches provide a prefetching mechanism from which programs with good spatial data locality can profit.

To account for spatial locality in our surrogate performance models, we determine the number $\mathcal{A}'$ of statements' memory accesses of a SCoP that have a forward pattern along the innermost array dimension, or that go to the same memory cell for one iteration of the respective loop nest's second-innermost loop.

Let us explain in detail how we determine whether a memory access in a transformed program exhibits one of the two patterns stated above. Let $\Theta_S : \mathbb{Z}^n \to \mathbb{Z}^m$ with $n, m \in \mathbb{N}$ be a schedule for a statement S that is injective. Consequently, each loop of the transformed program that encases S is explicitly encoded in Θ_S. Let $f : \mathbb{Z}^n \to \mathbb{Z}^k$ with $k \in \mathbb{N}$ be the subscript function of one of the array accesses of S. We determine the coefficient matrix M of

$f \circ \Theta^{-1}$ with Θ^{-1} being the inversion of Θ. Further, we determine the innermost dimension d^* of Θ that corresponds to a loop. This loop is the innermost loop in the transformed code that encases S. The column of M that contains the coefficients for dimension d^* of the range of Θ_S must contain 0 in all rows except the last. If the coefficient in the last row is 0, all accesses in one execution of the innermost loop go to the same array cell. If the coefficient in the last row is positive, the access pattern is forward along the innermost array dimension.

With $\mathcal{A}$ being the total number of different memory access functions in the SCoP the normalized feature is

$$F_{\text{MemAcc}} = \frac{\mathcal{A}'}{\mathcal{A}}.$$

We extract the feature from T_S.

Example 7.1.14. We demonstrate F_{MemAcc} by continuing Example 7.1.1. Listing 7.3 shows the code of `syrk` after transformation by the schedule in Equation 7.1.

Listing 7.3: The code of our running example `syrk` after transformation by the schedule in Equation 7.1.

```
for (int i = 0; i < n; i++) {
 for (int j = 0; j <= i; j++)
   C[i][j] *= beta; // statement R
 for (int k = 0; k <= i; k++)
   for (int j = 0; j < m; j++)
     C[i][k] += alpha * A[i][j] * A[k][j]; // statement S
}
```

After the transformation of the program by the schedule in Equation 7.1, the accesses of statement R to array `C` follow a forward pattern along the innermost array dimension. The accesses to array `C` by statement S in one execution of the innermost loop that encases S go to the same array cell. The accesses of S to array `A` follow a forward pattern along the innermost array dimension. Obviously, all accesses to the scalars `alpha` and `beta` go to the same respective memory address. Thus, we have $F_{\text{MemAcc}} = 1$. ◁

7.1.4 Discussion

The features that we have proposed in Section 7.1 cover the following performance aspects of schedules: (partial) fusion and distribution of statements, sparsity of the partial schedules of band nodes in schedule trees, parallelism, temporal and spatial data locality, and tiling. Danner [46] also covered out-of-order execution and the computational overhead yielded by conditional statements in loop bodies. While some of the features are meaningful only for SCoPs with more than one statement, no feature explicitly characterizes SCoPs. Thus, we avoid performance models that can be transferred only between programs of very similar structure. All features can be normalized (approximately) to the interval $[0, 1]$.

Currently, we do not cover loop vectorization in our feature set. During code generation, LLVM may vectorize loops if possible, but we keep Polly's more powerful polyhedral vectorization strategy that relies on strip-mining [162] inactivated. Activating polyhedral vectorization will likely make an additional vectorization feature necessary. The same holds for any other loop transformation that requires an enabling transformation. One such transformation is diamond tiling [29]. Our features cover multi-level tiling [80] in the rectangular case already. The criterion for rectangular tiling is sufficient for unroll-and-jam, which was noted by Sarkar [135] for the case of perfect loop nests. Other transformations, such as simple loop unrolling, require no extra feature.

Our schedule features are meaningful for optimizing loop programs that can be expected to profit from tiling and coarse-grained parallelization for execution on multicore CPUs. An optimization for single-threaded processors, GPUs, or Intel Xeon Phi would require a different set of features. Finding schedules that permit offloading of computations to

GPUs, for instance by using the PPCG compiler [158] as code generator, would require the restriction of the considered set of schedules to ones that yield tiles that can be computed in parallel. Otherwise, one would encounter numerous schedules without this property.

To evaluate the run-time complexity of feature extraction empirically, we used the partial loop unrolling technique proposed by Upadrasta and Cohen [148]. This technique allows us to scale the number of statements, schedule coefficient space dimensions, and dependence polyhedra of a SCoP. We measured the duration of schedule tree construction and simplification (we only constructed T_S and extracted all features from that representation) and of feature extraction. Per benchmark and number of unrolled loop iterations, we determined the algorithms' average execution times across 100 schedules generated randomly. The algorithms have been implemented in Scala 2.11 [110] and were executed on Oracle JDK 8 on an Intel®Xeon®E5-2690 v2 CPU @ 3.00GHz CPU with ten physical cores and 25MB of L3 cache. The operating system was Debian 9 with Linux 4.9. We used isl (commit `cfebc0c6` (Dec. 11, 2015) of `https://repo.or.cz/isl.git`) [152] and libbarvinok [157] (commit `91ba8f18` of `https://repo.or.cz/barvinok.git`, May 26, 2018) to represent polyhedra. To model the unrolled loop nests as SCoPs, we used Polly in the version of commit 2b618e01 (Jan. 27, 2016) of `http://llvm.org/git/polly.git`. Figure 7.8 shows exemplary results for several programs from PolyBench 4.1. Per program, we show the duration of the schedule tree transformation and simplification and the duration of feature extraction. Additionally, we show the duration of computing the data locality feature. We excluded F_{SpP} from the evaluation since we found that this feature is mostly insensitive to performance in other experiments (refer to Section 9.4).

While all other features can be extracted from the simplified schedule tree very quickly, the duration of computing F_{DataLoc} grows significantly with SCoP size. By analyzing the algorithm to calculate F_{DataLoc}, we found that the costliest part is to determine the communicated volumes of data by the use of Barvinok's counting algorithm (refer to Section 2.2.1.5). The cost of this step depends strongly on the number of memory access relations in the SCoP that address the same memory location. We mitigate this cost by caching the communicated data volumes of legality-affecting data dependence polyhedra already. This caching was enabled during the experiments reported above and the caches were initialized in a warm-up run. Despite the exponential run-time complexity of Barvinok's counting algorithm, we believe that its use in our data locality feature is not problematic since the dimensionality of the input polytopes is bounded by the dimensionality of the arrays accessed by the SCoP's statements. Their dimensionality is typically low. A data locality feature that is cheaper to compute could rely on an estimation of statements' memory traffic [18, 124], which can be precomputed entirely.

To cover additionally programs that do not profit from coarse-grained parallelism and tiling, we could include SCoP features such as the classification techniques for algorithms that have been proposed by Nugteren et al. [106] and Sioutas et al. [141]. Sioutas et al. [141] distinguish between loop nests that profit from temporal data locality and programs that profit from spatial data locality. The algorithm species by Nugteren et al. [106] is more finely grained. An *algorithm species* is a combination of five different array access patterns. In an outlook to compilation assisted by machine learning, Pouchet [119] states further program features, among them the number of statements, the maximum loop nest depth, a characterization of the data dependences in the program, and, as another category of features, a classification of programs by the kind of transformations and optimizations that are applicable to them. By using a large and comprehensive set of training programs and schedules, one may then be able to learn a widely applicable performance model. Depending on the class of program, the configuration of the polyhedral code generator has to be modified (for instance, by enabling or disabling loop parallelization).

As indicated in the introduction of Section 7.1 and illustrated in Section 5.6, we make a tradeoff between reduced run-time complexity and algorithmic complexity on the one hand,

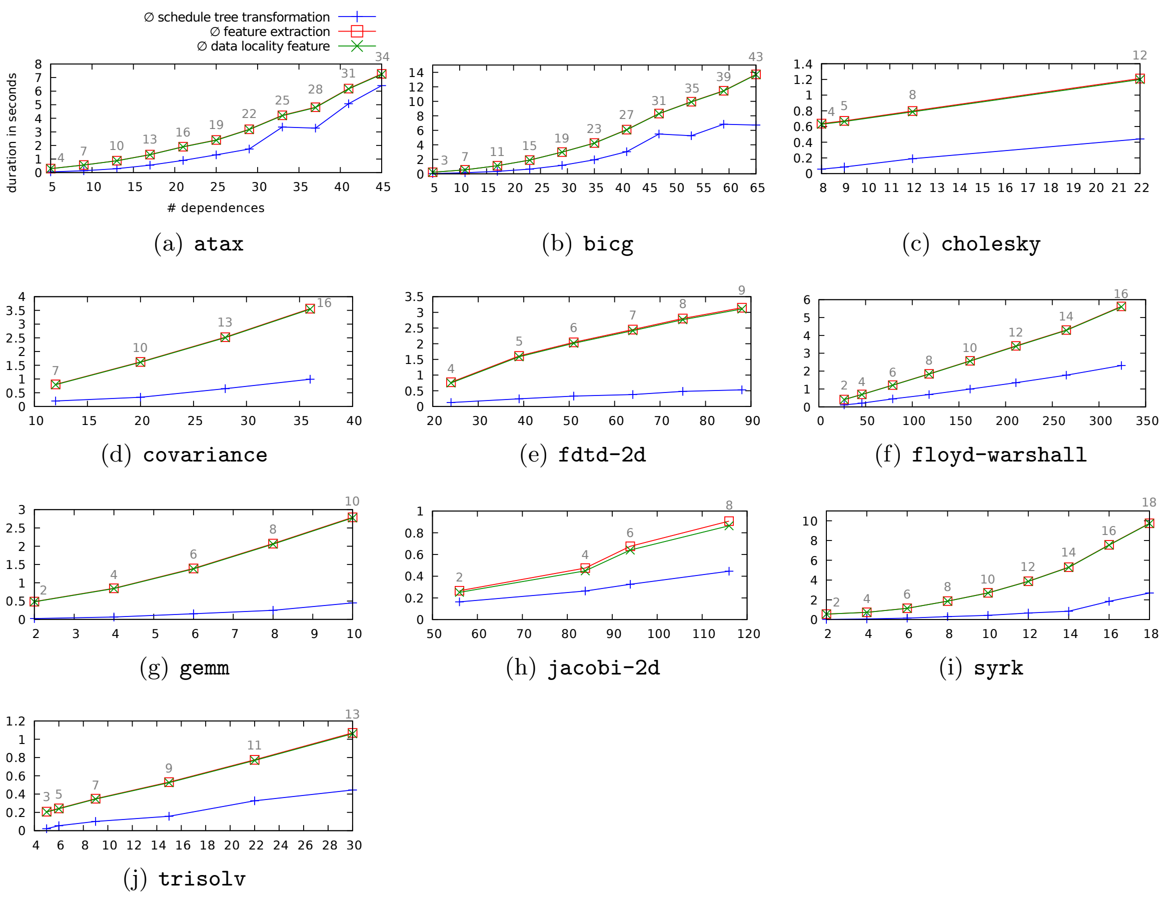


Figure 7.8: The duration of schedule tree transformation and simplification, and the duration of feature extraction relative to the number of data dependence polyhedra in a SCoP. We show the duration of calculating F_{DataLoc} separately to illustrate that it dominates the total duration of feature extraction. Above each data point, the number of statements in the respective SCoP is shown. Only the points correspond to measured values. They have been connected by lines to improve perceptibility.

and accuracy on the other hand by extracting feature vectors from schedules before the encoding of tiling. An empirical evaluation (refer to Section 9.4) must study whether the learned model can distinguish profitable from unprofitable schedules sufficiently well despite the tradeoff made. For instance, after tiling, a different schedule dimension may correspond to a specific loop of the transformed program, which may affect the accuracy of F_{Par} or F_{MemAcc}.

7.2 Learning a Schedule Classifier and Integrating it with the Genetic Algorithm

To learn a model for a schedule classifier, we use either CART or random forest (refer to Section 2.3.2). Both learning methods are supervised machine-learning techniques and, therefore, need labeled training data. For the intended approach, the classifier must be able to identify profitable schedules for a given SCoP. Thus, we must label schedules in the training set as either profitable or unprofitable. The generation of training data takes a set of schedules for a given SCoP as input. Per schedule, the execution time of the respective transformed SCoP must be known. Further, the execution time of the original code must be known. Here, we use the measured execution time of the binary that results

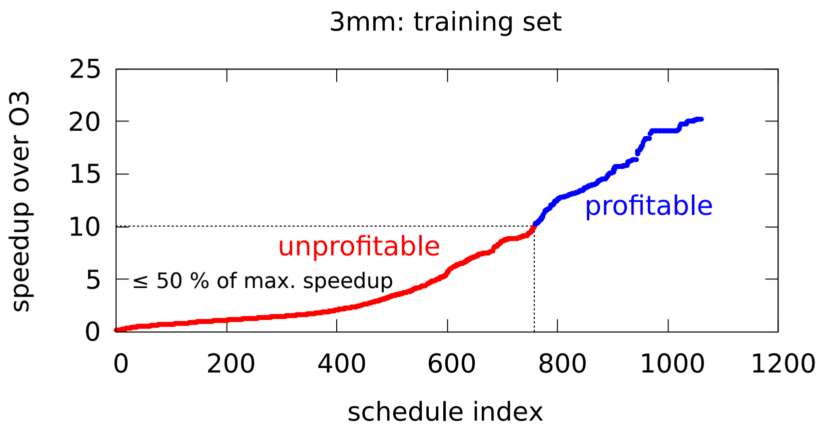


Figure 7.9: Labeling of schedules for classifiers' training data. The horizontal axis enumerates the schedules in a set of schedules for program `3mm` of PolyBench 4.1. The schedules are ordered by yielded speedup in execution time over `-O3`. The vertical axis shows this speedup. All schedules that yield a speedup smaller than 0.5 times the speedup yielded by the most effective schedule are labeled as unprofitable (lower left corner), all other schedules are labeled as profitable (upper right corner).

from the compilation of the original code with clang and `-march=native -O3`. We label all schedules as "profitable" that yield a speedup over the execution time of the original code compiled with `-O3` that is at least 50% of the highest speedup yielded by any schedule in the given set of schedules. We illustrate this labeling strategy in Figure 7.9. If all given schedules for a SCoP yield a speedup smaller than 1.2, we assume that this SCoP does not profit from the optimization with Polyite for an execution on multicore CPUs and the given configuration of Polyite and Polly/LLVM. This threshold was chosen to identify programs for which the maximum speedup yielded by Polly and Polyite is very low and the variance of the speedups yielded by the schedules in the respective training sets is low, as well. We wanted to avoid labeling many schedules in their training sets that are unprofitable in reality as profitable.

A classifier that labels schedules as "profitable" or "unprofitable" can be combined with the genetic algorithm that we propose in Chapter 6 in different ways. In the following, we describe two possible combinations.

7.2.1 Classification as a Guard for Benchmarking

A conservative approach to combining the classification of schedules with our genetic algorithm is to use the classifier as a guard for benchmarking. Instead of applying all schedules in the genetic algorithm's current population and measuring the transformed program's execution time in each case, one can classify the schedules and evaluate further only the schedules that the classifier labels as profitable. Following the genetic algorithm schema that is described in Section 6.1, we select the schedules evaluated by benchmarking that yield the shortest execution times until we have selected half of the schedules in the current population. If less than half of the schedules in the current population has neither been successfully evaluated by benchmarking nor is being considered profitable by the classifier, we fill up the set of schedules to be benchmarked with unprofitable schedules chosen randomly. Thereby, we retain the population's diversity. Figure 7.10 illustrates the use of a schedule classifier as a guard for benchmarking.

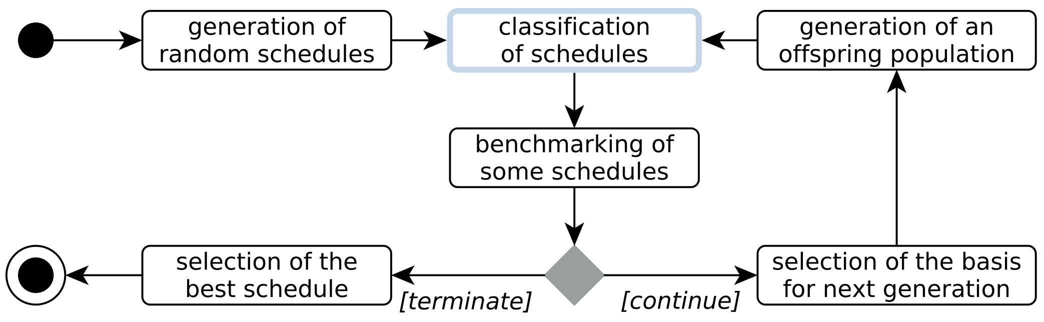

generation of random schedules
classification of schedules
generation of an offspring population
benchmarking of some schedules
selection of the best schedule
[terminate]
[continue]
selection of the basis for next generation

Figure 7.10: A classifier can be used as a guard for the benchmarking of schedules.

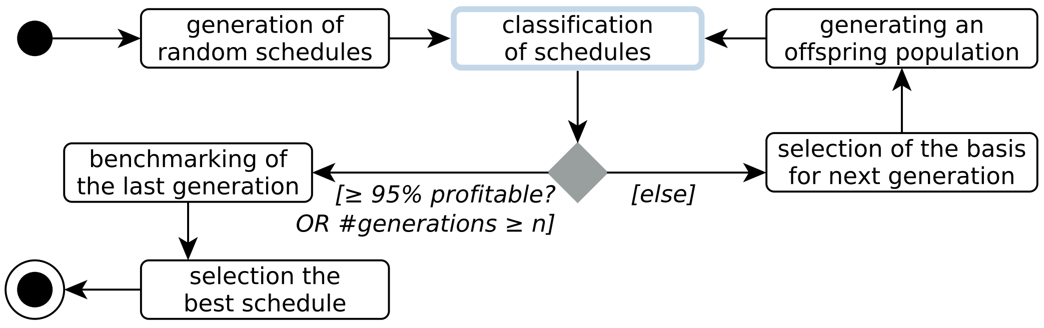

generation of random schedules
classification of schedules
generating an offspring population
benchmarking of the last generation
[≥ 95% profitable? OR #generations ≥ n]
[else]
selection of the basis for next generation
selection the best schedule

Figure 7.11: Compared to the procedure illustrated in Figure 7.10, the duration of the genetic algorithm can be reduced further by abstaining from benchmarking entirely until the genetic algorithm's last generation.

7.2.2 Two-Staged Approach

Remember our observation at the beginning of Chapter 7 that profitable schedules are likely to occur among the schedules generated by Polyite under the precondition that the search space exploration is biased to explore primarily specific subsets of the search space of legal schedules by configuration. A faster approach to integrating the classification into the genetic algorithm that is described in Section 6.1 is motivated by this observation: instead of reducing the benchmarking effort by benchmarking only schedules that the classifier labels as profitable, we may use classification alone until almost all schedules in the genetic algorithm's population are labeled as profitable.

We evaluate all yet unevaluated schedules in the current population by classification. Thereafter, we select randomly a subset of the profitable schedules that contains half as many schedules as the regular population size and use it as the subsequent population's basis. Since there may be few profitable schedules in the genetic algorithm's early populations, we may have to fill up the elected set of schedules with randomly chosen unprofitable schedules to retain the population's diversity. This process continues until half of the population has been selected. If 95% of the schedules in the population are considered profitable by the classifier or a maximum number of generations has been produced, the genetic algorithm is terminated and the most profitable schedule in the genetic algorithm's final population is determined by benchmarking. Figure 7.11 illustrates this procedure. In the sequel of this thesis, we refer to this variant of our genetic algorithm as GA_C.

Compared to GA_B and the approach presented in Section 7.2.1 that uses classification as a guard for benchmarking, the probability that GA_C keeps the most profitable schedule visited during its search space exploration is reduced. It may be necessary to switch to one of the two other variants of our genetic algorithm after a certain number of generations.

8 Implementation

This chapter is an overview of our iterative program optimizer Polyite. Polyite and software on which it relies are available from `http://github.com/stganser`. We start with an introduction to Polyite in Section 8.1 and continue by describing its usage in Section 8.2. We go on with a description of existing software on which Polyite is based (refer to Section 8.3). In Section 8.4, follows an overview of Polyite's software architecture and a detailed discussion of the schedule evaluation process and of parallel processing within Polyite. Section 8.5 describes how we comply with the (super)-exponential asymptotic run-time complexity of several algorithms used in Polyite.

8.1 Overview

Polyite is a stand-alone iterative polyhedral schedule optimizer that is written in Scala [110] (version 2.11). Polyite can be used in combination with any polyhedral compiler that can export polyhedral models of SCoPs in Polly's [65] JSCOP format. JSCOP is a JSON (Java Script Object Notation) [30] format. We illustrate the format in Example 8.1.1.

Example 8.1.1. To illustrate the JSCOP format, we show the model of our running example `syrk` in Listing 8.1. The representation was exported by Polly in the version after commit `2b618e01` (Jan. 27, 2016) of `http://llvm.org/git/polly.git`. A JSCOP representation starts with the description of the SCoP's structure parameters, the *context* (line 2). The structure parameters `p_2` and `p_3` do not correspond to any actual structure parameter of the SCoP. We found that, with release versions of Polly that are newer than version 3.8, these additional structure parameters would not be present in the SCoP's model. Unfortunately, Polyite cannot reliably identify such artificial structure parameters and adds dimensions that correspond to them to the schedule coefficient space. An inspection of transformed programs revealed, that each of these additional structure parameters refers to either `n` or `m`, which means that adding a non-zero coefficient for an artificial structure parameter to a schedule matrix does not yield incorrect program behavior. We verified this behavior for a number of the programs in PolyBench 4.1. The name of the SCoP (line 3) and the list of the SCoP's statements follow (lines 4-51). Per statement, the list of its memory accesses (lines 6-19 and lines 25-46), its domain (lines 20 and 47), and its schedule (lines 22 and 49) is given. All statement schedules have the same dimensionality. As can be seen, JSCOP represents statement schedules by coefficient matrices (or linearly affine expressions). The representation of iteration domains, schedules, and memory access functions in JSCOP corresponds to the string representation of the data structures of the Integer Set Library (isl) [152]. ◁

Furthermore, the polyhedral compiler must be able to import schedules in our extended JSCOP format and apply these schedules to the SCoP to be optimized. We extended JSCOP by a schedule tree representation of the represented SCoP's schedule to be able to pass schedule trees to the polyhedral compiler. The extension is necessary because we delegate schedule optimizations such as tiling to the polyhedral compiler. Polly can apply tiling and any other schedule transformation only to schedule trees and is unable to transform imported schedules further without our modification.

As illustrated in Figure 8.1, we achieve the decoupling of Polyite and the polyhedral compiler by wrapping the compiler with a script that has a specified interface, but internally

Listing 8.1: JSCOP representation of `syrk`, as exported by Polly.

```
{
 "context" : "[n, m, p_2, p_3] -> {  : -2147483648 <= n <= 2147483647 and -2147483648 <= m <= 2147483647 and 0 <= p_2 <= 4294967295 and 0 <= p_3 <= 4294967295 }",
 "name" : "entry.split => for.end38",
 "statements" : [
    {
       "accesses" : [
          {
             "kind" : "read",
             "relation" : "[n, m, p_2, p_3] -> { Stmt_for_body6[i0, i1] -> MemRef_C[i0, i1] }"
          },
          {
             "kind" : "read",
             "relation" : "[n, m, p_2, p_3] -> { Stmt_for_body6[i0, i1] -> MemRef_beta[] }"
          },
          {
             "kind" : "write",
             "relation" : "[n, m, p_2, p_3] -> { Stmt_for_body6[i0, i1] -> MemRef_C[i0, i1] }"
          }
       ],
       "domain" : "[n, m, p_2, p_3] -> { Stmt_for_body6[i0, i1] : i0 < n and 0 <= i1 <= i0 }",
       "name" : "Stmt_for_body6",
       "schedule" : "[n, m, p_2, p_3] -> { Stmt_for_body6[i0, i1] -> [i0, 0, i1, 0] }"
    },
    {
       "accesses" : [
          {
             "kind" : "read",
             "relation" : "[n, m, p_2, p_3] -> { Stmt_for_body14[i0, i1, i2] -> MemRef_A[i0, i1] }"
          },
          {
             "kind" : "read",
             "relation" : "[n, m, p_2, p_3] -> { Stmt_for_body14[i0, i1, i2] -> MemRef_alpha[] }"
          },
          {
             "kind" : "read",
             "relation" : "[n, m, p_2, p_3] -> { Stmt_for_body14[i0, i1, i2] -> MemRef_A[i2, i1] }"
          },
          {
             "kind" : "read",
             "relation" : "[n, m, p_2, p_3] -> { Stmt_for_body14[i0, i1, i2] -> MemRef_C[i0, i2] }"
          },
          {
             "kind" : "write",
             "relation" : "[n, m, p_2, p_3] -> { Stmt_for_body14[i0, i1, i2] -> MemRef_C[i0, i2] }"
          }
       ],
       "domain" : "[n, m, p_2, p_3] -> { Stmt_for_body14[i0, i1, i2] : i0 < n and 0 <= i1 < m and 0 <= i2 <= i0 }",
       "name" : "Stmt_for_body14",
       "schedule" : "[n, m, p_2, p_3] -> { Stmt_for_body14[i0, i1, i2] -> [i0, 1, i1, i2] }"
    }
 ]
}
```

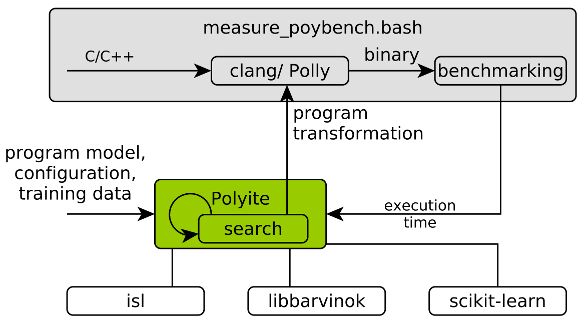


Figure 8.1: Tool chain of Polyite.

is specific to the compiler (in our case clang with Polly). The wrapper script does not only invoke the compiler, but it also measures the compiled program's execution time and verifies its computation result. The script that is currently implemented is specific not only to Polly, but also to the optimization of programs from PolyBench 4.1 [168]. This eased the preparation of our experimental evaluation. The presence of a generic wrapper script enables Polyite to optimize arbitrary SCoPs that can be modeled by Polly. The generality of Polyite's configuration options should support this goal. For practicality, loop nests that are to be optimized iteratively should be extracted into a small separate program and should be fed with artificial input data.

Internally, Polyite has a modular, extensible architecture. Alternative functionality, such as different sampling strategies for schedules (refer to Section 4.6) or the kind fitness assessment that is to be used by the genetic algorithm, can be selected by configuration.

8.2 Usage

To run Polyite, one must use one of the following three commands:

`run_genetic_opt` invokes an optimization by the genetic algorithm (refer to Chapter 6).

`run_rand_exploration` invokes random search space exploration (refer to Chapter 4).

`run_rand_exploration_letsee` invokes random exploration using an adaptation of the search space construction by Pouchet et al. [124] (refer to Section 4.8).

Each command takes two command line parameters:

```
> ./run_genetic_opt  syrk.jscop  syrk_ga_config.jscop
```

The first parameter specifies a file that contains the JSCOP representation of the SCoP that is to be optimized. The second parameter is a configuration file. It configures the behavior of Polyite. Among the configuration options are the sampling strategy to be used, the number of generations for the genetic algorithm and its population size, and the script that is to be called for the benchmarking of schedules. Polyite can be configured to import an existing set of schedules from a file and start or continue its genetic algorithm from the schedules given. Moreover, in random exploration mode, Polyite can import a given set of schedules and benchmark or classify all of the given schedules that have not been evaluated already. For the purpose of importing schedules, Polyite uses its own file format which, like the (extended) JSCOP format, is based on JSON.

We provide a separate tool for the labeling of training sets of schedules for schedule classifiers (refer to Chapter 7).

8.3 Existing Basic Building Blocks

As shown in Figure 8.1, besides clang/Polly, the main existing building blocks on which Polyite relies are the Integer Set Library (isl) [152], libbarvinok [157], and scikit-learn. This subsection introduces these and other dependencies of Polyite.

8.3.1 The Integer Set Library (isl)

The Integer Set Library (ISL) by Verdoolaege [152] is a library for the modeling of integer sets (refer to Definition 2.2.13) and relations between integer sets ("maps"). Besides operations on integer sets and maps, such as union, difference, projection, calculation of multiple types of hulls, application of maps to sets, calculation of lexicographic minimum/maximum, etc., ISL contains dedicated functionality for the polyhedron model. This comprises dependence analysis, a variant [156] of the PLuTo scheduling algorithm [25] (refer to Section 2.2.2.5), polyhedral code generation, and an implementation of schedule trees [66] (refer to Section 5.2). At the heart of ISL are an extension of Fourier-Motzkin variable elimination that is known as the Omega Test [129, 152] and a parametric integer programming solver [153].

Since ISL is written in C, SCALA programs run on the JAVA VIRTUAL MACHINE (JVM), and JAVA libraries can be invoked from SCALA code, POLYITE can use ISL via the JAVA NATIVE INTERFACE (JNI)[11]. The generator of the JNI bindings for ISL was built by Armin Größlinger and is available from `https://github.com/stganser/isl` together with the version of ISL used.

We use ISL in the version of commit `cfebc0c6` (Dec. 11, 2015) of `https://repo.or.cz/isl.git`.

8.3.2 libbarvinok

LIBBARVINOK by Verdoolaege et al. [157] is an extension of ISL that can count the number of elements of an integer set. Since we do not have JAVA/ SCALA bindings for LIBBARVINOK available, POLYITE uses LIBBARVINOK by running a subprocess that provides the library's functionality via a text-based interface.

We use LIBBARVINOK in the version of commit `91ba8f18` (May 26, 2018) of `https://repo.or.cz/barvinok.git`.

8.3.3 Größlinger's Implementation of Chernikova's Algorithm

Our implementation of Chernikova sampling uses Armin Größlinger's implementation of Chernikova's algorithm [107, 108, 109] (refer to Section 2.2.1.2) in SCALA. The library relies on ISL. Größlinger implemented the algorithm's extensions by Fernández and Quinton [55] and Le Verge [90].

8.3.4 LLVM

LLVM by Lattner and Adve [89] is a modular compiler and virtual machine infrastructure. Further, LLVM contains a just-in-time compiler for program optimization at run time. Internally, LLVM represents programs in its intermediate representation (LLVM IR). As illustrated by Figure 8.2, the compilation of programs in LLVM is divided into three phases from a macroscopic perspective: Initially, an LLVM front end must translate the code that is written in the original source language to LLVM IR. The LLVM front end for languages in the C language family is CLANG [12]. In the second phase, the program's representation in LLVM IR is optimized by different optimization passes. After the optimization of LLVM IR, a back end must be used to translate the program to code that is suitable for the intended run-time environment target, which is often a computer architecture. One back end translates to assembler code for the X86 processor architecture. Back ends perform optimizations that are specific to the corresponding target.

[11] `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/`

[12] `https://clang.llvm.org`

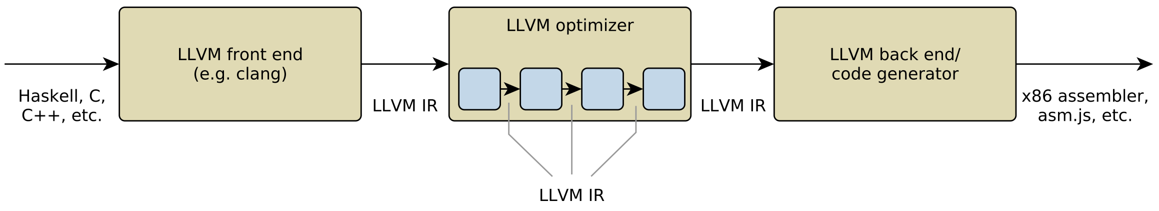


Figure 8.2: Macroscopic view of the architecture of compilers that are based on LLVM. The layout of the figure is inspired by Lattner [88].

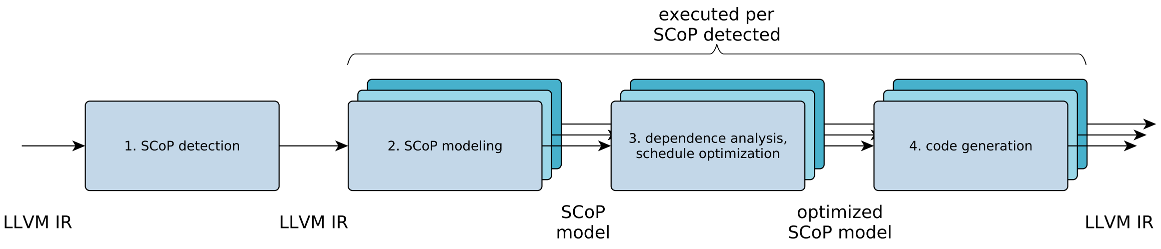


Figure 8.3: High-level steps taken by POLLY to detect and optimize SCoPs in a program.

We use LLVM in the version of commit `bf8415a8` (Jan. 10, 2016) of `http://llvm.org/git/llvm.git` in combination with CLANG in the version of commit `9909f323` (Jan. 27, 2016) of `http://llvm.org/git/clang.git`.

8.3.5 Polly

POLLY, which was initiated by Grosser et al. [65], is a set of LLVM (refer to Section 8.3.4) passes that provide polyhedral program analyses and optimization. Internally, POLLY relies on ISL (refer to Section 8.3.1).

As illustrated in Figure 8.3, POLLY identifies regions in LLVM IR that are SCoPs. POLLY builds these SCoPs' polyhedral models, optimizes the models' schedules, and generates optimized LLVM IR code from the transformed model. The version of POLLY used models each basic block in the SCoP as one statement. A *basic block* is a sequence of statements that does not contain branching instructions. The optimization of schedules consists of two steps. In the first step, POLLY uses a variant [156] of the PLUTO scheduling algorithm [25] (refer to Section 2.2.2.5) to compute a linearly affine schedule that minimizes the reuse distances of data dependences and enables tiling and parallelization. The resulting schedule is represented by a schedule tree (refer to Section 5.2). In a second step, POLLY optimizes this schedule tree further, for instance by tiling and strip-mining.

Besides optimizing a SCoP's schedule by itself, POLLY can export the SCoP's model via its JSCOP interface (refer to Example 8.1.1), reimport a transformed model, and replace the SCoP's code according to the imported model. As shown in the example, JSCOP is based on ISL maps. Thus, it is impossible to apply POLLY's schedule tree optimizations from the second phase of POLLY's internal schedule optimization. We extended JSCOP by an embedded schedule tree, to which POLLY can apply its schedule tree optimizations after the import. POLLY verifies the legality of imported schedules. For a sufficient legality criterion, we must check the legality of imported schedule trees after the application of schedule tree optimizations. Unfortunately, this check proved to be computationally too expensive. Therefore, we verify imported schedule trees only before their further transformation. This check is necessary, but not sufficient for the legality of an imported schedule. In other words,

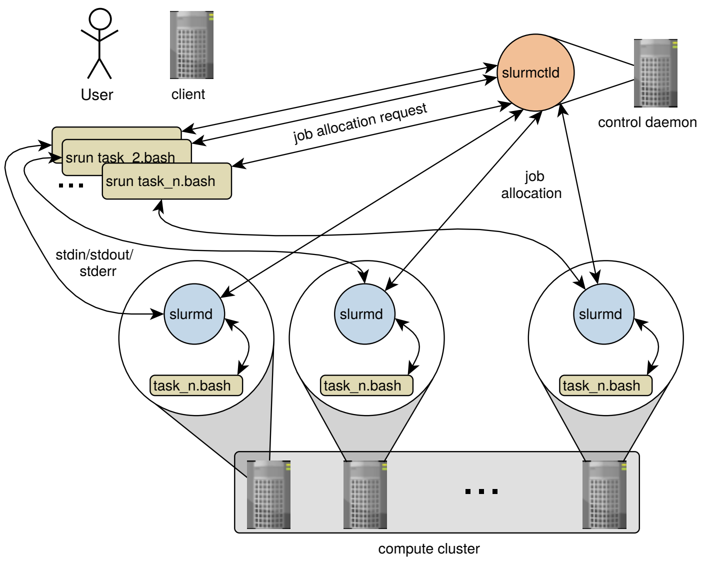


Figure 8.4: Architecture of the SLURM Workload Manager. This figure is a simplified reproduction of a figure by Yoo et al. [167]. We show only the aspects that are relevant to POLYITE.

if POLYITE groups schedule dimensions in a permutable band incorrectly, POLLY will not be able to recognize the schedule tree's illegality.

We use POLLY in the version of commit `2b618e01` (Jan. 27, 2016) of `http://llvm.org/git/polly.git`.

8.3.6 SLURM Workload Manager

The Simple Linux Utility Resource Manager (SLURM) [167] is a workload manager for compute clusters whose nodes run LINUX. Figure 8.4 shows the main architectural aspects of SLURM. To submit a computation job, clients invoke the `srun` program. Along with the script that executes the computation job, the client must specify the required computation resources. `srun` sends the job allocation request to the SLURM control daemon, which creates an allocation on a compute node that matches the resource specification. Using a direct connection of the client and a compute node, the standard input, output, and error streams of the computation job are connected to the `srun` process on the client computer. Thereby, the client can interact with the computation job. Besides interactive job allocations with `srun`, SLURM also supports the submission of non-interactive batch jobs. Inside a SLURM job, `srun` can be used to initiate a *job step*, which is a sub-computation that may be limited to a subset of the allocated computation resources.

8.3.7 scikit-learn

For the classification of schedules (refer to Section 7.2), POLYITE relies on the implementations of CART and random forest (refer to Sections 2.3.2.1 and 2.3.2.2) in the machine-learning library SCIKIT-LEARN [118] in version 0.19.2. The application programming interface of SCIKIT-LEARN is written in PYTHON [13]. To integrate SCIKIT-LEARN with POLYITE, we train models and classify schedules' feature vectors in interactive PYTHON 3.5.3 sessions that are subprocesses of POLYITE. We use multiple PYTHON sessions to be able to classify schedules in parallel. POLYITE's generic interface to interactive PYTHON sessions supports virtual

[13]`https://www.python.org/`

Python environments [14] and the execution of Python scripts. The former increases the portability of Polyite by making an existing installation of scikit-learn unnecessary.

An alternative to scikit-learn that would not require subprocesses is the Weka [58] data mining library. Weka is written in Java, and Polyite could therefore be linked to it.

8.3.8 MPI

The Message Passing Interface (MPI) [98] is a standard for inter-process communication in the context of distributed-memory parallelization. Processes that collaborate to compute a task in parallel and potentially run on different computers can communicate via MPI. MPI offers one-to-one communication and functions to distribute data to all processes in a group of processes or to collect data from all processes in a group. SLURM (refer to Section 8.3.6) can initialize the MPI communication among processes that run on nodes of a compute cluster. Bernasch [23] contributed a distributed genetic algorithm to Polyite (refer to Section 6.4) that uses Open MPI [15]. Polyite is known to be compatible with Open MPI in version 2.1.1. Open MPI is an implementation of the MPI standard that provides bindings for Java programs on the basis of the Java Native Interface (JNI).

8.3.9 The ExaStencils Code Generator

Basic operations of the schedule search space construction were originally implemented by Stefan Kronawitter as part of a polyhedral schedule search space exploration in the ExaStencils code generator [87]. Polyite was implemented on the basis of this code. The functionality that we reused comprises the basic layout of the schedule coefficient space (refer to Section 4.2), the dependence analysis (refer to Section 4.3), the functionality to build the sets of weakly and strongly satisfying schedule coefficient vectors for a given dependence polyhedron (refer to Section 2.2.2.3), functions to calculate sets of vectors that are linearly dependent or independent from a given set of vectors in certain dimensions (refer to Section 4.7), and, finally, the functionality to determine the direction of a dependence polyhedron relative to a given one-dimensional schedule (refer to Section 2.2.2.3).

8.4 Software Architecture

Most functionality in Polyite is provided by loosely-coupled replaceable modules. Most of these modules are stateless. To allow for variability within modules, we either pass strategies to respective functions as function parameters, or rely on factories. Polyite's configuration is stored in an object, which is accessible to many functions via a reference that we pass as a function argument.

We start by providing an overview of Polyite's most important (replaceable) components. We continue by describing how Polyite can utilize SLURM to benchmark multiple schedules in parallel on a compute cluster. Finally, we discuss the parallelization of tasks in Polyite.

8.4.1 Class Diagram

Figure 8.5 is a UML class diagram that shows the central components of Polyite and interfaces, which are called traits in Scala, for the most relevant kinds of replaceable functionality. The white box labeled `Polyite` represents the main module of Polyite and is, in reality, divided into several Scala objects. An *object* is a class that implicitly is a singleton, which means that only one instance of it exists. There are more objects that play a central role in Polyite. They are represented by the striped nodes in Figure 8.5. Polyite's genetic

[14] `https://docs.python.org/3/tutorial/venv.html`
[15] `https://www.open-mpi.org/`

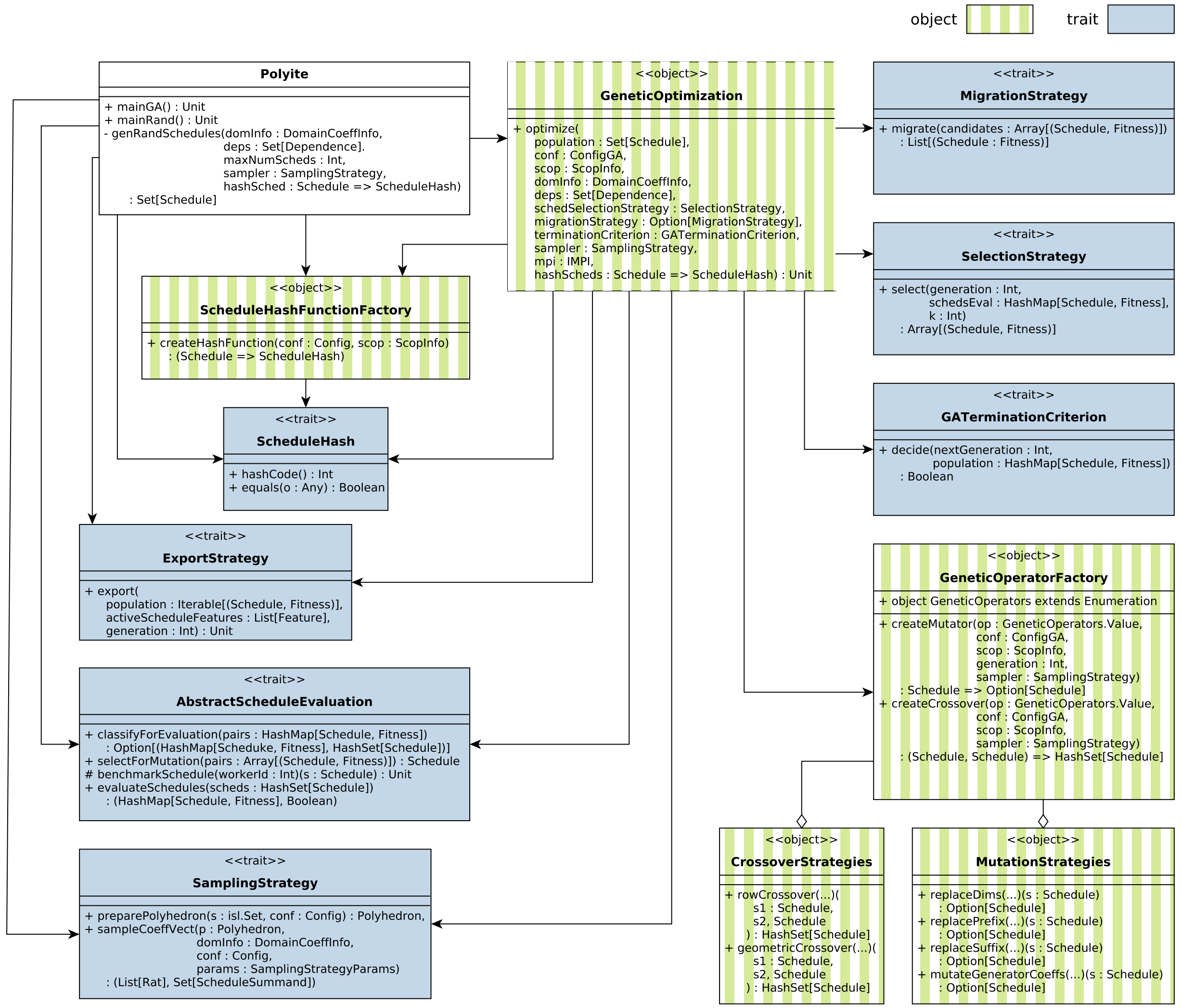

object
trait
Polyite
+ mainGA() : Unit
+ mainRand() : Unit
- genRandSchedules(domInfo : DomainCoeffInfo,
deps : Set[Dependence].
maxNumScheds : Int,
sampler : SamplingStrategy,
hashSched : Schedule => ScheduleHash)
: Set[Schedule]
<<object>>
GeneticOptimization
+ optimize(
population : Set[Schedule],
conf : ConfigGA,
scop : ScopInfo,
domInfo : DomainCoeffInfo,
deps : Set[Dependence],
schedSelectionStrategy : SelectionStrategy,
migrationStrategy : Option[MigrationStrategy],
terminationCriterion : GATerminationCriterion,
sampler : SamplingStrategy,
mpi : IMPI,
hashScheds : Schedule => ScheduleHash) : Unit
<<trait>>
MigrationStrategy
+ migrate(candidates : Array[(Schedule, Fitness)])
: List[(Schedule : Fitness)]
<<trait>>
SelectionStrategy
+ select(generation : Int,
schedsEval : HashMap[Schedule, Fitness],
k : Int)
: Array[(Schedule, Fitness)]
<<trait>>
GATerminationCriterion
+ decide(nextGeneration : Int,
population : HashMap[Schedule, Fitness])
: Boolean
<<object>>
ScheduleHashFunctionFactory
+ createHashFunction(conf : Config, scop : ScopInfo)
: (Schedule => ScheduleHash)
<<trait>>
ScheduleHash
+ hashCode() : Int
+ equals(o : Any) : Boolean
<<trait>>
ExportStrategy
+ export(
population : Iterable[(Schedule, Fitness)],
activeScheduleFeatures : List[Feature],
generation : Int) : Unit
<<object>>
GeneticOperatorFactory
+ object GeneticOperators extends Enumeration
+ createMutator(op : GeneticOperators.Value,
conf : ConfigGA,
scop : ScopInfo,
generation : Int,
sampler : SamplingStrategy)
: Schedule => Option[Schedule]
+ createCrossover(op : GeneticOperators.Value,
conf : ConfigGA,
scop : ScopInfo,
sampler : SamplingStrategy)
: (Schedule, Schedule) => HashSet[Schedule]
<<trait>>
AbstractScheduleEvaluation
+ classifyForEvaluation(pairs : HashMap[Schedule, Fitness])
: Option[(HashMap[Scheduke, Fitness], HashSet[Schedule])]
+ selectForMutation(pairs : Array[(Schedule, Fitness)]) : Schedule
# benchmarkSchedule(workerId : Int)(s : Schedule) : Unit
+ evaluateSchedules(scheds : HashSet[Schedule])
: (HashMap[Schedule, Fitness], Boolean)
<<trait>>
SamplingStrategy
+ preparePolyhedron(s : isl.Set, conf : Config) : Polyhedron,
+ sampleCoeffVect(p : Polyhedron,
domInfo : DomainCoeffInfo,
conf : Config,
params : SamplingStrategyParams)
: (List[Rat], Set[ScheduleSummand])
<<object>>
CrossoverStrategies
+ rowCrossover(...)(
s1 : Schedule,
s2, Schedule
) : HashSet[Schedule]
+ geometricCrossover(...)(
s1 : Schedule,
s2, Schedule
) : HashSet[Schedule]
<<object>>
MutationStrategies
+ replaceDims(...)(s : Schedule)
: Option[Schedule]
+ replacePrefix(...)(s : Schedule)
: Option[Schedule]
+ replaceSuffix(...)(s : Schedule)
: Option[Schedule]
+ mutateGeneratorCoeffs(...)(s : Schedule)
: Option[Schedule]

Figure 8.5: UML class diagram that shows the main components of POLYITE. Most functionality is provided by replaceable strategies.

algorithm (refer to Chapter 6) is implemented in the object `GeneticOptimization`. `Genetic Optimization` relies on the factory `GeneticOperatorFactory` for genetic operators. The genetic operators are implemented in two objects: one object for mutations and one for crossover operators. As described in Section 6.3, there are multiple ways to define the equality of two schedules. Based on the current configuration, `ScheduleHashFunctionFactory` can add a wrapper to a schedule matrix that implements the desired equivalence relation in its methods `equals` and `hashCode`.

Variable behavior is defined by a set of traits. We use the *strategy* design pattern [60] (1) to implement the schedule equivalence relations that are described in Section 6.3, (2) to allow the export of schedules and performance data to different file formats (currently CSV, JSCOP, and POLYITE's own JSON-based file format), (3) to implement the sampling techniques for schedule matrices that are described in Section 4.6, (4) to provide replaceable schedule migration strategies for POLYITE's distributed genetic algorithm (Section 6.4)[16], (5) replaceable selection strategies for our genetic algorithm, and (6) replaceable termination criteria (refer to Sections 6.1 and 7.2.2) for the genetic algorithm.

Further, we allow for the replacement of the schedule evaluation strategy. Here, we rely on the *template method* design pattern[60]. We can either evaluate schedules by the use of benchmarking or by one of the strategies that involve classification (refer to Section 7.2).

8.4.2 Schedule Evaluation

POLYITE can evaluate a configurable number of schedules in parallel. Each schedule evaluation thread selects a schedule that has not been evaluated yet and starts the schedules' evaluation by transforming it to a simplified schedule tree. If the schedule evaluation requires the classification of schedules, the thread will continue by computing the schedule's feature vector and will rely on its dedicated interactive PYTHON shell to classify the schedule. Consequently, for n schedule evaluation threads, we train n schedule classifiers that run in n interactive PYTHON shells.

To benchmark schedules in parallel, POLYITE relies on SLURM. A schedule evaluation thread that needs to apply a schedule to the program that is to be optimized and measure the transformed program's execution time starts `srun` as a subprocess of POLYITE and passes the name of the compiler wrapper script as a command line parameter. The call to `srun` will block until SLURM has allocated a compute cluster node for the benchmarking of the schedule. SLURM links the standard input and output streams of the `srun` process, which runs locally, and the benchmarking script, which executes on the cluster node. Therefore, the schedule evaluation thread can send a textual representation of the schedule to the benchmarking script via the standard input stream of `srun` and receive the measured execution times via `srun`'s standard output stream.

In summary, the remote execution of the schedule benchmarking procedure is transparent to POLYITE. Even if POLYITE and the benchmarking of schedules are executed on the same node of a compute cluster, for instance on different sockets, using `srun` can be handily to control the behavior of hyperthreading or INTEL TURBO BOOST.

8.4.2.1 Caching

Particularly in the case of small SCoPs and in late generations of the genetic algorithm, the schedule evaluation may encounter simplified schedule trees that have already been evaluated before. To avoid a repeated evaluation of a schedule tree, we cache evaluation results in a cache of unbounded size.

[16]These were contributed by Bernasch [23].

8.4.3 Parallelization

Two levels of parallelism can be enabled in Polyite. The first level is shared-memory parallelism. Enabling the second level turns Polyite's genetic algorithm into a distributed genetic algorithm (refer to Section 6.4) running in parallel using distributed memory.

8.4.3.1 Shared Memory

The shared-memory parallelization is a parallel version of our sequential genetic algorithm for schedule optimization (refer to Sections 6.1 and 7.2). We parallelize the fitness evaluation of the schedules, as described in Section 8.4.2. Further, we parallelize the generation of random schedules and the generating of offspring populations by mutating and crossing. This requires a synchronization of the threads that generate the schedules to avoid the insertion of duplicate schedules into the population.

As described in Section 8.3.1, Polyite relies on isl to represent many of its main data structures. Each object in isl (for instance, an integer set) is associated with a context object [154]. isl context objects are not thread-safe. Yet, multiple context objects are allowed to exist in one application. All isl objects that are involved in an operation must be associated with the same context. The JNI bindings for isl that Polyite uses achieve thread safety for any isl object by acquiring a lock on the object's associated context object before the start of any operation on the object. Naturally, this design causes a contention of the schedule generation threads. To overcome the contention, we transfer isl objects to temporary contexts before time-consuming library calls or invocations of algorithms that perform a large number of calls to isl library functions. Unfortunately, isl did not support the transfer of objects among contexts at the time of writing this thesis. We serialize isl objects to character strings and parse the serialization in the target context. This costly practice requires a careful consideration of when to use temporary contexts.

Apart from the fact that a genetic operator that is cheaper to apply than other genetic operators may be able to profit from its efficiency and produce more schedules by chance, the behavior of the shared-memory parallel genetic algorithm does not differ fundamentally from the sequential genetic algorithm.

Alba and Tomassini [3] testify that, for parallel versions of a sequential evolutionary algorithm, parallelizing parts of the algorithm other than the fitness evaluation of the population's individuals is typically not worthwhile. Yet, in our case, the cost of preserving a schedule's legality and of encoding all of the transformed program's loops in the schedule explicitly makes parallelizing random schedule generation, mutation, and crossover likely worthwhile.

8.4.3.2 Distributed Memory

Bernasch [23] implemented an extension of Polyite's sequential genetic algorithm that turns it into a distributed genetic algorithm (refer to Section 6.4). Each subpopulation of the distributed genetic algorithm resides in a dedicated Polyite process. For the migration of schedules between the subpopulations, Bernasch's implementation relies on MPI (refer to Section 8.3.8). Using Polyite's parallel schedule evaluation and SLURM, each Polyite process can allocate compute cluster nodes dynamically to benchmark schedules. Figure 8.6 illustrates the distributed setup.

8.5 Complying with the (Super-)Exponential Asymptotic Run-Time Complexity of Algorithms in Polyite

Many algorithms used by Polyite, such as Chernikova's algorithm [107, 108, 109] (refer to Section 2.2.1.2), Fourier-Motzkin variable elimination [138, 161], and parametric integer

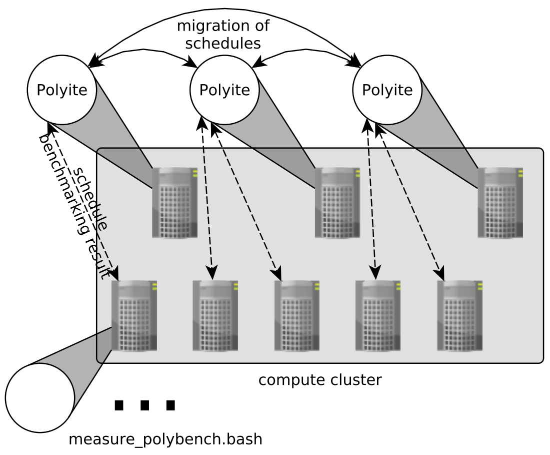


Figure 8.6: Polyite's genetic algorithm can be distributed across a compute cluster's nodes. Each node maintains a subpopulation. Via MPI, schedules can migrate between nodes. By using SLURM, the Polyite processes can allocate other cluster nodes for the benchmarking of schedules.

programming [50] (refer to Section 2.2.1.3), have a (super-)exponential asymptotic time complexity. While the average time complexity is much lower, we must be able to bail out of computations which take unacceptably long. For this purpose, we rely on isl's ability to limit the number of internal low-level steps of a function invocation and on timeouts that are based on execution time measurement. In the current implementation, we apply both timeout mechanisms to both the mutation and crossover of schedules and to the random sampling of schedules. In the case of a timeout, Polyite tries to sample a different schedule.

9 Evaluation

In Chapters 4 to 7 we propose a novel approach to iterative (or search-based) program optimization in the polyhedron model. Our focus is on numerical computations such as matrix multiplication, stencils, or algorithms in dynamic programming that profit from coarse-grained OpenMP parallelism and optimizations of data locality. The theoretical foundation is the iterative schedule optimization technique by Pouchet et al. [122, 124]. Our technique is implemented in the tool Polyite (refer to Chapter 8), which uses LLVM [89] and its polyhedral optimizer Polly [65] to detect SCoPs in programs, to build their polyhedral model, and to apply schedules to SCoPs. Polyite is compatible with a customized version of Polly (refer to Section 8.3.5). Polly may apply tiling and loop parallelization with OpenMP after the application of the schedules that are generated by Polyite. Currently, Polyite is incompatible with Polly's polyhedral prevectorization (strip-mining). A correct prevectorization appears to require an additional preparation of the schedules in Polyite. Thus, we excluded polyhedral vectorization from the scope of our evaluation entirely.

Our expectation is that, with our approach to schedule search space exploration, it is possible to find significantly better schedules under the preconditions set (refer to Section 9.1 for details on the experimental setup) than with the use of isl's version of the PLuTo scheduling algorithm [25, 152] (refer to Section 2.2.2.5). Furthermore, we expect that the surrogate performance models proposed in Chapter 7 can be useful to speed up our genetic algorithm and to reduce the benchmarking effort in comparison to random exploration without losing the benefit of iterative optimization.

The schedule search space exploration of our approach is necessarily incomplete. It is difficult or may even be impossible to identify guarantees for the quality of the optimization's result analytically. Instead, we resort to an empirical evaluation. As the baseline, we use the PLuTo scheduling algorithm and an exploration of the search space that is constructed by our adaptation of the algorithm for search space construction by Pouchet et al. [124].

We split the empirical evaluation into two parts: in Section 9.3, the focus is on the sampling techniques that Chapter 7 presents and the genetic algorithm described in Chapter 6. This genetic algorithm relies solely on benchmarking for the determination of schedules' fitness. Like in Chapter 7, we refer to it as GA_B. Section 9.4 describes the result of an evaluation of the surrogate performance models described in Chapter 7 and their combination with our genetic algorithm that we introduced as GA_C in Section 7.2.2.

We start with a description of the experimental setup in Section 9.1 and continue with a characterization of the benchmark set for the experiments in Section 9.2. In Sections 9.3 and 9.4, we present the two parts of our evaluation. Both sections have the same structure: first, we raise a number of research questions to shed light on our expectations to iterative schedule optimization for parallelization and the use of our surrogate performance models for schedules. Next, we present experiments that we conducted to study the research questions. Based on the experiments' results we give answers to the research questions. We conclude the chapter by a discussion of potential threats to validity in Section 9.5.

9.1 Experimental Setup

Our experimental setup comprises four components: the implementation of our approach, which includes the libraries used, the tool chain for the compilation of program versions, the execution environment and the settings used for the benchmarking of program versions, and the configuration of the search space exploration.

9.1.1 Implementation

Polyite is written in Scala (version 2.11). For machine learning, we use scikit-learn [118] (version 0.20.2). scikit-learn runs in a Python (version 3.5.3) session that is a child process of Polyite. To handle polyhedra, Polyite relies on JNI bindings for isl (commit `cfebc0c6` of `https://repo.or.cz/isl.git` on Dec. 11, 2015) and libbarnvinok (version 0.41). Details of the implementation can be found in Chapter 8.

9.1.2 Tool Chain

For SCoP extraction, tiling, and code generation, Polyite relies on a modified version of Polly that is based on commit `2b618e01` (Jan. 27, 2016) of `http://llvm.org/git/polly.git`, LLVM in the version of commit `bf8415a8` (Jan. 10, 2016) of `http://llvm.org/git/llvm.git`, and clang in the version of commit `9909f323` (Jan. 27, 2016) `http://llvm.org/git/clang.git`. The relevant compiler flags that we use to tile (with a fixed tile size of 64), generate code, and parallelize loops are: `-march=native -O3 -mllvm -polly -mllvm -polly-position=early -mllvm -polly-parallel=true -mllvm -polly-tiling=true -mllvm -polly-vectorizer=none -mllvm -polly-default-tile-size=64`. For the baseline measurements, we set `-polly-optimizer=isl`, if we import schedules that are produced by Polyite, we set `-polly-optimizer=none`. An additional exploration of other tile sizes on top of the polyhedral schedule space exploration would add further dimensions to our search space, which would make its exploration impractical. Thus, combing schedule search space exploration with an optimization of tile sizes such as by the autotuner of Sato et al. [136] or the approach of Shirako et al. [139] in a practical way remains an open challenge. Adding a static heuristics for tile size selection, for instance, the heuristics used by Pouchet et al. [127], who compute tile sizes such that the data accessed by each tile roughly fits into the L1 cache, would likely be unproblematic. Depending on the polyhedral compiler's ability to accept different vectors of tile sizes for different loop nests in the transformed SCoP, it may then be necessary to encode tiling already into the schedules inside Polyite. Polyite uses a debug build of LLVM, clang, and Polly because the original tool chain relied on LLVM's debug tools opt and llc for the injection of schedules into Polly. With the current setup, which starts the entire compilation process by a single invocation of clang, we could use a release build of the compiler, which would increase the efficiency of the program variants' compilation.

9.1.3 Benchmarking

The compilation of transformed program versions and the benchmarking of the transformed code was run on the Intel Xeon E5-2650 v2 CPU @ 2.6GHz with eight physical cores and 20MB of L3 cache. Hyperthreading and Intel Turbo Boost [36] were disabled. The operating system was Debian 9 with Linux 4.9. We ran Polyite on OpenJDK 8. To mitigate measurement bias, we executed each transformed program version five times and took the shortest measured execution time. In an additional preceding run, we verified the computation result. We purged any schedule that yielded incorrect computation results. As the reference result, we took the output that had been produced by a binary that was compiled by the same version of clang with `-O0`. While all schedules are legal in theory, they can, for instance, trigger integer overflows in the transformed programs. A schedule's evaluation may fail for the following reasons: a timeout (five minutes for compilation and 30 minutes overall[17]), failed compilation, and miscompilation resulting in a run-time error.

[17]For the programs `lu` and `ludcmp`, we use an overall timeout of 40 minutes.

9.1.4 Configuration

In Chapter 4, we could identify two sampling strategies for schedules that are potentially useful for a schedule search space exploration: projection sampling (refer to Section 4.6.6) and Chernikova sampling (refer to Section 4.6.5). While projection sampling scales better with SCoP size, Chernikova sampling is suitable for the SCoP sizes found in PolyBench 4.1. Furthermore, Chernikova sampling enables generator coefficient replacement (refer to Section 6.2.1.4) as an additional mutation operator for our genetic algorithm. Finally, controlling the number of rays and lines that Chernikova sampling may use to form a schedule coefficient vector is a straightforward way to regulate the sparsity of schedule matrices. It is widely believed that sparse schedule matrices are likely more profitable than dense matrices. Thus, we use Chernikova sampling as our primary sampling strategy for schedules.

For the number of rays and lines that may form a schedule coefficient vector together with a point on one of the corresponding polyhedron's minimal faces, we use three settings with Chernikova sampling: (1) If at most two rays and two lines may be added to a vertex to form a schedule coefficient vector, the schedule matrix has a high sparsity because many coefficients are zero. We call this the *sparse* setting. (2) If the allowed number of rays and lines is only limited by their total number, the matrices tend to be dense which we call the *dense* setting. (3) The *mixed* setting produces sparse and dense matrices. These settings apply to the generation of coefficient vectors by random exploration as well as by genetic operators.

Some experiments also cover projection sampling (refer to Section 4.6.6). Projection sampling takes the probability function of a discrete probability distribution as a parameter. We use a geometric distribution with $p = 0.7$. We accept a schedule coefficient $c \in \mathbb{Z}$ if the acceptance test according to the geometric probability function accepts $|c|$. If c was chosen from an interval $[lb, ub], lb, ub \in \mathbb{Z}$ with $0 \notin [lb, ub]$, we perform the acceptance test for $|c| - \min\{|lb|, |ub|\}$. By default, we require $c \in ([-4, 4] \cap \mathbb{Z})$. With these settings, we expect to obtain comparatively sparse schedule matrices with small coefficients.

A run of our genetic algorithm has at most 40 generations (excluding the initial population). GA_B may either be configured to run for a fixed number of generations (either 20 or 40 in our experiments) or it may terminate early if, over the last eight generations, the execution time yielded by the optimal schedule was reduced by less than 2% from one generation to the next. In the latter case, the maximum number of generations possible is 40. We had determined the parameters for the latter termination criterion by an analysis of GA_B's behavior when it runs for a fixed number of 40 generations. One observation was that, for most programs in the benchmark set used, the number of 40 generations is too high. GA_C terminates if at least 95 percent of the schedules in the population are classified as profitable or after 40 generations. The initial population is produced randomly using the sparse setting of random exploration.

The population size of GA_B is 30 schedules. A bigger population size or a larger number of generations conflicts with GA_B's costly fitness evaluation. As, in one generation, we replace half of the population, GA_B visits at most 630 schedules. In comparisons of GA_B and random exploration, we let random exploration visit the same number of schedules. To ensure a good distribution of randomly sampled schedules, we sample one or two schedules from the search space region currently visited. Still, the exploration may enter a search space region repeatedly. If, during random schedule generation, a thread has not been able to find a new schedule after 1000 tries it terminates. For the generating of the genetic algorithms initial population and for mutation and crossover, we use a smaller threshold of 20, which is justified by the genetic algorithm's small population size.

For GA_C, we increased the population size to 50 schedules to raise the population's diversity. Also, an increased population should not affect GA_C's processing time substantially because the costly evaluation by benchmarking is applied to only the schedules in the final population. The maximum number of schedules that GA_C may visit in one run is 1050.

The mutation operators are configured to mutate 10% of a schedule's dimensions at the start of the optimization process, but at least one row. Accordingly, generator coefficient replacement mutates 10% of the line, ray, and vertex coefficients at the beginning. Since, for schedule matrices with less than 11 rows, this always results in one row to be modified, the effect of simulated annealing diminishes for small SCoPs. The annealing function $f(p, g) = p/\sqrt{ln(g + \mathrm{e} - 1)}$ calculates the fraction of schedule dimensions to be mutated from the currently generated generation $g \in \mathbb{N}$ and the initial fraction $p \in [0, 1]$. Geometric crossover produces up to three schedules at once. The other operators produce just one schedule. The special role of geometric crossover is justified, since it produces otherwise unreachable rows. In each generation of the GA, we introduce two randomly generated schedules. The chosen configuration is the result of preliminary experiments. We challenge some of the choices in the research questions.

9.2 Benchmark Set

We have evaluated our approach on the widely used PolyBench 4.1 [121] benchmark set. PolyBench is dedicated to the evaluation of techniques in the context of the polyhedron model. PolyBench 4.1 is a collection of 30 small programs that contain each one function whose body is a SCoP. These functions represent the actual benchmarks. The programs perform numerical computations from different domains. Among these domains are stencils, algorithms from linear algebra, and dynamic programming. Among other features, PolyBench enables the verification of computation results, the measurement of the benchmark function's execution time, and variable configurable problem sizes. The problem sizes range from "mini" to "extra large". In our experiments, we use the extra large setting. Since we would like to optimize for parallelism and improved data locality, we need a problem size that yields a workload that is sufficiently large to profit from parallelization, and, ideally, we must ensure that the program processes an amount of data that does not fit into the processor cache. Table 9.1 characterizes the programs in PolyBench 4.1 and shows results of our baseline measurements with Polly and isl's variant of the PLuTo scheduling algorithm.

Per program, we show the number of statements in the SCoP's model. Recall from Section 8.3.5 that the version of Polly used for our evaluation of Polyite treats each basic block as one statement. Next, we show the number of dependence polyhedra that Polyite constructs to model a program's legality-affecting data dependences (these are the flow, output, and anti dependences). The fourth column shows the maximum loop nesting depth in each program's SCoP. As in Chapter 7, we heuristically use the maximum dimensionality of any statement iteration domain of the SCoP as its maximum loop depth.

Note that we use the SCoPs as they are modeled by Polly. Polly uses a heuristics to avoid adding statements to SCoPs that are unlikely to profit from its optimizations. We continue with the number of structure parameters in the SCoP's model. The last static property is the SCoP's relative dependence interference. Dependence interference is a metric that Pouchet et al. [124] used to characterize dependence polyhedra. Two dependence polyhedra *interfere* if it is not possible to construct a one-dimensional schedule that strongly satisfies both dependence polyhedra. With $n \in \mathbb{N}$ as the number of a SCoP's dependence polyhedra, one dependence polyhedron can interfere with at most $n - 1$ other dependence polyhedra. Let k_i be the number of dependence polyhedra with which dependence polyhedron $i, i \in \{1, \ldots, n\}$, interferes. We define the *relative dependence interference* of the SCoP as

$$\frac{\sum\limits_{i=1}^{n} k_i}{n \cdot (n-1)}.$$

Table 9.1: Characteristics of the programs in the PolyBench 4.1 benchmark set. Per program, we quantify several static properties and we show results of baseline measurements with the version of LLVM/ Polly used with Polyite and with LLVM/ Polly, version 8.

program	# statements	# dependence polyhedra	max. loop depth	# struct. parameters	dep. interference	exec. time -O3 (sec.)	speedup -O3 + isl + tiling + OpenMP	speedup -O3 + isl + tiling + OpenMP + vect.	exec. time -O3 (sec.)	speedup -O3 + isl + tiling + OpenMP	speedup -O3 + isl + tiling + OpenMP + vect.
									clang 8	**clang 8**	**clang 8**
`2mm`	4	6	3	7	0.00 %	37.75	13.81	38.25	37.64	16.82	44.64
`3mm`	6	10	3	9	0.00 %	51.33	13.39	41.72	50.93	17.11	46.95
`adi`	9	64	3	3	0.00 %	171.91	7.61	7.16	176.86	6.99	compile error
`atax`	3	4	2	3	0.00 %	0.01	0.99	1.13	0.01	0.89	1.25
`bicg`	2	4	2	3	0.00 %	0.01	1.05	1.05	0.01	2.10	compile error
`cholesky`	4	8	3	2	0.00 %	13.47	2.68	2.62	13.49	2.84	2.84
`correlation`	13	19	3	3	0.00 %	53.11	33.36	91.10	53.86	42.64	89.10
`covariance`	7	12	3	3	0.00 %	53.09	33.44	92.39	53.92	42.95	93.88
`deriche`	11	25	2	3	0.00 %	0.86	1.07	1.07	0.86	1.07	1.07
`doitgen`	3	8	4	5	0.00 %	5.48	4.06	8.04	5.38	4.64	8.62
`durbin`	10	43	2	1	0.11 %	0.02	0.99	1.00	0.02	1.01	1.00
`fdtd-2d`	4	24	3	4	0.00 %	26.81	0.92	0.92	26.64	1.23	1.23
`floyd-warshall`	1	18	3	2	11.11 %	196.72	0.15	0.15	196.46	0.15	0.15
`gemm`	2	2	3	5	0.00 %	8.95	4.23	10.13	8.82	4.80	11.11
`gemver`	4	6	2	2	0.00 %	0.10	3.78	4.54	0.10	3.79	4.33
`gesummv`	3	3	2	2	0.00 %	0.03	3.13	3.16	0.03	3.06	compile error
`gramschmidt`	9	23	3	3	0.40 %	42.20	8.10	9.07	41.95	8.49	8.62
`heat-3d`	2	171	4	2	0.00 %	37.31	2.68	2.69	37.81	0.53	0.53
`jacobi-1d`	2	16	2	2	0.00 %	0.01	0.88	0.89	0.01	0.75	0.75
`jacobi-2d`	2	56	3	3	0.00 %	32.82	1.13	1.13	33.36	1.02	1.04
`lu`	3	8	3	2	0.00 %	71.53	5.38	4.90	71.60	8.22	7.70
`ludcmp`	20	89	3	2	0.31 %	69.19	0.97	0.98	69.16	1.33	1.33
`nussinov`	8	24	3	3	1.45 %	80.98	0.96	0.95	63.40	1.01	1.00
`mvt`	2	2	2	2	0.00 %	0.08	4.65	5.71	0.08	5.18	6.05
`seidel-2d`	1	59	3	3	0.00 %	233.94	0.93	0.93	233.91	1.07	1.07
`symm`	5	21	3	4	1.90 %	27.41	1.01	1.01	27.40	2.35	2.33
`syr2k`	2	2	3	4	0.00 %	91.40	21.57	38.38	90.91	24.95	41.14
`syrk`	2	2	3	4	0.00 %	18.36	13.76	25.68	18.10	16.42	27.97
`trisolv`	3	5	2	2	0.00 %	0.01	0.36	0.37	0.01	0.36	0.36
`trmm`	2	4	3	4	0.00 %	18.86	2.02	1.99	18.77	1.81	1.77
minimum	1	2	2	1	0.00 %	0.01	0.15	0.15	0.01	0.15	0.15
maximum	20	171	4	9	11.11 %	233.94	33.44	92.39	233.91	42.95	93.88
median	3	11	3	3	0.00 %	27.11	2.68	2.66	27.02	2.59	2.33

In columns 6 to 12, we show the results of baseline measurements with LLVM's `-O3` optimization sequence and the PLuTo scheduling algorithm as it is implemented in isl (the "isl scheduler") and being used by Polly. We show the execution time in seconds of each program after its optimization by `-O3`. Further, we we show the speedup in execution time that results from enabling Polly with the isl scheduler and tiling and OpenMP parallelization enabled. These speedup values are the primary baseline in the experiments in Section 9.3. Additionally, we show the speedup that results from enabling Polly's polyhedral vectorizer, as well. One can see that some but not all programs profit from Polly's vectorization. As part of the `-O3` optimization sequence, LLVM's loop vectorizer is, of course, always active. As an additional reference, we show the same kind of baseline results for the most recent upcoming release versions of LLVM, clang, and Polly at the time of writing. We show results for the upcoming release 8 in its state on Feb. 1, 2019. One major difference between Polly 8 and the version used together with Polyite is that Polly 8 applies tiling to loop nests under more circumstances. Depending on the schedule's representation, versions of Polly that are prior to version 5, may not tile loop nests whose innermost loop encases more than one statement. Furthermore, Polly 8 adds fewer structure parameters to SCoPs' polyhedral models and it is capable of modeling SCoPs at the granularity level of actual statements, instead of basic blocks. In the optimization with Polly 8, we did not enable Polly's pattern-based detection of matrix multiplication because it frequently yielded different computation results.

In our adaptation of the search space construction by Pouchet et al. [124] (refer to Section 4.8) and in the computation of the data locality feature that is described in Section 7.1.3.3, we need to know the values of structure parameters. In general, the structure parameters' values would most likely have to be derived by running an instrumented version of the SCoP that is to be optimized with a typical workload. The instrumentation code would record the structure parameters' values. For all programs in PolyBench 4.1 except `nussinov`, we could statically derive the values of all structure parameters that are involved in the calculation of the statements' estimated memory traffic in our adaptation of the search space construction by Pouchet et al. [124]. The same holds for the calculation of the legality-affecting data dependences in our data locality feature. As described in Section 8.3.5, the version of Polly that we use in our experiments adds structure parameters to SCoPs' models that do not obviously correspond to variables in programs' source code. Thus, we could not automatically extract their values from the code and assumed their value to be 0 instead. This behavior of Polyite makes the calculation of input dependences' communicated data volumes in our data locality feature heuristic. We excluded `nussinov` from any experiments that require knowledge of the program's structure parameters' values.

For few combinations of the configuration of Chernikova sampling and the program to be optimized, we must make use of the techniques to purge rays with overly large components and to move points in polyhedra's geometric representation to adjacent points with integer coordinates described in Section 4.6.5.1. This affects our adaptation of the approach to search space construction by Pouchet et al. [124] with programs `adi` and `correlation`. Furthermore, random exploration with the mixed and the dense setting is affected in combination with program `3mm`.

9.3 Search Space Exploration

In the first part of the evaluation, we focus on the sampling techniques that are presented in Chapter 4 and the genetic algorithm described in Chapter 6.

9.3.1 Research Questions

With our approach to iterative schedule optimization for parallelization, we aim at finding more profitable schedules than can be found with model-based scheduling algorithms such as PLuTo and the existing iterative optimization technique by Pouchet et al. [124]. We expect a speedup in the execution time yielded by the optimal schedules found over the technique by Pouchet et al. since they optimized for sequential execution and bounded their schedule search space accordingly. To be able to challenge this assumption empirically, we have adapted their approach to schedule search space construction in Polyite (refer to Section 4.8). With our genetic algorithm, we expect to find better schedules in terms of speedup in execution time than by random exploration or, at least, to find acceptably profitable schedules with less effort.

We define the following research questions to verify our expectations and to examine further aspects:

RQ 9.3.1: *Does our approach find loop transformations that yield higher performance than the transformations found by established approaches to polyhedral schedule optimization?*
We evaluate whether our iterative search produces similar or better program schedules than established approaches. Answering this question gives us a baseline and demonstrates the practicality of our approach.

RQ 9.3.2: *Is our augmentation of the schedule search space justified?*
In contrast to Pouchet, we explore potentially the entire space of legal program transformations. We evaluate whether this expansion is actually meaningful (in terms of finding better

schedules) as it complicates the optimization process considerably. This question relates to Section 4.5. The question also concerns schedule completion as described in Section 4.7.

RQ 9.3.3: *Does optimization with our genetic algorithm have an advantage over random sampling?*
We evaluate whether a simple random search on the space of legal schedules can find schedules that perform equally well as our genetic algorithm. Again, answering this question provides a comparison to a baseline, which is often used for related problems due to their simplicity. The question relates to Chapter 6.

RQ 9.3.4: *Do schedule matrices with high sparsity yield better performance than dense matrices?*
This question sheds some light on the decisions made for the configuration of the experiments. We can control the maximum number of rays and lines that Chernikova sampling adds to a vertex to form a schedule coefficient vector (refer to Section 4.6.5). Here, we explore whether it is better to add only a small number of rays and lines, which yields schedule matrices with high sparsity, or whether we should leave the number unspecified.

RQ 9.3.5: *Are projection sampling and Chernikova sampling comparable with respect to the speedups in execution time over `-O3` yielded by the optimal schedules found?*
For the experiments that we present in this chapter, we used Chernikova sampling as the primary sampling strategy for schedules. Yet, Chernikova sampling does not scale for large SCoPs. Projection sampling has been implemented as an alternative sampling strategy in Polyite. Projection sampling has smaller run-time complexity than Chernikova sampling. It is relevant to understand how strongly the quality of schedules sampled by Chernikova sampling and projection sampling differs. This question relates to Sections 4.6.5, 4.6.6, and to the discussion of the different sampling strategies for schedules in Section 4.6.7.

RQ 9.3.6: *Is our configuration of the genetic algorithm justified?*
The presence of many tuning options is an intrinsic characteristic of iterative optimization tools. Their number makes it hard to find an optimal setup. Hence, we investigate whether the setup that we have chosen for the evaluation of our tool actually performs well.

RQ 9.3.7: *Does our genetic algorithm converge?*
The use of simulated annealing in the design of our genetic algorithm is based on the assumption that, after having visited a certain number of schedules, the genetic algorithm will not be able to make strong improvements over the fitness of the optimal schedule that have been found so far. Instead, the reachable optimum should be approached in steps that become gradually smaller. In other words, the changes to schedules made by mutations must shrink. This localization of search is only reasonable, if very small changes to schedules also lead to mostly small changes in the speedups yielded. Generally, this property cannot be guaranteed in the context of schedule matrices: changing a single coefficient in a schedule coefficient matrix can lead to a major difference in performance. For instance, a loop may be skewed after the change, or two previously fused statements may be distributed.

9.3.2 Experiments

In the following, we present the experiments that we conducted to receive answers to the research questions in Section 9.3.1. After the title of each experiment, we list in parentheses the research questions to which it relates.

E 9.3.1: *Convergence Rate of Iterative Optimization* (**RQ 9.3.3**, **RQ 9.3.4**, **RQ 9.3.5**)
We compare the number of schedules that different variants of iterative optimization have to visit, on average, to find the optimal reachable schedule. We compare the following configurations:

(1) random exploration with our adaptation of Pouchet's approach with the sparse setting

(2) random exploration with our adaptation of Pouchet's approach with the sparse setting and schedule completion (as described in Section 4.7, schedule completion adds dimensions to a schedule until all loops of the transformed program are encoded in the schedule explicitly)

(3) random exploration with the sparse setting (random sparse)

(4) random exploration with the dense setting (random dense)

(5) random exploration with the mixed setting (random mixed)

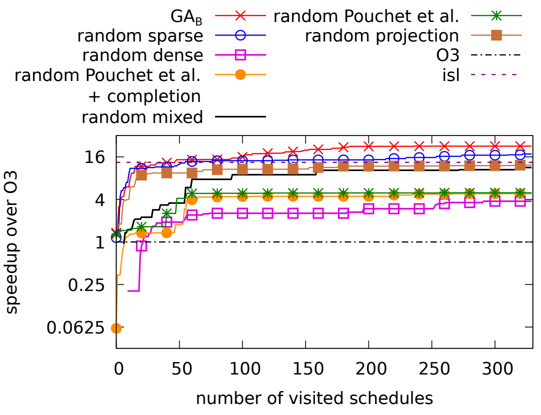
(6) random exploration with projection sampling (random rojection)

(7) GA_B with the sparse setting

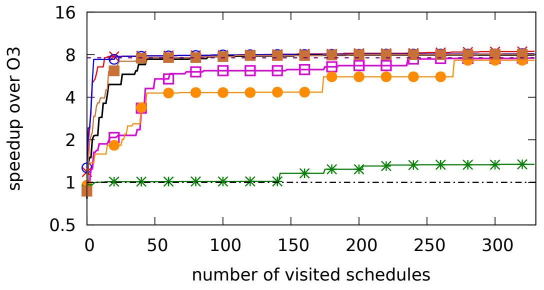
We selected programs `3mm`, `adi`, and `correlation`, which belong to different categories of PolyBench. They are promising candidates for this experiment, as the number of generators in the geometric representation of their search space regions is high compared to other programs and Polyite is capable of optimizing them. Thus, an actual difference between sparse and dense setting can be expected. Table 9.2 is a statistics regarding the average number and characterization of the generators of 1000 search space regions chosen randomly and independently per program in PolyBench 4.1. For the average number of rays, lines, and points, we analyzed P_1 of each search space region's representation since this polyhedron is among the most constrained polyhedra in each region's representation. To mitigate the effect of an incomplete exploration, we applied each configuration 10 times to each program. We fixed the number of generations of GA_B to 20, which corresponds to generating 330 schedules. The random exploration runs generated 330 schedules, as well. We reduced the number of replicated execution time measurements per generated program variant from 5, as described in Section 9.1.3, to 3. This is acceptable as no variant of any of the three programs selected has an extraordinarily short execution time far below one second. Figure 9.1 shows the median speedup over `-O3` that was reached by each optimization strategy after evaluating n schedules. The x-axis is n. The y-axis is the speedup in execution time over `-O3` yielded by the optimal schedule found after evaluating n schedules.

GA_B and random sparse converge at the same rate and to almost the same optimal speedup value in case of `adi` and `correlation`. The speedup yielded by GA_B is minimally higher. In the case of `3mm`, GA_B yields a higher maximum speedup than random sparse. Random mixed and random dense perform worse than random sparse. Random mixed lies between random sparse and random dense. Our adaptation of the approach by Pouchet et al. [124] performs worse than GA_B and random sparse. In the cases of `adi` and `correlation`, this sampling strategy profits from schedule completion. For comparison, we also marked the speedup in execution time over `-O3` yielded by the isl scheduler with the dashed line in each plot. `-O3` corresponds to the horizontal line that crosses the vertical axis at 1.

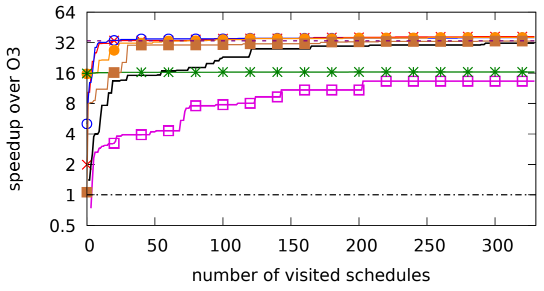
Figure 9.2 is a comparison of the configurations of search-based optimization that are enumerated above with respect to the distribution of the speedups in execution time over `-O3` that the schedules generated by the each configuration yield. The vertical axis is the speedup in execution time over `-O3`. The horizontal axis enumerates the schedules that were produced by the ten replicated runs of a configuration, from the schedule that yields the highest speedup to the schedule that yields the smallest speedup. On average, the schedules produced by GA_B yield higher speedups than the schedules produced by random exploration. Looking at random exploration, random sparse produces schedules that yield a higher average speedup than the schedules produced by random mixed. Random mixed performs better than random dense. In the case of `3mm`, the average speedup of the schedules produced



(a) `3mm`

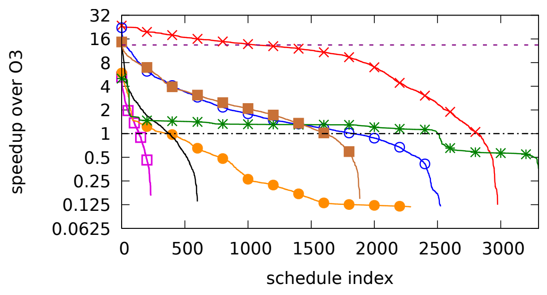


(b) `adi`

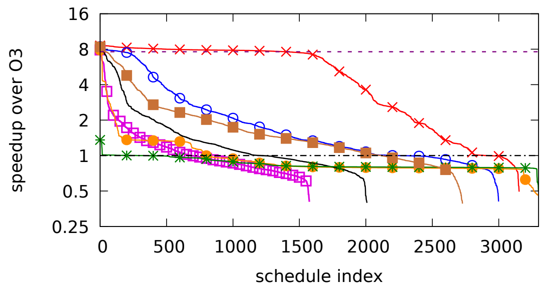


(c) `correlation`

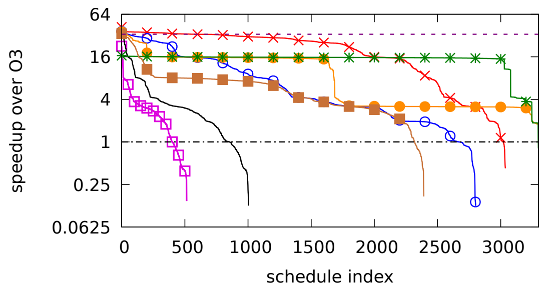
Figure 9.1: Median convergence speed of different configurations of iterative optimization for the programs `3mm`, `adi`, and `correlation` across 10 runs per configuration. Only every 20th data point is plotted.



(a) `3mm`



(b) `adi`



(c) `correlation`

Figure 9.2: Distribution of the speedups yielded by the schedules generated by different configurations of iterative optimization for the programs `3mm`, `adi`, and `correlation`. We show aggregated results from 10 runs per configuration. Note that not every data point is plotted.

Table 9.2: Characteristics of the PolyBench 4.1 programs' search space regions' geometric representations. Per program, we show the average numbers of generators of P_1 of 1000 schedule search space regions generated randomly and independently.

program	avg. # lines	avg. # rays	avg. # points
`2mm`	9	19	1
`3mm`	10	31	1
`adi`	4	27	4
`atax`	4	11	1
`bicg`	4	6	1
`cholesky`	3	27	4
`correlation`	17	37	1
`covariance`	4	26	1
`deriche`	4	70	26
`doitgen`	6	1	1
`durbin`	2	28	4
`fdtd-2d`	5	32	3
`floyd-warshall`	3	1	1
`gemm`	8	7	1
`gemver`	3	15	1
`gesummv`	4	7	1
`gramschmidt`	4	45	4
`heat-3d`	3	19	2
`jacobi-1d`	3	4	1
`jacobi-2d`	4	6	1
`lu`	3	23	3
`ludcmp`	3	66	10
`mvt`	8	2	1
`nussinov`	13	3	1
`seidel-2d`	4	3	1
`symm`	5	1	1
`syr2k`	7	7	1
`syrk`	7	6	1
`trisolv`	3	8	1
`trmm`	6	9	1

by our adaptation of Pouchet's search space construction is smaller with schedule completion than without completion. For `adi` and `correlation`, the influence on the distribution of the speedups yielded by the schedules that are produced by our adaptation of Pouchet's search space construction is smaller than for `3mm`.

The difference in the distribution of the speedups yielded by random sparse and random projection is small, but significant. We tested the differences' signficance using a Wilcoxon rank sum test [142]. We obtained the following p-values `3mm`: $p < 2.2e - 16$; `adi`: $p = 0.02$, `correlation`: $p = 0.04$. The p-values are the probability of erroneously assuming a difference between the two distributions of speedups.

From Figure 9.2, it is also apparent that not from all schedules a transformed program can be generated that operates correctly and that can be benchmarked within the timeout set. This happens although all schedules are conceptually legal. As described in Section 8.3.4, Polly verifies the schedules' legality before the application of tiling. Because this tests only a necessary but not sufficient condition for the schedules' legality after tiling, we fully verified the legality of a sample of schedules that were known to yield corrupted binaries and could not attest their illegality. Table 9.3 is a statistics of causes of failure.

During the evaluation of a schedule, the compilation of the transformed program can reach a timeout and the compiler can fail. Next, the execution of the resulting binary can fail, for instance, due to a segmentation fault or a bus error. The transformed code can produce wrong results, and the duration of the entire benchmarking procedure can reach a timeout. Our adaptation of the approach by Pouchet et al. yields almost no errors. Only in the case of `3mm` and the variant with schedule completion, the evaluation of 30.18 % of

Table 9.3: Percentage of schedules that cannot be benchmarked successfully for different reasons.

program	configuration	compile timeout	miscompiles	execution fails	wrong output	timeout	healthy
`3mm`	GA$_B$	0.61 %	0.00 %	7.76 %	0.73 %	0.73 %	90.18 %
`3mm`	random sparse configuration	1.82 %	0.03 %	16.91 %	1.33 %	3.39 %	76.52 %
`3mm`	random dense configuration	81.12 %	1.94 %	8.24 %	1.52 %	0.15 %	7.03 %
`3mm`	random mixed configuration	62.58 %	1.27 %	15.39 %	2.09 %	0.48 %	18.18 %
`3mm`	random Pouchet et al.	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	100.00 %
`3mm`	random Pouchet et al. + completion	0.39 %	0.00 %	0.00 %	0.00 %	30.18 %	69.42 %
`3mm`	random with projection	37.48 %	0.64 %	3.76 %	0.82 %	0.15 %	57.15 %
`adi`	GA$_B$	0.79 %	0.03 %	2.42 %	0.97 %	0.18 %	95.61 %
`adi`	random sparse configuration	0.85 %	0.06 %	5.48 %	1.97 %	0.73 %	90.91 %
`adi`	random dense configuration	7.52 %	0.06 %	22.42 %	17.79 %	4.45 %	47.76 %
`adi`	random mixed configuration	8.42 %	0.03 %	14.85 %	11.88 %	3.94 %	60.88 %
`adi`	random Pouchet et al	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	100.00 %
`adi`	random Pouchet et al + completion	0.00 %	0.03 %	0.00 %	0.00 %	0.03 %	99.94 %
`adi`	random with projection	9.61 %	0.06 %	4.82 %	1.85 %	1.00 %	82.67 %
`correlation`	GA$_B$	0.33 %	0.09 %	6.30 %	1.18 %	0.03 %	92.06 %
`correlation`	random sparse configuration	0.15 %	0.15 %	12.15 %	2.48 %	0.12 %	84.94 %
`correlation`	random dense configuration	55.27 %	5.79 %	19.09 %	4.00 %	0.21 %	15.64 %
`correlation`	random mixed configuration	36.70 %	3.52 %	24.09 %	4.91 %	0.33 %	30.45 %
`correlation`	random Pouchet et al	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	100.00 %
`correlation`	random Pouchet et al + completion	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	100.00 %
`correlation`	random with projection	14.27 %	0.91 %	8.79 %	3.30 %	0.24 %	72.48 %

the schedules reaches a time out. GA$_B$ and random sparse yield fewer failures than random dense, random mixed, and random projection.

E 9.3.2: *Comparison of GA$_B$ with existing Approaches and Random* (**RQ 9.3.1**, **RQ 9.3.2**, **RQ 9.3.3**) We compared the following configurations by the speedup that they yield over the execution of the original code optimized with `-O3`:

(1) ISL's adaptation of the PLuTo algorithm [152, 153, 156]

(2) random exploration with the sparse setting (random sparse)

(3) random exploration with our adaptation of Pouchet's approach and the sparse setting

(4) random exploration with our adaptation of Pouchet's approach combined with schedule completion and the sparse setting

(5) GA$_B$ running for 40 generations

The comparison to ISL's scheduler is fairer than a comparison to the original PLuTo algorithm [25] would be because the latter excludes some loop transformations that require negative coefficients for iteration variables [28]. The comparison is made solely in terms of the performance yielded by the best found program transformation.

In case of a few small SCoPs, configuration (3) did not find the intended number of 630 pairwise different schedule matrices. These are `atax` (588 schedules), `bicg` (290 schedules), `floyd-warshall` (627 schedules), `gemver` (293 schedules), `gesummv` (171 schedules), and `mvt` (129 schedules).

Figure 9.3 is a histogram that shows, per program and optimization method, the speedup over plain `-O3` yielded by the best schedule found. Table 9.4 shows the numbers plotted to allow for a detailed inspection. Also, It also shows, per configuration, the speedups' geometric mean. Using the arithmetic mean of normalized data can be misleading [56]. At mean, our iterative optimization technique yields more profitable schedules than all other configurations tested. The difference between GA$_B$ and random sparse is small and in

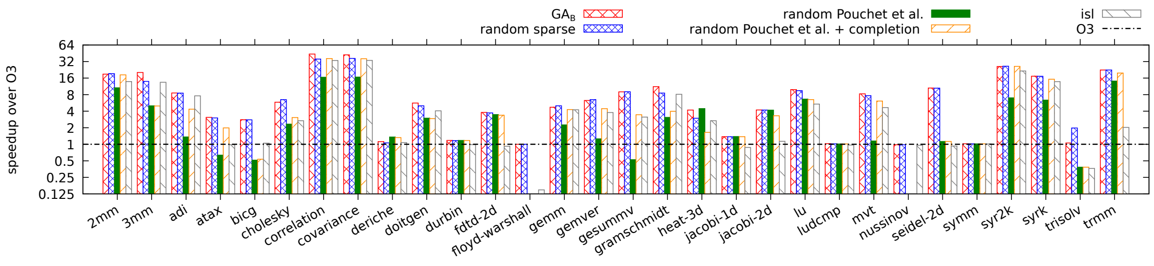


Figure 9.3: Speedups over plain `-O3` reached by different configurations.

some cases, random sparse yields a higher speedup than GA$_B$. Adding schedule completion to our adaptation of the approach by Pouchet et al. increases the mean speedup across the benchmark set yielded by the configuration. We failed to evaluate any schedule for `floyd-warshall` that was produced by our adaptation of the search space construction of Pouchet et al. within the timeout of 30 minutes that we configured for the evaluation of each schedule. This affects both the variant with schedule completion and the one without schedule completion.

Let us compare the speedups yielded by GA$_B$ to speedups yielded by the ISL scheduler in combination with POLLY's polyhedral vectorizer (`-polly-vectorizer=polly`) (refer to Table 9.1). In the case of 22 programs out of 30, the speedup yielded by GA$_B$ is higher.

We evaluated the statistical significance of the difference between the speedups yielded by the configurations tested using a pairwise Wilcoxon signed rank test [142] in combination with false discovery rate control [22]. Table 9.5 shows, per pair of configurations, the probability of erroneously assuming a difference between the configurations' results.

Table 9.6 lists the speedup over `-O3` of single-thread[18] and 8-threads[19] parallel execution of the optimal schedules found by GA$_B$. This gives an idea of the influence of the parallelization on the achieved speedups. The achieved speedups are largely due to parallelization.

Finally, we estimated the number of search space regions visited by GA$_B$ and random sparse. Each dimension of a schedule corresponds to a set of dependence polyhedra that are carried by the schedule dimension. Thus, each schedule corresponds to a list of sets of dependence polyhedra. For both GA$_B$ and random sparse, we grouped the respective sets of schedules by the schedules' associated lists of dependence polyhedra. In both cases, we counted the number of groups. Table 9.7 shows the results.

On average, GA$_B$ visits 41% of the search space regions visited by random exploration. It was to be expected that GA$_B$ visits fewer regions because our mutation operators (refer to Section 6.2.1) are designed to construct schedules that are not entirely different from the input schedules. For large SCoPs, the degree of similarity between the input and the output schedule of a mutation increases even further with an increasing number of generations of the genetic algorithm. This is due to our use of simulated annealing. Using Spearman's rank correlation [85][Chap. 4], we could show that the number of regions visited by both GA$_B$ and random sparse correlate strongly with the SCoP's number of statements. We obtain the following correlation coefficients and p-values:

$$\text{GA}_B\text{: } \rho = 0.91, p = 4.752e - 12; \text{ random sparse: } \rho = 0.96, p < 2.2e - 16.$$

The p-values are the results of a two-sided significance test.

Next, we tested for a correlation with the number of dependence polyhedra that resulted from the analysis of the legality-affecting dependences. We found that the number of

[18]In fact, we executed the parallel binary with `OMP_NUM_THREADS=1`.

[19]Different from the other experiments, we set `OMP_NUM_THREADS=8` explicitly.

Table 9.4: Speedups over plain `-O3` reached by different configurations. Figure 9.3 is a histogram of the same data.

program	**GA$_B$**	**random sparse**	**random following Pouchet et al.**	**random following Pouchet et al. + completion**	**isl**
`2mm`	18.88	19.25	10.66	18.28	13.81
`3mm`	20.20	13.92	5.01	4.98	13.39
`adi`	8.57	8.51	1.36	4.34	7.61
`atax`	3.09	3.02	0.64	1.99	0.99
`bicg`	2.79	2.78	0.52	0.54	1.05
`cholesky`	5.81	6.48	2.33	3.06	2.68
`correlation`	43.81	35.48	16.55	36.14	33.36
`covariance`	42.24	36.37	16.68	35.93	33.44
`deriche`	1.12	1.06	1.36	1.33	1.07
`doitgen`	5.63	5.01	2.99	2.93	4.06
`durbin`	1.18	1.17	1.18	1.17	0.99
`fdtd-2d`	3.80	3.72	3.46	3.38	0.92
`floyd-warshall`	1.00	1.00	–	–	0.15
`gemm`	4.72	5.00	2.25	4.29	4.23
`gemver`	6.27	6.47	1.26	4.44	3.78
`gesummv`	8.99	9.02	0.53	3.45	3.13
`gramschmidt`	11.17	8.53	3.09	3.98	8.10
`heat-3d`	4.19	2.97	4.46	1.65	2.68
`jacobi-1d`	1.37	1.37	1.38	1.37	0.88
`jacobi-2d`	4.21	4.20	4.18	3.30	1.13
`lu`	9.86	9.44	6.62	6.50	5.38
`ludcmp`	1.03	1.02	1.01	1.01	0.97
`mvt`	8.25	7.65	1.15	6.11	4.65
`nussinov`	0.98	0.98	–	–	0.96
`seidel-2d`	10.56	10.46	1.12	1.12	0.93
`symm`	1.02	1.02	1.02	1.02	1.01
`syr2k`	25.97	26.21	6.98	26.05	21.57
`syrk`	17.27	17.23	6.32	15.22	13.76
`trisolv`	1.06	1.98	0.38	0.38	0.36
`trmm`	22.43	22.40	14.04	19.75	2.02
geometric mean	5.31	5.16	2.43	3.68	2.74

dependence polyhedra and the number of search space regions visited for random sparse are weakly correlated; there is no correlation for GA$_B$:

$$\text{GA}_B\text{: } \rho = 0.40, p = 0.03;\ \text{random sparse: } \rho = 0.55, p = 0.002.$$

The test is inconclusive regarding a correlation of the estimated number of search space regions visited with the SCoP's maximum loop nest depth:

$$\text{GA}_B\text{: } \rho = 0.04, p = 0.85;\ \text{random sparse: } \rho = 0.10, p = 0.61.$$

Also, we cannot decide whether the estimated number of search space regions and the SCoP's number of structure parameters correlate:

$$\text{GA}_B\text{: } \rho = 0.04, p = 0.84;\ \text{random sparse: } \rho = 0.02, p = 0.93.$$

E 9.3.3: *Performance Distribution in the Generations of GA$_B$* (**RQ 9.3.7**) We analyzed the distribution of the speedups in execution time of the transformed program yielded by the schedules in each population of GA$_B$. The data are from **E 9.3.2**. We observed three kinds of behavior. The first kind corresponds to the majority of the GA$_B$ runs in **E 9.3.2**: the optimization starts from an initial population with a comparatively large variance of

Table 9.5: p-values obtained from a pairwise Wilcoxon signed rank test quantifying the significance of differences between different configurations of schedule optimization.

	random	**adapt. Pouchet et al.**	**adapt. Pouchet et al. + completion**	**isl**
GA$_B$	4.63e-02	5.42e-05	5.38e-05	1.12e-05
random	–	6.59e-05	6.59e-05	1.12e-05
adapt. Pouchet et al.	–	–	2.63e-02	4.63e-02
adapt. Pouchet et al. + completion	–	–	–	4.63e-02

Table 9.6: Speedups over `-O3` in execution time of the transformed program that are yielded by the optimal schedules found by GA$_B$. We show results for single-thread and 8-thread parallel execution.

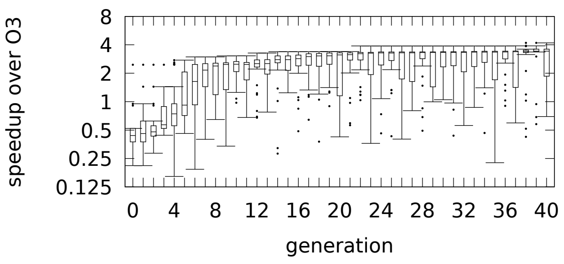
program	*2mm*	*3mm*	*adi*	*atax*	*bicg*	*cholesky*	*correlation*	*covariance*	*deriche*	*doitgen*	*durbin*	*fdtd-2d*	*floyd-warshall*	*gemm*	*gemver*
single-thread speedup	2.81	2.97	1.24	0.64	2.76	0.93	5.29	4.98	1.01	1.09	1.17	1.02	1.00	0.59	1.17
8-threads par. speedup	18.94	20.06	8.61	1.24	2.76	5.83	43.68	37.00	1.10	5.60	1.17	3.83	1.00	4.60	5.28

program	*gesummv*	*gramschmidt*	*heat-3d*	*jacobi-1d*	*jacobi-2d*	*lu*	*ludcmp*	*mvt*	*nussinov*	*seidel-2d*	*symm*	*syr2k*	*syrk*	*trisolv*	*trmm*
single-thread speedup	2.56	1.90	0.77	1.36	1.06	2.40	0.99	1.26	0.96	2.45	1.03	3.59	2.21	0.39	3.07
8-threads par. speedup	4.26	10.49	4.16	1.35	4.16	9.94	0.99	6.50	0.96	10.66	1.01	25.69	17.19	0.75	21.33

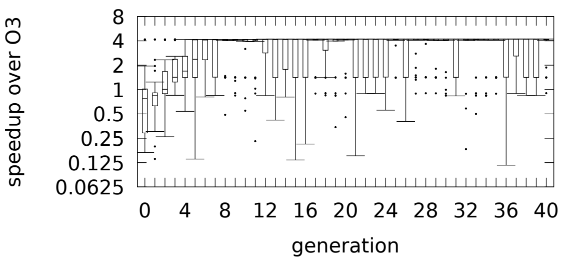
Table 9.7: The estimated number of search space regions per program that were visited by GA$_B$ and random sparse.

program	*2mm*	*3mm*	*adi*	*atax*	*bicg*	*cholesky*	*correlation*	*covariance*	*deriche*	*doitgen*	*durbin*	*fdtd-2d*	*floyd-warshall*	*gemm*	*gemver*
GA$_B$ **# regions**	104	145	265	118	47	129	236	177	244	145	182	81	30	19	142
random sparse # regions	415	542	609	284	121	388	626	552	617	369	575	348	30	43	418

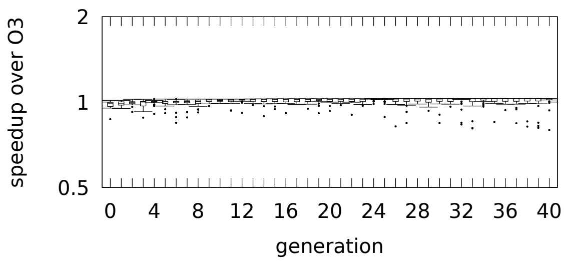
	gesummv	*gramschmidt*	*heat-3d*	*jacobi-1d*	*jacobi-2d*	*lu*	*ludcmp*	*mvt*	*nussinov*	*seidel-2d*	*symm*	*syr2k*	*syrk*	*trisolv*	*trmm*
	84	134	29	13	27	42	239	21	286	17	211	22	29	61	59
	149	587	153	52	166	268	629	17	624	39	531	36	36	229	150



(a) `heat-3d`



(b) `jacobi-2d`



(c) `ludcmp`

Figure 9.4: Box plots showing the distribution of performance within the generations of $\text{GA}_{\mathcal{B}}$. The data are from **E 9.3.2**. The whiskers extend from the ends of the box to the most distant point whose value lies within 1.5 times the interquartile range. Points that lie beyond that distance are drawn as outliers [165].

the speedups yielded by the schedules in the initial population. The maximum speedup is lower than the optimal speedup that $\text{GA}_{\mathcal{B}}$ reaches eventually. The variance of the speedups yielded by the schedules in the population diminishes while the optimization proceeds. We illustrate such a case in Figure 9.4(a). The horizontal axis enumerates the populations. The vertical axis shows speedup over `-O3`. The second largest group of runs corresponds to the behavior shown in Figure 9.4(b). The initial population contains a schedule that yields a speedup that is close to the optimal speedup that is reached. Over time, the variance of the speedups yielded by the schedules in the population diminishes and the median speedup converges towards the maximum. Finally, in the cases of programs `ludcmp`, `nussinov`, and `symm`, we are unable to find a schedule that reduces the execution time below the duration of the original sequential programs after its optimization with `-O3`. The variance of the speedups yielded by the schedules in the populations is very low throughout the optimization process. We illustrate this case for `ludcmp` in Figure 9.4(c).

Figure 9.4 exhibits a notable property of the genetic algorithm: it delivers a set of profitable schedules, while optimizing a SCoP requires just one. We wanted to know how many semantically different schedules the set contains. To determine the according equivalence classes, we prepared the schedules by transforming them to simplified schedule trees in the same way as we would prepare them for code generation by Polly (refer to Section 5.6). Subsequently, we encoded rectangular loop nest tiling in the schedule trees following the steps of Polly's tiling strategy. Following Vasilache [149], we consider the schedules Θ_1 and Θ_2 to be equivalent if

$$\Big(\forall I_X, I_Y \in I : \Big(\forall(\vec{i}_X, \vec{i}_Y) \in I_X \times I_Y : \big(\Theta(\vec{i}_X) \prec \Theta(\vec{i}_Y)\big) \Leftrightarrow \big(\Theta'(\vec{i}_X) \prec \Theta'(\vec{i}_Y)\big)\Big)\Big)$$

holds after the encoding of tiling in both schedules (refer to Section 5.5.4 for details). In Table 9.8, we show, per program, the number of equivalence classes in the final generation of the $\text{GA}_{\mathcal{B}}$ run that we performed in **E 9.3.2**.

We tested the number of equivalence classes in $\text{GA}_{\mathcal{B}}$'s final population for correlation with the SCoP's number of statements, the number of dependence polyhedra, and the maximum

Table 9.8: Per program, the number of equivalence classes in the final generation of the GA$_B$ run that we performed in **E 9.3.2**.

program	2mm	3mm	adi	atax	bicg	cholesky	correlation	covariance	deriche	doitgen	durbin	fdtd-2d	floyd-warshall	gemm	gemver
# equiv. classes	24	16	23	7	8	10	25	28	18	6	8	15	3	21	10

program	gesummv	gramschmidt	heat-3d	jacobi-1d	jacobi-2d	lu	ludcmp	mvt	nussinov	seidel-2d	symm	syr2k	syrk	trisolv	trmm
# equiv. classes	5	26	30	6	11	21	6	27	16	18	7	13	10	8	10

loop nest depth using Spearman's rank correlation [85][Chap. 4]. In all three cases, the test is inconclusive:

$$\begin{aligned}\text{number of statements: } & \rho = 0.17, p = 0.37\\ \text{number of dependence polyhedra: } & \rho = 0.15, p = 0.42\\ \text{maximum loop nest depth: } & \rho = 0.30, p = 0.10.\end{aligned}$$

The complexity of the remaining experiments that we present in Section 9.3.2 forced us to limit them to one program. Of the three programs that have been studied in more detail already, we chose `3mm` since its numbers of statements and dependence polyhedra and its maximum loop depth are closer to the respective median values across PolyBench 4.1 than those of `adi` and `correlation`. With respect to the average values, `3mm` is closer to average in the number of statements than `adi` and `correlation` but farther away from average than `correlation` in the number of dependence polyhedra.

E 9.3.4: *Effectiveness of the Genetic Operators* (**RQ 9.3.6**) To evaluate and compare the effectiveness of the genetic operators, we optimized the schedule of `3mm` with six configurations of GA$_B$. In each configuration, one operator was disabled. Otherwise, the configurations meet the specification in Section 9.1.4. We ran each configuration ten times, starting from one of ten initial populations. GA$_B$ was configured to run for a fixed number of 20 generations. As in **E 9.3.1**, we reduced the number of time measurements per schedule to three. A strong effect of disabling any of the operators on the convergence speed or the profitability of the optimal found schedule cannot be observed. Yet, from **E 9.3.1** and **E 9.3.3**, we know that the last generation consists mostly of profitable schedules and that the mean speedup in execution time yielded by GA$_B$ is higher than the speedup yielded by random sparse. So, in combination, our genetic operators are effective.

Figure 9.5 shows, per configuration tested, the median speedup of the optimal schedule known after visiting n schedules.

E 9.3.5: *Intensity of the Mutations* (**RQ 9.3.6**) We evaluated the effect of varying the genetic algorithm's mutations' intensity. We set the initial intensity to 10%, and later increased it to 30%, and then to 60%. GA$_B$ was configured to run for a fixed number of 20 generations. As in **E 9.3.1**, we reduced the number of time measurements per schedule to three. With each configuration, we optimized `3mm` ten times, starting from one out of ten initial populations. Increasing the mutations' intensity does not affect the convergence speed of the GA or the profitability of the optimal schedule found. This was to be expected: in **E 9.3.1**, we already noticed that GA$_B$, with its mutation intensity set to 10%, does not converge significantly faster than random sparse. Furthermore, we expect the genetic algorithm to degenerate to a purely random search as we increase the mutation intensity. So, in our case, increasing the intensity of mutations should not yield faster convergence.

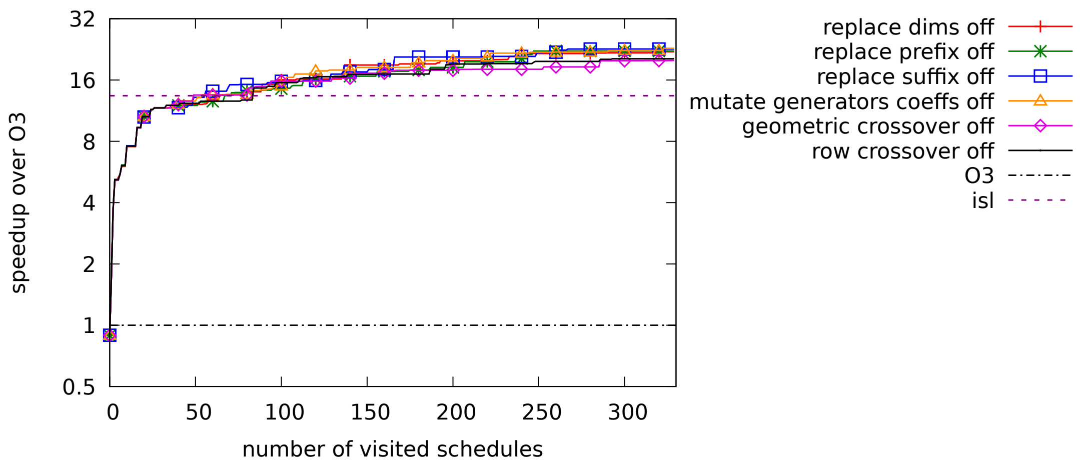


Figure 9.5: Per configuration tested in **E 9.3.4**, the median speedup of the optimal schedule known after visiting n schedules. The horizontal axis is n, the vertical axis is speedup in execution time over `-O3`.

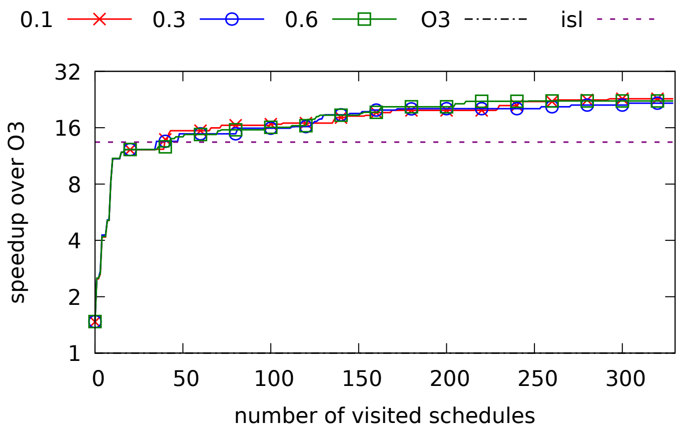


Figure 9.6: Per configuration tested in **E 9.3.5**, the median speedup of the optimal schedule known after visiting n schedules. The horizontal axis is n, the vertical axis is speedup in execution time over `-O3`.

Due to the small size of the programs considered, a mutation intensity below 10% should not yield different results, either.

Figure 9.6 shows, per configuration tested, the median speedup of the optimal schedule known after visiting n schedules.

9.3.3 Discussion

We can answer **RQ 9.3.1** positively, based on the results of **E 9.3.2**. Iterative optimization detects program transformations yielding a higher performance than those found by ISL's variant of the PLuTo algorithm. There are many possible reasons. The underlying assumptions of the PLuTo algorithm may not apply in every case: it maximizes the size of tilable bands, which our iterative approach does not. In their hybrid search-based and model-driven approach to schedule optimization, Pouchet et al. [126] search for a good partitioning of a SCoP's statements into classes with different execution date and then schedule the statement instances in each partition in a model-driven way. Their motivation is that cost models for scheduling inside each class are well understood, while those for the partitioning itself are not. Likewise, our iterative approach may profit from this fact by finding a better partitioning. It is likely that, on different hardware, the iterative optimization would identify different optimal schedules. Furthermore, the tile shapes chosen by the PLuTo algorithm may be suboptimal. Iterative optimization may find different ones. In the end, performance optimization is hardware-specific. Naturally, iterative optimization can adapt itself more easily than model-driven algorithms.

Let us explain the higher performance of the code optimized by Polyite compared to Polly's result exemplary for `trmm` and `jacobi-2d`. Polly yields a speedup of 2.02 over `-O3` for `trmm`, while Polyite yields 22.43 at best. Both, Polly and Polyite segregate `trmm`'s statements into fully separate loop nests. Polly can only tile the second loop nest. Polyite interchanges the loops in the first nest, skews them in another way than Polly, and can tile both loop nests. For `jacobi-2d`, the speedup yielded by Polly is 1.13 while Polyite yields 4.21. Polly fuses the two statements and cannot parallelize while Polyite splits the loop nest inside the time loop and both inner loop nests can be parallelized at their outermost level.

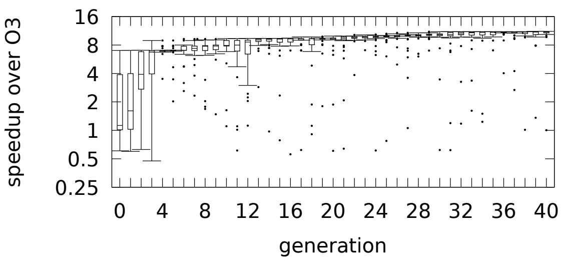
Some cases in which the speedup in execution time yielded by Polly is lower than the speedup yielded by Polyite may be due to isl not selecting a schedule that yields outer parallelism. Manually forcing outer parallelism, which is possible in isl's scheduling algorithm [156], may sometimes improve the speedup yielded by Polly. Versions of Polly that are newer than the version used in our experiment have a command line option that allows to force outer parallelism manually.

E 9.3.2 revealed that random exploration with the sparse configuration almost always finds schedules that yield better performance than random exploration with our adaptation of the search space construction by Pouchet et al. [124]. The latter profits significantly from schedule completion but, still, random sparse yields significantly higher speedups. These findings strengthen the assumption that an optimization for parallelization and tiling likely requires the exploration of a larger or different subset of the schedule search space in combination with schedule completion. Thus, we can answer **RQ 9.3.2** positively.

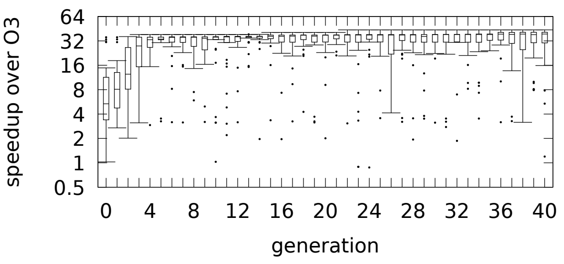
We continue by discussing **RQ 9.3.3**. **E 9.3.2** revealed that the maximum speedup in execution time yielded by the schedules found by GA_B are higher at mean than the maximum speedups yielded by random sparse. The speedups yielded by GA_B and random sparse differ significantly. In **E 9.3.1**, we found that, in the cases of `adi` and `correlation`, GA_B and random sparse converge at the same speed but, in the case of `3mm`, GA_B converges to a higher maximum speedup in the execution time of the program optimized. Furthermore, we found that GA_B tends to find fewer ineffective schedules than random sparse and yields fewer schedules that we cannot test successfully. So, in summary, we observed a small advantage of our genetic algorithm over random sampling and can answer **RQ 9.3.3** positively.

On the basis of **E 9.3.1**, we can answer **RQ 9.3.4** positively. Sparse schedule matrices generated randomly yield higher speedups than dense coefficient matrices generated randomly. Also, random sparse yields fewer schedules that we cannot test successfully than random dense. The effect is strong for `3mm` and `correlation`. It is less pronounced for `adi`. These findings conform to the prediction of Pouchet et al. [124] that picking schedule coefficients close to zero is beneficial. Also, schedule matrices formed from a large number of generators tend to contain larger coefficients, which may trigger integer overflows at run time.

Chernikova sampling has several advantages for our approach to explore the search space of legal schedules and, in particular, for our genetic algorithm. Among the disadvantages of Chernikova sampling are rarely occurring rays with extremely large components, points with rational coordinates that have large denominators (refer to Section 4.6.5.1), and the high run-time complexity of Chernikova's algorithm, which makes Chernikova sampling impractical for large SCoPs. Projection sampling does not have these disadvantages. In



(a) `gramschmidt`



(b) `correlation`

Figure 9.7: Box plots showing the distribution of performance within the generations of GA_B.

E 9.3.1, we compared the distribution of the speedups yielded by the schedules that resulted from random exploration with a configuration of projection sampling and random sparse. We found that the schedules generated by the tested configuration of projection sampling perform slightly worse than the schedules generated by random sparse. This difference is statistically significant. To answer **RQ 9.3.5**, we are confident that tuning of its configuration will allow projection sampling to perform equally well in random exploration.

In **E 9.3.4** and **E 9.3.5**, we challenged the configuration of our genetic algorithm that we used in the experiments presented. In **E 9.3.4**, we did not observe that disabling any genetic operator has an effect on the genetic algorithms convergence speed. Further, in **E 9.3.5**, we did not observe that a higher initial intensity of the mutations yields a better optimization result or leads to faster convergence. To answer **RQ 9.3.6**, we assume that our choices for the configuration of our genetic algorithm are justified. Of course, other aspects, such as the population size and the number of generations that the genetic algorithm performs, have an influence on the optimization result, as well. Finally, there may be an interaction between different configuration options.

Replacing a single coefficient in a schedule matrix can unarguably impact performance strongly, for instance, by skewing a loop of distributing two previously fused statements. Thus, one must not expect to find locality in a sense that a small change to a schedule usually leads to a similarly or even better performing schedule in the search space of schedule matrices. Yet, Pouchet et al. [124] control the intensity of the mutations in their genetic algorithm by an annealing factor and we adopted this choice. This leads us to the discussion of **RQ 9.3.7**.

Let us discuss **E 9.3.3**, in which we studied GA_B's behavior with a focus on its convergence. We analyzed the distribution of the speedups yielded by the schedules in each generation of the genetic algorithm. The reduction of the variance of the speedups that lie above the median is primarily a consequence of our genetic algorithm's elitism. Yet, as can be seen in Figure 9.7 for the programs `correlation` and `gramschmidt`, the variance of speedups below the median reduces as well. We observed this behavior also for several other programs. Consequently, the search does become more local over time. At the same time, the insertion of schedules generated randomly and the crossovers reduce the probability that the search becomes trapped at a local optimum. In particular, the geometric crossover has the ability to produce offspring schedules that differ strongly from their parents. We expect that the mutation of schedule matrices by partial schedule replacement, which we propose as an outlook in Section 6.2.1.5, would further increase the genetic algorithm's ability to escape from local optima more easily without losing all properties of the mutation's input schedule.

9.4 Genetic Algorithm with Schedule Classification

The second part of the evaluation focuses on the surrogate performance models for schedules that we propose in Chapter 7 and the combination of these models with our genetic algorithm for schedule optimization. In Section 7.2, we propose two ways of combining our genetic algorithm and a schedule classifier. In the evaluation, we focus on the two-staged approach, which entirely replaces benchmarking by classification in the genetic algorithm and uses benchmarking only to determine the most profitable schedule among the schedules in the genetic algorithm's final population. We refer to this combination by GA_C.

9.4.1 Research Questions

The overarching goal of what we present in Chapter 7 is to make POLYITE's genetic algorithm time-efficient to the point of practicality. To reach this goal, we replace most of the time-consuming fitness assessment of schedules that is based on benchmarking with a classifier that uses a machine-learned surrogate performance model. Thanks to the classifier, only the schedules in the genetic algorithm's final population require benchmarking. We train the classifier based on feature vectors that we extract from schedules that result from previous iterative optimizations, for instance, from earlier versions of the same program. The classification of schedules must be computationally cheap to achieve a saving of optimization time. In Section 7.1.4, we verified already that the extraction of the schedules' feature vectors is possible within a few hundred milliseconds to a small number of seconds for SCoPs with tens of statements and tens up to a few hundreds of dependence polyhedra. Moreover, the surrogate model must be transferable from the programs used for training to new ones. In the end, the whole optimization process must still obtain sufficient speedups.

RQ 9.4.1: *Are our schedule features sensitive to profitability of schedules?*
Given two schedules for a SCoP that yield strongly different execution times of the transformed code, we call a feature *sensitive* to performance if its value differs between the two schedules. Some of the features that Section 7.1 presents may not be sensitive to the profitability of schedules. The features that are insensitive to the transformed programs execution time can be excluded from the feature vector.

RQ 9.4.2: *Can our classifiers distinguish reliably between profitable and unprofitable program schedules?*
Is the difference between profitable and unprofitable schedules for a SCoP expressible using the features of Section 7.1 and decision tree classifiers? Can a model be learned on schedules of a set of programs and then be used to classify schedules of another program correctly?

RQ 9.4.3: *Is our schedule classifier an acceptable surrogate for benchmarking?*
We must compare running GA_C (refer to Section 7.2.2), GA_B and random exploration with respect to two criteria. The first criterion is the time saved by the use of the classifier. The second one is the optimization's outcome.

9.4.2 Training Sets

The experiments presented in Section 9.4.3 require training sets for schedule classifiers. A training sample refers to one schedule and consists of a vector of feature values and a label that indicates whether the corresponding schedule is profitable or unprofitable. A feature value is a real number.

To obtain a set of training samples for a program, we ran GA_B over 40 generations. This corresponds to generating 630 schedules. We generated another 630 schedules using random exploration with the sparse setting. We merged the two sets of schedules and filtered for schedules that we could benchmark successfully. The merging procedure considers two schedule matrices equivalent if they share the same rational coefficient matrix and if the

Table 9.9: Characteristics of the training sets. Per program, we show the total number of schedules in the training set and the numbers of profitable and unprofitable schedules.

program	2mm	3mm	adi	atax	bicg	cholesky	correlation	covariance	deriche	doitgen	durbin	fdtd-2d	floyd-warshall	gemm	gemver
# schedules	1112	1060	1166	1184	1175	805	1136	1192	1070	1255	1231	832	590	1222	1165
# profitable	488	301	472	421	639	106	462	605	0	550	0	421	0	756	395
# unprofitable	624	759	694	763	536	699	674	587	1070	705	1231	411	590	466	770
% profitable	43.88 %	28.40 %	40.48 %	35.56 %	54.38 %	13.17 %	40.67 %	50.76 %	0.00 %	43.82 %	0.00 %	50.60 %	0.00 %	61.87 %	33.91 %

program	gesummv	gramschmidt	heat-3d	jacobi-1d	jacobi-2d	lu	ludcmp	mvt	nussinov	seidel-2d	symm	syr2k	syrk	trisolv	trmm
# schedules	1172	996	857	1167	1103	907	793	1086	1227	1201	1241	1209	1190	1094	1193
# profitable	569	484	248	836	330	316	0	508	0	513	0	740	712	345	485
# unprofitable	603	512	609	331	773	591	793	578	1227	688	1241	469	478	749	708
% profitable	48.55 %	48.59 %	28.94 %	71.64 %	29.92 %	34.84 %	0.00 %	46.78 %	0.00 %	42.71 %	0.00 %	61.21 %	59.83 %	31.54 %	40.65 %

rows of the matrices result from the same linear combination of points, rays, and lines. This is the equivalence relation that we also apply to test whether a schedule should enter the genetic algorithm's population. The result of the merge is a set of up to 1260 schedules that covers reasonably the profitability range of the schedules in which we are interested.

We calculated the schedules' feature values and labeled each schedule as either profitable or unprofitable. Recall from Section 7.2, that a schedule is profitable if it yields an execution time of the transformed code that is not longer than twice the execution time yielded by the best known schedule for the program. When labeling schedules for a given SCoP as training data we label all schedules as unprofitable if none of the schedules yields a speedup over the execution time of the original sequential code after its optimization with `-O3` that is higher than 1.2. We use the optimal configuration found to determine the extent to which knowledge regarding schedules' profitability can be transferred from one program to another.

Table 9.9 characterizes the training sets. The total number of schedules in all training sets together is 29613.

9.4.3 Experiments

We conducted the experiments that we present in the following to shed light on the research questions in Section 9.4.1. Together with the title of each experiment, we reference the research questions to which the experiment relates.

As described in Section 9.2, we exclude the program `nussinov` from the experiments. We are unable to obtain sufficient information for the correct estimation of the data volumes that are communicated by the legality-affecting dependence polyhedra of `nussinov`. Furthermore, we exclude the programs `seidel-2d` and `floyd-warshall`, which are the only two programs in PolyBench 4.1 that Polly models with a single statement.

E 9.4.1: *Sensitivity of Features* (**RQ 9.4.1**) The sparsity of parameters' coefficients and the constant appears to be the least sensitive feature. Its value is almost always between 0.75 and 1 and barely differs between schedules with different profitability. Thus, we did not use the feature in the other experiments.

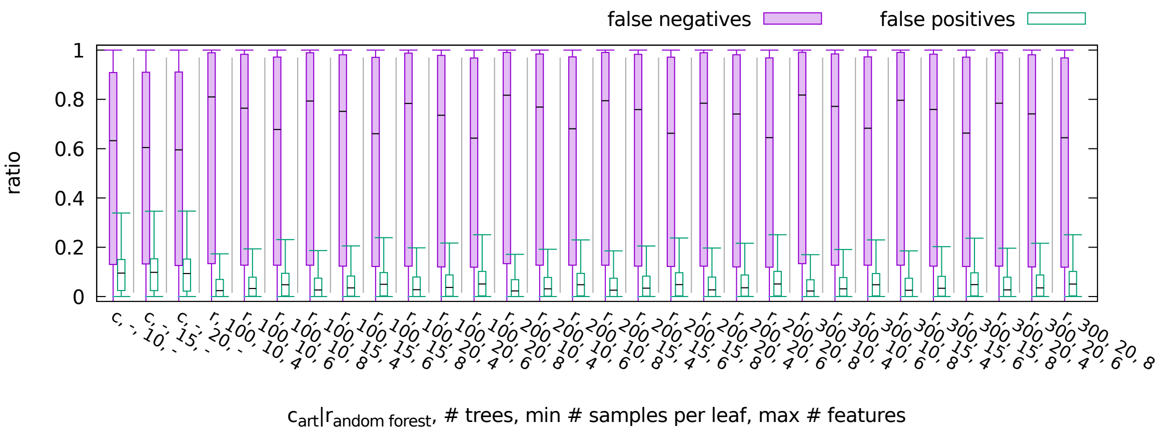


Figure 9.8: Comparison of different configurations of the learning algorithm for the schedule classifiers. Per configuration tested, we show two box plots that express the distributions of the rates of false negatives and false positives that we collected while testing the configuration for each p and combinations of training set and verification set of schedules.

E 9.4.2: *Leave-p-Out Across Programs* (**RQ 9.4.1**, **RQ 9.4.2**) In this experiment, we tested different configurations of the learning algorithms CART and random forest (refer to Section 2.3.2). Our aim was to determine an effective configuration for the learning of schedule classifiers that can distinguish between profitable and unprofitable schedules.

Regarding the learning algorithm's configuration, we investigated primarily the choice between random forest and CART with respect to their classification accuracy. Furthermore, both techniques have a number of configuration parameters that substantially influence the classifiers' accuracy. We tuned the most important ones. The minimum number of data points that are classified in a leaf node of a classification tree (`min_samples_leaf`) controls the tree's depth. By setting the parameter properly, overfitting of the model learned can be prohibited. We tested the values 10, 15, and 20. The motivation for the maximum tested value 20 is that, given the total number of schedules in the union of our training sets and assuming a balanced decision tree, each feature can be considered at least once on a path from a decision tree's root to a leaf and two features can be considered twice. The parameter `max_features` controls the number of features chosen randomly to consider when searching for the optimal splitting criterion for a node of a single classification tree. Since CART constructs only one decision tree, we allowed it to inspect all features. For random forest, we tested the values 4, 6, and 8. With random forest, the number of trees in a forest (`n_estimators`) is another parameter that requires tuning. More trees yield more stable predictions, but longer computations. We tested with 100, 200, and 300 trees. Since we use bootstrapping, each tree is learned from a subset of the training data chosen randomly.

To test each configuration, we removed the schedules of $p \in \mathbb{N}$ programs from the training set and learned a performance model on the remaining ones. Then, we classified the removed schedules and verified the predictions. In this way, we were able to determine the extent to which our performance models are transferable to unseen programs for each configuration. The schema is known as *leave-p-out*. We scaled p from 1 up to 5. Per program and configuration, we determined the ratio of actually profitable schedules classified wrongly (the *false negatives*) relative to the number of actually profitable schedules and the ratio of actually unprofitable schedules classified wrongly (the *false positives*) relative to the number of actually unprofitable schedules. Figure 9.8 shows, per configuration of the learning algorithm, the distribution of the ratios of false positives and false negatives across all values of p and combinations of training set and verification set of schedules. CART yields notably more false positives at median than random forest. The following configuration

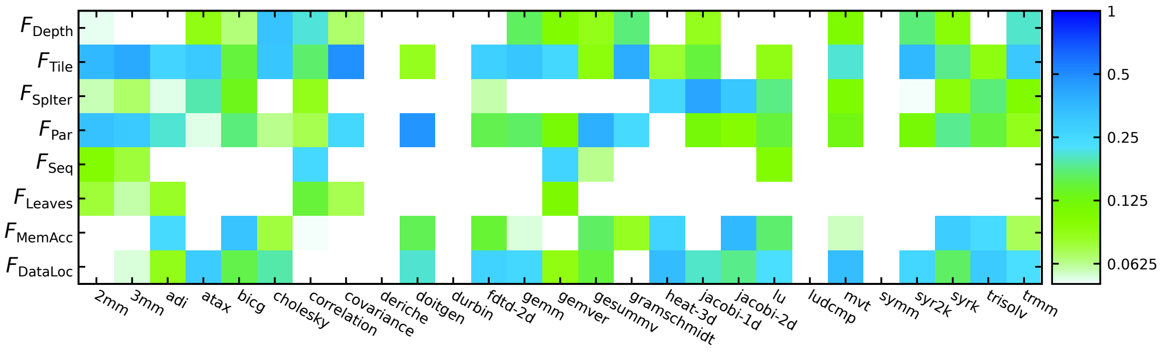


Figure 9.9: A heat map that shows the GINI importance per feature and program.

is among the random forest configurations that yield the lowest median numbers of false negatives: `n_estimators = 100`, `max_features = 8`, `min_samples_leaf = 20`. Among all comparable configurations it uses the smallest number of trees per random forest. We used this configuration in all other experiments.

Table 9.10 shows the results for this optimal configuration of random forest.

The average share of false negatives among the profitable schedules is 65% for $p = 1$ and 66% in the remaining cases. In contrast to that, the average share of false positives among the unprofitable schedules is only 8% for all values of p. For some programs, the share of false negatives among the profitable schedules is extreme.

To explain this situation, we started by computing the features' importance for splitting each program's training set into profitable and unprofitable schedules. The heat map in Figure 9.9 shows the GINI importance [103] per feature and program.

The *GINI importance* of a feature F is the sum of the decreases of GINI impurity at a decision tree's nodes whose splitting criterion is based on F. The *impurity decrease* is the difference between the GINI impurity at a node and the weighted sum of the GINI impurity at its children. In case of random forests, the value is normalized by the number of trees. To compute the GINI importances, we learned random forests with the following configuration: `min_samples_leaf = 20`, `max_features = 2`, and `n_estimators = 500`. Again, we use bootstrapping, which means that we learn each tree from a randomly chosen subset of a program's training set of schedules. The large number of trees per random forest and the low number of features considered for determining a splitting condition result in a diverse set of trees in each forest. In contrast, deriving the GINI importances from a single CART learned with a configuration that requires the consideration of all features for determination of each splitting condition would identify primarily the most important feature per program, but it would not yield strong evidence for the other features.

Per program, we assigned a rank to each feature according to its importance compared to the other features. Next, we summated each feature's ranks. In cases in which all features had an importance of 0 we assigned rank 0 to all features. In Table 9.11, we enumerate the features, from the most important to the least important, and show their rank sums. Furthermore, we learned a random forest from all programs' training sets together and determined the features' GINI importances. Table 9.12 shows the results. Apart from the ordering of the two least important features, the features' order is the same as in Table 9.11.

The ranking of the features that we derived does not fit all programs in the benchmark set. This is a first indicator that, with our set of features, we cannot learn a model that allows us to distinguish successfully profitable from unprofitable schedules of arbitrary programs.

Another aspect that we must take into account is the value range of the features. For instance, F_{Tile} is useful for the distinction of profitable from unprofitable schedules of both `3mm` and `adi` but, as can be seen from Figure 9.10, the feature's value ranges for the two

Table 9.10: Average results per combination of program and p for the optimal configuration of random forest that we determined in **E 9.4.2**. Listed are, per program and p, the average rate of false positives among the unprofitable schedules, the average rate of false negatives among all profitable schedules, among the schedules that yield $\geq 80\%$ of the speedup, and among the schedules that yield $\geq 95\%$ of the speedup. Finally, we show the average rate of mispredictions per combination of program and p. In the absence of profitable schedules for a program, the share of false negatives is undefined, which we indicate by dashes.

program	**p**	2mm	3mm	adi	atax	bicg	cholesky	correlation	covariance	deriche	doitgen	durbin	fdtd-2d	gemm	gemver	gesummv	gramschmidt	heat-3d	jacobi-1d	jacobi-2d	lu	ludcmp	mvt	symm	syr2k	syrk	trisolv	trmm	**row avg.**
% false pos.	1	4	16	3	14	4	1	10	9	0	2	0	2	10	8	14	0	44	5	3	2	0	51	0	10	7	0	8	8
% false neg.	1	13	4	97	70	97	100	91	55	–	100	–	99	13	88	21	100	51	97	68	97	–	45	–	7	41	100	49	65
% false neg. $\geq 80\%$	1	3	1	97	84	98	100	92	58	–	100	–	99	13	90	14	100	38	100	65	98	–	30	–	5	41	100	27	63
% false neg. $\geq 95\%$	1	2	2	100	96	98	100	100	92	–	100	–	99	57	100	13	100	0	100	65	99	–	0	–	2	56	100	25	65
total % mispred.	1	8	12	41	34	55	14	43	32	0	45	0	51	12	35	17	49	46	71	22	35	0	48	0	8	28	32	25	28
% false pos.	2	4	16	3	14	3	0	10	9	0	2	0	2	11	8	14	0	23	5	3	1	0	50	0	9	7	3	8	8
% false neg.	2	15	5	97	64	96	100	90	52	–	99	–	99	12	86	43	100	70	97	69	97	–	40	–	7	36	94	49	66
% false neg. $\geq 80\%$	2	5	1	97	75	97	99	93	54	–	100	–	99	13	88	40	100	46	100	68	99	–	22	–	5	34	98	28	63
% false neg. $\geq 95\%$	2	2	2	99	74	97	100	99	87	–	100	–	99	63	99	40	100	46	100	67	99	–	0	–	3	44	96	24	67
total % mispred.	2	9	13	41	32	54	13	43	31	0	45	0	51	12	34	28	49	36	71	23	35	0	45	0	8	24	31	25	28
% false pos.	3	5	16	4	14	4	0	10	10	0	4	0	2	11	8	13	0	17	5	3	2	0	49	0	9	7	3	9	8
% false neg.	3	17	5	95	63	95	100	90	50	–	97	–	99	12	84	54	100	74	97	73	97	–	35	–	7	37	93	49	66
% false neg. $\geq 80\%$	3	6	1	95	76	95	98	93	51	–	100	–	99	13	87	54	100	50	100	72	98	–	19	–	5	35	96	29	64
% false neg. $\geq 95\%$	3	3	2	98	75	95	100	99	80	–	100	–	99	64	96	54	100	57	100	72	98	–	0	–	3	46	96	25	68
total % mispred.	3	10	13	41	31	53	13	43	30	0	45	0	51	12	34	33	49	34	71	24	35	0	42	0	8	25	32	25	28
% false pos.	4	5	16	5	13	4	0	10	10	0	5	0	2	11	8	13	0	15	5	2	2	0	47	0	9	7	4	9	8
% false neg.	4	18	5	94	63	94	100	90	49	–	96	–	99	12	83	60	100	76	97	77	96	–	33	–	7	38	90	48	66
% false neg. $\geq 80\%$	4	8	1	94	77	94	98	92	49	–	99	–	99	12	86	61	100	53	100	76	98	–	16	–	5	37	96	29	64
% false neg. $\geq 95\%$	4	4	2	98	77	95	100	98	75	–	99	–	99	64	95	60	100	66	100	75	98	–	0	–	3	49	96	25	69
total % mispred.	4	11	13	41	31	53	13	43	30	0	45	0	51	12	33	36	49	33	71	25	35	0	40	0	8	25	31	25	28
% false pos.	5	5	16	5	13	5	0	10	10	0	6	0	2	11	8	13	0	14	5	2	2	0	46	0	9	7	5	9	8
% false neg.	5	20	5	93	64	93	99	90	48	–	95	–	99	12	82	63	100	77	97	78	96	–	31	–	7	39	89	48	66
% false neg. $\geq 80\%$	5	10	1	94	78	94	97	92	47	–	99	–	99	12	86	64	100	56	100	77	97	–	15	–	5	39	97	29	65
% false neg. $\geq 95\%$	5	4	2	98	79	94	100	98	71	–	99	–	99	63	94	64	100	69	100	77	97	–	0	–	3	52	96	25	69
total % mispred.	5	12	13	41	31	53	13	43	29	0	45	0	51	12	33	37	49	32	71	25	35	0	39	0	8	26	31	25	28

Table 9.11: Our schedule features ordered by their importance across the set of benchmark programs. We show each feature's rank sum.

feature	F_{Tile}	F_{Par}	F_{DataLoc}	F_{MemAcc}	F_{SpIter}	F_{Depth}	F_{Seq}	F_{Leaves}
rank sum	124	109	104	79	76	65	52	40

Table 9.12: GINI importances of our features for the union of all program's training sets.

feature	F_{Tile}	F_{Par}	F_{DataLoc}	F_{MemAcc}	F_{SpIter}	F_{Depth}	F_{Leaves}	F_{Seq}
GINI importance	0.21	0.20	0.16	0.15	0.11	0.08	0.06	0.05

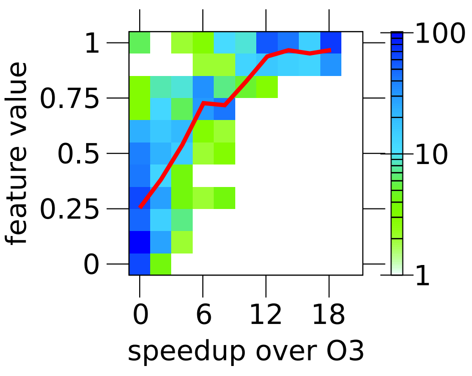
programs differ. This complicates the transfer of knowledge regarding the profitability of schedules among programs further.

To understand why the accuracy of our classification is particularly low in the cases of `adi`, `bicg`, `cholesky`, `doitgen`, `fdtd-2d`, `gramschmidt`, `jacobi-1d`, `lu`, and `trisolv`, we learned a strongly simplified decision tree from the union of the training sets of all programs in the benchmark set. We achieved the simplification by requiring that each leaf must correspond to at least 1500 schedules in the training set. Figure 9.11 shows this decision tree. We pruned subtrees whose leaf nodes share the same majority class of the corresponding training samples.

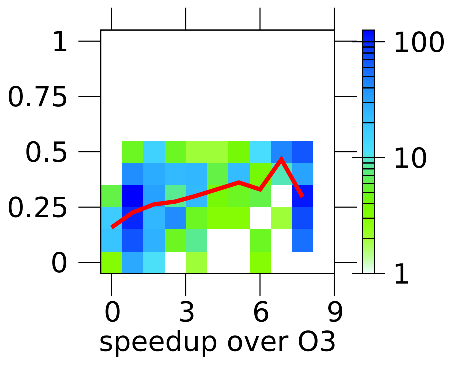
On the basis of the tree, we could identify three primary classes of schedules that correspond to good performance:

1. Strong parallelism and good applicability of tiling: $F_{\text{Par}} > 0.766 \;\wedge\; F_{\text{Tile}} > 0.845$
2. Strong parallelism and mostly beneficial memory access pattern:
   $F_{\text{Par}} > 0.766 \;\wedge\; F_{\text{Tile}} < 0.845 \;\wedge\; F_{\text{MemAcc}} > 0.558$
3. Medium or weak parallelism, some tiling, not too much skewing, very good memory access pattern: $F_{\text{Par}} \leq 0.766 \;\wedge\; F_{\text{Tile}} > 0.22 \;\wedge\; F_{\text{SpIter}} > 0.356 \;\wedge\; F_{\text{MemAcc}} > 0.881$

We marked the respective paths from the tree's root. With the exception of some schedules of `jacobi-1d` that are in class 3, the grand majority of the schedules of the programs that we listed above do not correspond to any of the three classes. The subtree that is marked by the



(a) `3mm`



(b) `adi`

Figure 9.10: Comparison of the value distribution of F_{Tile} between `3mm` and `adi`. The horizontal axis is speedup, the vertical axis is the feature value. The color of the squares expresses the number of schedules whose feature value is located at a position.

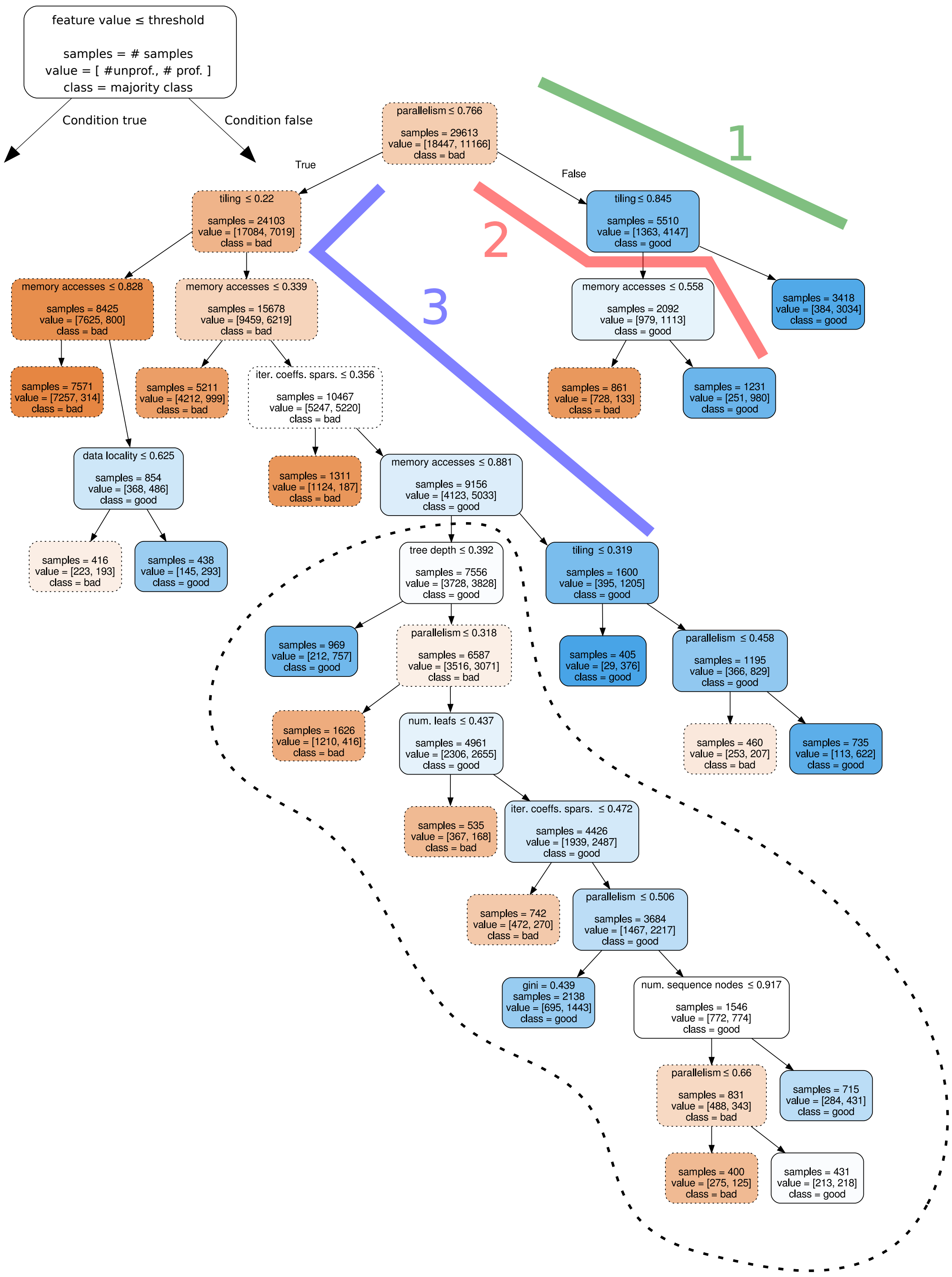

feature value ≤ threshold
samples = # samples
value = [#unprof., # prof.]
class = majority class
Condition true
Condition false
parallelism ≤ 0.766 samples = 29613 value = [18447, 11166] class = bad
True
False
1
2
3
tiling ≤ 0.22 samples = 24103 value = [17084, 7019] class = bad
tiling ≤ 0.845 samples = 5510 value = [1363, 4147] class = good
memory accesses ≤ 0.828 samples = 8425 value = [7625, 800] class = bad
memory accesses ≤ 0.339 samples = 15678 value = [9459, 6219] class = bad
memory accesses ≤ 0.558 samples = 2092 value = [979, 1113] class = good
samples = 3418 value = [384, 3034] class = good
samples = 7571 value = [7257, 314] class = bad
samples = 5211 value = [4212, 999] class = bad
iter. coeffs. spars. ≤ 0.356 samples = 10467 value = [5247, 5220] class = bad
samples = 861 value = [728, 133] class = bad
samples = 1231 value = [251, 980] class = good
data locality ≤ 0.625 samples = 854 value = [368, 486] class = good
samples = 1311 value = [1124, 187] class = bad
memory accesses ≤ 0.881 samples = 9156 value = [4123, 5033] class = good
samples = 416 value = [223, 193] class = bad
samples = 438 value = [145, 293] class = good
tree depth ≤ 0.392 samples = 7556 value = [3728, 3828] class = good
tiling ≤ 0.319 samples = 1600 value = [395, 1205] class = good
samples = 969 value = [212, 757] class = good
parallelism ≤ 0.318 samples = 6587 value = [3516, 3071] class = bad
samples = 405 value = [29, 376] class = good
parallelism ≤ 0.458 samples = 1195 value = [366, 829] class = good
samples = 1626 value = [1210, 416] class = bad
num. leafs ≤ 0.437 samples = 4961 value = [2306, 2655] class = good
samples = 460 value = [253, 207] class = bad
samples = 735 value = [113, 622] class = good
samples = 535 value = [367, 168] class = bad
iter. coeffs. spars. ≤ 0.472 samples = 4426 value = [1939, 2487] class = good
samples = 742 value = [472, 270] class = bad
parallelism ≤ 0.506 samples = 3684 value = [1467, 2217] class = good
gini = 0.439 samples = 2138 value = [695, 1443] class = good
num. sequence nodes ≤ 0.917 samples = 1546 value = [772, 774] class = good
parallelism ≤ 0.66 samples = 831 value = [488, 343] class = bad
samples = 715 value = [284, 431] class = good
samples = 400 value = [275, 125] class = bad
samples = 431 value = [213, 218] class = good

Figure 9.11: A decision tree that is learned from all programs' training sets. We identified three primary classes of profitable schedules and have marked the paths from the root node that correspond to these classes. The splitting conditions in the subtree marked are diffuse. Developing this subtree more deeply may be helpful. We pruned subtrees whose leaves share the same majority class.

dashed line corresponds to mostly diffuse splitting conditions. Developing this subtree more deeply may be helpful to distinguish better between profitable and unprofitable schedules in the training set that correspond to this subtree. It is remarkable that most of the subtree's leaves do not relate unambiguously to either profitable or unprofitable schedules.

E 9.4.3: *k-Fold Cross-Validation on the Full Training Set* (**RQ 9.4.1**, **RQ 9.4.2**) In addition to **E 9.4.2**, we carried out k-fold cross-validation on the union of all programs' training sets. Instead of leaving all schedules of p programs out of the training set, as we did in **E 9.4.2**, we removed a share of $1/k$ ($k = 5, 6, ..., 20$) randomly chosen schedules from the full training set. Then, we learned a performance model from the remaining schedules using the configuration that we had elected in **E 9.4.2**. Using the model, we classified the removed schedules. Per k, we repeated the experiment until the determined shares of false negatives among the profitable and false positives among the unprofitable schedules became significant. Table 9.13 shows the results.

Across the benchmark set, the average rate of false negatives among the actually profitable schedules is between 15% and 16%. The average rate of false positives among the actually unprofitable schedules is approximately 6%.

E 9.4.4: *Cost-Benefit-Analysis of Replacing Benchmarking with Classification* (**RQ 9.4.3**) In this experiment, we evaluated the cost and the benefit of replacing our genetic algorithm's fitness function that is based on benchmarking with a classifier that relies on a machine-learned surrogate performance model. Section 7.2 describes two ways of combining a classifier with our genetic algorithm. The first way uses the classifier as a filter for the actual fitness evaluation that relies on benchmarking. The second way relies completely on classification for the fitness evaluation of schedules and uses benchmarking only to select the best schedule from the genetic algorithm's final population. As already described, we focus on the second way of combining the classification of schedules with our genetic algorithm here. We refer to this combination by GA_C.

For this experiment, we used a NUMA (non-uniform memory access) [82] computer architecture with two CPU sockets. An INTEL XEON E5-2650 v2 CPU @ 2.6GHz CPU with eight physical cores and 20MB of L3 cache is mounted on each of the sockets. We use `numactl` [82] to pin POLYITE to one CPU socket and the benchmarking of schedules (i.e., the script that invokes CLANG with POLLY and performs the benchmarking; refer to Section 8.1) to the other CPU socket. We use a single thread to generate schedules. The classification of schedules classifies 20 schedules in parallel. We wrap the benchmarking script by the `srun` command of SLURM, which initiates a SLURM job step on the already allocated local compute node (refer to Section 8.3.6) and allows us to deactivate INTEL TURBO BOOST during the benchmarking process. We measure the execution time of GA_C, the total time needed for the evaluation of the schedules in GA_C's final population, and, per schedule in the final population, the execution time of the program to be optimized after the application of the schedule. From the latter, we can calculate the speedup over the sequential execution of the original program after its optimization by `-O3`. To account for the randomness of GA_C, we run each configuration ten times and use the median of the maximum speedups yielded by the ten runs as GA_C's result for a program.

The experiment has two steps. In the first step, we guided GA_C with a model learned from schedules of the program to be optimized. This allowed us to determine how well GA_C performs when it is guided by a performance model that resembles well the program to be compiled. Here, we labeled any schedule as profitable that yields at least 50% of the speedup yielded by the best schedule in the program's training set. This deviates from the labeling rule described in Section 7.2.

In the second step, we used the leave-one-out schema to evaluate how well transfer-learned performance models direct GA_C. We learned the models from all programs' training sets except the program to be optimized. We labeled the training data as described in Section 7.2.

Table 9.13: Results of **E 9.4.3** (*k-Fold Cross-Validation on the Full Training Set*). Per program and value of k, we show the share of false negatives among the profitable schedules and the share of false positives among the unprofitable schedules.

	total **k**	2mm	3mm	adi	atax	bicg	cholesky	correlation	covariance	deriche	doitgen	durbin	fdtd-2d	gemm	gemver	gesummv	gramschmidt	heat-3d	jacobi-1d	jacobi-2d	lu	ludcmp	mvt	symm	syr2k	syrk	trisolv	trmm	
% false neg.	5	15.62	11.58	6.16	23.37	10.30	7.84	15.23	31.77	20.60	–	17.16	–	14.55	5.83	10.69	13.97	12.96	42.20	14.42	33.32	30.80	–	18.70	–	6.90	11.26	16.78	16.06
% false pos.	5	6.21	4.75	7.48	9.22	9.78	5.89	1.92	10.28	9.49	0.19	4.40	0.00	7.06	7.50	9.14	11.75	3.04	10.72	3.27	12.53	8.62	0.00	10.50	0.00	6.45	6.28	11.55	8.85
% false neg.	6	15.53	11.68	6.10	23.00	10.63	7.97	15.00	31.61	20.59	–	16.94	–	14.26	5.76	10.62	13.65	12.64	41.43	14.33	33.64	30.22	–	18.85	–	6.77	11.51	16.75	16.11
% false pos.	6	6.09	4.45	6.95	9.24	9.35	5.60	1.91	10.51	9.36	0.18	4.10	0.00	6.88	7.78	9.34	11.60	3.01	11.39	2.94	12.27	8.57	0.00	9.39	0.00	6.28	6.10	11.36	8.33
% false neg.	7	15.57	11.53	5.94	23.08	10.95	7.79	14.77	31.63	21.00	–	17.00	–	14.37	5.71	10.71	13.84	12.40	41.96	14.53	34.47	30.36	–	18.62	–	6.56	11.20	17.10	15.95
% false pos.	7	6.09	4.41	7.00	9.03	9.45	5.86	1.94	10.38	9.29	0.22	4.00	0.00	6.48	7.44	9.20	11.39	3.20	11.36	2.96	12.14	8.65	0.00	10.52	0.00	6.43	5.88	11.61	8.02
% false neg.	8	15.50	11.56	6.15	23.12	10.75	7.88	15.26	32.08	20.89	–	16.71	–	13.66	5.75	10.81	13.84	12.27	41.83	14.49	32.61	30.17	–	18.72	–	6.55	11.04	16.97	16.02
% false pos.	8	6.11	4.43	7.03	9.01	9.35	5.68	2.00	10.41	9.45	0.21	4.13	0.00	6.45	7.29	9.26	11.58	3.10	11.55	2.82	12.32	8.54	0.00	9.83	0.00	6.38	6.18	11.79	8.28
% false neg.	9	15.43	11.57	6.09	22.93	10.37	7.55	15.04	31.92	20.95	–	16.08	–	14.07	5.69	10.56	13.47	12.04	40.36	14.44	32.98	30.67	–	18.78	–	6.82	11.57	17.11	15.93
% false pos.	9	6.10	4.48	6.99	8.75	9.34	5.68	1.93	10.54	9.23	0.24	4.09	0.00	6.44	7.47	9.25	11.47	3.35	11.41	2.83	12.47	8.71	0.00	10.70	0.00	6.27	5.60	11.56	8.00
% false neg.	10	15.36	11.83	6.04	22.28	10.79	7.68	15.36	31.57	20.66	–	16.40	–	13.83	5.81	10.23	13.50	11.86	39.98	14.40	33.59	29.56	–	18.87	–	6.68	11.12	16.90	16.16
% false pos.	10	6.06	4.40	7.06	9.12	9.29	5.54	1.95	10.12	9.25	0.19	4.11	0.00	6.41	7.57	9.20	11.13	3.35	11.68	2.88	12.19	8.28	0.00	10.06	0.00	6.71	5.82	11.52	7.94
% false neg.	11	15.33	11.47	6.24	22.26	10.96	7.63	15.63	32.04	20.71	–	16.47	–	14.24	5.82	10.08	13.26	11.73	40.12	14.46	32.74	30.27	–	18.50	–	6.54	10.96	16.45	16.40
% false pos.	11	6.13	4.97	7.05	9.31	8.90	5.58	2.02	10.28	9.36	0.21	4.06	0.00	6.52	7.62	9.36	11.45	3.36	11.97	2.89	12.51	8.36	0.00	10.55	0.00	6.35	5.87	11.54	8.17
% false neg.	12	15.34	11.53	6.11	23.51	10.66	7.62	15.09	32.28	20.41	–	16.33	–	13.94	5.83	10.23	13.28	11.71	40.10	14.36	33.35	30.30	–	18.89	–	6.63	10.93	16.67	15.97
% false pos.	12	6.07	4.77	6.85	9.03	9.34	5.64	1.94	10.46	9.33	0.22	4.05	0.00	6.37	7.43	9.06	11.34	3.35	12.04	2.68	12.14	8.25	0.00	10.46	0.00	6.45	5.44	11.47	7.91
% false neg.	13	15.39	11.57	6.20	23.48	11.20	7.45	14.83	31.84	20.50	–	16.33	–	13.86	5.99	10.01	13.47	11.68	40.24	14.55	32.31	30.21	–	18.97	–	6.43	11.18	17.20	16.14
% false pos.	13	6.04	4.68	6.96	9.01	8.97	5.67	1.91	10.22	9.34	0.19	4.17	0.00	6.41	7.39	8.93	11.29	3.57	11.96	2.63	12.17	8.34	0.00	10.08	0.00	6.33	5.73	11.66	7.89
% false neg.	14	15.34	11.40	6.21	22.21	10.71	7.43	16.16	32.58	20.77	–	16.22	–	13.94	6.10	10.07	12.93	11.68	39.20	14.38	33.09	30.15	–	18.95	–	6.56	11.15	16.75	16.33
% false pos.	14	6.06	4.82	6.77	8.95	9.08	5.50	1.89	10.14	9.33	0.20	4.11	0.00	6.55	7.24	9.06	11.40	3.19	12.28	3.14	12.30	8.50	0.00	10.57	0.00	6.27	5.48	11.68	7.88
% false neg.	15	15.30	11.39	6.23	22.26	11.13	7.60	15.56	31.95	20.51	–	16.28	–	13.70	5.82	10.20	13.12	12.06	40.01	14.39	32.69	30.03	–	19.35	–	6.58	11.16	16.62	16.33
% false pos.	15	6.07	4.55	7.02	9.09	9.12	5.51	1.98	10.55	9.15	0.24	4.13	0.00	6.26	7.44	9.04	10.94	3.43	12.13	2.93	12.09	8.40	0.00	10.47	0.00	6.43	5.75	11.72	8.03
% false neg.	16	15.24	11.39	6.39	22.61	11.02	7.40	14.56	31.72	20.53	–	16.32	–	13.84	5.89	10.13	12.87	11.46	39.08	14.10	32.83	30.06	–	19.13	–	6.37	11.08	16.80	16.08
% false pos.	16	6.05	4.74	6.84	8.83	8.98	5.51	1.97	10.26	9.07	0.22	3.99	0.00	6.29	7.58	9.24	11.10	3.38	12.10	2.78	12.44	8.53	0.00	10.63	0.00	6.28	5.80	11.81	7.90
% false neg.	17	15.25	11.42	6.20	23.01	10.93	7.67	15.11	32.41	20.63	–	16.15	–	14.07	5.95	10.10	13.19	11.72	38.53	14.30	32.62	29.35	–	18.57	–	6.48	10.95	16.82	16.22
% false pos.	17	6.09	4.69	6.84	8.97	9.23	5.45	1.99	10.12	9.39	0.20	4.16	0.00	6.18	7.67	9.23	11.25	3.45	12.31	3.01	12.48	8.24	0.00	10.68	0.00	6.27	5.81	11.55	7.65
% false neg.	18	15.24	11.39	6.25	22.57	11.25	7.58	14.80	31.96	20.39	–	16.19	–	13.76	5.62	10.28	13.03	11.36	39.16	14.44	32.61	29.67	–	18.84	–	6.37	10.99	16.75	16.28
% false pos.	18	6.07	4.59	7.22	8.72	8.74	5.67	1.95	10.36	8.88	0.20	4.09	0.00	6.21	7.64	9.27	11.44	3.38	12.06	2.89	12.57	8.44	0.00	10.26	0.00	6.55	5.95	11.74	7.74
% false neg.	19	15.19	11.22	6.19	22.20	10.80	7.57	15.06	32.32	20.49	–	16.18	–	13.94	5.85	10.05	12.88	11.47	38.45	14.27	32.04	29.85	–	18.97	–	6.51	10.93	16.73	16.20
% false pos.	19	6.08	4.70	6.93	8.99	8.98	5.54	1.94	10.29	9.22	0.18	3.97	0.00	5.99	7.63	9.37	11.27	3.43	12.38	2.94	12.60	8.47	0.00	10.44	0.00	6.66	5.71	11.54	7.80
% false neg.	20	15.20	11.35	6.31	22.86	10.83	7.57	15.87	31.51	20.40	–	16.14	–	13.39	5.80	10.15	12.93	11.66	39.19	14.36	32.39	29.75	–	18.85	–	6.41	11.11	16.33	16.01
% false pos.	20	6.08	4.59	6.86	9.13	8.78	5.51	1.97	10.54	9.10	0.20	4.19	0.00	6.00	7.51	9.15	11.15	3.39	12.28	2.81	12.21	8.34	0.00	10.61	0.00	6.68	5.71	11.72	7.83

As the baseline, we ran GA_B with the window-based early termination criterion described in Section 6.1, using the configuration specified in Section 9.1.4 five times per program. Running GA_B for a fixed number of generations that is chosen independently from the program to be optimized would have been unfair in terms of the total duration of the optimization because GA_C's termination criterion also allows for an early termination. As the optimal speedup reached by GA_B, we used the median of the maximum speedups reached by the five runs.

The second baseline are, per program, ten sets of schedules generated by random exploration with the sparse configuration. Each set contained 50 schedules, which equals the size of GA_B's final population.

We expect that the benchmarking effort to determine the most profitable schedule in GA_C's final population is lower than the effort that is required to determine the most profitable schedules among the schedules generated randomly. Further, we expect that the optimal schedule found by GA_C yields an equal or higher speedup in execution time of the transformed program than the best schedule found by random exploration. In addition, we compare the distributions of the speedups yielded by the successfully applicable schedules found by GA_C and those found by random exploration. If the distribution of the speedups do not differ, this is an indicator that GA_C is nothing but random exploration. As a simple test, we check whether the schedules in GA_C's final population yield significantly higher average speedups than the schedules generated randomly.

In the comparison to GA_B, we expect to be able to reduce the overall duration of the optimization by using GA_C. Furthermore, we evaluate the effect on the speedup yielded by the optimal schedule found that results from replacing benchmarking by classification.

Table 9.14 shows the results from an analysis of the speedups yielded by GA_C, GA_B, and random exploration. We also show the speedups yielded by the configuration of GA_B that we used in **E 9.3.2**. This configuration fixed GA_B's number of generations to 40. Here, we refer to this configuration by GA_B^{40}. Thereby, we can evaluate the quality of our criterion for the early termination of GA_B.

Across the benchmark set, the speedup yielded by GA_B is reduced by a mean factor of 0.95 compared to GA_B^{40}. On the other hand, instead of computing 40 generations of the genetic algorithm, which corresponds to testing 630 schedules, GA_B computes less than 10 generations on average, which corresponds to less than 180 schedules. The speedups yielded by GA_C with the well fitting classifier are reduced by a mean factor of 0.90 compared to GA_B. In case of GA_C in the leave-one-out schema, the mean speedup reduces by a factor of 0.82 compared to GA_B. If we compare GA_C to GA_B^{40} directly, the optimal speedup reduces by a factor of 0.86 in the case of the well fitting classifier and by the factor 0.78 in the case of the leave-one-out schema. Compared to the ISL scheduler, GA_C with the well fitting classifier yields speedups that are higher by a mean factor of 1.53 compared to the ISL scheduler. The speedups yielded by GA_C in the leave-one-out schema are 1.39 times higher at mean than the speedups yielded by the ISL scheduler. By looking at random exploration, one notices that random often performs surprisingly well. GA_C with the well fitting classifier yields speedups that are higher by a factor of 1.04 than the maximum speedups yielded by random. In the leave-one-out schema, GA_C yields a mean maximum speedup that is reduced by 0.94 compared to random. The average speedup yielded by all successfully applicable schedules in GA_C's final populations is higher for most benchmarks than the average speedup yielded by all schedules generated randomly. This indicates that the benchmarking effort that is required to evaluate all schedules in GA_C's final population may be reduced compared to benchmarking the schedules generated randomly. Yet, the actual benchmarking effort also depends on the number of schedules that result in miscompilation or incorrect binaries or that cannot be benchmarked within the timeout for schedule evaluation. Figure 9.12 compares the distribution of the speedups yielded by the schedules in GA_C's final population and the speedups yielded by schedules generated randomly. In most cases, the median

Table 9.14: Speedups yielded by the configurations that we evaluated in **E 9.4.4**. Columns 2 to 6 show the baseline. The seventh and the eighth column show the results of GA$_C$. In these columns we show two values per cell: the value to the left corresponds to the first step of the experiment in which we used a classifier that resembles the program to be optimized well. The second value corresponds to the leave-one-out case. The last four columns compare ISL, GA$_C$, GA$_B$, GA$_B^{40}$, and random.

program	**isl speedup**	**GA$_B^{40}$ speedup**	**GA$_B$ median max. speedup**	**rand. mean speedup**	**rand median max. speedup**	**GA$_C$ median max. speedup**	**GA$_C$ mean speedup**	**GA$_C$ median max. speedup / GA$_B$ median max. speedup**	**GA$_C$ median max. speedup / random median max. speedup**	**GA$_C$ median max. speedup / isl speedup**	**GA$_B$ median max. speedup / GA$_B^{40}$ speedup**
`2mm`	13.81	18.88	19.88	4.46	15.47	18.17 | 18.21	10.93 | 10.80	0.91 | 0.92	1.17 | 1.18	1.32 | 1.32	1.05
`3mm`	13.39	20.20	18.83	2.55	11.37	17.94 | 13.72	10.56 | 6.13	0.95 | 0.73	1.58 | 1.21	1.34 | 1.02	0.93
`adi`	7.61	8.57	8.41	2.55	7.90	7.90 | 7.62	5.31 | 2.69	0.94 | 0.91	1.00 | 0.96	1.04 | 1.00	0.98
`atax`	0.99	3.09	3.02	0.78	2.78	2.32 | 2.70	1.75 | 1.32	0.77 | 0.89	0.83 | 0.97	2.35 | 2.74	0.98
`bicg`	1.05	2.79	2.78	1.19	2.77	2.78 | 2.76	2.38 | 1.43	1.00 | 0.99	1.00 | 1.00	2.65 | 2.63	1.00
`cholesky`	2.68	5.81	6.40	1.31	3.32	2.82 | 4.08	1.16 | 2.41	0.44 | 0.64	0.85 | 1.23	1.05 | 1.53	1.10
`correlation`	33.36	43.81	36.69	7.19	32.27	31.39 | 33.99	11.62 | 16.29	0.86 | 0.93	0.97 | 1.05	0.94 | 1.02	0.84
`covariance`	33.44	42.24	36.40	11.35	35.17	36.23 | 36.22	24.63 | 23.54	1.00 | 1.00	1.03 | 1.03	1.08 | 1.08	0.86
`deriche`	1.07	1.12	1.07	0.67	1.01	1.02 | 0.95	0.68 | 0.60	0.95 | 0.89	1.01 | 0.94	0.95 | 0.88	0.96
`doitgen`	4.06	5.63	4.86	1.87	4.10	4.37 | 4.23	2.93 | 3.02	0.90 | 0.87	1.07 | 1.03	1.08 | 1.04	0.86
`durbin`	0.99	1.18	1.00	0.66	1.00	1.00 | 1.00	0.70 | 0.63	1.00 | 1.00	1.00 | 1.00	1.01 | 1.01	0.85
`fdtd-2d`	0.92	3.80	3.32	0.74	2.67	3.00 | 2.55	1.46 | 1.30	0.90 | 0.77	1.12 | 0.96	3.26 | 2.77	0.88
`gemm`	4.23	4.72	4.93	2.09	4.39	4.40 | 4.39	2.96 | 3.22	0.89 | 0.89	1.00 | 1.00	1.04 | 1.04	1.04
`gemver`	3.78	6.27	6.07	0.83	2.75	4.73 | 2.90	2.72 | 1.24	0.78 | 0.48	1.72 | 1.05	1.25 | 0.77	0.97
`gesummv`	3.13	8.99	9.04	2.96	8.93	8.92 | 8.91	6.55 | 5.63	0.99 | 0.99	1.00 | 1.00	2.85 | 2.85	1.01
`gramschmidt`	8.10	11.17	8.56	1.94	7.99	8.21 | 4.64	4.49 | 1.75	0.96 | 0.54	1.03 | 0.58	1.01 | 0.57	0.77
`heat-3d`	2.68	4.19	3.65	0.82	2.81	3.50 | 2.43	1.82 | 0.88	0.96 | 0.67	1.25 | 0.87	1.31 | 0.91	0.87
`jacobi-1d`	0.88	1.37	1.37	0.64	1.36	1.37 | 1.00	1.01 | 0.52	1.00 | 0.73	1.01 | 0.74	1.56 | 1.14	1.00
`jacobi-2d`	1.13	4.21	4.19	0.97	4.11	4.19 | 4.17	2.29 | 3.02	1.00 | 0.99	1.02 | 1.01	3.69 | 3.68	1.00
`lu`	5.38	9.86	9.44	2.24	6.99	7.67 | 3.54	4.91 | 1.73	0.81 | 0.38	1.10 | 0.51	1.43 | 0.66	0.96
`ludcmp`	0.97	1.03	1.02	0.97	1.01	1.01 | 1.01	0.97 | 0.97	0.99 | 0.98	1.00 | 0.99	1.05 | 1.04	0.99
`mvt`	4.65	8.25	8.10	2.51	6.44	6.11 | 5.58	4.13 | 2.08	0.75 | 0.69	0.95 | 0.87	1.31 | 1.20	0.98
`symm`	1.01	1.02	1.02	1.00	1.01	1.02 | 1.02	1.00 | 1.00	1.00 | 1.00	1.00 | 1.00	1.00 | 1.00	1.00
`syr2k`	21.57	25.97	25.99	11.89	25.80	25.92 | 25.98	20.68 | 20.33	1.00 | 1.00	1.00 | 1.01	1.20 | 1.20	1.00
`syrk`	13.76	17.27	16.14	6.97	16.47	15.98 | 15.97	11.59 | 12.08	0.99 | 0.99	0.97 | 0.97	1.16 | 1.16	0.93
`trisolv`	0.36	1.06	1.00	0.48	1.33	1.00 | 1.00	0.86 | 0.90	1.00 | 0.99	0.75 | 0.75	2.74 | 2.74	0.95
`trmm`	2.02	22.43	22.41	6.42	21.38	21.44 | 21.84	12.04 | 9.52	0.96 | 0.97	1.00 | 1.02	10.59 | 10.79	1.00
geometric mean	3.30	5.87	5.58	1.83	4.86	5.04 | 4.58	3.28 | 2.61	0.90 | 0.82	1.04 | 0.94	1.53 | 1.39	0.95

Table 9.15: p-values that resulted from a pairwise Wilcoxon signed rank test that quantified the significance of differences in the distribution of the values in columns 2 to 8 of Table 9.14. From the results, the significance of differences in the outcome of the schedule optimization techniques that were evaluated in **E 9.4.4** can be derived.

	GA$_B^{40}$ speedup	**GA$_B$ median max. speedup**	**rand. mean speedup**	**rand. median max. speedup**	**GA$_C$ median max. speedup (well fitt. class.)**	**GA$_C$ median max. speedup (leave-p-out)**	**GA$_C$ mean speedup (well fitt. class.)**	**GA$_C$ mean speedup (leave-p-out)**
isl speedup	3.00E-08	5.60E-08	2.60E-05	1.78E-03	1.00E-05	3.68E-03	6.55E-02	1.51E-02
GA$_B^{40}$ speedup	–	6.03E-03	3.00E-08	1.00E-06	3.00E-08	8.00E-08	3.00E-08	3.00E-08
GA$_B$ median max. speedup	–	–	3.00E-08	1.40E-05	3.00E-08	3.00E-08	3.00E-08	3.00E-08
rand. mean speedup	–	–	–	3.00E-08	3.00E-08	3.00E-08	3.60E-07	5.70E-04
rand. median max. speedup	–	–	–	–	9.81E-02	8.04E-01	3.00E-08	3.00E-08
GA$_C$ median max. speedup (well fitt. class.)	–	–	–	–	–	4.47E-02	3.00E-08	3.00E-08
GA$_C$ median max. speedup (leave-p-out)	–	–	–	–	–	–	3.20E-06	3.00E-08
GA$_C$ mean speedup (well fitt. class.)	–	–	–	–	–	–	–	1.96E-02

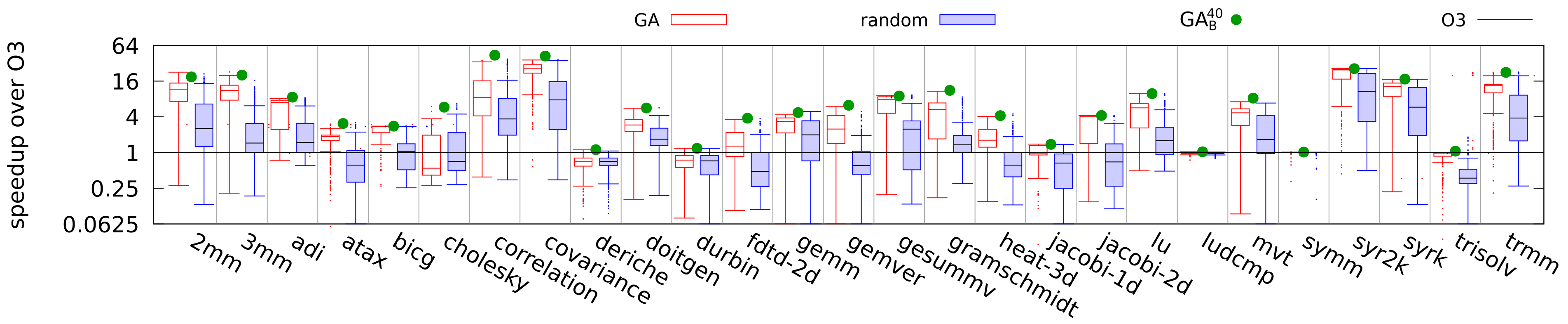
speedup yielded by the schedules in GA$_C$'s final population is higher than the median speedup yielded by the schedules generated randomly.

We tested whether the observations listed above correspond to significant differences in the respective results' distribution across the benchmark set. Table 9.15 shows the results of a pairwise Wilcoxon signed rank test [142] in combination with false discovery rate control [22] for columns 2 to 8 of Table 9.14.

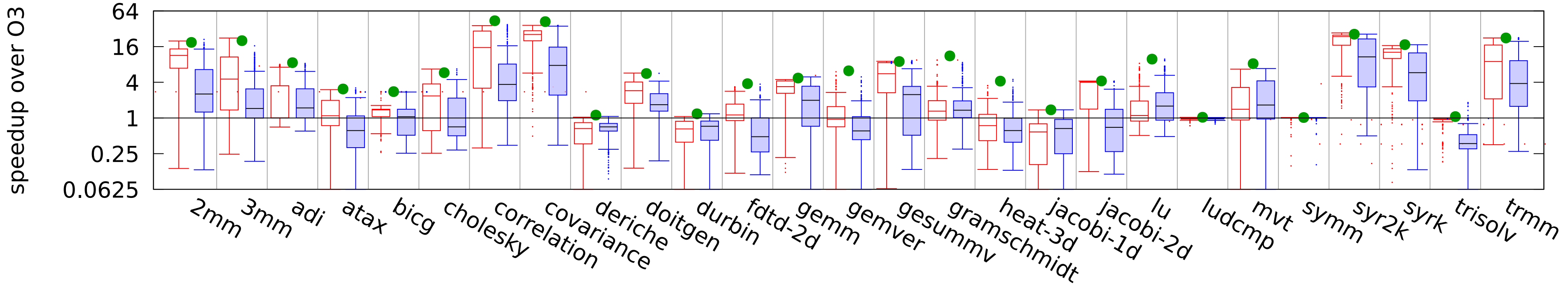
The optimal speedups yielded by GA$_B^{40}$ are significantly higher than the maximum speedups yielded by GA$_C$. GA$_B$ yields significantly higher speedups than GA$_C$ for both the well fitting classifiers and the classifiers learned in the leave-one-out schema. GA$_C$ yields significantly higher speedups than ISL in the case of the well fitting classifiers and in the leave-one-out schema. The test is inconclusive regarding the difference between the optimal speedups yielded by random exploration and GA$_C$. The average speedup yielded by the schedules in GA$_C$'s final population is significantly higher than the average speedup yielded by the schedules generated randomly for both the well fitting classifiers and the classifiers that were trained in the leave-one-out schema. Table 9.15 makes further comparisons.

We continue by comparing the total duration of optimizing a program with GA$_B$ and the total duration of optimizing it with GA$_C$ in the leave-p-out schema. Table 9.16 shows that across the benchmark set GA$_C$ is faster by the mean factor 2.26. Particularly in the case of small SCoPs with a short execution time, the lead of GA$_C$ is likely to reduce if we disable INTEL TURBO BOOST for the entire optimization process and, consequently, avoid the interaction with SLURM per schedule. As stated in Section 9.1.2, switching to a release build of LLVM would decrease the effort that is required to benchmark transformed program versions.

Table 9.16 compares the configurations tested with respect to the duration of the optimization and the benchmarking effort on the target hardware that is required. Furthermore, the table shows the median number of generations produced by GA$_B$. For all programs except `gramschmidt`, it was possible to reach at least 84% at median of the speedup in execution time of the optimized program over the execution time of the original sequential code after its optimization by `-O3` within less than half the number of generations produced



(a) well fitting classifier



(b) leave-one-out

Figure 9.12: Comparison of the speedups yielded by the schedules in the final generations of GA_C and the speedups yielded by schedules generated randomly. Per program, the box plot on the left corresponds to GA_C and the box plot on the right corresponds to random. The whiskers extend from the ends of the box to the most distant point whose value lies within 1.5 times the interquartile range. Points that lie beyond that distance are drawn as outliers [165]. The horizontal line corresponds to the sequential execution of the original code after its optimization with `-O3`. The dot in between the box plots marks the speedup yielded by GA_B^{40}.

by GA_B^{40}. The median speedup reached by the five replicated runs of GA_B for `gramschmidt` was 77% of the speedup reached by GA_B^{40}.

Finally, we compare the time needed by GA_C in the leave-p-out schema to benchmark the schedules in its final population and the time needed to benchmark the schedules generated randomly. Table 9.16 shows that the benchmarking of the schedules in GA_C's final population is faster than the benchmarking of the same number of schedules generated randomly by the factor 1.56 at mean. We cannot include the result of `gesummv` in the comparison for technical reasons. Using the Wilcoxon signed rank test, we found that, in the configuration tested, the median duration of optimization with GA_C is significantly faster across the benchmark set than the median duration of the optimization with GA_B. Also, the median duration of benchmarking the schedules in GA_C's final population is significantly lower than the duration of benchmarking an equally sized set of schedules generated randomly.

Of course, we make the presence of suitable training data for classifiers a precondition. Without training data, one cannot profit from the faster optimization of GA_C compared to GA_B or the reduced benchmarking effort compared to random exploration.

9.4.4 Discussion

In **RQ 9.4.1**, we asked whether all of the features in Section 7.1 are useful for prediction. In **E 9.4.1**, we found that the sparsity of structure parameters is largely insensitive to speedup and purged it.

In **E 9.4.2**, we found that classifiers based on our transfer-learned performance models fail to recognize many profitable schedules. In contrast, the models yielded few false positives. An analysis of the features' importance and value distribution per program largely explains why transfer to some programs is complicated. There are several programs, such as `gemm` and `syrk` in PolyBench 4.1, that are transformed best by fully tiling each loop nest and parallelizing its outermost loop. These programs' simplicity may bias our performance models and decrease their applicability to more complex programs. Our features' approximative nature may add to the classifiers' inability to recognize some profitable schedules: in the presence of partially fused loops, our parallelism feature may be unable to recognize some parallelism. Yet, a precise identification of parallelism is generally impossible due to parametric loop bounds and unknown parameter values. While we could identify partial loop fusion and model it using sequence nodes in schedule trees, we would still be unable to decide for every loop whether it will actually be executed at run time. Also, as described in Section 9.2, the computation of F_{DataLoc} suffers from spurious structure parameters that the version of Polly that we used in our experiments introduces in many SCoPs' models. This is no longer an issue with newer versions of Polly. An improved schedule tree transformation as suggested in Section 5.6 would increase the features' accuracy and could possibly increase performance by enabling tiling in more situations. Generally, the extraction of the feature vectors before tiling adds to the inaccuracy. The findings in **E 9.4.2** indicate that a normalization of the feature values may also be helpful to increase the classifiers' precision. To interpret the result of the k-fold cross-validation on the entire training data in **E 9.4.3**, one must be aware that the programs' training sets are not free of redundancy. Multiple schedules per program that correspond to the same pair of simplified schedule trees and, thus, to the same feature vector can exist. Yet, the experiment shows that the programs' differences do not lead to the effect that they trigger mispredictions mutually.

In summary, the answer to **RQ 9.4.2** is that, to a good extent, models learned from the results of previous iterative optimization can be used to recognize profitable schedules of unseen programs. The success depends on the similarity of the programs involved. We do not recognize many very profitable schedules of many programs as such. The inclusion of program features in our feature vector would certainly improve our performance models'

Table 9.16: Comparison of GA$_B$ and GA$_C$ in the leave-one-out schema with respect to the total duration of the optimization and of GA$_C$ and random exploration with respect to the benchmarking effort required.

program	**GA$_B$ median # gen.**	**GA$_B$ median # schedules evaluated**	**GA$_B$ median duration (min.)**	**GA$_C$ median duration (min.)**	**GA$_C$ final gen. benchm. median duration (min.)**	**rand. expl. benchm. median duration (min.)**	**GA$_C$ speedup in optimization time over GA$_B$**	**GA$_C$ speedup in benchm. duration over rand. expl.**
`2mm`	15	255	340.37	67.66	49.23	150.62	5.03	3.06
`3mm`	18	300	783.15	160.96	135.85	250.85	4.87	1.85
`adi`	9	165	1139.65	752.24	664.79	582.28	1.52	0.88
`atax`	13	225	22.77	14.43	4.94	6.35	1.58	1.28
`bicg`	7	135	18.18	9.85	2.54	6.17	1.85	2.43
`cholesky`	9	165	3139.32	1002.25	985.15	1004.68	3.13	1.02
`correlation`	10	180	345.82	200.36	71.08	119.98	1.73	1.69
`covariance`	7	135	160.18	61.23	41.91	88.27	2.62	2.11
`deriche`	7	135	282.65	162.23	94.71	102.85	1.74	1.09
`doitgen`	16	270	202.45	63.63	38.96	60.75	3.18	1.56
`durbin`	7	135	24.55	58.04	8.07	8.07	0.42	1.00
`fdtd-2d`	9	165	620.13	223.38	196.23	280.65	2.78	1.43
`gemm`	7	135	111.83	39.04	34.92	66.63	2.86	1.91
`gemver`	15	255	34.37	22.02	8.30	9.33	1.56	1.12
`gesummv`	7	135	15.70	8.40	4.22	–	1.87	–
`gramschmidt`	11	195	489.08	243.29	190.74	192.32	2.01	1.01
`heat-3d`	13	225	1211.03	539.21	381.58	431.82	2.25	1.13
`jacobi-1d`	7	135	21.35	17.44	3.65	7.68	1.22	2.11
`jacobi-2d`	7	135	600.45	141.59	113.29	433.17	4.24	3.82
`lu`	14	240	5772.47	1193.67	1176.91	1294.05	4.84	1.10
`ludcmp`	7	135	3190.17	1104.58	709.94	927.55	2.89	1.31
`mvt`	8	150	19.23	15.73	10.45	8.28	1.22	0.79
`symm`	7	135	303.42	154.85	118.86	156.67	1.96	1.32
`syr2k`	7	135	197.87	50.63	48.92	126.73	3.91	2.59
`syrk`	7	135	105.07	30.03	27.45	63.83	3.50	2.33
`trisolv`	7	135	15.35	12.54	2.32	6.57	1.22	2.83
`trmm`	10	180	146.17	45.58	37.48	58.43	3.21	1.56
average:	9.67					**geometric mean:**	2.26	1.56

accuracy, but would also require a larger and comprehensive training set of programs. If learned from a carefully chosen training set, such a model might be widely applicable. Here, we have studied the extent to which one can abstain from program features and a fully precise feature extraction and learn from data that is available.

The answer to **RQ 9.4.3** is that a classifier that is based on a transfer-learned surrogate performance model can help to accelerate the iterative optimization of new programs, at least in our experimental setup. Optimizing the experimental setup may reduce the lead of GA_C particularly in the case of very short-running program. Sometimes, the profitable schedules cannot be recognized, though, because the characteristics of profitable schedules for the program to be optimized differ too much from those of profitable schedules for the programs in the training set. Also, the schedules found by GA_C are a bit less profitable than the schedules that can be found with GA_B. While GA_B never loses its best schedule found, GA_C is not guaranteed to keep it. Yet, the average performance of the schedules in GA_C's final population is significantly higher than the average performance of schedules generated by random sparse. Regarding the speedup yielded by the optimal schedule found, random exploration often performs surprisingly well, but it tests more ineffective schedules on the target hardware than GA_C. Under the precondition that suitable training data is available the reduced benchmarking effort that results from the use of GA_C, is a benefit.

For GA_B, we set the population size to 30 and let the genetic algorithm run for 40 generations (630 schedules). We had increased the population size to 50 for GA_C and still let the genetic algorithm run for at most 40 generations (1050 schedules). We intended to increase the search space coverage by raising the population size, since our classifiers often miss actually very profitable schedules. To investigate this choice, we let our genetic algorithm with classification run again, but with a population size of 30 and for at most $\lceil(50/30)\cdot 40\rceil = 67$ generations (1035 schedules). Again, we optimized each program ten times. On average, GA_C with the reduced population size terminates after 54 generations (840 schedules). Compared to the original configuration, the median share of schedules classified as profitable is 2.46% higher, on average. The Wilcoxon signed rank test is inconclusive regarding a difference in the number of schedules classified as profitable. Although the larger number of generations appears to improve the share of profitable schedules in GA_C's final population, the larger population size of 50 schedules increases the coverage of the search space.

Table 9.17 compares GA_C in the leave-one-out schema of **E 9.4.4** and with the well fitting classifiers of **E 9.4.4** to GA_C in the leave-one-out schema with a population size of 30 and at most 67 generations. Like in **E 9.4.4**, we compare results of ten runs per configuration and program. We look at the median number of generations of the genetic algorithm computed and the median number of profitable schedules in the genetic algorithm's final generation. On average, GA_C with the reduced population size terminates after 54 generations (840 schedules). Compared to the configuration of GA_C in **E 9.4.4** with the leave-one-out schema, the average of the median shares of schedules that are classified as profitable is 2.46% higher. Using the Wilcoxon signed rank test we found that there is a significant difference in the distribution of the median shares of profitable schedules between the two configurations. Although the larger number of generations appears to improve the share of profitable schedules in GA_C's final population, the larger population size of 50 schedules increases the coverage of the search space.

Table 9.17: Comparison of three configurations of GA$_C$ with respect to the median share of profitable schedules in the final generations, the median number of generations computed and the percent of GA$_C$ runs that terminated early. A value of 0 for the number of generations indicates that the genetic algorithm terminated immediately after the evaluation of the initial population.

	E 9.4.4, well fitting classifier			**E 9.4.4, leave-one-out schema**			**population size 30, max. 67 generations, leave-one-out schema**		
program	**median % prof. schedules**	**median # generations**	**% runs with early exit**	**median % prof. schedules**	**median # generations**	**% runs with early exit**	**median % prof. schedules**	**median # generations**	**% runs with early exit**
`2mm`	90	40	30	83	40	0	93	58	50
`3mm`	88	40	20	85	40	40	97	29	60
`adi`	84	40	20	80	40	10	82	67	10
`atax`	88	40	0	78	40	0	78	67	20
`bicg`	96	9.5	90	83	40	10	83	67	40
`cholesky`	68	40	0	67	40	10	82	67	20
`correlation`	84	40	20	78	40	10	97	24.5	80
`covariance`	97	9	100	96	19.5	80	97	17.5	100
`deriche`	100	0	100	62	40	0	40	67	0
`doitgen`	82	40	10	71	40	10	72	67	20
`durbin`	98	4	100	0	40	0	0	67	0
`fdtd-2d`	84	40	0	68	40	0	67	67	0
`gemm`	96	5	100	96	17.5	90	97	6.5	100
`gemver`	90	40	40	74	40	0	92	60	50
`gesummv`	85	40	20	96	18	60	97	18	100
`gramschmidt`	90	40	30	0	40	0	0	67	0
`heat-3d`	72	40	0	79	40	0	80	67	20
`jacobi-1d`	96	7	100	77	40	0	80	67	20
`jacobi-2d`	70	40	0	61	40	0	65	67	0
`lu`	78	40	10	67	40	0	63	67	0
`ludcmp`	100	0	100	0	40	0	0	67	0
`mvt`	77	40	0	78	40	0	83	67	20
`symm`	100	0	100	0	40	0	0	67	0
`syr2k`	96	18	80	96	21	60	97	13	100
`syrk`	96	23	70	96	13.5	80	97	16	100
`trisolv`	84	40	20	78	40	0	72	67	0
`trmm`	80	40	0	68	40	0	75	67	0
average	88	28	43	67	36	17	70	54	34

9.5 Threats to Validity

In the following, we discuss aspects that may affect the reproducibility of our empirical results and the generality of the conclusions drawn.

Threats to Internal Validity (Reproducibility of results) The search space of legal schedules for a SCoP has an enormous size. Conceptually, it is infinite, but even the size of the finite subset of reasonable schedules is thousands to millions strong. Our search space exploration is necessarily incomplete and repeated runs may yield different results. To mitigate the effect of randomness, we executed most tested configurations repeatedly and used the aggregated results. An exception are the runs of different configurations of our search space exploration in **E 9.3.2**. Here, we reduced the influence of the search's incompleteness by the high number of generations (40) computed by our genetic algorithm and the high number of schedules (630) tested by the configuration of random exploration evaluated. Moreover, we found a number of apparent regularities in the data. Especially, random exploration and the genetic algorithm often yield very similar performance.

Although all schedules visited are theoretically legal, some result in broken binaries or compilation failures. Any schedule that can be identified as mathematically illegal before the application of tiling will be caught by POLLY and will be reported to POLYITE. POLYITE will then abort. One of the run-time failures is the presence of overly large schedule coefficients. We tested the validity of a sample of the schedules drawn from the results of **E 9.3.1** after the application of tiling to the schedules and found no mathematically illegal schedules. To verify generally the legality of schedules after tiling appears to be computationally too expensive. As already mentioned in the discussion, we noticed that dense schedule matrices are likely to cause failure. They are also likely to increase compilation time tremendously. The problematic schedules can simply be discarded but, the more schedules are discarded, the longer it takes to find the desired number of healthy schedules. An examination of the schedules from **E 9.3.1** revealed that, in contrast to random dense, the GA and random sparse discover only few schedules that fail (refer to Table 9.3).

To control execution time measurement bias, we took several precautions. We benchmarked every transformed program version five times. In **E 9.3.1**, **E 9.3.4**, and **E 9.3.5** we performed three measurements per program version. Five measurements are hardly enough to reach statistical significance but a higher number of measurements conflicts with the extremely large total number of program versions that had to be evaluated in our experiments. Figure 9.13 provides an insight into the stability of our measurements. We analyzed the data that resulted from the search space exploration with random sparse in **E 9.3.2**. Per schedule, we determined the relative standard error of the five execution time measurements' results. The plot shows per program a box plot of the respective 630 schedules' computed standard errors. The measured execution times' relative standard errors per schedule are low, except for the very short-running programs `mvt`, `gesummv`, and `atax` for which we observed increased values. Our benchmarking machines ran no other workloads in parallel. **E 9.4.4** ran on machines with a two-socket NUMA design. We pinned POLYITE to one socket and the benchmarking of program versions to the other.

Threats to External Validity (Generality of conclusions drawn) We use the benchmark set POLYBENCH 4.1. It contains algorithms that occur in application domains for which the polyhedron model is relevant. Yet, the benchmark set's small size limits our conclusions' generality. Furthermore, our approach currently targets primarily programs that profit from coarse-grained parallelism. Its applicability to very short-running loop nests that operate on small data sets is limited. This limitation is enlarged by the fact that POLYITE cannot be used in combination with POLLY's polyhedral vectorizer, yet.

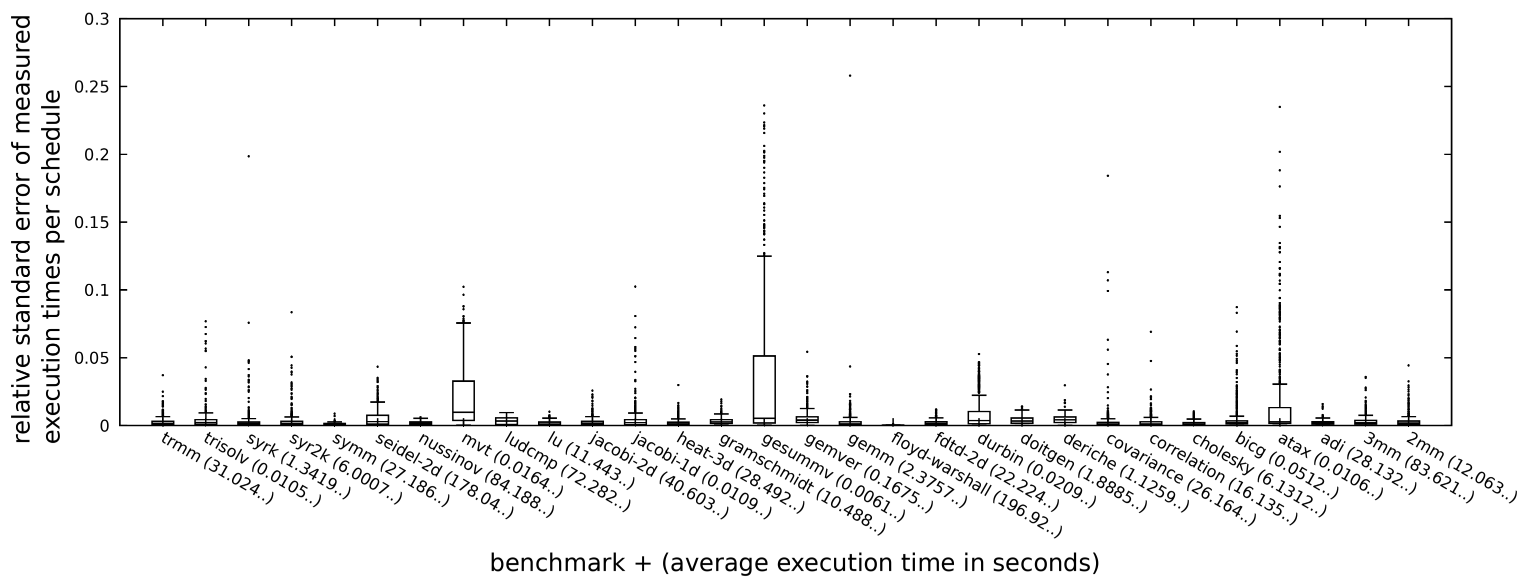


Figure 9.13: Box plots that show the distribution of the standard errors of measured execution times per schedule that resulted during the evaluation of set of schedules generated randomly. In parentheses, the mean execution time in seconds of each program is shown.

10 Conclusion and Future Work

We propose an approach of iterative schedule optimization for OpenMP parallelization and tiling in the polyhedron model. Our approach's theoretical foundation is the iterative polyhedral schedule optimization of the sequential execution time of programs in the absence of tiling by Pouchet et al. [122, 124].

We remove restrictions that Pouchet et al. imposed on the schedule search space that are likely detrimental to parallelization and tiling. Our approach combines the wider search space with sampling strategies for a non-uniform random search space exploration and an adaptation of the genetic algorithm schema by Pouchet et al. that uses a novel set of genetic operators. These operators facilitate the traversal of our wider search space. Our implementation Polyite relies on the compiler infrastructure LLVM [89] and its polyhedral optimizer Polly [65]. In an evaluation on the PolyBench 4.1 benchmark set [121], we were able to reach significantly higher speedups than the PLuTo scheduling algorithm [25] as it is adapted by the Integer Set Library [152, 156] and being used by Polly. This finding applies to programs with long-running loops that operate on larger data sets since we used PolyBench with its extra large data set configuration.

While our genetic algorithm does not yield strongly higher speedups in the execution time of the optimized program than an informed non-uniform random sampling, it requires the evaluation of fewer comparatively unprofitable schedules than a random exploration that visits the same number of schedules as the genetic algorithm.

Determining the fitness of a program transformation by applying it to the program to be optimized and measuring the transformed code's execution time in iterative program optimization induces a strong overhead. This makes iterative optimization impractical to use in many cases. To mitigate the overhead, we propose to train a classifier on results of previous iterative optimizations with Polyite and to replace benchmarking of transformed program versions by classification in Polyite's genetic algorithm to the extent possible. Our empirical evaluation reveals that a reduction of the number of program versions that need to be evaluated by benchmarking is possible without severely reducing the speedup yielded by the optimal program transformation found. A prerequisite for a successful acceleration of the optimization is that training data from an optimization of programs exists that are sufficiently similar to the program to be optimized.

In the following, we provide a more detailed summary. Section 10.5 addresses open questions and research directions.

10.1 Sampling of Schedules

The search space of legal linearly affine schedules for a program is a set of lists of polyhedra. We refer to these lists as search space regions. Basically, each search space region corresponds to one mapping of legality-affecting data dependences to schedule dimensions that carry them if they were not already carried by a previous dimension. In theory, the number of search space regions is infinite, since the first schedule dimension that carries a data dependence may be preceded by an arbitrary number of dimensions that satisfy the dependence weakly. We modified the algorithm for search space construction by Pouchet et al. [124] to sample the set of search space regions.

For the sampling of schedules from search space regions, Danner [46] evaluated four techniques that permit to sample schedules uniformly from a single search space region. We recall these techniques and partly broaden the description and the algorithm's description and assessment. Danner rejected a sampling of search space regions by enumeration, acceptance rejection sampling – we demonstrate its inefficiency by an example – and pattern hit-and-run sampling. Danner suggested to sample search space regions using geometric divide-and-conquer sampling. An empirical evaluation of the algorithm's run-time complexity revealed its impracticality. Danner had limited his empirical study to programs with a small complexity. In addition to the sampling strategies evaluated by Danner, we propose Chernikova sampling, which relies on the geometric representation and Chernikova's algorithm, and projection sampling, which is an adaptation of the sampling strategy by Pouchet et al. [124] for non-uniform random sampling without an underlying statistical model of the search space. We evaluated the run-time complexity of geometric divide-and-conquer sampling, Chernikova sampling, and projection sampling empirically and observed that, while the first is impractical, Chernikova sampling and projection sampling are viable for schedule search space exploration.

We enable the traversal of a wide polyhedral schedule search space that does not require prior knowledge. Configuration parameters exist to bias both the sampling of search space regions and the sampling of schedules from search space regions.

The schedules sampled by the approach of Pouchet al. do not necessarily encode all loops of the transformed code explicitly. To facilitate tiling, we complete each schedule by appending additional dimensions such that all loops are encoded explicitly.

Lastly, we describe an adaptation of the search space construction by Pouchet et al. in Polyite.

Our evaluation reveals that, with our approach to schedule search space exploration, it is possible to outperform the widely used PLuTo scheduling algorithm and random sampling from the search space that results from our adaptation of the search space construction of Pouchet et al. Polyite is not able to perform polyhedral prevectorization (strip-mining) and is inferior to PLuTo in the case of some benchmark programs if the latter is combined with vectorization.

10.2 Schedule Simplification and Analysis

Different schedule representations facilitate different tasks. For the sampling of schedules, we rely on their representation by coefficient matrices. While coefficient matrices are convenient for the purpose of sampling, they are less suitable for an analysis or transformation of schedules. A representation that is more convenient for these purposes is the schedule tree [66]. We propose a conversion from schedule coefficient matrix to schedule tree that is more sophisticated than storing the entire schedule function in one schedule tree node.

Our schedules generated randomly contain schedule coefficients or even entire schedule dimensions that are useless because they do not influence the SCoP's statement instances' execution order. Other information may influence the execution order of statement instances, for instance after tiling by specific tile sizes, yet we consider it to be unwanted noise that complicates schedules' analysis and increases the complexity of the generated code without being apparently beneficial. We propose a sequence of simplification steps for schedule trees that remove such unwanted information. The simplifications retain schedules' legality and preserve or increase the applicability of tiling to loop nests in the transformed program.

With this schedule transformation and simplification, we mitigate the effect of noise that is due to random schedule generation, and facilitate the analysis, characterization, and further optimization of schedules.

10.3 Genetic Algorithm

Besides non-uniform random exploration of the schedule search space, we propose a genetic algorithm for a more guided search space traversal. The genetic algorithm's schema is derived from the genetic algorithm for schedule optimization by Pouchet et al. [124], who observed that a polyhedral schedule search space contains mostly schedules that are no improvement. Well performing schedules are reported to be scarce. Our derived genetic algorithm schema comprises many aspects that reflect this finding. In particular, we refer to the use of elitism, which guarantees the survival of the best performing schedules in a generation to its consecutive generation, a strong diversity in the initial population, and the attenuation of mutations with time. Since legal schedules, which preserve program semantics, are rare compared to illegal schedules, the set of legal schedules for a program must be closed under mutation and crossover.

We propose a set of novel mutation and crossover operators because the operators designed by Pouchet et al. cannot traverse our wider schedule search space. Our empirical evaluation suggests that the genetic algorithm does not yield strongly better schedules than an informed random exploration but that it requires the benchmarking of fewer ineffective schedules.

10.4 Schedule Classification

Iterative compilation can be impractical due to the large time and resource consumption that is mainly due to the evaluation of program versions with benchmarking. We mitigate this drawback partly by replacing benchmarking by a prediction from a random forest classifier to the extent possible. The classifier can label a schedule either as profitable or as unprofitable. It relies on structural and directly performance-related features of schedules. In the performance models, we do not characterize programs using static or dynamic program features. Instead, we suggest to train the classifier on schedules and the respective measured execution times that originate from the previous optimization of other programs or earlier versions of the program to be optimized. We confirmed empirically that, under the precondition that there is sufficient similarity between the training programs and the program to be optimized, the classifier trained will be useful in reducing the benchmarking overhead of our genetic algorithm for schedule optimization without impairing the optimization result seriously. The evaluation was performed in a leave-one-program-out schema on the PolyBench 4.1 benchmark set.

10.5 Open Questions and Research Directions

Tuning of Tile Sizes In our empirical evaluation, we used a fixed tile size of 64. Yet, to gain maximum performance for a given combination of program, linearly affine schedule as it results from Polyite's search space exploration, and target hardware, it is important to tune tile sizes per tilable loop nest und loop dimension. The combined purely iterative optimization of schedule and tile sizes remains an open question. We expect that combining the schedule search space exploration with an optimization of tile sizes by autotuning, such as by the autotuner of Sato et al. [136] or the approach of Shirako et al. [139], may lead to an increase in the search space's dimensionality that makes exploration impractical. Alternatively, tile sizes could be optimized by an analytical model, for instance, the heuristics used by Pouchet et al. [127], who compute tile sizes such that the could data accessed by each tile fits roughly into the L1 cache.

Short-Running Loops Our evaluation on PolyBench 4.1 with the extra large data set setting can be generalized primarily to long-running loops that profit from data locality optimization and coarse-grained parallelism. Moreover, the schedule features described in Section 7.1 target loop nests of this kind. To optimize short-running loop nests that operate on small data sets, Polyite must be developed further to prevectorize loop nests by strip-mining or to be able to interoperate with Polly's polyhedral vectorizer. A schedule feature that quantifies the applicability of vectorization similar to the existing feature for tiling will be necessary.

Prediction of Speedups We train classifiers that predict whether a given schedule is profitable. In our evaluation, this coarse-grained classification sufficed to reduce the benchmarking effort in our iterative schedule optimization and retain an acceptable optimization result. Revisiting the approach by Danner [46] of learning a regression model to predict the speedup yielded by a given schedule would allow to rank schedules by their profitability at a more finely grained level and could further reduce the benchmarking overhead of iterative schedule optimization. Another direction that seems promising is the kind of classifier proposed by Ruvinskiy and van Beek [134]. Their classifier takes a tuple of program transformations as its input and decides whether the first transformation its more profitable than the second. If successfully adapted to Polyite, such a classifier would allow to rank the schedules in the genetic algorithm's population without the requirement to learn a regression model.

Program Features Our classifiers are trained on feature vectors that characterize schedules with respect to the kind of transformation that they encode. We do not include features that characterize the associated program, though. On the one hand, this increases the transferability of our models somewhat, on the other hand, the models are transferable only between programs that require similar transformations. Specifically, as our evaluation indicates, we cannot learn models that are useful for the optimization of a wide range of programs that differ strongly in the kind of transformation from which they benefit. Including program features in our feature vector and training a performance model from a large and comprehensive set of training programs could yield a model that could be widely applicable and could be distributed together with Polyite. Pouchet [119] [Chap. 9] lists likely useful program features.

Small Improvements Throughout the thesis we make several suggestions for small improvements that may increase the quality of program optimization with Polyite.

Bibliography

[1] A. Acharya and U. Bondhugula. PLUTO+: Near-complete modeling of affine transformations for parallelism and locality. In A. Cohen and D. Grove, editors, *Proceedings of the 20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 54–64. ACM, Feb. 2015.

[2] F. V. Agakov, E. V. 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. 4th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 295–305. IEEE, Mar. 2006.

[3] E. Alba and M. Tomassini. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, 2002.

[4] E. Alba and J. M. Troya. Improving flexibility and efficiency by adding parallelism to genetic algorithms. *Statistics and Computing*, 12(2):91–114, 2002.

[5] J. R. Allen and K. Kennedy. Automatic loop interchange. In M. S. V. Deusen and S. L. Graham, editors, *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 233–246. ACM, June 1984.

[6] A. H. Ashouri, A. Bignoli, G. Palermo, and C. Silvano. Predictive modeling methodology for compiler phase-ordering. In C. Silvano, J. M. P. Cardoso, G. Agosta, and M. Hübner, editors, *Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms (PARMA-DITAM)*, pages 7–12. ACM, Jan. 2016.

[7] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano. COBAYN: compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):21:1–21:25, 2016.

[8] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos. MiCOMP: Mitigating the compiler phase-ordering problem using optimization subsequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):29:1–29:28, 2017.

[9] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51 (5):96:1–96:42, 2019.

[10] J. Avigad and J. Zach. The epsilon calculus. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab Center for the Study of Language and Information Stanford University Stanford, CA 94305, winter 2013 edition, 2013.

[11] R. Baghdadi, A. Cohen, S. Verdoolaege, and K. Trifunovic. Improved loop tiling based on the removal of spurious false dependences. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):52:1–52:26, 2013.

[12] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In *Proceedings of the 12th International Conf. on Machine Learning (ICML)*, pages 38–46. Morgan Kaufmann, 1995.

[13] W. Bao, C. Hong, S. Chunduri, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, and P. Sadayappan. Static and dynamic frequency scaling on multicore CPUs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):51:1–51:26, 2016.

[14] W. Bao, S. Krishnamoorthy, L.-N. Pouchet, and P. Sadayappan. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(POPL):32:1–32:26, 2018.

[15] A. I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Mathematics of Operations Research*, 19(4):769–779, 1994.

[16] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of thr 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 7–16. IEEE, Sept. 2004.

[17] C. Bastoul. *Improving Data Locality in Static Control Programs.* PhD thesis, Université Pierre-et-Marie-Curie, 2004.

[18] C. Bastoul and P. Feautrier. Improving data locality by chunking. In G. Hedin, editor, *Proceedings of the 12th International Conference on Compiler Construction (CC)*, pages 320–334. Springer, Apr. 2003.

[19] U. Beaugnon, A. Pouille, M. Pouzet, J. A. Pienaar, and A. Cohen. Optimization space pruning without regrets. In P. Wu and S. Hack, editors, *Proceedings of the 26th International Conference on Compiler Construction (CC)*, pages 34–44. ACM, 2017.

[20] K. Belkadi, M. Gourgand, and M. Benyettou. Parallel genetic algorithms with migration for the hybrid flow shop scheduling problem. *Journal of Applied Mathematics and Decision Sciences (JAMDS)*, 2006:65746:1–65746:17, 2006.

[21] M. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In R. Gupta, editor, *Compiler Construction (CC)*, LNCS 6011, pages 283–303. Springer, Mar. 2010.

[22] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995.

[23] F. Bernasch. A distributed genetic algorithm for Polyite. Bachelor thesis, University of Passau, 2018.

[24] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1996.

[25] U. Bondhugula, M. M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In L. J. Hendren, editor, *Compiler Construction (CC)*, LNCS 4959, pages 132–146. Springer, Mar. 2008.

[26] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In R. Gupta and S. P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, pages 101–113. ACM, June 2008.

[27] U. Bondhugula, O. Günlük, S. Dash, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In V. Salapura, M. Gschwind, and J. Knoop, editors, *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 343–352. ACM, 2010.

[28] U. Bondhugula, A. Acharya, and A. Cohen. The Pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(3):12:1–12:32, May 2016.

[29] U. Bondhugula, V. Bandishti, and I. Pananilath. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(5):1285–1298, May 2017.

[30] T. Bray. The JavaScript object notation (JSON) data interchange format. *RFC*, 8259: 1–16, 2017.

[31] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[32] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984. ISBN 0-534-98053-8.

[33] P. Brémaud. *Markov Chains*, volume 31 of *Texts in Applied Mathematics*. Springer, 1st edition, 2010.

[34] J. Cavazos, G. Fursin, F. V. Agakov, E. V. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the 5th International Symposium on Code Generation and Optimization (CGO)*, pages 185–197. IEEE, Mar. 2007.

[35] L. Chang, D. J. Frank, R. K. Montoye, S. J. Koester, B. L. Ji, P. W. Coteus, R. H. Dennard, and W. Haensch. Practical strategies for power-efficient computing technologies. *Proceedings of the IEEE*, 98(2):215–236, 2010.

[36] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova. Evaluation of the Intel® Core™ i7 Turbo Boost feature. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 188–197. IEEE, 2009.

[37] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In M. Burke and M. L. Soffa, editors, *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, pages 286–297. ACM, June 2001.

[38] T. Chen, L. Zheng, E. Q. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy. Learning to optimize tensor programs. In S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 3393–3404. NIPS, 2018.

[39] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. Evaluating iterative optimization across 1000 datasets. *SIGPLAN Notices*, 45(6):448–459, June 2010.

[40] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(3):21:1–21:30, Sept. 2012.

[41] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 676–687. IEEE, 2011.

[42] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings – Software*, 150(3):161–175, June 2003.

[43] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache. Facilitating the search for compositions of program transformations. In *Proceedings of the 19th International Conference on Supercomputing (ICS)*, pages 151–160. ACM, 2005.

[44] K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. ACME: adaptive compilation made efficient. In Y. Paek and R. Gupta, editors, *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 69–77. ACM, June 2005.

[45] B. Cosenza, J. J. Durillo, S. Ermon, and B. H. H. Juurlink. Autotuning stencil computations with structural ordinal regression learning. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 287–296. IEEE, 2017.

[46] D. K. Danner. A performance prediction function based on the exploration of a schedule search space in the polyhedron model. Master thesis, University of Passau, 2017.

[47] G. B. Dantzig. Linear programming. *Operations Research*, 50(1):42–47, 2002.

[48] A. Darte, Y. Robert, and F. Vivien. *Scheduling and automatic parallelization*. Birkhäuser, 2000.

[49] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideovt, E. Bassous, and A. R. Leblanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Solid-State Circuits Society Newsletter*, 12(1):38–50, 2007.

[50] P. Feautrier. Parametric integer programming. *RAIRO – Operations Research*, 22(3): 243–268, 1988.

[51] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming (IJPP)*, 20(1):23–53, 1991.

[52] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *International Journal of Parallel Programming (IJPP)*, 21(5): 313–347, 1992.

[53] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming (IJPP)*, 21(6): 389–420, 1992.

[54] P. Feautrier and C. Lengauer. Polyhedron model. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, volume 3, pages 1581–1591. Springer, 2011.

[55] F. Fernández and P. Quinton. Extension of Chernikova's algorithm for solving general mixed linear programming problems. Technical Report RR-0943, INRIA, 1988.

[56] P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, 1986.

[57] F. Franchetti, T. M. Low, D. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura. SPIRAL: extreme performance portability. *Proceedings of the IEEE*, 106(11):1935–1968, 2018.

[58] E. Frank, M. A. Hall, and I. H. Witten. *Data Mining: Practical Machine Learning Tools and Technique*, chapter The WEKA Workbench, Online Appendix, pages 1–128. Morgan Kaufmann, 4th edition, 2016.

[59] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. V. Bonilla, J. Thomson, C. K. I. Williams, and M. F. P. O'Boyle. Milepost GCC: machine learning enabled self-tuning compiler. *International Journal of Parallel Programming (IJPP)*, 39(3):296–327, 2011.

[60] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, chapter Behavioral Patterns, pages 221–349. Addison-Wesley Publishing Company, 1994.

[61] S. Ganser, A. Größlinger, N. Siegmund, S. Apel, and C. Lengauer. Iterative schedule optimization for parallelization in the polyhedron model. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):23:1–23:26, Aug. 2017.

[62] S. Ganser, A. Größlinger, N. Siegmund, S. Apel, and C. Lengauer. Speeding up iterative polyhedral schedule optimization with surrogate performance models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4):56:1–56:27, Jan. 2019.

[63] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN 0-7167-1044-7.

[64] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *International Journal of Parallel Programming (IJPP)*, 28(6):607–631, Dec. 2000.

[65] T. Grosser, A. Größlinger, and C. Lengauer. Polly – Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters (PPL)*, 22 (4), 2012.

[66] T. Grosser, S. Verdoolaege, and A. Cohen. Polyhedral AST generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(4):12:1–12:50, Aug. 2015.

[67] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Chapman and Hall / CRC Computational Science Series. CRC Press, 2011.

[68] M. Harman. The current state and future of search based software engineering. In L. C. Briand and A. L. Wolf, editors, *International Conference on Software Engineering (ICSE)), Proc. Workshop on the Future of Software Engineering (FOSE)*, pages 342–357. IEEE, May 2007.

[69] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[70] T. Hiroyasu, M. Miki, and M. Negami. Distributed genetic algorithms with randomized migration rate. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, volume 1, pages 689–694. IEEE, Oct 1999.

[71] J. H. Holland. *Adaptation in Natural and Artificial Systems.* The MIT Press, 1992.

[72] *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Intel, Apr. 2018. URL `https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf`.

[73] F. Irigoin. Tiling. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, volume 4, pages 2040–2049. Springer, 2011.

[74] F. Irigoin and R. Triolet. Supernode partitioning. In J. Ferrante and P. Mager, editors, *Confernence Record of the 15th Annanual ACM Symposium on Principles of Programming Languages (POPL)*, pages 319–329. ACM Press, 1988.

[75] R. M. Karp. Reducibility among combinatorial problems. In M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors, *Proceedings of a Symposium on the Complexity of Computer Computations*, pages 85–103. Springer, 1972.

[76] C. Kartsaklis, O. R. Hernandez, C. Hsu, T. Ilsche, W. Joubert, and R. L. Graham. HERCULES: A pattern driven code transformation system. In *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPS)*, pages 574–583. IEEE, 2012.

[77] V. I. Kelefouras. A methodology pruning the search space of six compiler transformations by addressing them together as one problem and by exploiting the hardware architecture details. *Computing*, 99(9):865–888, Sept. 2017.

[78] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. In *Proceedings of the 1st International Conference on Algorithms and Architectures for Parallel Processing (ICAPP)*, volume 1, pages 153–162. IEEE, Apr. 1995.

[79] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, editors, *6th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, LNCS 768, pages 301–320. Springer, Aug. 1993.

[80] D. Kim, L. Renganarayanan, D. Rostron, S. V. Rajopadhye, and M. M. Strout. Multi-level tiling: M for the price of one. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 51:1–51:12. ACM, 2007.

[81] V. Klee and G. J. Minty. How good is the simplex algorithm? In *Inequalities, III (Proc. Third Sympos., Univ. California, Los Angeles, Calif., 1969; dedicated to the memory of Theodore S. Motzkin)*, pages 159–175. Academic Press, New York, 1972.

[82] A. Kleen. An NUMA API for Linux. Technical report, SUSE Labs, 2004.

[83] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24(1):43–67, 2003.

[84] P. M. W. Knijnenburg, T. Kisuki, K. A. Gallivan, and M. F. P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. *Concurrency and Computation: Practice and Experience*, 16(2-3):247–270, 2004.

[85] P. R. Krishnaiah and P. K. Sen, editors. *Handbook of Statistics*, volume 4. North-Holland, 1984.

[86] S. Kronawitter. *Automatic Performance Optimization of Stencil Codes*. PhD thesis, University of Passau, 2020.

[87] S. Kronawitter and C. Lengauer. Polyhedral search space exploration in the exastencils code generator. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4):40:1–40:25, 2019.

[88] C. Lattner. LLVM and Clang: Next generation compiler technology. In *BSDCan: The BSD Conference*, May 2008.

[89] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 75–88. IEEE, Mar. 2004.

[90] H. Le Verge. A note on Chernikova's algorithm. Res. Report RR-1662, INRIA, 1994.

[91] S. Long and G. Fursin. A heuristic search algorithm based on unified transformation framework. In *34th International Conference on Parallel Processing Workshops (ICPP 2005 Workshops)*, pages 137–144. IEEE, 2005.

[92] S. Long and G. Fursin. Systematic search within an optimisation space based on unified transformation framework. *International Journal of Computational Science and Engineering (IJCSE)*, 4(2):102–111, 2009.

[93] S. Long and M. F. P. O'Boyle. Adaptive Java optimisation using instance-based learning. In P. Feautrier, J. R. Goodman, and A. Seznec, editors, *Proceedings of the 18th International Conference on Supercomputing (ICS)*, pages 237–246. ACM, June 2004.

[94] S. Long and W. Zhu. A solution to the can or cannot problem of learning based compilation. In *Proceedings of the 6th International Conference on Natural Computation (ICNC)*, pages 3261–3265. IEEE, Aug. 2010.

[95] S. Long and W. Zhu. Outlier detection for learning-based optimizing compiler. In I. Stojmenovic, G. E. Farin, M. Guo, H. Jin, K. Li, L. Hu, X. Wei, and X. Che, editors, *Proceedings of the 5th International Conference on Frontier of Computer Science and Technology (FCST)*, pages 570–575. IEEE, Aug. 2010.

[96] J.-F. Mai and M. Scherer. *Simulating Copulas: Stochastic Models, Sampling Algorithms, and Applications*, volume 6 of *Series in Quantitative Finance*. World Scientific, 2nd edition, 2017.

[97] T. H. Matheiss and D. S. Rubin. A survey and comparison of methods for finding all vertices of convex polyhedral sets. *Mathematics of Operations Research*, 5(2):167–185, 1980.

[98] Message Passing Interface Forum. MPI: A message-passing interface standard. Standard, Revision 3.1, 5 University of Tennessee, Knoxville, June 2015.

[99] H. O. Mete and Z. B. Zabinsky. Pattern hit-and-run for sampling efficiently on polytopes. *Operations Research Letters*, 40(1):6–11, 2012.

[100] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.

[101] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In D. Scott, editor, *Artificial Intelligence: Methodology, Systems, and Applications (AIMSA)*, LNCS 2443, pages 41–50. Springer, Sept. 2002.

[102] G. E. Moore. Progress in digital integrated electronics. In *Proceedings of the 1975 International Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[103] S. Nembrini, I. R. König, and M. N. Wright. The revival of the Gini importance? *OUP Bioinformatics*, pages 1–8, 2018.

[104] A. Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In P. Sloot, M. Bubak, and B. Hertzberger, editors, *International Conference and Exhibition on High-Performance Computing and Networking (HPCN Europe)*, LNCS 1401, pages 987–989. Springer, Apr. 1998.

[105] A. Nisbet. Towards retargettable compilers – feedback directed compilation using genetic algorithms. In *Proceedings of the 9th International Workshop on Compilers for Parallel Computers (CPC)*, page 12 pages, 2001.

[106] C. Nugteren, P. Custers, and H. Corporaal. Algorithmic species: A classification of affine loop nests for parallel programming. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):40:1–40:25, 2013.

[107] N.V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities (in russian). *U.S.S.R. Computational Mathematics and Mathematical Physics*, 4(4):151–158, 1964.

[108] N.V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities (in russian). *U.S.S.R. Computational Mathematics and Mathematical Physics*, 5(2):228–233, 1965.

[109] N.V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem (in russian). *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6), 1968.

[110] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala.* Artima, 2008.

[111] I. Pak. *Foundations of Computational Mathematics*, chapter On Sampling Integer Points in Polyhedra, pages 319–324. World Scientific, 2002.

[112] P.-Q. Pan. *Linear Programming Computation.* Springer, 2014. ISBN 978-3-642-40754-3.

[113] E. Park, S. Kulkarni, and J. Cavazos. An evaluation of different modeling techniques for iterative compilation. In R. K. Gupta and V. J. Mooney, editors, *Proceedings of the 14th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 65–74. ACM, Oct. 2011.

[114] E. Park, L.-N. Pouchet, J. Cavazos, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *Proceedings of the 9th International Symposium on Code Generation and Optimization (CGO)*, pages 119–129. IEEE, Apr. 2011.

[115] E. Park, J. Cavazos, and M. A. Alvarez. Using graph-based program characterization for predictive modeling. In C. Eidt, A. M. Holler, U. Srinivasan, and S. P. Amarasinghe, editors, *Proceedings of the 10th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 196–206. ACM, Apr. 2012.

[116] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *International Journal of Parallel Programming (IJPP)*, 41(5):704–750, 2013.

[117] E. Park, C. Kartsaklis, and J. Cavazos. HERCULES: strong patterns towards more intelligent predictive modeling. In *Proceedings of the 43rd International Conference on Parallel Processing (ICPP)*, pages 172–181. IEEE, Sept. 2014.

[118] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research (JMLR)*, 12:2825–2830, 2011.

[119] L.-N. Pouchet. *Iterative Optimization in the Polyhedron Model.* PhD thesis, Université de Paris-Sud, 2010.

[120] L.-N. Pouchet. LeTSeE – The LEgal Transformation SpacE Explorator, 2012. URL `http://web.cs.ucla.edu/~pouchet/software/letsee/`.

[121] L.-N. Pouchet and T. Yuki. PolyBench 4.1, 2015. `http://web.cse.ohio-state.edu/~pouchet/software/polybench/`.

[122] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, One-dimensional time. In *Proceedings of the 5th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 144–156. IEEE, Mar. 2007.

[123] L.-N. Pouchet, C. Bastoul, J. Cavazos, and A. Cohen. A note on the performance distribution of affine schedules. In *2nd Workshop on Statistical and Machine learning approaches applied to ARchitectures and compilaTion (SMART), Göteborg, Sweden*, Jan. 2008.

[124] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, Multidimensional time. In R. Gupta and S. P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, pages 90–100. ACM, June 2008.

[125] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, R. Ramanujam, and P. Sadayappan. Hybrid iterative and model-driven optimization in the polyhedral model. Research Report RR-6962, INRIA, 2009.

[126] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, Nov. 2010.

[127] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: Convexity, pruning and optimization. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 549–562. ACM, Jan. 2011.

[128] B. Pradelle, P. Clauss, and V. Loechner. Adaptive runtime selection of parallel schedules in the polytope model. In L. T. Watson, G. W. Howell, W. I. Thacker, and S. Seidel, editors, *Proceedings of the 19th High Performance Computing Symposia (HPC)*, pages 81–88. SCS/ACM, 2011.

[129] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In R. Elliott, editor, *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC)*, pages 4–13. ACM, 1991.

[130] M. Püschel, J. M. F. Moura, J. R. Johnson, D. A. Padua, M. M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

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

[132] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In H. Boehm and C. Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 519–530. ACM, 2013.

[133] J. B. Rogers. *A Prolog Primer.* Addison-Wesley Longman Publishing Co., Inc., 1986.

[134] R. Ruvinskiy and P. van Beek. An improved machine learning approach for selecting a polyhedral model transformation. In D. Barbosa and E. E. Milios, editors, *Advances in Artificial Intelligence (Canadian AI)*, LNAI 9091, pages 100–113. Springer, 2015.

[135] V. Sarkar. Optimized unrolling of nested loops. In *Proceedings of the 14th International Conference on Supercomputing (ICS)*, pages 153–166. ACM, 2000.

[136] Y. Sato, T. Yuki, and T. Endo. An autotuning framework for scalable execution of tiled code via iterative polyhedral compilation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4):67:1–67:23, Jan. 2019.

[137] C. Schmitt, S. Kuckuk, F. Hannig, H. Köstler, and J. Teich. Exaslang: a domain-specific language for highly scalable multigrid solvers. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 42–51. IEEE, 2014.

[138] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1994.

[139] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar. Analytical bounds for optimal tile size selection. In M. F. P. O'Boyle, editor, *Proceedings of the 21st International Conference on Compiler Construction (CC)*, pages 101–121. Springer, 2012.

[140] A. Simbürger and A. Größliger. On the variety of static control parts in real-world programs: from affine via multi-dimensional to polynomial and just-in-time. In S. Rajopadhye and S. Verdoolaege, editors, *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Jan. 2014.

[141] S. Sioutas, S. Stuijk, H. Corporaal, T. Basten, and L. J. Somers. Loop transformations leveraging hardware prefetching. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*, pages 254–264. ACM, 2018.

[142] P. J. Smith. *Into Statistics.* Springer, 1997.

[143] L. Song and K. M. Kavi. A technique for variable dependence driven loop peeling. In *Proceedings of the 5th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 390–395, Oct 2002.

[144] K. Stock, L.-N. Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):50:1–50:23, 2012.

[145] J. Thomson, M. F. P. O'Boyle, G. Fursin, and B. Franke. Reducing training time in a one-shot machine learning-based compiler. In G. R. Gao, L. L. Pollock, J. Cavazos, and X. Li, editors, *22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC) 2009, Revised Selected Papers*, pages 399–407. Springer, 2009.

[146] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 327–337. IEEE, 2009.

[147] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta. GRAPHITE two years after: First lessons learned from real-world polyhedral compilation. In D. N. Grigori Fursin, editor, *Proceedings of the International Workshop on GCC Research Opportunities (GROW)*, pages 1–13, 2010.

[148] R. Upadrasta and A. Cohen. Sub-polyhedral scheduling using (unit-)two-variable-per-inequality polyhedra. In R. Giacobazzi and R. Cousot, editors, *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 483–496. ACM, Jan. 2013.

[149] N. Vasilache. *Scalable Program Optimization Techniques in the Polyhedral Model.* PhD thesis, Université Paris-Sud, 2007.

[150] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *Proceedings of the 15th International Conference on Compiler Construction (CC)*, pages 185–201. Springer, May 2006.

[151] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *Computing Research Repository (CoRR)*, 2018.

[152] S. Verdoolaege. *isl*: An integer set library for the polyhedral model. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software – ICMS 2010*, LNCS 6327, pages 299–302. Springer, 2010.

[153] S. Verdoolaege. Presburger formulas and polyhedral compilation. Technical report, Polly Labs and KU Leuven, 2016.

[154] S. Verdoolaege. *Integer Set Library: Manual.* INRIA, 2018. Version isl-0.19.

[155] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In U. Bondhugula and V. Loechner, editors, *Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Jan. 2012.

[156] S. Verdoolaege and G. Janssens. Scheduling for PPCG. Technical Report CW706, CS Department, KU Leuven, 2017.

[157] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48 (1):37–66, June 2007.

[158] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54:1–54:23, Jan. 2013.

[159] J. von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[160] Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O'Boyle. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(1):2:1–2:26, 2014.

[161] V. Weispfenning. Parametric linear and quadratic optimization by elimination. Technical Report MIP-9404, University of Passau, Jan. 1994.

[162] M. Weiss. Strip mining on SIMD architectures. In *Proceedings of the 5th International Conference on Supercomputing, (ICS)*, pages 234–243. ACM, Nov. 1991.

[163] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[164] S. Williams, A. Waterman, and D. A. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4): 65–76, 2009.

[165] T. Williams and C. Kelly. *gnuplot 5.2: An Interactive Plotting Program*, Jan. 2019.

[166] M. Wolfe. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming (IJPP)*, 15(4):279–293, Aug. 1986.

[167] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: simple linux utility for resource management. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 44–60. Springer, 2003.

[168] T. Yuki. Understanding PolyBench/C 3.2 kernels. In S. Rajopadhye and S. Verdoolaege, editors, *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Jan. 2014.

[169] X. Zhang, A. Ramachandran, C. Zhuge, D. He, W. Zuo, Z. Cheng, K. Rupnow, and D. Chen. Machine learning on fpgas to face the iot revolution. In S. Parameswaran, editor, *International Conference on Computer-Aided Design (ICCAD)*, pages 819–826. IEEE, 2017.

[170] O. Zinenko, L. Chelini, and T. Grosser. Declarative transformations in the polyhedral model. Research Report RR-9243, Inria, ENS Paris - Ecole Normale Supérieure de Paris, ETH Zurich, TU Delft, IBM Zürich, Dec. 2018.

[171] O. Zinenko, S. Verdoolaege, C. Reddy, J. Shirako, T. Grosser, V. Sarkar, and A. Cohen. Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In C. Dubach and J. Xue, editors, *Proceedings of the 27th International Conference on Compiler Construction (CC)*, pages 3–13. ACM, Feb. 2018.