**University of Passau**
Faculty of Computer Science and Mathematics


# A Performance Prediction Function based on the Exploration of a Schedule Search Space in the Polyhedron Model

## Master's thesis

Dominik Karl Danner

February 28, 2017

Supervisors: Prof. Christian Lengauer, Ph.D.
Prof. Dr.-Ing. Sven Apel
Stefan Ganser

# Contents

## Abstract

The Polyhedron Model is an abstract representation of loop programs. Optimized schedules are usually obtained by heuristics solving linear inequalities [55]. This master's thesis explores the search space of legal multi-dimensional schedules for a Static Control Part (SCoP) by randomly selecting regions of the search space and generating samples from each region. Furthermore, a performance prediction function is trained with machine learning techniques.

Static analysis tools, such as Polly [4], extract the Static Control Parts (SCoPs) of a program. With the use of dependency information, a given algorithm selects regions of the search space and models them as a list of polyhedra. Each polyhedron corresponds to one schedule dimension.

The result of normalization and performance measurement on a high number of generated schedules provides the input data for machine learning a performance prediction function. Features like the possibility for parallelization, out-of-order execution and tiling, cache hit rate and computational overhead prediction are extracted from the schedule representation and form the basic factors of the learned function. Benchmarks from the Polybench suite [5] are used to evaluate the prediction function.

# Chapter 1

# Introduction

Automatic loop optimization is highly relevant in high performance computing [17, 18, 31]. Most algorithms, for example the PLuTo algorithm [18], build an optimized parallel execution order by solving linear inequalities. Recent work has shown that exhaustive search in the space of legal loop transformations can lead to better transformation sequences that are out of the range of classical compiler optimizations [53, 55]. But the huge number of runtime evaluations needed for iterative compilation optimization is computationally expensive.

This chapter motivates the need for a surrogate function that approximates the performance of different loop transformations in the polyhedron model. Furthermore, an overview of the components of such a function is given.

## 1.1 Motivation

*Iterative compilation* techniques usually optimize compiler flags and parameters. Among others, Pouchet et al. [53, 55] extended this approach to the variety of different transformation sequences of the program using a genetic algorithm. The genetic algorithm iteratively improves a set of schedules by joining up the transformations of the best schedules, which are determined by the runtime of the transformed program. For this purpose, the whole set of programs must be executed at each iteration of the algorithm. A fitness function that classifies a schedule by its performance-related key features without executing them can accelerate this process. Furthermore, if the fitness function performs significantly faster than the actual program execution takes, the number of generations and/or schedules per generation can be increased.

*Just-In-Time(JIT)* compilers can also benefit from such a surrogate function that approximates the runtime of a transformed program by its schedule. The performance of a schedule highly depends on the target platform, e.g. the cache size and number of cores. Furthermore, some optimizations can only be applied with the knowledge of the runtime parameters. With a surrogate function, JIT compilers can verify whether a given (pre-optimized) schedule is likely to perform well on the target hardware. Since JIT compilers operate at runtime, it is necessary to decide rapidly whether code generated from a specific schedule is worthwhile.

## 1.2 Approach

This master's thesis proposes to build a surrogate function that predicts the performance of a schedule in the polyhedron model and performs faster than the actual execution time. In order to learn such a function with machine learning techniques, different features, which are expected to highly influence the performance of a schedule, are inspected:

- **Parallelism.** Parallel computation of (transformed) loops can reduce the execution time drastically, especially if the target platform has a huge number of CPU cores [18, 42].
- **Data Locality.** Programs whose performance is bounded by memory bandwidth can benefit from CPU caches, because recently used data are kept for reuse in faster memory levels. A transformed loop program will utilize the caches better if the number of distinct memory accesses between two memory references to the same cell is small. Loop tiling can further increase data locality [18, 36].
- **Overhead.** Loop transformations that result in expensive boundary checks and/or require additional if-statements inside the loops produce computational overhead. This affects the performance of the transformed program, especially if the program is computation bound.
- **Inner Parallelism.** The absence of dependencies at the innermost loop can exploit instruction level parallelism and hide cache latency [37, 67]. Both influence the performance of a loop program.

These features are evaluated on several programs of the Polybench Benchmark Suite [5]. For each program, one thousand schedules are sampled well distributed from 50 different regions of the search space of legal transformation sequences. The measured runtime of the generated programs and the result of the features, that are calculated on the normalized schedule representation, form the input data of machine learning a performance prediction function.

## 1.3 Evaluation

First, we evaluate the runtime of the proposed sampling algorithm. Furthermore, the feature calculation time is of high interest, since it must not exceed the measured runtime of the generated programs, otherwise the performance prediction function has no advantage over simple runtime measurement.

Second, the correlation between the feature values and the measured data is examined. The feature results form the input vector for the machine learning algorithms and, hence, must correlate with the execution times in order to get a accurate performance prediction.

Finally, we learn performance prediction models with linear regression and the k-nearest neighbor algorithm. For the input data set we chose the following six benchmark programs from the Polybench Suite: *cholesky*, *gemm*, *seidel-2d*, *syrk*,

*syr2k* and *trmm*, which are some variation of matrix multiplications, a matrix decomposition and a two-dimensional 9-point stencil. Using both machine learning algorithms, the prediction models are trained on three of the six chosen benchmark programs and validated on the other three benchmarks.

## 1.4 Related Work

Nisbet [49] used a prediction function in the GAPS framework for *Single Program Multiple Data (SPMD)* execution, in order to evaluate the performance of different loop transformations in a genetic algorithm approach. The naive fitness function simply calculates the sum of loop and synchronization costs of the program. Assuming that not the whole program but single loops are computed in parallel, our fitness function includes the granularity of parallelism of the program. Furthermore, fast sequential parts of a program rely on good cache reuse, which is not part of the prediction function by Nisbet.

Caşcaval and Padua [19] proposed a *stack histogram algorithm* to estimate the number of cache misses. For each memory reference $m$ to a memory cell $C$ the stack distance (number of distinct memory references between $m$ and the last reference to the same cell $C$) is computed using a stack processing algorithm [43]. The number of memory references is accumulated for each stack distance to obtain the stack histogram. The sum of all memory references that have a larger stack distance than the cache size estimates the total number of cache misses. Even with efficient algorithms [10], the stack histogram computation is not applicable at run-time. For so-called *loop-carried* dependencies (two loop iterations reference the same memory cell) the number of loop iterations between the two memory references is computed and referred to as the *distance vector*. The set of intermediate memory references between two depending loop iterations is computed symbolically for the entire loop by one single formula, which depends on the iteration variable of the surrounding loop. The cardinality of this set is the stack distance for the target memory reference and is identical for all dependency instances of the loop, as long as the dependency is *uniform* (the distance vector is constant for all dependency instances), loop boundaries are constant or loop invariant and there are no conditional instructions in the loop body. But loop skewing [69] transforms nested loops such that the variables of inner loops depend on outer loop variables. This changes the execution order of the memory references inside this loop nest and can, hence, enable additional loop tiling [36] to improve data locality. This thesis includes schedules that can have skewed loops. In that case, the loop boundaries are neither constant nor loop invariant and the stack distance can only be computed inaccurately with the stack histogram algorithm. Some other loop transformations, like loop fusion, can introduce conditional instructions inside a loop body, which also leads to inaccuracy. Furthermore, the dependency analysis, needed for the stack histogram algorithm, is computational expensive on highly skewed loops [56].

Chatterjee et al. [21] propose an alternative way to compute cache misses based on the cache architecture. The commonly used $n$-way cache is organized in different cache sets, which store $n$ cache lines. A cache line is a set of data that is transfered from RAM to the caches. A memory reference to a cache line $B$ results in a cache

miss if there is an earlier reference to a different cache line mapping to the same cache set as $B$ and there is no access to $B$ between this earlier access and the current access. These conditions are expressed by a set of Presburger formulas and define a polytope, whose number of elements is equal to the number of cache misses of the loop nest. Since the initial cache state is not known, the analysis is divided into interior misses, which can be counted by analyzing the loop in isolation, and potential boundary misses, which depend on the initial cache state.

## 1.5 Result

The proposed sampling algorithms is fast enough to obtain a huge number of schedules in a reasonable time. Experiments showed that this is valid for at least half of the benchmark programs of the Polybench Suite.

During the experiments we made one interesting observation, that a good cache hit rate (independent of the cache level) does not imply a fast execution time for all of the six chosen benchmark programs. That explains that only some of the proposed features correlate directly with the measured data of the generated programs. For some features the outcome is totally different with schedules from different SCoPs. The learned prediction functions are over-fitted to the trainings benchmark programs, such there is no correlation between the predicted and measured execution times of schedules from some of the testing benchmarks. This means that the prediction model cannot be used with arbitrary programs.

From this result we conclude that some of the proposed feature metrics are not suitable for performance prediction of a schedule. We believe that there must be other features that lead to a better correlating performance prediction model.

# Chapter 2

# Polyhedron Model

Automatic parallelization and data locality optimization is an important topic in high performance computing [17, 18, 31]. The polyhedron model permits to explore a wide range of loop transformations systematically. Among others, the set of expressible transformations comprises loop reversal, loop interchange, and loop skewing [9, 11, 12, 69]. This chapter starts with background knowledge about the polyhedron model and describes some loop transformations that can be applied. A short part at the end shows different schedule representations in the polyhedron model.

## 2.1 Basics

The *polyhedron model* is an abstract representation of a loop program as a computation graph [30]. Each node of the graph, representing an iteration of a statement, is associated with a point in a $\mathbb{Z}$-polyhedron, which is inferred from the bounds of the surrounding loops. The set of code regions that are expressible in the polyhedron model is limited to so-called *Static Control Parts (SCoP)*, although recent work tries to enhance the application field [32, 16]. A SCoP is the maximal set of consecutive static control statements, like for-loops and if-statements, where conditions and array subscript functions are limited to affine functions of iteration variables and global parameters, that are known at compile time. The exact mathematical definition of an affine function and a $\mathbb{Z}$-polyhedron, also called lattice-polyhedron, is given in Definitions 2.1 and 2.2. A more detailed view on polyhedral theory and integer programming is presented by Schrijver [58].

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

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

**Definition 2.2** ($\mathbb{Z}$-Polyhedron)**.** *A set $\mathcal{P} \in \mathbb{Z}^m$ is a $\mathbb{Z}$-polyhedron if there exists a system with a finite number of inequalities $A\vec{x} \leq \vec{b}$, such that:*

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

```
for(i = 0; i < n_i; i++) {
  C[i] = 0;                              // statement S[i]
  for(j = 1; j < n_j; j++)
    C[i] = C[i] + A[i][j - 1] + B[j]; // statement R[i,j]
}
```

**Example 2.3:** *Simple SCoP example*

The program code in Example 2.3 represents one SCoP over two nested loops with a total of two different non-control statements $S$ and $R$.

For each statement of a SCoP, $S$ and $R$ in our example, the control and the data flow are expressed by three algebraic structures: an iteration domain, subscript functions and a schedule [14, 53]. Several algorithms and tools exist that can regenerate code from this abstract model [14, 57].

**Iteration Domain.** The *iteration domain* of a statement $S$ contains all dynamic instances that are executed within the statement's surrounding loops. Each statement instance is identified uniquely by the values of the iteration variables of the statement's surrounding loops, also called the *iteration vector* $\vec{x}$. Since all loop bounds are affine inequalities, the iteration domain of each statement in a SCoP can be represented by a polyhedron. This polyhedron is bounded by constraints that are derived from the bounds of the statement's surrounding loops.

Statement $R$ from Example 2.3 is executed within two loops $i$ and $j$. The iteration domain, as a polyhedron, is spanned according to the bounds of the loops and is defined by

$$\mathbb{D}_R = \{(i, j) \mid 0 \leq i < n_i \wedge 1 \leq j < n_j\}.$$

**Subscript Function.** Inside a SCoP, all memory accesses are considered to be array references (a reference on a variable is just a specific array reference). The *subscript function* is an affine function that calculates the data location on which a statement operates. There is one subscript function for each memory access of each statement. This allows to analyze the data flow within the SCoP.

In Example 2.3 the subscript function for the read access on $A[i][j-1]$ of statement $R$ is defined by $f(i, j) = (i, j - 1)$.

**Schedule.** A *schedule* is a function which provides an execution date for each instance of a statement. All instances are executed according to the increasing order of the execution dates. As a simplification, only affine scheduling functions are considered, since non-affine functions yield significant problems in code generation [34]. Given a statement $S$, the scheduling function $\theta_S$ can be written in the form

$$\theta_S : \mathbb{Z}^{dim(\mathbb{D}_S)} \to \mathbb{Z}^p : x_S \to K \begin{pmatrix} x_S \\ \vec{n} \\ 1 \end{pmatrix}, K \in \mathbb{Z}^{p \times (dim(\mathbb{D}_S) + dim(\vec{n}) + 1)}$$

where $x_S$ is the vector of iteration variables, $\vec{n}$ is the parameter vector specified at compile time and $K$ is a constant coefficient matrix. In case $K$ has multiple

rows ($p > 1$), $\theta_S$ is a multi-dimensional schedule and the scheduling function of dimension $d$ is referred to as $\theta_S^d$. The p-dimensional result of the scheduling function $(c_1, \ldots, c_p)$ is interpreted like a clock. The first schedule dimension is the most significant one and is treated like the hours, next one like minutes, and so on. According to Feautrier [29](Theorem 2) every SCoP has a multi-dimensional schedule. The schedule of a SCoP is a set of scheduling functions for all its statements and is named $\theta$.

A schedule for a statement $S$ is defined as *complete*, if the scheduling function $\theta_S$ is injective. This means that each statement instance of $S$ is assigned a different execution date. The schedule of a SCoP $\theta$ is complete, if all scheduling functions are injective and it does not map instances of different statements to the same execution date.

Example 2.3 has the following multi-dimensional scheduling functions: $\theta_S(i) = (i, 0)$ and $\theta_R(i, j) = (i, j)$. The instances of both statements are executed according to $i$ in the first schedule dimension. The second dimension orders the statement instances, such that one instance of $S$ is executed at time 0 and the instances of $R$ are executed after $S$ according to $j$, because $j \in [1, n_j - 1]$.

**Dependencies.** The space of possible coefficients in the scheduling function's matrix for multi-dimensional schedules is large and contains a wide range of different schedules for the same SCoP. But choosing coefficients for the schedule matrix $K$ randomly can likewise result in an illegal program version. Among others, Nisbet observed that arbitrary coefficients likely lead to an illegal schedule [48, 50].

To avoid this problem, the coefficients must be selected by additionally considering the dependencies between statement instances. There is a *dependency* between two statement instances, if both access the same memory location and at least one access is a write operation. A dependency can be modeled as a relation between two sets of statement instances. Instances of the second set depend on instances from the first set, according to the relation. The commonly used mathematical representation of a dependency is the *dependency polyhedron*, which is a subset of the Cartesian product of the iteration domains of two statements $R$ and $S$. Each point inside this polyhedron corresponds to a dependency between two statement instances $\vec{x}_R$ and $\vec{x}_S$ [27, 55].

A schedule is legal, if it *satisfies* all dependencies of the SCoP. The mathematical description for dependency satisfaction is given in Definition 2.4.

**Definition 2.4** (Dependency Satisfaction)**.** *Let $S$ and $R$ be two statements, $D_{S,R}$ be a dependency relation, and $\theta_S$ and $\theta_R$ be the scheduling functions of the two statements. $\theta_S$ and $\theta_R$ solves the dependency $D_{S,R}$, if*

$$\forall d : \vec{x}_S \to \vec{x}_R \in D_{S,R} : \theta_S(\vec{x}_S) \prec \theta_R(\vec{x}_R) \quad (\prec \textit{ denotes the lexicographical ordering.}^1)$$

One-dimensional scheduling functions must carry all dependencies within the single time dimension. Using multi-dimensional schedules, a dependency needs to be *weakly solved* ($\theta_R(\vec{x}_R) - \theta_S(\vec{x}_S) \succeq 0$) for the first time dimensions until it is *strongly solved* ($\theta_R^d(\vec{x}_R) - \theta_S^d(\vec{x}_S) > 0$) at a given time dimension $d$. This schedule dimension

[1] $(a_1, \ldots, a_n) \prec (b_1, \ldots, b_m)$ iff there exists an index $1 < i < \min(n, m)$ such that $(a_1, \ldots, a_{i-1}) = (b_1, \ldots, b_{i-1})$ and $a_i < b_i$

*d satisfies* the dependency. Once a dependency is solved strongly, no further time dimension $d' > d$ need to solve this dependency [55].

In this context one also talks about dependency directions. The direction of a dependency is defined in Definition 2.5.

**Definition 2.5** (Direction of a Dependency)**.** *Let $S$ and $R$ be two statements, $D_{S,R}$ be a dependency relation and $\theta_S$ and $\theta_R$ be the scheduling functions.*

*At schedule dimension $d$, the direction function of the dependency between two statement instance $\vec{x}_S \to \vec{x}_R \in D_{S,R}$ is defined as:*

$$dir_{\theta_S^d, \theta_R^d}(\vec{x}_S \to \vec{x}_R) = \begin{cases} 1 & \textit{(forward)} \quad , \textit{ if } \theta_S^d(\vec{x}_S) - \theta_R^d(\vec{x}_R) > 0 \\ 0 & \textit{(no direction)} \quad , \textit{ if } \theta_S^d(\vec{x}_S) - \theta_R^d(\vec{x}_R) = 0 \\ -1 & \textit{(backward)} \quad , \textit{ if } \theta_S^d(\vec{x}_S) - \theta_R^d(\vec{x}_R) < 0 \end{cases}$$

*The total dependency relation $D_{S,R}$ has a direction, if all $\vec{x}_S \to \vec{x}_R \in D_{S,R}$ have the same direction.*

Example 2.3 contains two data dependencies. The first one

$$\{S[i] \to R[i, 0] : 0 \leq i < n_i\}$$

is the dependency between the two instances $S[i]$ and $R[i, 0]$ and indicates, that the statement instance $R[i, 0]$ must be executed after $S[i]$ for all $i \in [0; n_i - 1]$, because both access the same memory cell $C[i]$. The second dependency

$$\{R[i, j] \to R[i, j + 1] : 0 \leq i < n_i \wedge 1 \leq j < n_j - 1\}$$

only affects the statement $R$ and implies that the inner loop cannot be executed in parallel, since there is a read and write access on the same memory cell $C[i]$ in each $j$-iteration.

Most scheduling algorithms, e.g. Pluto [17, 18] or the iterative approach by Pouchet et. al. [55, 53], use dependency information to construct a schedule for a given SCoP. More details on constructing a search space, containing only legal schedules, are given in Chapter 3.

## 2.2 Loop Transformations

The schedule, as the iteration domain transformation function, describes the exact execution date of each statement instance. The execution sequence of the statement instances can be rearranged by applying a different loop transformation on the iteration domain. The resulting target code can potentially have more data locality and the possibility to execute more loops in parallel, which improves the runtime of the SCoP [17].

Within the polyhedron model each unique coefficient matrix for the schedule represents a different iteration domain transformation for the given SCoP. It is recalled that this schedule is only legal if it satisfies all the dependencies. This chapter will illustrate the following transformations: loop interchange, loop reversal, loop skewing, loop distribution, loop fusion and loop peeling [52, 54]. Additionally, the more complex tiling transformation is discussed.

### 2.2.1 Loop Interchange

In perfectly nested loops, which means that all statements occur in the innermost loop [8], the order of the loops can be *interchanged*. This is only legal if it does not invalidate the program. In Example 2.6 *(a)* the statement $S$ is surrounded by two loops. After swapping the two schedule dimensions of $\theta_S$ in part *(b)*, now the loop with the iteration variable $j$ is the outermost loop.

$$\theta_S(\vec{x}_S) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} = (i, j)$$

```
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++)
    S(i,j)
```

*(a) original schedule and program code*

$$\theta_S(\vec{x}_S) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} = (j, i)$$

```
for(j = 0; j < n; j++)
  for(i = 0; i < n; i++)
    S(i,j)
```

*(b) schedule with interchanged loops and program code*

**Example 2.6:** *Example of loop interchange: (a) the original schedule and (b) after loop interchange transformation.*

### 2.2.2 Loop Reversal

The direction, in which a loop processes its iteration range, can be *reversed*. This can be done for each nested loop separately. Example 2.7 shows a loop reversal transformation. The iteration over the instances of $S$ is reversed by negation of the iteration variable's coefficient. After the transformation *(b)*, the resulting code executes the statement instances of $S$ in reversed order, beginning with the last element of the iteration range.

$$\theta_S(\vec{x}_S) = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} = (i)$$

```
for(i = 0; i < n; i++)
  S(i)
```

*(a) original schedule and program code*

$$\theta_S(\vec{x}_S) = \begin{pmatrix} -1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} = (-i)$$

```
for(i = n - 1; i >= 0; i--)
  S(i)
```

*(b) schedule reversed loop iteration and program code*

**Example 2.7:** *Example of loop reversal: (a) the original schedule and (b) after loop reversal transformation.*

### 2.2.3 Loop Skewing

With *skewing*, the boundary of an inner loop depends on an outer loop variable [69]. Consider the SCoP in Example 2.8, which has the following three dependencies: $S[i,j] \to S[i+1,j+1]$, $S[i,j] \to S[i,j+1]$ and $S[i,j] \to S[i+1,j]$. Part *(a)* and *(b)* show the schedule and the program code before and after the transformation. The graphs in *(c)* and *(d)* show the execution date of the statement instances. Each point corresponds to a statement instance and an arrow represents a dependency instance. For both graphs, the parameter $n$ is set to 4. As can be seen in *(c)*, neither the $i$ nor the $j$-loop can be executed in parallel, because each statement instance depends on instances that are executed before in both dimensions. After the skewing transformation *(d)*, the inner loop can now be computed in parallel, since the dependencies are shifted towards the first dimension ($j$-loop).

$$\theta_S(\vec{x}_S) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} = (i, j)$$

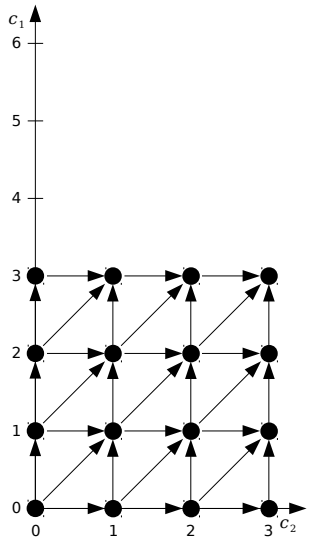
```
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++)
    S(i,j)
```

*(a) original schedule and program code*

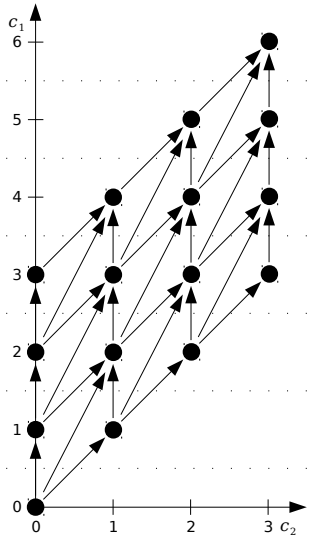
$$\theta_S(\vec{x}_S) = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} = (i+j, j)$$

```
for(i = 0; i < 2*(n-1)+1; i++)
  for(j = max(0, i-n+1);
      j < min(i+1, n);
      j++)
    S(i - j,j)
```

*(b) schedule with skewed loop and program code*




*(c) Neither the $i$ nor the $j$ loop can be executed in parallel.*




*(d) The inner $j$ loop does not carry dependencies and can be parallelized.*

**Example 2.8:** *Example of loop skewing: (a) the original schedule and (b) after loop skewing transformation. (c) and (d) shows the execution date of the statement instances. Each point corresponds to an instance and an arrow represents a dependency.*

### 2.2.4 Loop Distribution and Fusion

Loop *distribution* splits a single nested loop, with more than one statement, into many loop nests. This can create a total order on statements, where all instances of one statement are executed before or after the instances of another statement. It is also possible to *fuse* two statements into the same nested loop. Example 2.9 shows the distribution of the loop that surrounds the two statements $R$ and $S$. In part *(a)*, both statements are executed in the same loop. The second schedule dimension determines the execution order inside the loop. In the transformed code *(b)* all instances of $S$ are shifted back by the value of $n$, such that all instances of $R$ are executed before $S$. The second schedule dimension of the new schedule does not influence the execution order of the statement instances anymore and can be removed. A more detailed description of multi-dimensional schedule normalization is presented in Section 3.3.

$$\theta_R(\vec{x}_R) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} = (i, 0)$$

$$\theta_S(\vec{x}_S) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} = (i, 1)$$

```
for(i = 0; i < n; i++) {
  R(i)
  S(i)
}
```

*(a) original schedule and program code*

$$\theta_R(\vec{x}_R) = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} = (i)$$

$$\theta_S(\vec{x}_S) = \begin{pmatrix} 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} = (i{+}n)$$

```
for(i = 0; i < n; i++)
  R(i)
for(i = n; i < 2*n; i++)
  S(i - n)
```

*(b) schedule with distributed loops and program code*

**Example 2.9:** *Example of loop distribution: (a) the original schedule and (b) after loop distribution transformation.*

### 2.2.5 Loop Peeling

The *peeling* transformation extracts one or more statement instances of a given loop. If the coefficient value is at least as high as the parameter $n$, the execution sequence of two or more statements can be reordered. The loop peeling transformation is depicted in Example 2.10. The original schedule in part *(a)* executes $S[i]$ straight after $R[i]$. After the transformation *(b)* the execution date of $S$ is shifted back by one. As a result, $S[i-1]$ is now directly executed after $R[i]$ and the first instance of $R$ and the last one of $S$ are executed before and after the loop.

$$\theta_R(\vec{x}_R) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} = (i, 0)$$

$$\theta_S(\vec{x}_S) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} = (i, 1)$$

```
for(i = 0; i < n; i++) {
  R(i)
  S(i)
}
```

*(a) original schedule and program code*

$$\theta_R(\vec{x}_R) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} = (i, 0)$$

$$\theta_S(\vec{x}_S) = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} = (i{+}1, 1)$$

```
R(0)
for(i = 1; i < n; i++) {
  R(i)
  S(i-1)
}
S(n-1)
```

*(b) schedule with an extracted single statement instance from the loop and program code*

**Example 2.10:** *Example of loop peeling: (a) the original schedule and (b) after loop peeling transformation.*

### 2.2.6 Loop Tiling

On modern processors the reuse of cached data is of high importance. *Tiling* is a transformation that can improve data locality of a loop nest by changing the execution order of the statement instances without invalidating the semantics of the program. From a mathematical perspective, tiling is a partitioning of the loop iteration space that induces a renumbering and a reordering of the iterations. A partition of the iteration space is called a *tile*. Unlike many other transformations, tiling is not unimodular because it modifies the iteration domain [36]. Tiling can have a huge performance impact [35].

The tiling transformation doubles the number of loops $n$ around a statement. The outer $n$ loops enumerate the tiles and the inner $n$ loops are used to execute all the iterations of each tile. Tiling is only legal if there are no dependency cycles between the tiles. The schedule dimensions that generate the tiled loops must solve all of the remaining dependencies of the SCoP at least weakly as a sufficient condition for tiling [17, 36, 70].

Example 4.19*(a)* shows a schedule with these two dependencies: $D_1 : S[i, j] \to S[i + 1, j + 1]$ and $D_2 : S[i, j] \to S[i + 1, j]$. The iteration domain is given as $\mathbb{D}_S = \{(i, j) \mid 0 \leq i < 6 \wedge 0 \leq j < 6\}$. The two dependent statement instances of $D_1$ access the same memory cell, but they are executed in different iterations of the outer loop $i$ *(c)*. If the number of array references from iterations of the inner loop $j$ is to high, the data of the memory cell may not be present in the cache at the second access. The tiling transformation adds two new iteration coefficients to the iteration domain of $S$, that iterate over the tiles. Each tile contains only a partition of the original iteration domain. The new execution order is depicted in *(d)*. The outer two dimensions enumerate the four rectangular tiles. Inside a tile, the statement

$$\mathbb{D}_S = \{(i,j) | 0 \leq i < 6 \wedge 0 \leq j < 6\}$$

$$\theta_S(\vec{x}_S) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} = (i,j)$$

```
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++)
    S(i,j)
```

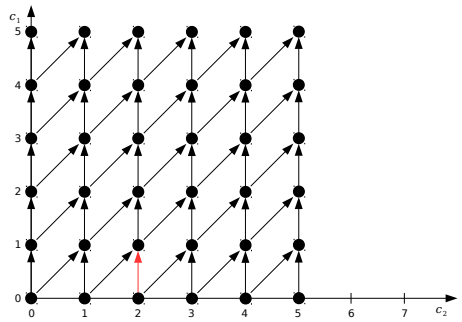
*(a) iteration domain, original schedule and program code*

$$\mathbb{D}_S = \{(ti,tj,i,j) | 0 \leq i < 6 \wedge 0 \leq j < 6 \wedge ti = \lfloor i/3 \rfloor \wedge tj = \lfloor j/3 \rfloor\}$$

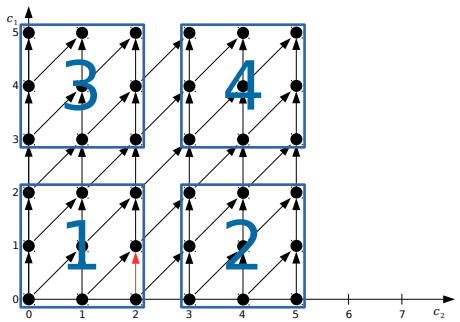
$$\theta_S(\vec{x}_S) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} ti \\ tj \\ i \\ j \\ n \\ 1 \end{pmatrix} = (ti,tj,i,j)$$

```
for(ti = 0; i < n; ti+=3)
  for(tj = 0; j < n; tj+=3)
    for(i = 3ti; i < 3ti+3; i++)
      for(j = 3tj; j < 3tj+3; j++)
        S(i,j)
```

*(b) iteration domain, loop tiled schedule and program*



*(c) 5 statement instances are executed between the two instances of the red dependency.*




*(d) The $i$ and $j$ loop are tiled with a tile size of 3. Now only 2 statement instances must be executed between the two instances of the red dependecy.*

**Example 2.11:** *Example of loop tiling: (a) the original schedule and (b) after loop tiling transformation with tile sizes of 3. (c) and (d) shows the execution order of both schedules. Each point corresponds to a statement instance and an arrow represents a dependency.*

instances are executed in the order of the original schedule. It is possible to define another execution order for the statement instances inside a tile as well. For a given dependency instance inside a tile (e.g. the marked one in the graph) the number of statement instances that are executed between the two dependent instances is smaller than with the original schedule. This dramatically increases data locality inside a tile.

Unlike the loop transformations before, tiling modifies the iteration domain. Thus, it is hard to integrate tiling in the search space of legal schedules. But tiling can also be seen as a post-scheduling transformation. Some tools, like Polly [4], recognize tilable schedule dimensions and can perform the tiling transformation on them.

## 2.3 Schedule Representation

There are different ways to represent multi-dimensional schedules in the polyhedron model. This chapter discusses the representation of a schedule as a matrix, a union map and as a schedule tree. Further tree-like representations are illustrated and compared by Verdoolaege et al. [65].

The schedule of each statement of a SCoP may be specified by a scheduling function with a coefficient matrix, like it is introduced in Section 2.1. The overall schedule for all statements is a set of scheduling functions with the specified matrices. From this representation, it is hard to derive the order in which the statement instances are executed.

A very similar schedule representation is the *union map*. It provides a multi-dimensional affine linear expression for each statement $S_k$, which is the result of the matrix vector product of the schedule matrices $K_{S_k}$ with the vectors $(\vec{x_k}, \vec{n}, 1)^T$. The union map representation omits unnecessary zero coefficients, but the execution order relation between two statements is still hard to identify.

Schedules in the polyhedron model naturally have the form of a tree [65]. Both, the schedule matrix and the union map, only express the tree implicitly. Verdoolaege et al. [65] provide a comparison of different schedule representations and introduce a generic schedule tree, which is also used in this thesis. The tree is built with the following node types.

- *Context*: The context node introduces new constants or constraints on known constants.
- *Domain*: The domain node defines the iteration domain of all statements that are scheduled by the descendant nodes.
- *Sequence*: The sequence node defines the specific order, in which the children nodes are processed.
- *Set*: The set node processes its children nodes in arbitrary order.
- *Filter*: The filter node is typically the child of a sequence or set node and selects a subset of statement instances introduced by outer domain nodes.

- *Band*: The band node contains a multi-dimensional affine scheduling function for statement instances defined by outer domain nodes and selected by outer filter nodes. Additionally, properties like the possibility to compute schedule dimensions in parallel or the option to perform tiling are included.
- *Mark*: The mark node can be used to mark its subtree with a user-specified label.

Each node of a schedule tree can have only one child, except sequence and set nodes.

Example 2.12 shows the schedule of Example 2.3 in different representations. The iteration domains of the two statements are: $\mathbb{D}_S = \{i \mid 0 \leq i < n\}$ and $\mathbb{D}_R = \{(i,j) \mid 0 \leq i < n \wedge 1 \leq j < n\}$. In part *(a)* the schedule is represented with single scheduling functions for each statement. The representation in *(b)* is the matrix vector product and is called union map, whereas in *(d)* the same schedule is given in schedule tree notation.

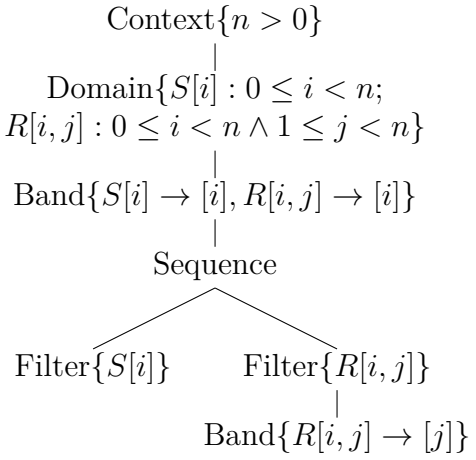
$$\theta_S(\vec{x}_S) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} i \\ n \\ 1 \end{pmatrix}$$

$$\theta_R(\vec{x}_R) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}$$

*(a) scheduling functions for each statement.*

$\{S[i] \to [i,0],\ R[i,j] \to [i,j]\}$

*(b) schedule as a union map.*




*(c) schedule as a tree.*

**Example 2.12:** *Different schedule representations with the iteration domains* $\mathbb{D}_S = \{i \mid 0 \leq i < n\}$ *and* $\mathbb{D}_R = \{(i,j) \mid 0 \leq i < n \wedge 1 \leq j < n\}$.

# Chapter 3

# Sampling Schedules from the Schedule Search Space

Choosing the coefficients of a schedule matrix at random likely leads to an illegal program version [48, 50]. There exist algorithms for computing an optimized schedule, one example is PLUTO, an automatic parallelization tool by Bondhugula et. al., which computes a multi-dimensional affine schedule with parallel loops on outer dimensions and loops with dependencies on inner dimensions [17, 18].

While it is easy to verify a posteriori that a given schedule preserves all dependencies, one would like to create a search space containing only the legal schedules. This faces two combinatorial problems. First, if using multi-dimensional schedules, the earliest dimension in which each dependency is solved strongly can vary. This results in a large number of different search polyhedra for each schedule dimension. Second, the search polyhedra can contain an infinite number of schedules and the coefficients must therefore be limited to a specific range [55]. Section 3.1 shows different solutions for how to construct a search space for multi-dimensional schedules.

In order to produce input data for the machine learning algorithms, a schedule sampling strategy is needed to obtain uniformly distributed schedules from different regions of the search space. In Section 3.2, this problem is reduced to picking points from a $\mathbb{Z}$-polyhedron uniformly. For that, different strategies are presented. Each sampled point from a polyhedron corresponds to the coefficient vector of a matrix row in the scheduling function.

The selected schedules are then transformed into schedule trees. Since two different schedules can produce the same generated imperative target code, a normalization pass on schedule trees is presented in Section 3.3. This normalization can improve analysis and readability of different schedules because it eliminates unused coefficients, which have no influence on the execution order of the statement instances. It also enhances further compiler optimizations, like tiling.

## 3.1 Generation of the Search Space

Only multi-dimensional schedules are considered here, because according to Feautrier not every program has a one-dimensional schedule [29]. Thus, there is one search polyhedron for each schedule dimension.

**Building Search Polyhedra.** One solution to the enormous number of possible polyhedra is the greedy algorithm developed by Feautrier [29]. It tries to maximize the number of dependencies that are strongly solved in the first schedule dimension and continues recursively. The resulting search space has one search polyhedron per schedule dimension and as few schedule dimensions as possible [29, 55].

Pouchet modified the algorithm of Feautrier slightly [55]. Additionally to solve as many dependencies in the outer dimensions as possible, his algorithm processes the dependencies in a specific order, determined by two criteria. First, the dependencies are sorted by the amount of data traffic, that they generate. The more memory cells are accessed by the statements of a dependency, the better it is placed in an inner schedule dimension. This minimizes traffic in outer loops, whereas the reuse of data in inner loops is maximized. If the order of two dependencies is not decidable by the amount of traffic, the second criterion applies. Similar to Feautrier, the number of dependencies that are solved in one schedule dimension should be maximized. This is done by prioritizing the dependency that interferes with the lowest number of other dependencies. Two dependencies interfere, if no one-dimensional schedule exists that satisfies both two dependencies strongly.

The approach in *Polyite* proposed by Ganser et. al. expands the search space by the dependency-to-dimension mapping, that means which dependency is satisfied by which schedule dimension. The algorithm selects the dependencies that must be solved strongly for each dimension randomly and returns a region of the total search space. As a result, several of such regions must be inspected to obtain schedules from the complete search space. The resulting schedule potentially has more dimensions than necessary, but the normalization passes developed by Ganser et. al. (described in Section 3.3) remove redundant dimensions. A step-by-step explanation of the search space construction is given in Figure 3.1. First *(1)*, the total set of dependencies $G$, that are necessary for the correctness of the SCoP, is computed. For each dependency $D_{R,S} \in G$ the space of legal schedules $\mathcal{W}_{D_{R,S}}$ weakly solving $D_{R,S}$ and $\mathcal{S}_{D_{R,S}}$ strongly solving $D_{R,S}$ is precomputed. The order in which the dependencies of the SCoP are processed is random. The following steps *(4)(a)-(f)* are executed until $G$ is empty. Each iteration creates a new schedule dimension $d$. In order to span the search space polyhedron of the new schedule dimension in step *(a)*, a set of dependencies $G_S \subseteq G$, that should be solved strongly in this dimension, is randomly determined. The probability for the set to be empty is configurable. Thus, outer parallel schedule dimensions are possible. The next steps, *(b)* and *(c)*, initialize a polyhedron $\mathcal{L}_d$ to the full space and add the constraints for solving all remaining dependencies from $G$ weakly. It is recalled that a dependency must be solved weakly, until it is solved strongly in any dimension $d$. All further dimensions $d' > d$ do not need to solve this dependency anymore. Additionally, in step *(d)* the algorithm tries to solve the dependencies from $G_S$, which should be solved strongly in this dimension. If two dependencies interfere and, therefore, the resulting polyhedron would be empty, only the first dependency is chosen to be solved strongly in this schedule dimension. Remember that the order of the dependencies is random. As a last step *(e)*, all dependencies from $G$ that are solved strongly by this schedule dimension are removed from the set of unsatisfied dependencies $G$.

**Data:** SCoP Information (Read/Write access and iteration variable for each Statment)
**Result:** Schedule Search Space
(1) Compute the set $G$ of pairwise dependences for the SCoP;
(2) **foreach** *dependency* $D_{R,S} \in G$ **do**
  ($a$) Compute $\mathcal{W}_{D_{R,S}}$ - the space of legal schedules weakly solving $D_{R,S}$: $\theta_S(\vec{x}_S) - \theta_R(\vec{x}_R) \geq 0$;
  ($b$) Compute $\mathcal{S}_{D_{R,S}}$ - the space of legal schedules strongly solving $D_{R,S}$: $\theta_S(\vec{x}_S) - \theta_R(\vec{x}_R) > 0$ ;
**end**
(3) $d \leftarrow 1$;
(4) **while** $G \neq \emptyset$ **do**
  ($a$) Select random number of dependecies to carry strongly in this schedule dimension and put them in $G_S \subseteq G$ ($P(G_S = \emptyset)$ is configurable);
  ($b$) Initialize $\mathcal{L}_d$ - the space for legal schedules for dimension $d$ - to full-space polyhedron;
  ($c$) **foreach** *dependency* $D_{R,S} \in G$ **do**
    $\mathcal{L}_d \leftarrow \mathcal{L}_d \cap \mathcal{W}_{D_{R,S}}$;
  **end**
  ($d$) **foreach** *dependency* $D_{R,S} \in G_S$ **do**
    **if** $\mathcal{L}_d \cap \mathcal{S}_{D_{R,S}} \neq \emptyset$ **then**
      $\mathcal{L}_d \leftarrow \mathcal{L}_d \cap \mathcal{S}_{D_{R,S}}$;
    **end**
  **end**
  ($e$) Remove all dependences from $G$ that are strongly solved in $\mathcal{L}_d$;
  ($f$) $d \leftarrow d + 1$;
**end**

**Figure 3.1:** *Construction of a Schedule Search Space*

All the aforementioned algorithms can find only affine schedules. Vivien et. al. [66] showed that the approach by Feautrier is known to be non-optimal, in sense of finding all possible parallelism of the program. This is not due to the design of the algorithm, but it relies on the limitation of the underlying framework. First, that only affine schedules are considered, but handling more general scheduling functions will cause problems in code generation [34]. Second, the polyhedron model has the limitation that each statement only has one scheduling function. Griebl et al. [33] developed a solution to split the iteration domain of a statement.

This master's thesis uses the algorithm by Ganser et. al. to span multiple regions of the search space, because later the performance prediction function is intended to work within the Polyite project.

**Bounding Search Polyhedra.** After modeling a search space region, there is one polyhedron for each schedule dimension, but each can contain an infinite number of points. For a reasonable search space exploration, it is necessary to limit the value of the coefficients, so that the polyhedra are bounded. Pouchet et. al. allowed coefficient values within $\{-1, 0, 1\}$, because they only considered sequential

codes and wanted to minimize control-flow overhead [55]. With iteration coefficients bounded to $\{-1, 0, 1\}$ only unit-skewing is possible. This restriction eliminates a lot of schedules with dimensions that expresses parallelism.

But what is a reasonable boundary for the coefficient values? Obviously, if a value $a \in \mathbb{N}$ can be assigned to a coefficient, the negative $-a$ should be assignable as well. If it is an iteration coefficient, this expresses the reversal transformation, otherwise the loop or instance-wise shift is inverted. If $a$ is a large number, the quantity of schedules in the search space will explode, which makes a detailed search space exploration impossible. Further, as it will be discussed in Section 3.3, the two schedules

$$\theta_S(\vec{x}) = (5, 0, 0) \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} = 5 \cdot i$$

$$(\theta_S)'(\vec{x}) = (1, 0, 0) \begin{pmatrix} i \\ n \\ 1 \end{pmatrix} = 1 \cdot i$$

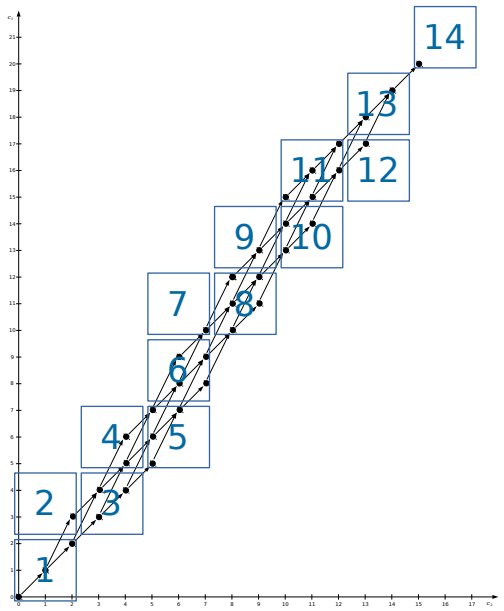
will produce code that iterates the statement instances of $S$ in the same order. Allowing high values for the coefficients will add a lot of redundant schedules to the search space.

$$\mathbb{D}_S = \{(i, j) | 0 \leq i < 6 \wedge 0 \leq j < 6\}$$

$$D_1 : S[i, j] \to S[i + 1, j + 1]$$
$$D_2 : S[i, j] \to S[i + 1, j - 1]$$

$$\theta_S(\vec{x}_S) = \begin{pmatrix} 3 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}$$
$$= (3i + j, 2i + j)$$

*(b) iteration domain, dependencies and schedule that can be tiled and parallized.*




*(d) The $i$ and $j$ loop is tiled by a tile size of 3. Inside each tile the $j$ loop can be computed in parallel.*

**Example 3.2:** *Example for a schedule with coefficients limited to* $\{-3, \ldots, 3\}$.

The direction of simple dependencies that only span one or two iterations of each surrounding loop can be inverted by skewing with coefficient values up to 3. This can result in the possibility to compute dimensions in parallel and/or in tiles. The

iteration domain of the statement $S$ in Example 3.2 is $\mathbb{D}_S = \{(i,j) \mid 0 \leq i < 6 \wedge 0 \leq j < 6\}$ and there are two simple dependencies that only span one iteration of each loop $D_1 : S[i,j] \to S[i+1, j+1]$ and $D_2 : S[i,j] \to S[i+1, j-1]$. With the given schedule $\theta_S(\vec{x}_S) = (3i + j, 2i + j)$ the iteration instances of the statement $S$ can be computed in tiles and the inner loop $j$ inside each tile can be parallelized.

The skewing transformation can produce better performing code by enabling tiling and parallel computation if the dependencies only span a small number of iterations of the surrounding loops. Otherwise that leads to high iteration coefficients and the tiles will only contain a few statement instances producing more overhead than speed-up. Consequently, the coefficient values are limited to $\{-3, \ldots, 3\}$ in this thesis.

## 3.2 Schedule Sampling Strategies

The algorithms presented in Section 3.1 construct one $\mathbb{Z}$-polytope for each schedule dimension to model the search space or a region of the space, respectively. Picking one point from each $\mathbb{Z}$-polytope forms the matrix of a multi-dimensional scheduling function. With the aim of exploring the search space, the schedule should be uniformly chosen, but this yields two problems. First, because of the huge number of possible different regions of the search space of a SCoP it is infeasible to inspect all regions. Furthermore, the different regions are not pairwise distinct and can have diverse dimensionality, which makes schedule comparison impossible. Second, different selected schedules (even from one region) can produce code that processes the statement instances in the same execution order.

Even though it is impossible to uniformly sample schedules from a search space, we create several regions of the search space and sample a huge number of schedules from each region. The row vectors of the schedule matrix are uniformly picked from the corresponding $\mathbb{Z}$-polytope. The selected schedules are normalized by the normalization passes given in Section 3.3 and duplicate schedules are removed. This does not produce a set of uniformly sampled schedules, but guarantees a good distribution inside the search space.

There are several methods how to uniformly pick points from a $\mathbb{Z}$-polytope. Below, four sampling strategies are discussed regarding these two questions: How many points are needed to obtain a uniform distribution on the $\mathbb{Z}$-polytope, and what is the computational effort?

### 3.2.1 Enumeration Sampling

The simplest way to obtain a uniformly sampled point from a discrete geometric body $x \in P \subseteq \mathbb{Z}^d$, is to enumerate all points of $P$ and select one randomly. Let $x_1, \ldots, x_{card(P)}$ be all points in $P$ and $\mathcal{X}$ a discrete random variable with the discrete uniform distribution

$$P(\mathcal{X} = x_i) = \frac{1}{card(P)}.$$

Sampling the random variable $\mathcal{X}$ returns a uniform point from $P$.

This method does not need a minimum number of points to get a uniform distribution. The computational effort can be specified with $\mathcal{O}(card(P))$, since first,

all points must be enumerated, and sampling one point only costs $\mathcal{O}(1)$. It also needs $\mathcal{O}(card(P))$ memory. Enumerating all points is computationally infeasible for $\mathbb{Z}$-polytopes with a very large number of points. It can only be used with thin polytopes, containing only a few thousand points.

### 3.2.2 Rejection Sampling

Another way to sample points from a $\mathbb{Z}$-polytope is *rejection sampling*, as a type of the Monte Carlo method [47]. First, build the minimal bounding box $B_{min}$ around the polytope $P$ and then uniformly select points from it. A point can be used, if it is inside $P$ too, otherwise reject it. Obviously, a uniformly sampled point $p$ in $B_{min}$, is also uniform in $P$, if $p \in P$.

Let $min_i$ and $max_i$ be the minimum and maximum value for dimension $i$ of the minimal bounding box $B_{min}$. For each dimension $i$, define $\mathcal{X}_i$ as a discrete uniform random variable with the outcome $\Omega_i = [min_i, max_i] \cap \mathbb{Z}$. Now a vector $p$ can be constructed by combining the uniformly sampled results $\omega_i$ of each random variable $\mathcal{X}_i$ to a vector.

$$p = (\omega_1, \ldots, \omega_d)$$

Each obtained point $p$ is uniformly distributed in the minimal bounding box $B_{min}$. This experiment must be repeated, until a point $p \in P$ is obtained. The probability to hit a point in $P$ is $\frac{card(P)}{card(B_{min})}$. Hence, to retrieve $n$ points from the polytope $P$, one must statistically generate and check a total of $n * \frac{card(B_{min})}{card(P)}$ points. If the polytope $P$ only covers a small area of the minimum bounding box $B_{min}$ this number explodes.

### 3.2.3 Hit and Run - Markov Chain Monte Carlo Method

There are other stochastic solutions to sample points from a $\mathbb{Z}$-polytope. The *Markov Chain Monte Carlo (MCMC)* methods are a class of sampling algorithms based on *Markov Chains* [47]. A *Markov Chain* is a state machine with state transitions that are weighted with probabilities. The probabilities of all outgoing transitions of a state must sum up to 1. The method of sampling points in a geometric body is also called *Hit and Run (HR)*. For a convex polytope in $P \subseteq \mathbb{R}^d$, hit and run methods, also known as random walk, start with an arbitrary point $p \in P$. This is the current state of the *Markov Chain*. Then a point $y \in \mathbb{R}^d$ from the unit ball is chosen uniformly random, such that the intersection of the line $l = \{p + \alpha \cdot y \mid \alpha \in \mathbb{R}\}$ and $P$ is not empty ($l \cap P \neq \emptyset$). An oracle $O$ can determine the point of interception of the line $l$ and the convex hull of $P$. With this information, it is possible to uniformly sample one point on $l \cap P$, which is the next point in the *Markov Chain* [47]. This may possibly be the same point $p$. The hit and run method needs a specific number of experiment runs, until it approximately reaches the desired distribution. The more samples are generated, the better they convergence to the desired distribution.

Several algorithms have been developed to efficiently sample uniform points from a convex continuous polytope in $\mathbb{R}^d$ [23, 24, 26]. But, sampling lattice points from a $\mathbb{Z}$-polytope is harder. Consider a high-dimensional $\mathbb{Z}$-polytope $P_{\mathbb{Z}} = P \cap \{-3, -2, -1, 0, 1, 2, 3\}^d$, similar to our search space polytopes. Then, it is hard to find a line $l$ in $\mathbb{R}^d$ through a given lattice point $p \in P_{\mathbb{Z}}$, that hits, at least, one point of

$P_{\mathbb{Z}} \setminus \{p\}$. Furthermore, an arbitrarily chosen line through a point $p \in P$ can contain a maximum of 7 points. As a consequence, it is unlikely to find a suitable line and, if one is found, the number of points on that line is very small, which likely leads to staying at the starting point. Both concerns highly increase the computational effort to get a reasonable number of uniformly distributed lattice points.

Baumert et. al. developed a discrete hit and run algorithm (DHR), that samples integer points from a subset of an integer hyper-rectangles [15]. It uses two independent nearest neighbor random walks instead of a line, but for each sampled point a new random biwalk is computed. Mete et. al. improved the runtime of the discrete hit and run algorithm by doing fixed, pattern based biwalks [45]. Thus, a random biwalk must not be generated in every iteration of the algorithm. They also extended their algorithm to perform efficiently on polytopes with knapsack constraints [44]. A knapsack constraint is of the form $\sum_{j=1}^{n} a_j x_j < b$, where $a_j$ is non-negative and $b$ is positive. With only one knapsack constraint, the convergence rate of PHR to a uniform distribution on $P$ is in $\mathcal{O}(n^{5.5})$. With $m$ knapsack constraints it is also polynomial in $n$, but the exponent contains $m$ as a factor [44].

The search space considered in this thesis can have constraints that cannot be expressed as a knapsack constraint. Furthermore, the dimensionality of the polytopes is very high; thus sampling with hit and run methods would not scale, because the number of points that must be sampled to obtain approximately uniform distribution explodes.

### 3.2.4 Geometric Sampling

Igor Pak [51] provides a *divide and conquer* algorithm for sampling integer points from a polytope. This method requires an oracle that can determine the number of lattice points in a polytope. For example, Barvinok's algorithm can be used as the oracle [13]. Figure 3.3 outlines Pak's approach. First, it tries to find a hyperplane $H$ that halves the given polytope $P \subset \mathbb{R}^n$ by the number of lattice points inside it. This is the case, if $\alpha = |P \cap H_+|/|P| < \frac{1}{2}$ and $\beta = |P \cap H_-|/|P| < \frac{1}{2}$ holds, where $H_+$ and $H_-$ are the half-spaces of $\mathbb{R}^n \setminus H$. Then, one of the polytopes $P \cap H_+$, $P \cap H_-$ or the hyperplane $P \cap H$ is chosen randomly, depending on the number of included lattice points. These two steps are recursively applied to the chosen sub-polytope, until one single point remains. At each reduction step, either the total number of remaining points is reduced by a factor of $> 2$ (in case $P \cap H_+$ or $P \cap H_-$ is chosen) or the dimensionality of the subproblem is decremented by one (in case choosing $P \cap H$).

The normal vector of the required hyperplane $H$ is of the form $H = (x_1, \ldots, x_n)$, where $x_i = c$ and $x_{j \neq i} = 0$. Clearly, for some dimension $i$ and some constant value $c$, a hyperplane with the given constraints exists. The constant $c$ is determined by binary search. For both steps, finding the hyperplane $H$ and selecting the subproblem, the oracle, respectively Barvinok's algorithm, is needed.

Pak's algorithm runs in polynomial time. It calls the oracle $\mathcal{O}(n^2 L^2)$ times. $L$ is the bit size of the input. Additionally, Pak proposes an improved version, which only calls the oracle $\mathcal{O}(n^2 \log L)$ times. In theory, a run of Barvinok's algorithm costs $L^{\mathcal{O}(n)}$, but it highly depends on the complexity of the constraints [51].

**Data:** $\mathbb{Z}$-polytope $P \subset \mathbb{R}^n$
**Result:** uniform sampled point from $P$
PAK($P$)
(1) Find a Hyperplane $H$, such that $\alpha = |P \cap H_+|/|P| < \frac{1}{2}$ and
$\beta = |P \cap H_-|/|P| < \frac{1}{2}$, where $H_+$ and $H_-$ are the half-spaces of $\mathbb{R}^n \setminus H$.
(With $\gamma = |P \cap H|/|P|$ this equation holds: $\alpha + \beta + \gamma = 1$.)
(2) Sample a random variable $\mathcal{X}$ with three outcomes at the probabilities of $\alpha$, $\beta$ and $\gamma$.
(3) Set $P'$ to $P \cap H_+$, $P \cap H_-$ or $P \cap H$, depending on the output of $\mathcal{X}$.
(4) **if** $|P'| = 1$ **then** return the element of $P'$ **else** PAK($P'$);

**Figure 3.3:** *Sampling uniformly from a $\mathbb{Z}$-polytope.*

A hybrid approach of geometric and enumeration sampling is used in this master's thesis to obtain uniformly distributed integer points from a polytope. Starting with the geometric divide and conquer method, the bounded polytope of each schedule dimension is shrunk, until the number of remaining points reaches a threshold and enumeration sampling can be performed on them.

Since none of the sampling strategies perfectly scales with the dimensionality of the arising polytope, the task of uniformly sampling points from a high-dimensional $\mathbb{Z}$-polytope remains for future investigation.

A schedule is built by uniformly picking one point for each schedule dimension from the corresponding search space polytope. The coordinates of each point reveals the values for the coefficients of the matrix row. It is only necessary to generate new schedule dimensions, until all dependencies are solved strongly. This keeps the number of schedule dimensions low. After that, further schedule dimensions with linear independent iteration coefficients are added. Each selected schedule is converted to the schedule tree representation [65]. During this step, the normalization passes described in the following Section 3.3 are applied.

## 3.3 Schedule Normalization

Two schedules of a SCoP are equivalent iff they define the same lexicographic order on all statement instances and all unsatisfied dependencies at each schedule dimension have the same direction. A formal description is given in Definition 3.4. Pouchet [52] specified a definition for semantic-preserving schedules, but in addition to that the normalization must preserve the execution order and the possibility to compute dimensions in parallel or to apply the tiling transformation.

**Definition 3.4** (Schedule Equivalence)**.** *Let $S_{1,\dots,k}$ be all statements of a SCoP with the following p-dimensional scheduling functions for each statement $S_{j \in \{1,\dots,k\}}$:*

$$\theta_{S_j} : \mathbb{Z}^{dim(\mathbb{D}_{S_j})} \to \mathbb{Z}^p : x_j \to K \begin{pmatrix} x_j \\ \vec{n}_j \\ 1 \end{pmatrix}, K \in \mathbb{Z}^{p \times (dim(\mathbb{D}_{S_j}) + dim(\vec{n}_j) + 1)}$$

*where $x_j$ is the vector of iteration variables and $\vec{n}_j$ is the vector of parameters of the statement $S_j$.*

*For a SCoP, the two schedules $\theta$ and $\theta'$ are equivalent, if*

*(a) The execution order of all statement instances is the same.*
$\forall S_a, S_b : a, b \in \{1, \ldots, k\} : \forall \vec{x}_a \in S_a, \vec{x}_b \in S_b :$

$$\theta_{S_a}(\vec{x}_a) \prec \theta_{S_b}(\vec{x}_b) \Leftrightarrow (\theta_{S_a})'(\vec{x}_a) \prec (\theta_{S_b})'(\vec{x}_b)$$

*(b) and for each dimensions $d$ all unsatisfied dependencies between two statement instances have the same direction.*
$\forall d \in \{1, \ldots, p\} : \forall D_{S_a,S_b} \in \mathcal{D}_d : \forall \vec{x}_a \rightarrow \vec{x}_b \in D_{S_a,S_b} :$

$$dir_{\theta^d_{S_a},\theta^d_{S_b}}(\vec{x}_a \rightarrow \vec{x}_b) = dir_{(\theta^d_{S_a})',(\theta^d_{S_b})'}(\vec{x}_a \rightarrow \vec{x}_b)$$

*where $\mathcal{D}_d$ is the set of unsatisfied dependencies at schedule dimension $d$.*

It is possible that two schedules with different matrices express the same execution order and direction of dependencies. For a better comparison one would like to normalize the representation of the schedule. The following normalization steps, introduced by Ganser et. al., are based on the schedule tree representation. During the construction of the schedule tree in Polyite, the total order on statements is extracted to sequence nodes as a first step of normalization. Sequence and set nodes cannot be further normalized, because they are already the elementary representation for executing their children in a specific order [65]. Among the other node types, only band nodes hold execution order information that can be normalized. If a band node contains a multi-dimensional schedule, it is processed dimension by dimension. For simplification we assume that the schedule is complete.

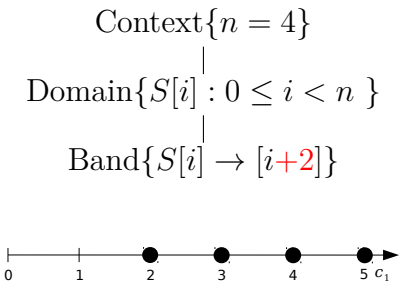
The first four normalization passes are developed by Ganser, whereas their proof of correctness is part of this thesis. Due to the lexicographic order of multi-dimensional schedules each pass starts at the root node (outer schedule dimension) of the schedule tree and is then processed recursively. The result is an equivalent schedule with a higher number of coefficients equal to zero and a possibly lower dimensionality.

In addition to the four normalization passes of the Polyite project, a further normalization step is introduced in this master's thesis. This step detects *hidden statement sequences* in schedule dimensions that express both, loop iterations by linear independent iteration coefficients and a sequence of statements by ascending constant coefficients.

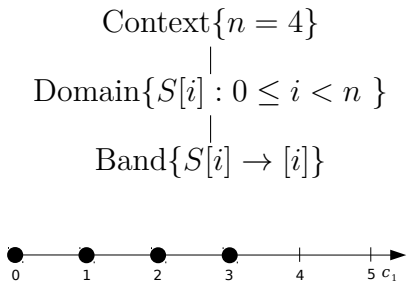
### 3.3.1 Common Constant Offset

The parameter and constant coefficients produce a constant offset for all statement instances. If all statements have a common offset $o \in \mathbb{Z}$ at dimension $d$, this offset can be subtracted from each of the scheduling functions. The effect is a shift of the execution dates of all statement instances by the constant offset in the time dimension $c_d$. Because the common offset $o$ can be expressed as a linear expression over the parameters and the constant vector, the resulting scheduling functions have more parameter and constant coefficients equal to zero. If only one statement is scheduled at the band node, the constant offset can be completely omitted.

The schedule $S[i] \to [i+2]$ of Example 3.5*(a)* has an offset of 2. It maps the statement instances $S[0], S[1], S[2]$ and $S[3]$ to the execution dates $2, 3, 4$ and 5. Since no statement instance is scheduled at time date 0 and 1 the shift of $+2$ is unnecessary and can be omitted. The equivalent schedule is shown in part *(b)*.




*(a) original schedule tree and execution order.*




*(b) normalized schedule tree and execution order.*

**Example 3.5:** *Example with a common offset that can be reduced.*

**Theorem 3.6.** *Let $S_{1,\dots,k}$ be all statements that are scheduled at the current band node at schedule dimension $d$ with the scheduling functions for each statement $S_{j\in\{1,\dots,k\}}$:*

$$\theta^d_{S_j} : \mathbb{Z}^{dim(\vec{x}_j)} \to \mathbb{Z} : (\vec{x}_j) \to (\vec{i}_j, \vec{p}_j, c_j) \begin{pmatrix} \vec{x}_j \\ \vec{n}_j \\ 1 \end{pmatrix}$$

*Define $(\theta^d_{S_j})'(\vec{x}_j) = \theta^d_{S_j}(\vec{x}_j) - o$ as the new scheduling function with a constant offset $o \in \mathbb{Z}$ for each statement $S_j$.*

*Then $\theta$ and $\theta'$ are equivalent.*

*Proof.*

(a) **Execution Order:** Obviously, the ordering at dimension $d' \neq d$ is equivalent. Now, we show that the execution order of the statement instances at dimension $d$ does not change, when subtracting a common offset $o \in \mathbb{Z}$. $\forall S_a, S_b : a, b \in \{1, \dots, k\} : \forall \vec{x}_a \in S_a, \vec{x}_b \in S_b$ :

$$\theta^d_{S_a}(\vec{x}_a) < \theta^d_{S_b}(\vec{x}_b) \Leftrightarrow (\theta^d_{S_a})'(\vec{x}_a) < (\theta^d_{S_b})'(\vec{x}_b)$$

$\Rightarrow$:
Let $S_a$ and $S_b$ be two statements scheduled at the band node, $\vec{x}_a$ be a instance of $S_a$ and $\vec{x}_b$ be a instance of $S_b$, such that $\theta^d_{S_a}(\vec{x}_a) < \theta^d_{S_b}(\vec{x}_b)$. Obviously, this is true:

$$(\theta^d_{S_a})'(\vec{x}_a) = \theta^d_{S_a}(\vec{x}_a) - o < \theta^d_{S_b}(\vec{x}_b) - o = (\theta^d_{S_b})'(\vec{x}_b)$$

$\Leftarrow$: Similar to $\Rightarrow$.

(b) **Dependency Direction:** Obviously, the direction of unsatisfied dependencies at dimension $d' \neq d$ is equivalent. Now, we show that the direction of the

unsatisfied dependencies at dimension $d$ does not change, when subtracting a common offset $o \in \mathbb{Z}$. $\forall D_{S_a,S_b} \in \mathcal{D}_d : \forall \vec{x}_a \to \vec{x}_b \in D_{S_a,S_b}$ :

$$dir_{\theta^d_{S_a},\theta^d_{S_b}}(\vec{x}_a \to \vec{x}_b) = dir_{(\theta^d_{S_a})',(\theta^d_{S_b})'}(\vec{x}_a \to \vec{x}_b)$$

Let $D_{S_a,S_b}$ be an unsatisfied dependency between two statements $S_a$ and $S_b$, $\vec{x}_a$ be a instance of $S_a$ and $\vec{x}_b$ be a instance of $S_b$, such that $\vec{x}_a \to \vec{x}_b \in D_{S_a,S_b}$. The condition for the direction of the dependency is equal for both scheduling functions $\theta^d$ and $(\theta^d)'$.

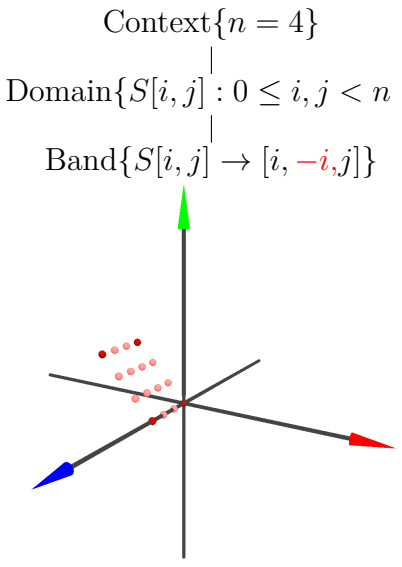
$$(\theta^d_{S_a})'(\vec{x}_a) - (\theta^d_{S_b})'(\vec{x}_b) = \theta^d_{S_a}(\vec{x}_a) - o - \left(\theta^d_{S_b}(\vec{x}_b) - o\right) = \theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b) - o + o$$

□

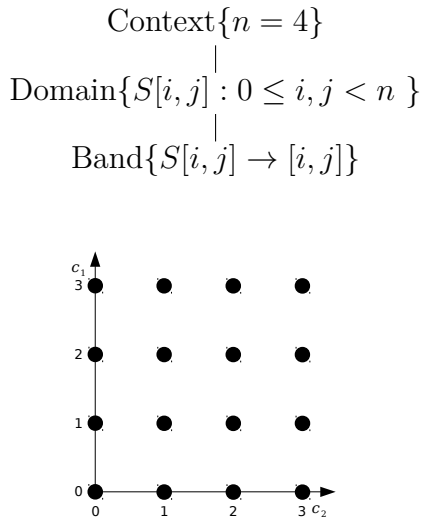
### 3.3.2 Uninfluential Schedule Dimension

There may be schedule dimensions that do not affect the executable code at all. These schedule dimensions do not introduce a new ordering on the statement instances of the band node and can be removed completely. The resulting scheduling function has one time dimension less.

The schedule of the Example 3.7 *(a)* produces execution dates with three dimensions. At the second dimension, the value of $i$ is already fixed by the outer dimension and therefore only one unique statement instance is scheduled for each step of the outer dimension. This schedule is equivalent to the one in the Example 3.7 *(b)*. The second dimension of the execution dates is projected out.




*(a) original schedule tree and execution order. The green axes denotes the first time dimension $c_1$, the red one the second $c_2$ and the blue one the third dimension $c_3$.*




*(b) normalized schedule tree and execution order.*

**Example 3.7:** *Example with an uninfluential dimension that can be omitted.*

**Theorem 3.8.** *Let $S_{1,\ldots,k}$ be all statements that are scheduled at the current band node at schedule dimension d with the scheduling functions for each statement $S_{j\in\{1,\ldots,k\}}$:*

$$\theta^d_{S_j} : \mathbb{Z}^{dim(\vec{x}_j)} \to \mathbb{Z} : (\vec{x}_j) \to (\vec{i}_j, \vec{p}_j, c_j) \begin{pmatrix} \vec{x}_j \\ \vec{n}_j \\ 1 \end{pmatrix}$$

*Define the prefix schedule as:*

$$Pre(\theta^d_{S_j})(\vec{x}_j) = (\theta^1_{S_j}, \ldots, \theta^{d-1}_{S_j})(\vec{x}_j)$$

*Further, let $\theta^d_{S_j}$ not introduce a new ordering on the statement instances. That equals to the following condition:*

$$\forall S_a, S_b : a, b \in \{1, \ldots, k\} : \forall \vec{x}_a, \vec{x}_b : Pre(\theta^d_{S_a})(\vec{x}_a) = Pre(\theta^d_{S_b})(\vec{x}_b) \Rightarrow \theta^d_{S_a}(\vec{x}_a) = \theta^d_{S_b}(\vec{x}_b)$$

*Define $(\theta_{S_j})'(\vec{x}_j) = (\theta^1_{S_j}, \ldots, \theta^{d-1}_{S_j}, 0, \theta^{d+1}_{S_j}, \ldots, \theta^p_{S_j})(\vec{x}_j)$ as the new scheduling function, where the dimension d is omitted.*

*Then $\theta$ and $\theta'$ are equivalent.*

*Proof.*

(a) **Execution Order:** The execution order of the statement instances does not change, when omitting schedule dimension $d$. $\forall S_a, S_b : a, b \in \{1, \ldots, k\} : \forall \vec{x}_a \in S_a, \vec{x}_b \in S_b :$

$$\theta_{S_a}(\vec{x}_a) \prec \theta_{S_b}(\vec{x}_b) \Leftrightarrow (\theta_{S_a})'(\vec{x}_a) \prec (\theta_{S_b})'(\vec{x}_b)$$

$\Rightarrow$:
Let $S_a$ and $S_b$ be two statements, $\vec{x}_a$ be a instance of $S_a$ and $\vec{x}_b$ be a instance of $S_b$, such that $\theta_{S_a}(\vec{x}_a) \prec \theta_{S_b}(\vec{x}_b)$. Obviously, the order on $Pre(\theta^d_{S_j})$ and $Pre((\theta^d_{S_j})')$ is equivalent. Now, consider only statement instances that are mapped to the same execution date $Pre(\theta^d_{S_a})(\vec{x}_a) = Pre(\theta^d_{S_b})(\vec{x}_b)$. Since dimension $d$ does not introduce a new ordering on those instances one has: $\theta^d_{S_a}(\vec{x}_a) = \theta^d_{S_b}(\vec{x}_b)$. Under that condition the lexicographic ordering of those statement instances relies on the next time dimensions, independent of the value of $\theta^d_{S_j}(\vec{x}_j)$. One can set this value to zero for all scheduling functions without changing the execution order, resulting in the schedule $\theta'$ with equivalent execution order.

$$(\theta^1_{S_j}, \ldots, \theta^{d-1}_{S_j}, 0, \theta^{d+1}_{S_j}, \ldots, \theta^p_{S_j})(\vec{x}_j) = (\theta_{S_j})'(\vec{x}_j)$$

$\Leftarrow$: Similar to $\Rightarrow$.

(b) **Dependency Direction:** Obviously, the direction of unsatisfied dependencies at dimension $d' \neq d$ is equivalent. Now, we show that the direction of the unsatisfied dependencies at dimensions $d$ does not change, when omitting schedule dimension $d$. $\forall D_{S_a,S_b} \in \mathcal{D}_d : a, b \in \{1, \ldots, k\} : \forall \vec{x}_a \to \vec{x}_b \in D_{S_a,S_b} :$

$$dir_{\theta^d_{S_a},\theta^d_{S_b}}(\vec{x}_a \to \vec{x}_b) = dir_{(\theta^d_{S_a})'(\theta^d_{S_b})'}(\vec{x}_a \to \vec{x}_b)$$

Let $D_{S_a,S_b} \in \mathcal{D}_d$ be an unsatisfied dependency between two statements $S_a$ and $S_b$, $\vec{x}_a$ be a instance of $S_a$ and $\vec{x}_b$ be a instance of $S_b$, such that $\vec{x}_a \rightarrow \vec{x}_b \in D_{S_a,S_b}$. The dependency $D_{S_a,S_b}$ is unsatisfied in all previous schedule dimensions. That means that the prefix execution dates of both dependent statement instances are equal $Pre(\theta^d_{S_a})(\vec{x}_a) = Pre(\theta^d_{S_b})(\vec{x}_b)$. (Otherwise the dependency is already satisfied.) Since dimension $d$ does not introduce a new ordering on both instances one has: $\theta^d_{S_a}(\vec{x}_a) = \theta^d_{S_b}(\vec{x}_b)$. Obviously, the condition for the direction of the dependency is equal for both scheduling functions $\theta^d$ and $(\theta^d)'$.

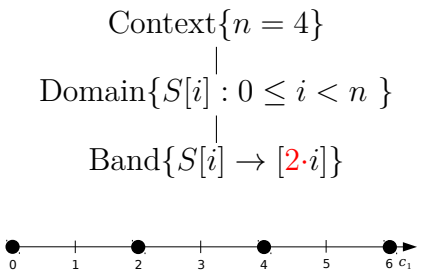
$$\theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b) = 0 = (\theta^d_{S_a})'(\vec{x}_a) - (\theta^d_{S_b})'(\vec{x}_b)$$

□

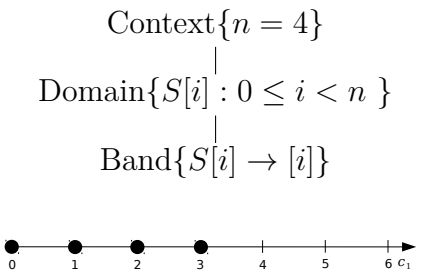
### 3.3.3 Common Factor

If the scheduling functions $\theta_{S_j}$ of all statements that are scheduled at the same schedule dimension $d$ at a band node have a common factor $f \in \mathbb{N} \setminus \{0\}$, the coefficients of the scheduling functions can be reduced by this factor.

The schedule $S[i] \rightarrow [2 \cdot i]$ in Example 3.9 *(a)* produces a grid of execution times, where not every iteration step contains a statement instance. It maps the statement instances $S[0], S[1], S[2]$ and $S[3]$ to the execution dates $0, 2, 4$ and 6. Since no other statement instances are scheduled at the time dates in between, the factor 2 is unnecessary and can be omitted. The new schedule is shown in part *(b)*.




*(a) original schedule tree and execution order.*




*(b) normalized schedule tree and execution order.*

**Example 3.9:** *Example with a common factor that can be reduced.*

**Theorem 3.10.** *Let $S_{1,\ldots,k}$ be all statements, that are scheduled at the current band node at schedule dimension d with the scheduling functions for each statement $S_{j\in\{1,\ldots,k\}}$:*

$$\theta^d_{S_j} : \mathbb{Z}^{dim(\vec{x}_j)} \rightarrow \mathbb{Z} : (\vec{x}_j) \rightarrow (\vec{i}_j, \vec{p}_j, c_j) \begin{pmatrix} \vec{x}_j \\ \vec{n}_j \\ \vec{1}_j \end{pmatrix}$$

*Define $(\theta^d_{S_j})'(\vec{x}_j) = f \cdot \theta^d_{S_j}(\vec{x}_j)$ as the scheduling function with common multiplication factor $f \in \mathbb{N} \setminus \{0\}$ for each statement $S_j$.*

*Then $\theta$ and $\theta'$ are equivalent.*

*Proof.*

(a) **Execution Order:** Obviously, the ordering at dimension $d' \neq d$ is equivalent. Now, we show that the execution order of the statement instances at dimension $d$ does not change, when multiplying with a common factor $f \in \mathbb{N} \setminus \{0\}$. $\forall S_a, S_b : a, b \in \{1, \ldots, k\} : \forall \vec{x}_a \in S_a, \vec{x}_b \in S_b :$

$$\theta^d_{S_a}(\vec{x}_a) < \theta^d_{S_b}(\vec{x}_b) \Leftrightarrow (\theta^d_{S_a})'(\vec{x}_a) < (\theta^d_{S_b})'(\vec{x}_b)$$

$\Rightarrow$:
Let $S_a$ and $S_b$ be two statements scheduled at the band node, $\vec{x}_a$ be a instance of $S_a$ and $\vec{x}_b$ be a instance of $S_b$, such that $\theta^d_{S_a}(\vec{x}_a) < \theta^d_{S_b}(\vec{x}_b)$. Obviously, this is true:

$$(\theta^d_{S_a})'(\vec{x}_a) = f \cdot \theta^d_{S_a}(\vec{x}_a) < f \cdot \theta^d_{S_b}(\vec{x}_b) = (\theta^d_{S_b})'(\vec{x}_b)$$

$\Leftarrow$: Similar to $\Rightarrow$.

(b) **Dependency Direction:** Obviously, the direction of unsatisfied dependencies at dimension $d' \neq d$ is equivalent. Now, we show that the direction of the unsatisfied dependencies at dimension $d$ does not change, when multiplying with a common factor $f \in \mathbb{N} \setminus \{0\}$. $\forall D_{S_a,S_b} \in \mathcal{D}_d : \forall \vec{x}_a \rightarrow \vec{x}_b \in D_{S_a,S_b} :$

$$dir_{\theta^d_{S_a},\theta^d_{S_b}}(\vec{x}_a \rightarrow \vec{x}_b) = dir_{(\theta^d_{S_a})'(\theta^d_{S_b})'}(\vec{x}_a \rightarrow \vec{x}_b)$$

Let $D_{S_a,S_b}$ be an unsatisfied dependency between two statements $S_a$ and $S_b$, $\vec{x}_a$ be a instance of $S_a$ and $\vec{x}_b$ be a instance of $S_b$, such that $\vec{x}_a \rightarrow \vec{x}_b \in D_{S_a,S_b}$.
Case 1: $\theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b) > 0$

$$\begin{aligned}
0 &< \theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b) \\
&\leq f \cdot \left(\theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b)\right) \\
&= f \cdot \theta^d_{S_a}(\vec{x}_a) - f \cdot \theta^d_{S_b}(\vec{x}_b) \\
&= (\theta^d_{S_a})'(\vec{x}_a) - (\theta^d_{S_b})'(\vec{x}_b)
\end{aligned}$$

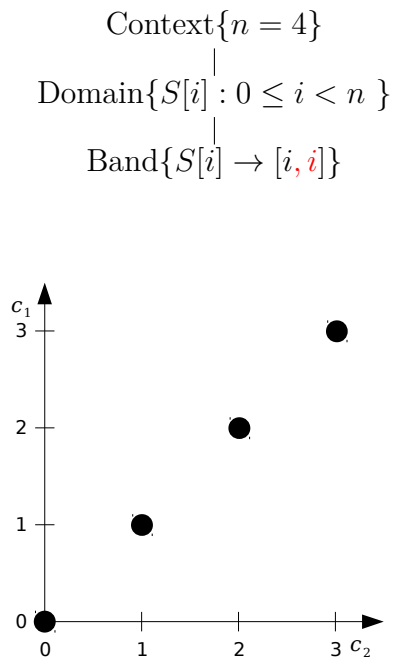
Case 2: $\theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b) = 0$
Obviously true.

□

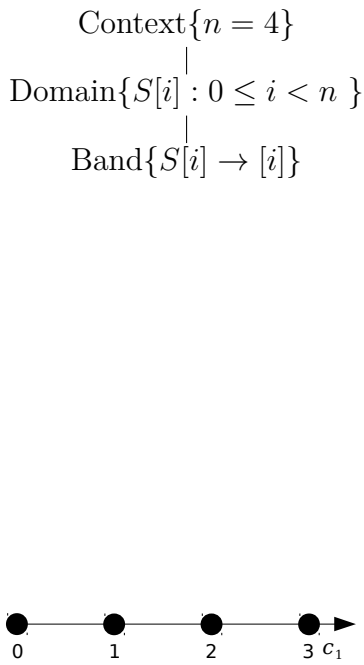
### 3.3.4 Injective Prefix

If the prefix schedule of dimension $d$ is injective for all statement instances that are scheduled in a subtree, one can cut off the subtree. An injective schedule assigns a globally unique execution date to each statement instance. Further schedule dimensions can only define the order on one single statement instance and therefore cannot change the execution order.

The schedule $S[i] \rightarrow [i, i]$ in Example 3.11 *(a)* has two time dimensions $c_1$ and $c_2$. The schedule for dimension $c_1$ is already injective $\theta^1_S(i) = i$. Therefore the second schedule dimension can be removed. The resulting schedule is shown in part *(b)*.




*(a) original schedule tree and execution order.*




*(b) normalized schedule tree and execution order.*

**Example 3.11:** *Example with an injective subtree that can be omitted.*

**Theorem 3.12.** *Let $S_{1,\dots,k}$ be all statements that are scheduled at the current schedule node at schedule dimension $d$ with the scheduling functions for each statement $S_{j\in\{1,\dots,k\}}$:*

$$\theta^d_{S_j} : \mathbb{Z}^{dim(\vec{x}_j)} \to \mathbb{Z} : (\vec{x}_j) \to (\vec{i}_j, \vec{p}_j, c_j) \begin{pmatrix} \vec{x}_j \\ \vec{n}_j \\ \vec{1}_j \end{pmatrix}$$

*Define the prefix schedule as:*

$$Pre(\theta^d_{S_j})(\vec{x}_j) = (\theta^1_{S_j}, \dots, \theta^{d-1}_{S_j})(\vec{x}_j)$$

*Further, let $Pre(\theta^d_{S_j})(\vec{x}_j)$ be injective. Define $(\theta_{S_j})'(\vec{x}_j) = Pre(\theta^d_{S_j})(\vec{x}_j)$ as the new schedule.*

*Then $\theta$ and $\theta'$ are equivalent.*

*Proof.*

(a) **Execution Order:** Obviously, the ordering at dimension $d'' < d$ is equivalent. Because $Pre(\theta^d_{S_j})$ is injective, it defines the total order on all elements of $\vec{x}_j$. Schedule dimensions $d' \geq d$ does not change the order on the execution date, due to the definition of lexicographical ordering.

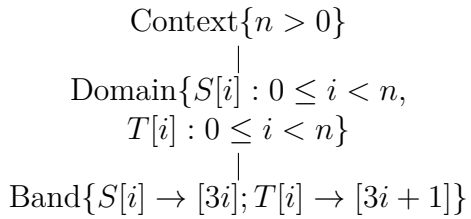
(b) **Dependency Direction:** Obviously, the direction of unsatisfied dependencies at dimension $d' \neq d$ is equivalent. Because $Pre(\theta^d_{S_j})$ is injective, it defines the total order on all elements of $\vec{x}_j$. There are no unsatisfied dependencies left for schedule dimensions $d' \geq d$.

□

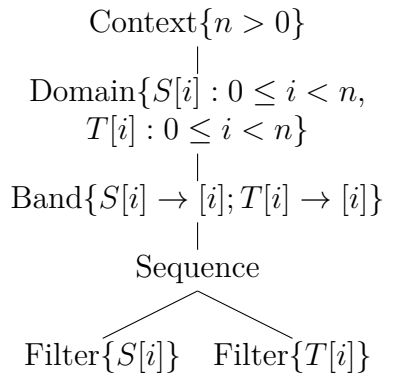
### 3.3.5 Hidden Sequence

Each schedule dimension of a band node is inspected whether it encodes a *hidden sequence* on the statements. In this context, *hidden* means that there is not a total ordering of the statements in this schedule dimension (which is detected during schedule tree construction), but instances of different statements are executed alternately, always with the same statement order. The hidden sequence can be extracted to a separate sequence node that orders the statements according to the value of their constant coefficients. The conditions that must hold in order to perform this normalization step are described in Theorem 3.14 *(a) - (d)*.

Consider the schedule in Example 3.13*(a)*. The number and the value of the iteration coefficients are identical for both statements $S$ and $T$. Furthermore, the parameter coefficients of both statements are zero and the constant coefficients are in the range $[0, 1]$, which is lower than 3 (the value of the iteration coefficients). Hence, the hidden sequence normalization can be applied and part *(b)* of Example 3.13 shows the resulting schedule tree.




*(a) original schedule tree and execution order.*




*(b) normalized schedule tree and execution order.*

**Example 3.13:** *Example with a hidden sequence that can be extracted to a seperate sequence node.*

**Theorem 3.14.** *Let $S_{1,\dots,k}$ be all statements that are scheduled at the current band node at schedule dimension $d$ with the scheduling functions $\theta^d_{S_j}$ for each statement $S_{j\in\{1,\dots,k\}}$:*

$$\theta^d_{S_j} : \mathbb{Z}^{dim(\vec{x}_j)} \to \mathbb{Z} : (\vec{x}_j) \to (\vec{i}_j, \vec{p}_j, c_j) \begin{pmatrix} \vec{x}_j \\ \vec{n}_j \\ 1_j \end{pmatrix}$$

*Further, let all scheduling functions $\theta^d_{S_j}$ fulfill the following conditions:*

*(a) The number of iteration coefficients in $\vec{i}_j$ that are unequal to zero are identical for all statements $S_j$.*

*(b) The values of all iteration coefficients in $\vec{i}_j$ that are unequal to zero are identical for all statements $S_j$.*

*(c) The parameter coefficients $\vec{p}_j$ are identical for all statements $S_j$.*

(d) *The range of the constant coefficients $c_j$ of all statements $S_j$ is lower than the value of the iteration coefficients.*

*Define* $(\theta^d_{S_j})'(\vec{x}_j) = (\vec{i}_j, \vec{p}_j, 0) \begin{pmatrix} \vec{x}_j \\ \vec{n}_j \\ 1_j \end{pmatrix}$ *as the new schedule dimension* $d$ *and* $(\theta_{S_j})'(\vec{x}_j) = (\theta^1_{S_j}, \ldots, \theta^{d-1}_{S_j}, (\theta^d_{S_j})', c_j, \theta^{d+1}_{S_j}, \ldots, \theta^p_{S_j})$ *as the new schedule for statement* $S_j$.
*Then* $\theta$ *and* $\theta'$ *are equivalent.*

*Proof.*

(a) **Execution Order:** The execution order of the statement instances does not change, when extracting a sequence from the schedule dimension $d$. $\forall S_a, S_b : a, b \in \{1, \ldots, k\} : \forall \vec{x}_a \in S_a, \vec{x}_b \in S_b :$

$$\theta_{S_a}(\vec{x}_a) \prec \theta_{S_b}(\vec{x}_b) \Leftrightarrow (\theta_{S_a})'(\vec{x}_a) \prec (\theta_{S_b})'(\vec{x}_b)$$

$\Rightarrow$:
Let $S_a$ and $S_b$ be two statements, $\vec{x}_a$ be an instance of $S_a$ and $\vec{x}_b$ be an instance of $S_b$, such that $\theta_{S_a}(\vec{x}_a) \prec \theta_{S_b}(\vec{x}_b)$. Obviously, the order on the first $d-1$ dimensions is identical. Now, consider only statement instances that are mapped to the same execution date $Pre(\theta^d_{S_a})(\vec{x}_a) = Pre(\theta^d_{S_b})(\vec{x}_b)$. Dimension $d$ does not change the order in which the instances of one statement are executed, because all instances are shifted simultaneously by the constant coefficient. The execution order between instances of different statements is dissolved in this dimension, but restored in the following sequence node. Note, that this is only legal due to the constraint that the stride (minimum number of time space iterations between two instances of the same statement) is greater than the range of the coefficients $c_j$ if the statements.
$\Leftarrow$: Similar to $\Rightarrow$.

(b) **Dependency Direction:** Obviously, the direction of unsatisfied dependencies at dimension $d' < d$ is identical. Now, we show that the direction of the unsatisfied dependencies at dimensions $d$ does not change, when extracting a sequence from the schedule dimension $d$.

Let $D_{S_a,S_b} \in \mathcal{D}_d$ be an unsatisfied dependency between two statements $S_a$ and $S_b$, $\vec{x}_a$ be a instance of $S_a$ and $\vec{x}_b$ be a instance of $S_b$, such that $\vec{x}_a \rightarrow \vec{x}_b \in D_{S_a,S_b}$. The dependency $D_{S_a,S_b}$ is unsatisfied in all previous schedule dimensions. That means that the prefix execution dates of both dependent statement instances are equal $Pre(\theta^d_{S_a})(\vec{x}_a) = Pre(\theta^d_{S_b})(\vec{x}_b)$. (Otherwise the dependency is already satisfied.)

Case 1: The dependency $D_{S_a,S_b}$ is only weakly solved in dimension $d$ of the original schedule $\theta$.

$$\theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b) = 0$$

In that case, this dependency must be weakly solved in both of the two new schedule dimensions $d$ and $d+1$ of the new schedule $\theta'$.

$$(\theta^d_{S_a})'(\vec{x}_a) - (\theta^d_{S_b})'(\vec{x}_b) = 0 \wedge (\theta^{d+1}_{S_a})'(\vec{x}_a) - (\theta^{d+1}_{S_b})'(\vec{x}_b) = 0$$

Obviously, this is true, since the new schedule has the same execution order of all statement instances.

Case 2: The dependency $D_{S_a,S_b}$ is strongly solved in dimension $d$ of the original schedule $\theta$.

$$\theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b) > 0$$

This dependency can either be strongly solved in dimension $d$ (band node) or in the newly introduced dimension $d+1$ (sequence node) of the new schedule $\theta'$.

$$\begin{aligned} & (\theta^d_{S_a})'(\vec{x}_a) - (\theta^d_{S_b})'(\vec{x}_b) > 0 \\ \vee \quad & (\theta^{d+1}_{S_a})'(\vec{x}_a) - (\theta^{d+1}_{S_b})'(\vec{x}_b) > 0 \end{aligned}$$

Case 2.1: $\theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b) \geq x$, where $x$ is the value of the iteration coefficient $\vec{i}_j$ of condition *(b)*. Further, according to condition *(d)* the following holds: $|c_b - c_a| < x$. The dependency is satisfied by the band node in dimension $d$ of the new schedule, because it is spanned between two statements instances of different loop iterations. The subtraction of the constant value $c_j$ from $\theta^d_{S_j}$ for each statement $S_j$ does not change the direction of the dependency, because

$$\begin{aligned} (\theta^d_{S_a})'(\vec{x}_a) - (\theta^d_{S_b})'(\vec{x}_b) = \\ (\theta^d_{S_a}(\vec{x}_a) - c_a) - (\theta^d_{S_b}(\vec{x}_b) - c_b) = \\ \underbrace{\theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b)}_{\geq x} + \underbrace{(c_b - c_a)}_{> -x} > 0 \end{aligned}$$

Case 2.2: $\theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b) < x$, where $x$ is the value of the iteration coefficients $\vec{i}_j$ of condition *(b)*. The execution date at dimension $d$ of a statement instance $\vec{x}_j$ can be expressed by a composition of three parts and is computed with

$$\theta^d_{S_j}(\vec{x}_j) = \underbrace{\vec{i}_j \bullet \vec{x}_j}_{A} + \underbrace{\vec{p}_j \bullet \vec{n}_j}_{B} + \underbrace{c_j}_{C}$$

According to *(c)*, the parameter coefficients are identical for all $S_j$ and, therefore, part $B$ of the execution date is identical, as well. Further, as the value of the iteration coefficients $\vec{i}_j$ is $x$, the value of part $A$ cannot differ between the statement instances $\vec{x}_a$ and $\vec{x}_b$, otherwise $\theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b)$ would be greater than $x$. As a result, $|c_a - c_b| > 0$ and the dependency is satisfied by the sequence node in dimension $d+1$ of the new schedule, because it is spanned between two statement instances of the same loop iteration.

$$\begin{aligned} (\theta^{d+1}_{S_a})'(\vec{x}_a) - (\theta^{d+1}_{S_b})'(\vec{x}_b) = \\ c_a - c_b > 0 \end{aligned}$$

In both cases, the direction of the dependency remains unchanged.

□

These normalization passes do not produce the minimal normalized schedule, but reduce the number of redundant coefficients and schedule dimensions drastically.

As mentioned in Section 2.2, schedule normalization can further improve the runtime of the schedule. As an example, consider a SCoP with only one statement $S$ and one dependency $D : S[i, j] \to S[i + 1, j]$. The schedule is given by $\theta_S(\vec{x}_S) = (i, -i, j)$. This scheduling function satisfies the dependency $D$ in the first schedule dimension.

$$\theta_S^1(i, j) - \theta_S^1(i + 1, j) = 1 > 0$$

It is neither possible to perform tiling on the first two nor the last two dimensions, because $\theta_S^2(i, j) = -i$ does not satisfy the dependency $D$ weakly. Since $\theta_S^2$ does not introduce a new ordering on the statement instances, it is omitted by one of the normalization passes and the resulting simplified schedule is $\theta'_S(i, j) = (i, j)$. The two remaining dimensions of the schedule can now be executed in tiles.

# Chapter 4

# Performance Prediction Function

The performance prediction function, introduced in this chapter, aims to predict the performance of different schedules of a given SCoP. With this prediction, the schedules can be classified into schedules producing well-performing codes and schedules producing codes with a lower performance. Using this classification, iterative optimization algorithms can detect good/bad schedules of a given SCoP without time critical runtime measurement of the transformed program.

The performance prediction function proposed in this master's thesis is a composition of different features based on the control and data-flow information provided by the polyhedron model. We propose features classifying the following performance aspects:

- **Parallelism.** CPUs with more than one core can benefit from parallel execution of loop iterations [18, 42]. The parallelization feature determines the amount of parallelism in the (transformed) program. All schedule dimensions that produce parallel loops are identified and ranked by the necessary amount of synchronization and thread management overhead.

- **Data Locality.** Good cache utilization and loop tiling can improve the performance of a program, especially if the performance of the program is limited by the memory bandwidth [18, 36]. The cache feature approximates the expected cache hit rate and the tiling feature inspects the schedule tree for possibilities to perform tiling.

- **Overhead.** Additional *if*-conditions inside a loop and computational expensive loop boundaries can arise of loop fusion and loop skewing. The conditional overhead feature classifies a schedule by the placement of *if*-guards in the generated code and the skewing overhead feature determines the skewing level of a schedule.

- **Out-of-Order Execution.** In modern CPUs the upcoming instructions are stored in an instruction buffer. If the processing units of the CPU stall, e.g. waiting for data from the cache, any other independent micro-operation is picked from the instruction buffer and executed out-of-order. The absence of loop-carried dependencies at the innermost loop allows out-of-order execution of instructions from different loop iterations.

The point of interest is comparing different schedules of the same SCoP. The execution time of the operations of the statements itself is not part of the prediction function, since it is equal for all different schedules of a SCoP. The different features are combined by machine learning techniques to a performance prediction function. The runtime of this prediction function should be less than the actual runtime of the generated program, otherwise the practical usage of such a function is very restricted.

**Polyite Toolchain.** The prediction function is designed to work within the Polyite toolchain. In Polyite, the input program is inspected by the SCoP detection of Polly [4, 59], which transforms each identified SCoP into the corresponding polyhedron model representation. This representation contains the domain of all statements, all memory references and the original schedule of the SCoP. Polly can regenerate executable code from the polyhedron model representation, whereby the schedule of the SCoP can be changed.

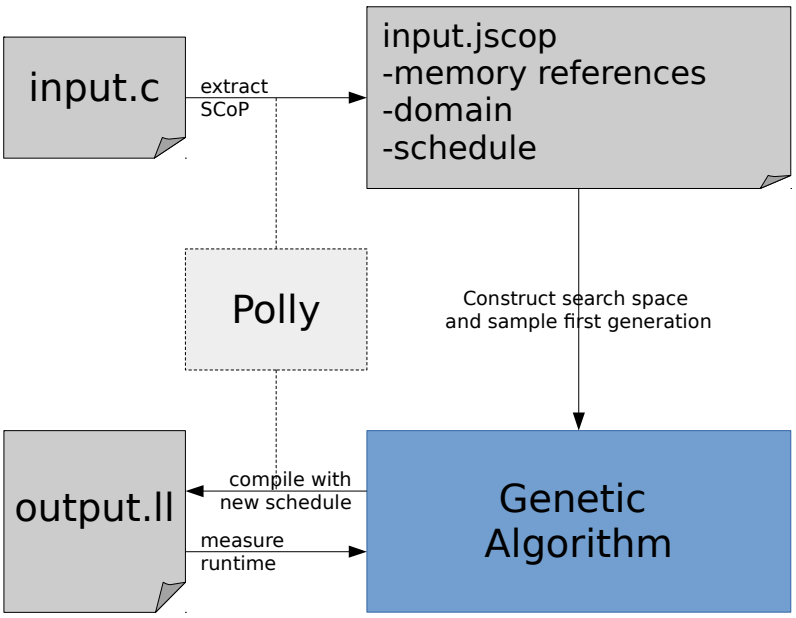



**Figure 4.1:** *Polyite toolchain.*

Using data flow analysis, provided by the *isl library* [63, 64], the dependencies between statement instances of the input program can be computed. Polyite executes a genetic algorithm that iteratively searches for good performing schedules. The initial population is sampled randomly from different regions of the search space of legal schedules. The construction of a region of the search space is described in Section 3.1. Using Polly, executable code is then generated for all schedules of the current generation and the measured runtime of the transformed program is used for schedule comparison. The best performing schedules are joined up to the population for the next iteration, until the configured number of populations is reached.

The machine learned prediction function of this master's thesis is intended to (partly) replace the expensive runtime measurement of the transformed programs.

## 4.1 Features

The feature vector of a schedule consists of the features described in the following section. Each feature is quantified by a normalized value between zero and one. Higher values refer to good performance, whereas lower values imply bad performance in the aspect of the feature.

### 4.1.1 Parallelization

The parallelization feature classifies a schedule on the basis of the possibility to compute loops around statements in parallel. This yields two questions: First, which schedule dimensions generate loops and second, which of these loops can be computed in parallel?

#### Parallel Loops

A schedule dimension $d$ generates a loop nest around a statement $S$ if the iteration coefficients of the scheduling function $\theta_S^d$ at dimension $d$ are *linearly independent* to all previous dimensions. Obviously, if the iteration coefficients are zero at a certain schedule dimension, all statement instances, which are assigned to equal execution dates by the prefix-schedule, have the same constant execution date at this dimension. Thus, this dimension does not produce a loop around these statement instances. Similar to this, schedule dimensions with linearly dependent iteration coefficients produce the same execution date, which depends on outer loop variables, for all statement instances with equal prefix execution dates. The mathematical description for a loop generating schedule dimension is given in Definition 4.2. Loops around different statements are fused if their prefix schedule is equal. In the schedule tree representation, naturally, only the band nodes can produce loops.

**Definition 4.2** (Loop Generating Dimension)**.** *Let $S$ be a statement with the scheduling function $\theta_S^d$ at dimension $d$:*

$$\theta_S^d : \mathbb{Z}^{dim(\vec{x}_S)} \to \mathbb{Z} : \vec{x}_S \to (\vec{i}_S^d, \vec{p}_S^d, c_S^d) \begin{pmatrix} \vec{x}_S \\ \vec{n} \\ 1 \end{pmatrix},$$

*where $\vec{i}_S^d$ are the iteration coefficients, $\vec{p}_S^d$ the parameter coefficients and $c_S^d$ the constant coefficients.*

*The schedule dimension $d$ generates a loop around the statement $S$ iff the iteration coefficient vector $\vec{i}_S^d$ is linearly independent of the iteration coefficient vectors $\vec{i}_S^1, \ldots \vec{i}_S^{d-1}$ of the previous dimensions.*

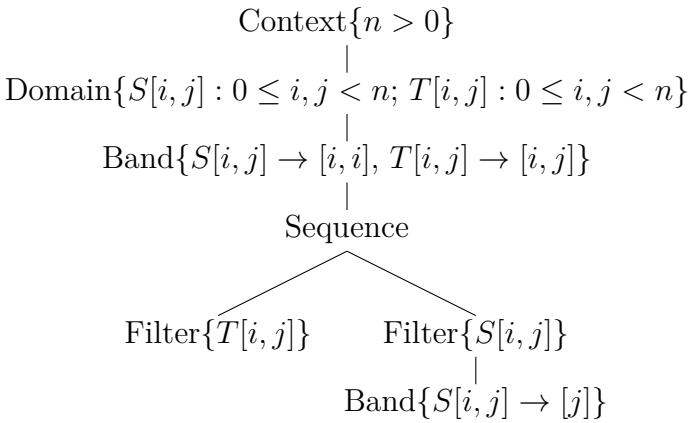
Consider the normalized schedule tree in Example 4.3. The scheduling functions, that can be extracted from the tree, are:

$$\theta_S(\vec{x}_S) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}, \theta_T(\vec{x}_T) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}$$

The statement instances are mapped to four-dimensional execution dates, but not every schedule dimension creates a loop around a statement. For example, schedule dimension 2 of the statement $S$ with the scheduling function

$$\theta_S^2(\vec{x}_S) \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} = (i)$$

has non-zero iteration coefficients, but they are linearly dependent of the coefficients of $\theta_S^1$. Therefore, this schedule dimension does not create a nested loop around $S$, it simply defines the iteration of the loop around statement $T$, in which the instances of $S$ are executed (compare the corresponding code given in Example 4.3). The generated code has an *if*-instruction that selects a certain iteration step of the $j1$-loop around the statement $T$, at which the instances of $S$ are computed.




```
for (i = 0; i < n; i++)            // generated by θ_S^1 and θ_T^1
  for (j1 = 0; j1 < n; j1++) {     // generated by θ_T^2
    T(i, j1);
    if (j1 == i)                   // θ_S^2 only defines position of S
      for (j2 = 0; j2 < n; j2++)   // generated by θ_S^3
        S(i, j2);
}
```

**Example 4.3:** *Example of loop generating dimensions: The given schedule maps the statement instances to a four-dimensional execution date, but not every dimension produces a loop.*

A schedule dimension can be computed in parallel, if no unsatisfied dependency is solved strongly in this dimension. If such a dependency exists, the different iterations at this dimension depend on each other and must be executed sequentially. The

schedule dimension that represents a sequence node in the schedule tree cannot be parallelized, since a sequence node describes a predefined sequential execution order of its children. The children of a set node can be processed in arbitrary order. This includes the possibility of parallel computation, but the level of parallelization is limited by the number of children, which is usually very small. In order to check the parallelism of a schedule dimension $d$ in a band node, all unsatisfied dependencies $\mathcal{D}_d$, that only affect the statements at this band node, must be considered. The formal conditions for parallel schedule dimensions at a band node are given in Definition 4.4.

**Definition 4.4** (Parallel Dimension)**.** *Let $S_1, \ldots, S_k$ be all statements that are scheduled at a band node with the scheduling functions $\theta^d_{S_1}, \ldots \theta^d_{S_k}$ for dimension $d$. Let $\mathcal{D}_d$ be the set of unsatisfied dependencies at dimension $d$, only containing dependencies between $S_1, \ldots, S_k$.*

*The schedule dimension $d$ of this subtree is parallel if*

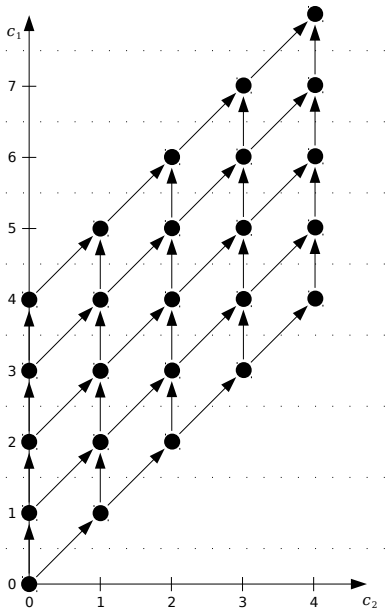
$$\forall D_{S_a,S_b} \in \mathcal{D}_d : \forall \vec{x}_a \to \vec{x}_b \in D_{S_a,S_b} : \theta^d_{S_a}(\vec{x}_a) - \theta^d_{S_b}(\vec{x}_b) = 0$$

$$\text{Context}\{n > 0\}$$
$$|$$
$$\text{Domain}\{S[i,j] : 0 \leq i < n \wedge 0 \leq j < n\}$$
$$|$$
$$\text{Band}\{S[i,j] \to [i+j, j]\}$$

```
for(i = 0; i < 2*n-1; i++)
  parfor(j = max(0, i-n+1);
         j < min(i+1, n);
         j++)
    S(i - j,j)
```

*(a) Above: schedule tree and program.*

*(b) Right: Execution dates with $n = 5$. Each point represents a statement instance and the arrows illustrate the dependencies between them. The inner loop $c_2$ can be parallelized.*




**Example 4.5:** *Example of loop parallelization: (a) the schedule and (b) execution order of the statement instances with all the dependencies.*

The schedule tree in Example 4.5 has only one band node and the scheduling function is $\theta_S(i,j) = (i+j, j)$. The given SCoP has the following two dependencies:

$$D_1 : S[i,j] \to S[i, j+1]$$
$$D_2 : S[i,j] \to S[i+1, j]$$

Obviously, the two schedule dimensions of $\theta_S$ produce loops. The set of unsatisfied dependencies $\mathcal{D}_1$ contains both dependencies at the first schedule dimension. Since

$D_1$ and $D_2$ are solved strongly, the first dimension cannot be parallelized.

$$D_1 : \theta_S^1(i,j) - \theta_S^1(i,j+1) = (i+j) - (i+j+1) = 1$$
$$D_2 : \theta_S^1(i,j) - \theta_S^1(i+1,j) = (i+j) - (i+1+j) = 1$$

For the second dimension, the set of unsatisfied dependencies $\mathcal{D}_2$ is empty and all iterations of this dimension can be computed in parallel. The execution dates of the statement instances are shown in Example 4.5 *(b)*. All instances at the same iteration of the outer loop $c_1$ can be computed in parallel, because no dependencies exists among them.

**Parallelization Feature**

The parallelization feature focuses on the parallelism of loops. Now, the task is to identify the schedule dimensions that produce parallel loops. Both properties can be verified by a depth-first search on the schedule tree. The search starts at the root node and processes the schedule tree in pre-order. Band nodes with more than one schedule dimension are processed dimension-by-dimension. The complete algorithm is outlined in Figure 4.6.

It starts with the root node $N$, the set of linearly dependent iteration vectors $\mathcal{L}_{S_j}$ for each statement $S_j$, which is initially empty, and the set of unsatisfied dependencies $\mathcal{D}$ containing all dependencies of the SCoP. Context, domain and mark nodes are skipped and the algorithm is called recursively on their children with unaltered parameters. Set nodes and sequence nodes are processed similarly, with the difference that all dependencies that are solved strongly at these nodes are removed from the set of unsatisfied dependencies $\mathcal{D}$. At filter nodes, only the dependencies between the filtered statements are passed to the recursive call on their child nodes.

For each schedule dimension $d$ at a band node the set of strongly solved dependencies $\mathcal{S}_d$ is computed in step *(1)*. If $\mathcal{S}_d$ is empty, this dimension is marked as parallel *(2)*. After that, the set of unsatisfied dependencies $\mathcal{D}$ is updated *(3)*. Finally *(4)*, the algorithm checks whether dimension $d$ produces a loop. This is the case if the iteration vector $\vec{i}_{S_k}^d$ of any statement $S_k$ scheduled at the band node is linearly independent of the previous iteration vectors.

The marking of parallel and loop generating dimensions can be done in one single depth-first search as depicted in Figure 4.6, but during the schedule tree construction by Ganser all parallel dimensions are already marked for code generation. Therefore, only the loop generation property must be inspected by a separate run of a depth-first search.

After identification of all schedule dimensions that produce parallel loops, the level of parallelization must be rated. Parallel execution of the outermost loop of a loop nest only needs one synchronization step after the loop and is hence better than inner parallel loops, which need synchronizations after each iteration of the outer loop. Additionally, more thread management overhead results from inner parallelism. Consequently, Polly only generates parallel loops for the outermost parallel dimensions. These loops can be easily extracted from the marked schedule tree.

```
Data: tree node: 𝒩, set of linearly dependent iteration vectors L_{S_j} for each
      statement S_j: ℒ, set of unsatisifed dependencies: 𝒟
Result: marked schedule node 𝒩 (loop generation and parallel)
PARLOOP(𝒩, ℒ, 𝒟)
switch 𝒩 do
    case does not exist do
        done;
    end
    case context, domain or mark node do
        PARLOOP(𝒩.child, ℒ, 𝒟);
    end
    case sequence or set node do
        (1) Compute 𝒮 - the set of strongly solved dependencies at that node;
        (2) foreach 𝒞 of 𝒩.children do
            PARLOOP(𝒞, ℒ, 𝒟 \ 𝒮);
        end
    end
    case filter node do
        (1) Compute ℛ - the set of dependencies of 𝒟 only affecting the
         filtered statements;
        (2) PARLOOP(𝒩.child, ℒ, ℛ);
    end
    case band node do
        foreach schedule dimension d do
            (1) Compute 𝒮_d - the set of strongly solved dependencies;
            (2) if 𝒮_d = ∅ then
                mark dimension d as parallel
            end
            (3) 𝒟 ← 𝒟 \ 𝒮_d;
            (4) foreach statement S_k do
                if iteration vector i⃗^d_{S_k} is non-zero and independent to L_{S_k} then
                    (a) mark dimension d as loop generating;
                    (b) add i⃗^d_{S_k} to L_{S_k} and update ℒ;
                end
            end
        end
        PARLOOP(𝒩.child, LDS, 𝒟);
    end
end
```

**Figure 4.6:** *Algorithm for marking all schedule dimensions that produce loops and that are parallel. The initial parameters are the root node of the schedule tree, an empty set $L_{S_j}$ for each statement $S_j$ and the full set of dependencies of the SCoP.*

The overhead of a parallel loop is defined as the number of loop executions multiplied by a constant parallelization cost $C$. Parallel loops can only be generated by band nodes and the calculation of the overhead value at a band node is shown in Definition 4.7. The prefix schedule of the parallel schedule dimension $d$ provides a polytope containing all execution dates where the parallel loop at dimension $d$ is executed.

**Definition 4.7** (Overhead of parallel loop)**.** *Let $S_1, \ldots, S_k$ be all statements that are scheduled at a band node at schedule dimension $d$ with the scheduling functions $\theta^d_{S_1}, \ldots, \theta^d_{S_k}$. Further, let dimension $d$ generate a parallel loop.*
*Define the prefix schedule as:*

$$Pre(\theta^d_{S_j})(\vec{x}_j) = (\theta^1_{S_j}, \ldots, \theta^{d-1}_{S_j})(\vec{x}_j)$$

*We define the overhead for the parallel loop produced by schedule dimension $d$ as*

$$Oh(d) = C \cdot \underbrace{\Big| \bigcup_{S=S_1}^{S_k} \Big( \bigcup_{\vec{x}_S \in \mathbb{D}_S} Pre(\theta^d_S)(\vec{x}_S) \Big) \Big|}_{\#\ \textit{loop executions}}$$

*where $C$ is the constant cost of synchronization and thread management for one parallel loop.*

```
for(i = 0; i < n; i++)
  parfor(j = 0; j < n; j++)
    S(i,j)
```

*(a) Without tiling the parallelization overhead of the $j$ loop is $C \cdot n$.*

```
for(it = 0; it < n; it += 32)
  for(jt = 0; jt < n; jt += 32)
    for(i = 0; i < 32; i++)
      parfor(j = 0; j < 32; j++)
        S(it + i,jt + j)
```

*(b) With tiling applied the outermost parallel loop is the $j$ loop and the overhead is $C \cdot \frac{n^2}{32}$.*

**Example 4.8:** *Example of loop parallelization with tiles.*

There are two special cases: First, if neither a band node nor its subtree produce any parallel loops, the calculated overhead is equal to zero. Second, in case the band node which produces a parallel loop will later be tiled by Polly and the outermost parallel loop is not the outermost loop of this band node, the number of loop executions increases because the loops are reordered by the tiling transformation. Polly only tiles the schedule dimensions at a tilable band node if the band node is a leaf of the schedule tree. Example 4.8 *(a)* shows a loop nest where the outermost parallel loop is the $j$ loop. The overhead of this loop nest without tiling is $C \cdot n$. After tiling of both loops *(b)*, the parallel loop is moved inside the loop nest and is executed $\frac{n^2}{32}$ times. To overcome this issue, one could apply all tiling transformations on the schedule tree as Polly would do. But with highly skewed loops, the cardinality computation to obtain the number of loop executions with Barvinok's algorithm [13] becomes computationally expensive. A simpler, but less exact approach is to divide the total number of statement instances - scheduled at the tilable band node

- by the number of statement instances executed inside the parallel part of the tiles (in Example 4.8 the tile size is 32). We assumed that each iteration inside a tile executes one statement instance. This calculation is exact as long as only non-skewed schedules are used. The tiles at the edge regions of skewed schedules are neglected and the actual execution number of the parallel loop is slightly higher.

The parallelization features needs to calculate the parallelization overhead of the total schedule tree and weight the different parallel loops according to the number of statement instances executed inside the body. Therefore, the total parallelization cost is determined by a depth-first search. Sequence and set nodes sum up the cost of their children $C_1, \ldots, C_k$.

$$Cost(N) = Cost(C_1) + \ldots + Cost(C_k)$$

At a band node, the parallelization overhead is calculated like in Definition 4.7 if none of the two special cases applies. Otherwise, it is computed as described above. Additionally, the total number of sequential loop nest iterations is added to the overhead.

$$Cost(N) = \begin{cases} Cost(N.child) & \text{no dimension at N produces a parallel loop} \\ Oh(d) + \frac{Inst(N)}{cores} & \text{dimension } d \text{ produces a parallel loop} \\ 0 + Inst(N) & \text{special case 1} \\ C \cdot \frac{Inst(N)}{Tile} + \frac{Inst(N)}{cores} & \text{special case 2} \end{cases}$$

where $Inst(N)$ is the total number of statement instances scheduled at the node $N$, $cores$ is the number of available processor cores and $Tile$ is the number of statement instances inside the parallel part of the tiles.

If the depth-first-search reaches a non-parallel leaf node, all statement instances in this subtree are executed sequentially and the parallelization cost is equal to the number of statement instances $It(N)$. All other node types are skipped and the parallelization cost of the child is returned.

$$Cost(N) = \begin{cases} It(N) & \text{if } N \text{ is leaf node} \\ Cost(N.child) & otherwise \end{cases}$$

The value of the parallelization feature is computed as shown in Definition 4.9. If all loops are processed sequentially the parallelization cost of the schedule tree is equal to the number of statement instances of the SCoP, which yields a zero parallelization feature value. If the cost exceeds the number of statement instances the schedule receives a zero parallelization value as well. A well parallelized program has less overhead cost and the result is close to one.

**Definition 4.9.** *Let $T$ be a schedule tree and $I$ be the total number of statement instances scheduled by this schedule tree. Further, let $Cost(T)$ be the parallelization cost of $T$.*

*We define the parallelization feature value as*

$$F_{par}(T) = \begin{cases} 1 - \frac{Cost(T)}{I} & if \ \frac{Cost(T)}{I} \leq 1 \\ 0 & otherwise \end{cases}$$

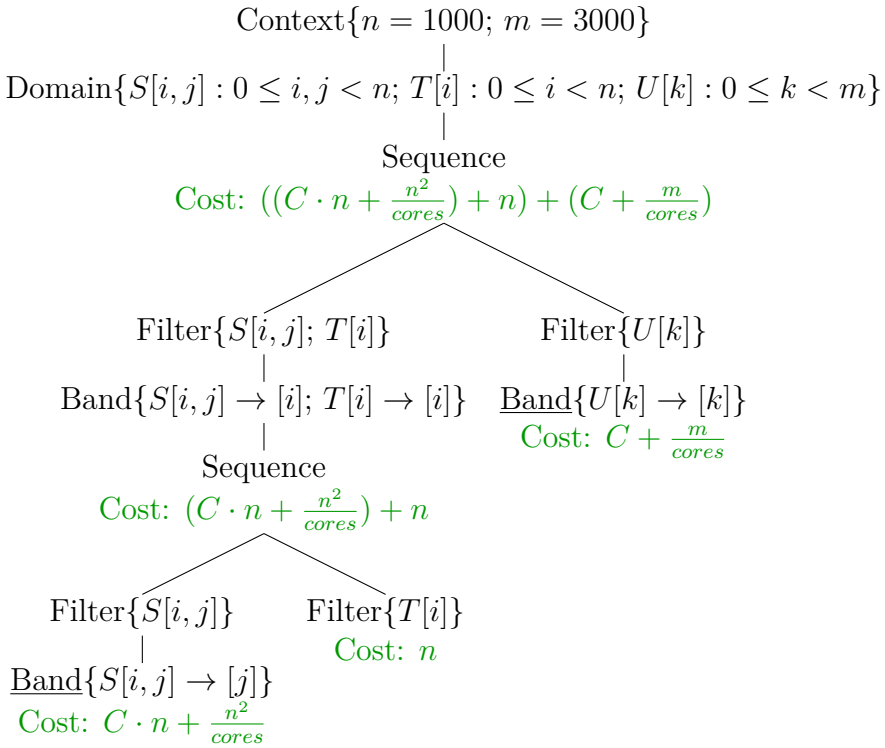
In Example 4.10, the $j$ and $k$ loops can be computed in parallel. Note, that the parameters are $n = 1000$, $m = 3000$, $C = 200$ and the number of processor cores is 10. The overhead cost of the schedule tree is

$$((C \cdot n + \frac{n^2}{cores}) + n) + (C + \frac{m}{cores}) = 301500$$

and the SCoP has a total of

$$n^2 + n + m = 1004000$$

statement instances. The parallelization value for this example is 0.700, which correlates to a good parallelization level.




```
for(i = 0; i < n; i++) {
  parfor(j = 0; j < n; j++)
    S(i,j)
  T(i)
}
parfor(k = 0; k < m; k++)
  U(i)
```

**Example 4.10:** *Example of parallelization overhead cost. The overhead cost is denoted at each node of the schedule tree.*

### 4.1.2 Cache Behavior

This feature aims to approximate how much cached data is reused by a schedule. The reuse of cached memory cells depends on the number of different memory accesses between two references to the same cell. If the number of accesses is too high, the

data will likely not be present in the cache at the second reference and must be transfered from a slower memory level. Tilable schedule dimensions can increase data locality by reducing the average number of memory accesses in between.

### Cache

The memory hierarchy of a computer architecture orders the memory levels by increasing latency and storage size. The pyramid in Figure 4.11 shows the four main memory levels [62]: internal memory (like CPU registers and cache), Random Access Memory (RAM), mass storage and off-line bulk storage. For computational purpose, only the first two levels, which are close to the processor, are relevant.

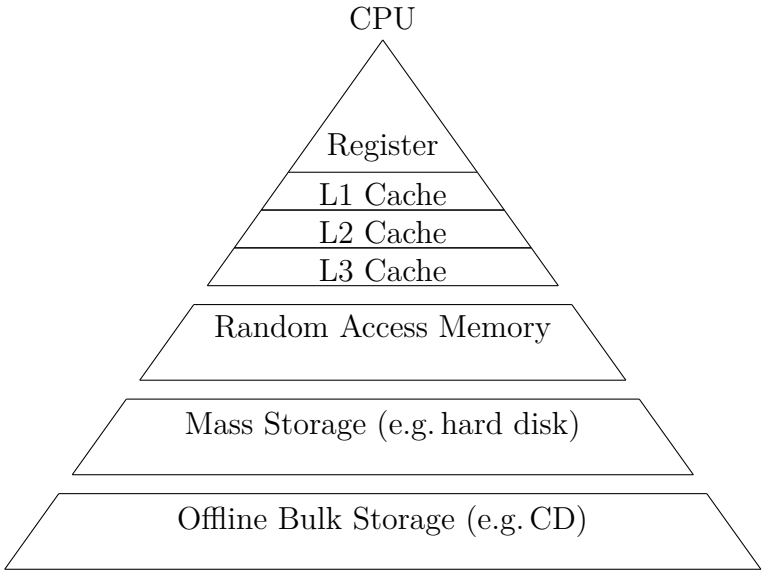



**Figure 4.11:** *Pyramid of computer memory hierarchy.*

Modern CPUs have several cache levels with increasing sizes and access delays. The purpose of these caches is to store recently used or prefetched data and instructions such that they can be (re-)accessed fast. The data is transfered and stored in blocks with a fixed size, the so-called *cache lines*. For each memory reference, it is checked whether the corresponding data is located in the cache. If the requested data is found, one speaks of a *cache hit*. The cached copy may directly be used for computation. In case of a *cache miss*, the cache line containing the memory cell must be transfered from slower RAM to the processor, which causes a delay.

The caches can be organized in three different ways: *direct mapped*, *n-way set associative* or *fully associative*. In direct mapped caches each block of the main memory has its predefined location in the cache, which can be calculated by the memory address of the block. It is easy to check whether the cache contains a given data block. The $n$-way set associative caches are partitioned into sets and each set can hold $n$ different cache lines. The set number is derived from the memory address and the memory bank inside the set is free to choose, depending on the cache replacement policy. Fully associative caches are equal to $n$-way caches with only one set. Each memory address can reside anywhere in the cache. Modern high-performance processors, like the Intel Xeon, whose cache configuration is shown in

Table A.1, use $n$-way caches. Since the data of a program might be stored at different locations each time the program is executed, it is impossible to model the correct $n$-way associative cache. Instead, we assume that the cache is fully associative and uses the *least recently used (LRU)* cache replacement policy.

Modern CPU architectures permit to queue one data request and one instruction request in parallel. Therefore, cache levels must have separate areas for data and instructions. They are implemented as *banked*. On the other hand, *unified* caches mix data and instruction blocks and process the requests sequentially. Since all requests of the processor always reach the L1 cache first, the L1 cache is usually banked. Nested loops typically have a low number of different instructions that tend to fit into the L1 instruction cache. Our cache feature focuses on the data cache.

Multi-level caches can have different inclusion strategies between the cache levels. An *inclusive* cache level stores a copy of all blocks located in the upper cache level. For instance, if the L2 cache is inclusive, the data contained by the L1 cache is a subset of the data in the L2 cache. An *exclusive* cache level only contains blocks that are not part of the upper cache levels. This increases the total usable cache size, but it is more expensive to check whether a block is present in the cache at any level. The *Non-Inclusive Non-Exclusive (NINE)* cache strategy enforces none of the aforementioned policies. Our cache feature approximates the total cache hit rate, whether data is fetched from any cache level or must be transfered from the slower main memory. We assume that all cache levels are inclusive and therefore that the total cache size is at least as large as the size of the *Last Level Cache (LLC)*.

In multi core CPUs a cache can be *privately* attached to one CPU core or be *shared* among all cores of the CPU. Since one cannot be aware of which loop iterations of a parallel loop are executed at the other CPU cores that have access to the same shared L3 cache, it is hard to determine the exact number of different memory references between two references to the same cell. Therefore, we assume that the total capacity of the cache is equally distributed among all cores, i.e., the total L3 cache size is divided by the number of CPU cores. Again, the assumption introduces an inaccuracy to the cache hit rate approximation.

### Cache Feature

The caches are transparent for the program, hence one cannot allocate memory at a specific cache level. But if there are few different memory accesses between two references to the same cell the data will likely be present in the cache. The cache feature actually tries to approximate the cache hit rate, which is a value between zero and one.

At SCoP detection, Polly constructs the *read* and *write access relations*, which are given in the following two Definitions 4.12 and 4.13. With these access relations, all pairs of statement instances, that accesses the same memory cell and are executed in a specific order, can be computed using *dependence analysis*. In theory, there are four different dependence types: flow (RAW: Read-After-Write), anti (WAR: Write-After-Read), output (WAW: Write-After-Write) and input (RAR: Read-After-Read) [25, 28, 40]. For the legality of the schedule only the dependence types flow, anti

**Definition 4.12** (Read Access Relation)**.** *The read access relation $R$ is a binary relation that maps each statement instance to each data element read by the statement instance.*

$$R = \{\vec{x}_{S_k} \to m \mid S_k \text{ statement of the SCoP} \wedge \\ \vec{x}_{S_k} \in \mathbb{D}_{S_k} \wedge \\ m \text{ memory cell read by } \vec{x}_{S_k}\}$$

**Definition 4.13** (Write Access Relation)**.** *The write access relation $W$ is a binary relation that maps each statement instance to each data element written by the statement instance.*

$$W = \{\vec{x}_{S_k} \to m \mid S_k \text{ statement of the SCoP} \wedge \\ \vec{x}_{S_k} \in \mathbb{D}_{S_k} \wedge \\ m \text{ memory cell written by } \vec{x}_{S_k}\}$$

and output are needed, since two statement instances cannot influence each other by reading from the same memory cell. In order to analyze the cache reuse, all four dependence types are inspected, because every memory reference passes the cache and, therefore, consecutive read accesses on the same memory cell influence the cache behavior, as well. The dependency relations can be computed similarly as described by Verdoolaege [64]. Let us consider the read-after-write dependency as an example. First, the pairs of statement instances, one performing a write and one performing a read on the same memory cell, are obtained by

$$R^{-1} \circ W \quad (\circ \text{ denotes the composition of binary relations.}^{1}).$$

The dependency relation should only hold those pairs of statement instances, where the first instance is executed before the second one. Therefore, the relation is intersected with the ordering relation $<_\theta$ of the schedule $\theta$

$$(R^{-1} \circ W) \cap <_\theta .$$

The ordering relation can be computed from the schedule $\theta$ and is given by

$$<_\theta = \{\vec{x}_{S_k} \to \vec{x}_{S_l} \mid S_k, S_l \text{ statements of the SCoP} \wedge \\ \vec{x}_{S_k} \in \mathbb{D}_{S_k} \wedge \vec{x}_{S_l} \in \mathbb{D}_{S_l} \wedge \\ \theta_{S_k}(\vec{x}_{S_k}) \prec \theta_{S_l}(\vec{x}_{S_l})\}.$$

The other dependency relation types are computed similarly by

$$(W^{-1} \circ W) \cap <_\theta \text{ (WAW)}, \\ (R^{-1} \circ R) \cap <_\theta \text{ (RAR)}, \\ (W^{-1} \circ R) \cap <_\theta \text{ (WAR)}.$$

The resulting dependency relations might still contain pairs of statement instances that are not relevant for cache reuse analysis. Consider the code and the access

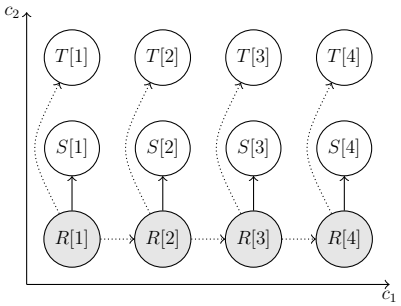
[1] $B \circ A = \{i \to j \mid \exists k : i \to k \in A \wedge k \to j \in B\}$

relations of Example 4.14 *(a)*. The read-after-write dependency relation, depicted in part *(c)*, can be expressed as the union of these three relations: $\{R[i] \to S[i] : 1 \leq i < 5\}$, $\{R[i] \to T[i] : 1 \leq i < 5\}$ and $\{R[i] \to R[i+1] : 1 \leq i < 4\}$. Between each two depending statement instances of the last two relations (dotted arrows), other statement instances which references the same memory cell are executed.

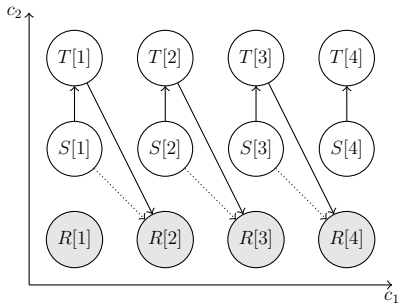
$$
\begin{aligned}
R = &\{S[i] \to A[i] : 1 \leq i < 5\} \cup \\
&\{T[i] \to A[i] : 1 \leq i < 5\} \cup \\
&\{R[i] \to A[i-1] : 1 \leq i < 5\} \\
W = &\{R[i] \to A[i] : 1 \leq i < 5\}
\end{aligned}
$$

```
for(i = 1; i < 5; i++) {
  A[i] = A[i - 1]    // R
  B[i] = A[i] + 1    // S
  C[i] = A[i] * 2    // T
}
```

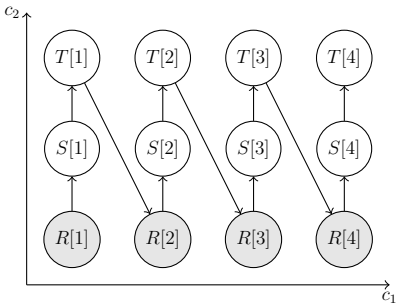
*(a) read and write access relations and code.*




*(c) RAW dependency relation.*




*(b) RAR dependency relation.*



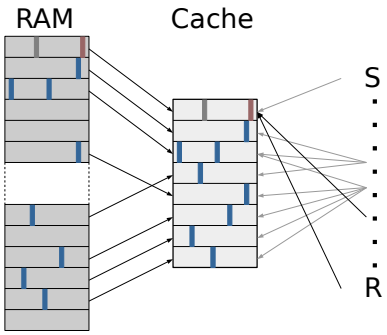

*(d) memory-based dataflow dependency for cache analysis.*

**Example 4.14:** *Dependency example: (a) code and access relations, (b) read-after-read dependency relation, (c) read-after-write dependency relation and (d) memory-based dataflow analysis. The example does not have any write-after-read nor write-after-write dependencies. The filled statement instances in (b)-(d) are read and write accesses to a memory cell and the unfilled are only read accesses.*

With the purpose of analyzing the cache reuse we actually need all pairs of statement instances, where the source is executed directly before the sink of the relation without an intermediate memory reference to the same cell. For this reason *dataflow analysis* is used. Conceptually, dataflow analysis removes all dependencies with an intermediate memory access to the same cell. The removed dependencies are called to be *killed* [64]. There are two kinds of dataflow analysis. *Memory-based analysis* simply removes all dependencies with an intermediate memory access independent whether it is a read or a write access, whereas *value-based analysis* deletes dependencies which reference a different value of the memory cell (in case the intermediate memory access is a write). The cache reuse feature uses the memory-based dataflow

analysis. Consequently, each statement instance with a read or write access to a memory cell is the sink of only one dependency relation regarding this memory cell. The union of the remaining dependencies build a partial order on the statement instances. In Example 4.14, the dotted dependency arrows of the read-after-read *(b)* and read-after-write *(c)* dependency relations are *killed* by the dataflow analysis. Part *(d)* shows the union of the remaining dependency relations used by the cache reuse analysis.

Since the data is transmitted and stored in cache lines, the target cache line (referenced by the sink of a dependency instance) might be accessed in between the two depending instances, as well. Thus, the cache reuse feature must determine the last reference to the target cache line before the sink of the dependency instance is executed. If the number of different cache lines, that are accessed in between, fit into the cache, a cache hit is expected. In Example 4.15, the two statement instances $S$ and $R$ are in dependence and both access the same data cell of the first cache line. There is a second reference to another data element of the same cache line shortly before the execution of the sink statement instance $R$. In that case, the target cache line is reloaded or marked as shortly used (the cache will first replace all the other cache lines according to the $LRU$ replacement policy) and only the number of different cache lines referenced after the second access are required for the cache hit rate computation.




**Example 4.15:** *$S$ and $R$ are in dependence but there is one reference to the same cache line in between.*

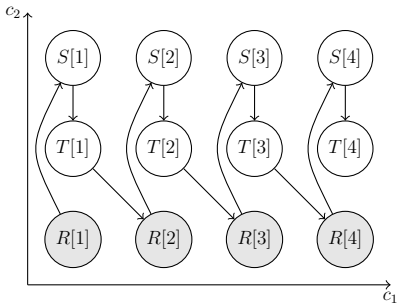
There are two limitations: First, dataflow analysis on a loop program is related to finding an integer solution for a system of equations which is NP-complete and inefficient with highly skewed schedules [56]. Second, it is infeasible to test every dependency instance in a reasonable time.

The first problem can be softened by computing the dataflow analysis on the original schedule and, then, using the new schedule for cache hit approximation. Dataflow analysis is practicable on the original schedule, because it is also needed for the construction of the search space. Since the new schedule must preserve the semantics of the program, only the direction of read-after-read dependencies can be inverted by the new schedule. Using the sink of a dependency instance for cache hit approximation is still accurate if the sink is part of a read-after-write dependency relation, too. Example 4.16 *(a)* shows a legally transformed program of Example 4.14, where the statements $T$ and $S$ are interchanged. In part *(b)*

the dataflow dependency relations are shown with the execution dates of the new schedule. Taking all sink instances into account, one has exactly the same depending statement instances as with the original schedule.

```
for(i = 0; i < 4; i++) {
  A[i] = A[i - 1]    // R
  C[i] = A[i] * 2    // T
  B[i] = A[i] + 1    // S
}
```

*(a) transformed code.*




*(b) memory-based dataflow dependency of the original schedule with the new execution order.*

**Example 4.16:** *Transformation of Example 4.14.*

Since it is infeasible to check whether the referenced memory cell of each dependency instance is present in the cache, the feature approximates the cache hit rate on the basis of a limited number of dependency instances. First, the dependency relation is split into several relations with the same source and sink statements, that can be expressed by a *basic map* in the *isl library* [63]. Additionally, the involved memory cell of this dependency is annotated along with the dependency relation. Test dependency instances can now be sampled from the sink space of each relation, which is a subset of the iteration domain of a statement. Dimension-by-dimension this statement iteration polytope is divided into slides of a fixed size, which is defined by the number of samples that should be generated. From each slide, the center hyperplane is chosen and weighted by the number of statement instances of the slide, until a single point is obtained. All memory references of the sampled dependency instances are tested for a cache hit and the overall cache hit rate of the dependency relation can be computed as a weighted average over the cache samples.

The algorithm in Figure 4.17 approximates the cache hit rate for one dependency $D$ of the SCoP. A set of dependency instances (cache samples) is computed in the first step *(1)*. For all of these cache samples the memory cells, accessed by both statement instances, are extracted. Then, for each of these memory cells $m$, the algorithm builds the time space interval $I_m$, which spans from the last access to the target cache line of $m$ to the execution time of the sink statement instance. At this step, the tiling transformation, applied by Polly, must be considered, because the statement instances are reordered by the tiling transformation. If the number of different cache lines accessed within this interval is lower than the total number of cache lines that fit into the *Last Level Cache (LLC)*, the memory reference to $m$ at the sink of the dependency instance will result in a cache hit.
The cache reuse feature approximates the cache hit rate for all dependency relations of the SCoP and weights the result by the number of dependency instances. The formal description is given in Definition 4.18.

**Data:** dependency of the input program $D$, schedule $\theta$ and memory reference information $Mem(x_j)$
**Result:** approximated cache hit rate of all instances of $D$
CACHEHITRATE($D$, $\theta$, $Mem(x_j)$)
(1) Compute a set of dependency instances $D_T \subseteq D$;
(2) **foreach** $x_a \to x_b \in D_T$ **do**
  **foreach** $m \in Mem(x_a) \cap Mem(x_b)$ **do**
    ($a$) Compute the interval $I_m$ in the time space between the last reference to the cache line containing $m$ and $\theta(x_b)$;
    ($b$) Compute the number of diverse cache lines $L$ referenced within $I_m$;
    ($c$) **if** $L < LLC$ *cache lines* **then**
      cache hit of memory cell $m$ for this dependency $D$;
    **end**
  **end**
**end**
(3) return the average cache hit rate over all tested dependency instances;

**Figure 4.17:** *Algorithm for calculating the cache hit rate for one dependency relation.*

**Definition 4.18** (Cache reuse feature)**.** *Let $D_1, \ldots, D_k$ be all dependency relations of a SCoP, that are computed for the cache reuse analysis. Further, let $p_i$ be the approximated cache hit rate for the dependency relation $D_i$.*

*We define the cache reuse feature value as*

$$F_{cache}(T) = \sum_{i=1}^{k} \frac{|D_i|}{\sum_{i=1}^{k} |Di|} \cdot p_i$$

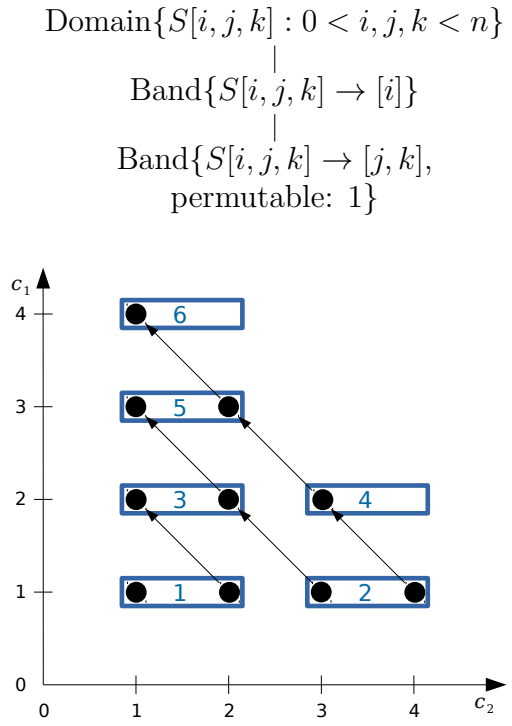
This approach for cache hit approximation has several limitations and inaccuracies:

- The parameters, especially the loop bounds, must be known at analysis time, as well as the overall cache size of the target system.
- The cache hit rate is only calculated for several dependency instances and then extrapolated to the whole dependency relation, which introduces inaccuracy.
- Edge regions of the time space polytope can achieve a 100% cache hit rate, but are not inspected separately by the cache hit rate approximation.
- In contrast to Caşcaval et al. [20], only the interior cache hit rate can be computed. The first reference to a memory cell is neither treated as a cache hit nor a cache miss, because the initial cache state is unknown.
- Every rerun of the generated loop program can produce a different memory cell to cache line mapping, depending on the kernel's memory placement in the RAM.
- In order to count the number of distinct cache line references in the interval $I_m$, we use Barvinok's algorithm. This can result in a huge execution time of the feature calculation.

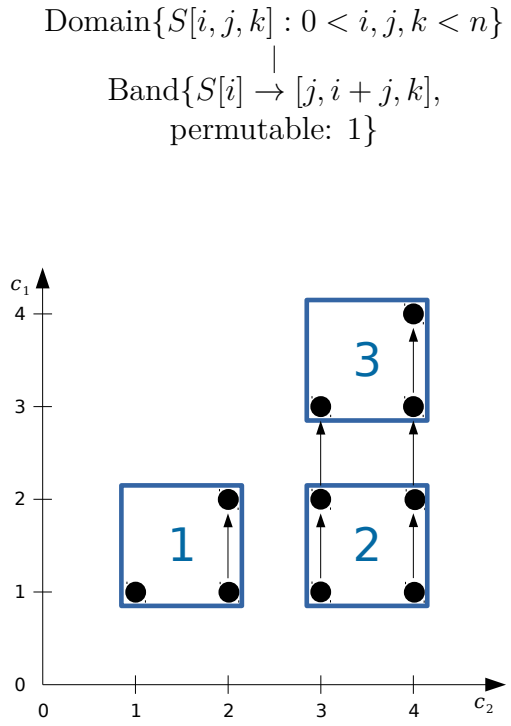
### 4.1.3 Tiling

A more efficient way to identify schedules with a good cache hit rate is inspecting the schedule tree regarding for possibilities of tiling. Typically, consecutive loop iterations reference the same cache line or even the same memory cell. Schedule dimensions are blocked and reordered by the tiling transformation, such that the number of intermediate memory references between depending statement instances is reduced. This transformation can improve the data locality and, hence, a higher cache hit rate is expected [17, 36]. Tiling is only a heuristic to obtain data locality but does not necessarily improve the cache hit rate.

The more schedule dimensions are involved in the tiling transformation, the more advantage can be taken from data locality. Consider the two schedules in Example 4.19. Both satisfy the dependency $D_1 : S[i,j,k] \to S[i+1, j-1, k]$ in the outermost schedule dimension. Obviously, every schedule dimension produce a loop at the code generation step. The schedule in part *(a)* allows tiling of only the innermost two loops. As a result, the dependency $D_1$ spans between statement instances of two different tiles and a low data reuse is expected. In part *(b)*, all three generated loops can be tiled and most of the depending memory references are resolved inside one tile, improving data reuse.




*(a) Only the innermost two schedule dimension can be tiled.*




*(b) All three schedule dimension can be processed in tiles.*

**Example 4.19:** *Example of different tile dimensions. The top shows schedules in the schedule tree representation and the bottom depicts the projection of the execution dates on the first two dimensions of the time space. Additionally, the tiles are marked and numbered according to their execution order.*

The schedule optimizer in Polly applies the tiling transformation on the innermost band nodes of the schedule tree, if the schedule dimensions of that band nodes are interchangeable. Polly then introduces new loops that enumerate the single tiles. The conditions that must hold in a band node for the tiling transformation in Polly are described in Definition 4.20.

**Definition 4.20** (Tilable Band Node)**.** *Let $S_1, \ldots, S_k$ be all statements that are scheduled at a band node $B$. Further, let $B$ contain $e$ schedule dimensions with the scheduling functions $\theta^d_{S_i}, \ldots, \theta^{(d+e-1)}_{S_i}$. Let $G$ be the set of dependencies that are not solved strongly by schedule dimensions $d' < d$ and only concern the statements $S_1, \ldots, S_k$.*

*Polly applies the tiling transformation on all $e$ schedule dimensions of $B$ if*

*(a) $B$ contains at least two schedule dimensions ($e \geq 2$).*

*(b) $B$ is the innermost band node of the subtree ($d + e - 1 = \dim(\theta_{S_i})$).*

*(c) The schedule dimensions of $B$ are interchangeable*

$$\forall D_{S,R} \in G : \forall \vec{x}_S \to \vec{x}_R : \forall d' \in \{d, \ldots, d + e - 1\} : \theta^{d'}_S(\vec{x}_S) \leq \theta^{d'}_R(\vec{x}_R)$$

**Tiling Feature**

The tiling feature processes the schedule tree statement-by-statement. First, the innermost band node of each statement is identified. If the tiling transformation can be applied on that band node, the number of surrounding loops is used to classify the impact of the transformation. If not, the tiling value for this statement is rated with zero. The overall feature value is the weighted sum of the tiling values of all statements of the SCoP. The exact calculation is given in Definition 4.21.

**Definition 4.21.** *Let $S_1, \ldots, S_k$ be all statements of the SCoP and $T$ be a schedule tree. Further, let $p_i \in \{0, 1\}$ indicate whether the tiling transformation can be applied on the innermost band node $B_i$, which schedules the statement $S_i$. Let $d_i$ be the number of loops generated around $B_i$.*

*We define the tiling feature value as*

$$F_{tiling}(T) = \sum_{i=1}^{k} \frac{|\mathbb{D}_{S_i}|}{\sum_{i=1}^{k} |\mathbb{D}_{S_i}|} \cdot p_i \cdot \left(\frac{1}{2}\right)^{d_i}$$

Overhead, produced by the tiling transformation, is not considered in this feature. A typical tile size is 32 or 64 iterations per dimension, but if only each fourth iteration of each loop executes a statement instance, the number of instances grouped in one tile is significantly lower as with full tiles. Hence, the performance overhead of the additional loops, introduced by the tiling transformation, might exceed the benefit of data locality.

### 4.1.4 Computation Overhead

Two major aspects that produce computational overhead are addressed by the two overhead features. First, additional *if*-instructions inside a loop can influence the runtime of the program. Second, if a schedule is highly skewed - inner loop variables depend on many outer loop variables - the calculation of the loop boundaries can impact the performance of the schedule as well.

**Feature: Conditional Overhead**

The conditional overhead feature aims to classify a schedule by the number of *if*-instructions generated inside the loop nest. It compares the iteration polytopes of all statements of a generated loop and verifies whether all iterations execute the same statements. In case not, *if*-conditions are placed inside the loop body as guards around the statements.

The iteration polytope of a statement describes the execution dates of a loop on which instances of this statement are executed. Given a loop at dimension $d$, the iteration polytope of a statement simply contains all execution dates produced by the prefix-schedule of the statement at dimension $d+1$.

**Definition 4.22** (Iteration Polytope)**.** *Let $S$ be a statement of the SCoP with the scheduling function $\theta_S$.*

*The iteration polytope of statement $S$ at schedule dimension $d$ is defined as*

$$P_{S,d} = \{(\theta_S^1(\vec{x}), \ldots, \theta_S^d(\vec{x})) \mid \vec{x} \in \mathbb{D}_S\}$$

The overhead cost for *if*-instructions is obtained by depth-first search on the schedule tree and is the overall number of loop iterations containing an *if*-guard. Sequence and set nodes sum up the overhead values of their children $C_1, \ldots, C_k$.

$$\texttt{IF}(N) = \texttt{IF}(C_1) + \ldots + \texttt{IF}(C_k)$$

Among the other node types, only band nodes can generate loops. For each loop generating dimension $d$ at a band node the iteration polytopes $P_{S_i,d}$ are computed for all statements $S_i$ scheduled at that band node. If the polytopes are congruent, no *if*-guards are needed. Otherwise, *if*-instructions are placed to guard at least one of the statements. Since the additional *if*-guard must be checked in every iteration step, the total number of loop iterations is used as the overhead cost of this loop. The band node returns the sum of the overhead cost of all its schedule dimensions $d' \in \{d, \ldots, d+e-1\}$.

$$\texttt{IF}(N) = \sum_{d'=d}^{d+e-1} \begin{cases} 0 & \text{if } \bigcup_{S_i} P_{S_i,d'} = \bigcap_{S_i} P_{S_i,d'} \\ |\bigcup_{S_i} P_{S_i,d'}| & \text{otherwise} \end{cases}$$

The computation of the conditional overhead feature value is described in Definition 4.23.

**Definition 4.23.** *Let $T$ be a schedule tree and $I$ be the total number of statement instances scheduled by $T$. Further, let $\texttt{IF}(T)$ be the overhead cost for if-instructions extracted from $T$.*

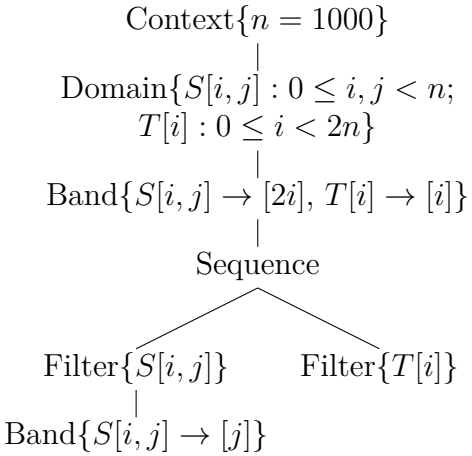
*We define the conditional overhead feature value as*

$$F_{condition}(T) = 1 - \frac{\texttt{IF}(T)}{I}$$

Perfectly structured loop nests without any *if*-instructions obtain the best feature value 1.0, whereas the conditional overhead cost increases the deeper an *if*-guard is placed in the loop nest and the overhead feature converges to zero.

Consider the schedule tree in Example 4.24*(a)*. There is only one band node generating a single loop around both statements. The iteration polytopes for that loop generating dimension are $P_{S,1} = \{c \mid 0 \leq i < 2n \wedge 2c = i\}$ and $P_{T,1} = \{c \mid 0 \leq c < 2n\}$. Since the two polytopes are not congruent, an *if*-guard around the statement $S$ is placed in the generated code in part *(c)*. The statement $S$ is executed on every second iteration step of the loop, whereas $T$ is executed at all iteration steps. The conditional overhead feature value is computed as:

$$F_{OB}(T) = 1 - \frac{2n}{2n + n^2} = 0.998$$




*(a) Schedule tree representation.*

$$P_{S,1} = \{c : 0 \leq i < 2n, 2c = i\}$$
$$P_{T,1} = \{c : 0 \leq c < 2n\}$$

*(b) Iteration polytopes of the first band node.*

```
for(c0 = 0; c0 < 2000; c0++){
  if (c0 % 2 == 0)
    for(c1 = 0; c1 < 1000; c1++)
      S(c0 / 2, c1);
  T(c0);
}
```

*(c) Generated Code.*

**Example 4.24:** *Schedule with conditional overhead.*

Note, that the overhead cost calculation introduces two inaccuracies. First, in some cases, more than one *if*-guard is placed inside a loop, but the cost calculation only counts one *if*-instruction per iteration step of the loop in case the iteration polytopes are not congruent. Second, the compiler might pass the *if*-guards down the loop nest in order to reduce the code size of the generated program and the overhead cost calculation is slightly higher than the number of *if*-instructions inside the loop nest.

This feature is only applicable on SCoPs with more than one statement. SCoPs with only a single statement will not generate conditional instructions and, hence, the feature value will always be equal to 1.0. Furthermore, loop unrolling, which is not considered by this feature, can decrease the number of if-instructions in the code.

### Feature: Skewing Overhead

With the skewing transformation, inner loop variables depend on the values of outer loop variables. The skewing overhead feature classifies schedules regarding to the complexity of loop boundary computation. The iteration variables of highly skewed loops depend on the values of many outer loop variables. Before an inner loop can be executed, the lower and upper bound of the loop variable must be computed. In

some cases, integer min, max or rounding operations are part of the loop boundary conditions. The complexity increases with the number of iteration coefficients unequal to zero.

For a statement $S$, the minimum number of iteration coefficients unequal to zero is given by the dimensionality of its iteration domain. The maximum number is reached if all iteration variables of $S$ are part of the affine functions of all loop generation schedule dimension and can be computed by $(\dim(\mathbb{D}_S))^2$. The skewing overhead feature calculates the ratio between additional iteration coefficients and the maximal number of additional iteration coefficients for each statement of the SCoP. The result is then weighted by the number of statement instances. The skewing overhead feature value computation is described in Definition 4.25.

**Definition 4.25.** *Let $S_1, \ldots, S_k$ be all statements of the SCoP with the scheduling functions $\theta_{S_i}$. Further, let $I_i$ be the number of iteration coefficients of the scheduling matrix $A_i$ from $\theta_{S_i}$ that are unequal to zero.*

*We define the skewing overhead feature value as*

$$F_{skewing} = 1 - \sum_{i=1}^{k} \frac{|\mathbb{D}_{S_i}|}{\sum_{i=1}^{k} |\mathbb{D}_{S_i}|} \cdot \frac{I_i - \dim(\mathbb{D}_{S_i})}{\dim(\mathbb{D}_{S_i})^2 - \dim(\mathbb{D}_{S_i})}$$

Consider the two schedules in Example 4.26. The iteration domain of the statement $S$ is given by $\mathbb{D}_S = \{(i,j,k) \mid 0 \leq i,j,k < 10\}$. The first schedule, in part *(a)*, has the minimum number of iteration coefficients and the loop boundaries in the resulting code are constants. The schedule in part *(b)* has the maximal number of iteration coefficients unequal to zero. The resulting code contains a lot of *min*, *max* and *floord* operations in order to compute the loop bounds.

$$\theta_S(\vec{x}_S) = (i,j,k)$$

```
for (c0 = 0; c0 <= 9; c0 += 1)
  for (c1 = 0; c1 <= 9; c1 += 1)
    for (c2 = 0; c2 <= 9; c2 += 1)
      S(c0, c1, c2);
```

*(a) Schedule and code with no skewing overhead. The feature value is* 1.0*.*

$$\theta_S(\vec{x}_S) = (2i + j - k, i + 3j + 2k, i + j + 2k)$$

```
for (c0 = -9; c0 <= 27; c0 += 1)
  for (c1 = max(max(3*c0-45, -2*c0), -2*c0+5*floord(c0-1, 2)+5);
       c1 <= min(min(3*c0+45, -2*c0+90), (c0+10)/2+40);
       c1 += 5)
    for (c2 = max(max(c1-18, (-4*c0+3*c1)/5), (2*c0+c1)/5);
         c2 <= min(min(c1, ((-4*c0+3*c1)/5)+18), ((2*c0+c1)/5)+18);
         c2 += 2)
      S((4*c0-3*c1+5*c2)/10, (c1-c2)/2, (-2*c0-c1+5*c2)/10);
```

*(b) Schedule and code with high skewing overhead. The feature value is* 0.0*.*

**Example 4.26:** *Two schedules with different skewing overhead. The iteration domain of $S$ is defined by $\mathbb{D}_S = \{(i,j,k) \mid 0 \leq i,j,k < 10\}$.*

Note, not all skewing constellations with additional iteration coefficients unequal to zero produce expensive loop boundary computations with min, max or rounding operations. In some further cases, compiler optimizations like strength reduction can reduce the computation overhead inside the loop body [22]. Moreover, the skewing transformation can enable parallel loop execution or loop tiling. The advantage will likely predominate the overhead of complex boundary computation.

### 4.1.5 Out-of-Order Execution

Out-of-order execution is a paradigm in modern CPUs in order to exploit instruction level parallelism and to hide cache latency. Instead of wasting clock cycles waiting on slower processing units or data from the cache, CPUs can execute instructions from the instruction buffer out-of-order if there is no data dependency.

Iteration domain: $\mathbb{D}_S = \{i \mid 0 < i < 10\}$; Schedule: $\theta_S(\vec{x}_S) = (i)$

Dependency: $S[i] \rightarrow S[i+1]$.

```
for(c0 = 1; c0 < 10; c0++)
  A[i] = A[i-1] + 1   // S
```

```
1  // iteration 1:
2  RD  %r1 A[0]
3  ADD %r1 1
4  SR  A[1] %r1
5
6  // iteration 2:
7  RD  %r1 A[1]
8  ADD %r1 1
9  SR  A[2] %r1
10
11 // ...
```

*(a) Single loop with a loop carried dependency and the assembly-like CPU instruction for the first two loop iterations.*

No data dependency.

```
for(c0 = 1; c0 < 10; c0++)
  A[i] = A[i] + 1   // S
```

```
1  // iteration 1:
2  RD  %r1 A[1]
3  ADD %r1 1
4  SR  A[1] %r1
5
6  // iteration 2:
7  RD  %r1 A[2]
8  ADD %r1 1
9  SR  A[2] %r1
10
11 // ...
```

*(b) Single loop without loop carried dependencies and the assembly-like CPU instruction for the first two loop iterations.*

**Example 4.27:** *Example with out-of-order execution.*

Consider the two loops in Example 4.27. The iteration domain of the statement $S$ is $\mathbb{D}_S = \{i \mid 0 < i < 10\}$ and the schedule is given by $\theta_S(\vec{x}_S) = (i)$. In part *(a)*, each statement instances of $S$ depends on the instance of the previous loop iteration, whereas in part *(b)* the loop does not carry any data dependency. The assembly code in part *(a)* loads the memory element `A[0]` into the register `%r1` at line 2. After that, the constant `1` is added in line 3 and, finally, the value of register `%r1` is stored back to the memory location of `A[1]`. These three micro-operations must be executed in-order. At each memory load or store the CPU stalls, waiting for the data from the cache or main memory. Since all CPU instructions of the successive loop iteration depend on the data of the same memory cell `A[1]`, which is written by the last micro-operation of the first loop iteration, no out-of-order execution is possible. In contrast, the second loop of the assembly code of part *(b)* in line 7 is data independent of the first loop in lines 2-4. The only dependency in this

assembly code is introduced by referencing the same register `%r1`. This so-called *false* dependency is solved with hardware algorithms, e.g. the Tomasulo algorithm [38, 61], that renames the used registers at runtime. While waiting for the read and store instructions in line 2 and 4 to finish, the CPU can execute the instructions of the second loop iteration in lines 7-9. For that reason, a loop with independent loop iterations can achieve a better performance than with loop carried dependencies.

#### Out of Order Feature

The out-of-order execution feature verifies whether the innermost loop of each statement carries any dependency. If not, the loop iterations are independent and instructions of different loop iterations can be executed out-of-order. Otherwise, the out-of-order paradigm is limited to the number of instructions inside the loop body. The out-of-order feature value is computed statement-by-statement and weighted by the number of instances. The formal computation is described in Definition 4.28.

**Definition 4.28.** *Let $S_1, \ldots, S_k$ be all statements of the SCoP. Further, let $p_i \in \{0, 1\}$ indicate whether the innermost loop of the statement $S_i$ carries any dependency.*

*We define the out-of-order feature value as*

$$F_{\textit{out-of-order}}(T) = \sum_{i=1}^{k} \frac{|\mathbb{D}_{S_i}|}{\sum_{i=1}^{k} |\mathbb{D}_{S_i}|} \cdot p_i$$

Note, if the dependencies of the innermost loop span more than one loop iteration or the number of independent statements inside the loop body is big enough, the performance of the schedule is less or even not influenced by the carried dependencies of the innermost loop.

### 4.1.6 Feature Work: Additional Features

The six features described beforehand are only a subset of a wide range of possible metrics that can classify the performance of a schedule. This section describes two other features that possibly correlates with the performance of a schedule.

**Vectorization.** There might be a vectorization feature that determines the number of statement instances that can be computed with vector operations. The code generator in Polly can automatically detect and insert vector operations if a loop can be computed in parallel, no control flow operations are placed inside the loop body and the number of loop iterations is an integer multiple of the vector width of the target system [4]. Obviously, the data accessed by a vector operation should be stored aligned in the RAM, otherwise the vector load or store operation is executed sequentially. Stock et al. [60] use machine-learned models to predict the performance of Single Instruction Multiple Data (SIMD) codes. Their features are extracted from the generated assembly code. Similar approach can be used to classify the vectorization capability of a schedule.

We explored that Polly's auto-vectorization cannot improve the runtime of almost all randomly sampled schedules of programs from the Polybench Suite. Most

of the time the reason is data miss-alignment or a varying number of iterations of the innermost loop.

**Code Size.** Pouchet [52] described a procedure for machine-learning the optimal transformation sequence of an arbitrary program. For this purpose, the best transformation sequences of different programs are detected by several performance metrics. Among execution time, parallelism and memory behavior, Pouchet proposed the code size of the generated output program as a performance related metric.

## 4.2 Machine Learning a Performance Prediction Function

The outcome of the features, described in the previous section, form the feature vector of the schedule. It is computed only from the abstract schedule tree description in the polyhedral model without the need of code generation and performance measurement.

In general, machine learning techniques are used for two purposes: classification and regression. A machine learned classifier can assign each new input feature vector to a certain class, whereas a regression approximates a continuous value. This master's thesis uses performance regression which approximates the measured runtime of the generated program.

The input dataset (feature vectors of schedules for different programs) is split into two partitions. One partition is used as the training set and the other one as the testing set to evaluate the learned function. Usually, the ratio is 60% training and 40% testing set. In this thesis, the training and testing set is chosen among different benchmark programs in order to verify the applicability of the learned performance prediction function on arbitrary programs. In the evaluation experiments, the training set consists of three different programs of the Polybench Suit [4]. The learned model is then verified on three other benchmark programs. A second option that is considered in this thesis is *k-fold cross-validation* [39]. This technique randomly splits the data set into $k$ equally sized partitions. A prediction function is trained $k$ times, each time with $n$ - $1$ partitions as trainings set and one partition as test set. The learned cross-validation function is the average of the $k$ prediction functions.

The following two machine learning algorithms are used in this master's thesis in order to obtain the performance prediction function.

**Linear Regression.** This algorithm simply defines the output of the learned function as a linear combination of the values of the feature vector. We assume that the feature values are normalized in $[0, 1]$. Linear regression tries to find the best fitting parameters $w_1, \ldots, w_{d+1}$ to define the following linear function.

$$func : [0,1]^d \rightarrow [0,1] : \vec{v} \rightarrow w_1 \cdot v_1 + \ldots + w_d \cdot v_d + w_{d+1}$$

Each instance of the trainings set produce a linear equation. Obviously, the trainings set must contain at least as many instances as available features, otherwise the parameters cannot be determined. The parameters $w_1, \ldots, w_{d+1}$ are chosen, such

that the variance between the estimation and the actual value is minimal for all trainings instances.

Linear regression might fail if the input features highly depend on each others or the output (runtime of the generated program) is not linear dependent to the feature values. Furthermore, it can also fail if the training data set is to small or contains a lot of noise.

**K-Nearest Neighbor Algorithm.** The *k-nearest neighbor (KNN) algorithm* is an instance based lazy learner [7]. The feature vectors are elements of $[0,1]^d \subset \mathbb{R}^d$ and can be compared by their distance, i.e. the Euclidean distance. In the training phase of this algorithm all instances of the trainings set are stored in the model. For each new input feature vector only the $k$ nearest neighbors in the feature space are considered. The estimated output variable (in our case the performance of the schedule) is the average of the values stored at the $k$ nearest neighbors. It can be useful to introduce weights to the contributions of the neighbor vectors. The number of neighbors $k$ is chosen depending on the number of training instances. With a huge number of trainings instances it is usually set to 20, likewise in this thesis.

For machine learning the performance prediction function we use the data mining and machine learning framework WEKA [68]. In addition to *linear regression* and *k-nearest neighbor*, the framework provides a wide range of different machine learning algorithms including function regressions, lazy learners, stochastic models and decision trees.

# Chapter 5

# Evaluation

In this chapter we describe our evaluation of the performance of the geometric schedule sampling method of Section 3.2, the significance and computation time of the features described in Section 4.1 and, finally, the accuracy of the performance prediction functions learned by the machine learning algorithms described in Section 4.2.

## 5.1 Aim of this thesis

This master's thesis aims to machine learn a performance prediction function that can approximate the runtime of a schedule. In order to learn such a surrogate function, several different features are defined in Section 4.1, which extract performance related aspects from the schedule tree representation of the schedule.

## 5.2 Research Questions

**Q1: Is geometric schedule sampling efficient enough to obtain a huge number of schedules in reasonable time?** As we need a huge number of well distributed schedules for the input data of the machine learning algorithms, the schedule sampling algorithm must efficiently run in reasonable time. The performance of the geometric sampling method, introduced in Section 3.2, depends on the runtime of Barvinok's algorithm that counts the number of integer points inside a polytope and, therefore, depends on the structure of the search space polytopes.

**Q2: How can normalization improve schedule comparison?** The chosen geometric sampling method produces schedules that have different schedule matrices, but actually represent the same schedule. By applying the normalization steps described in Section 3.3 we expect to detect equal schedules that are expressed by different schedule matrices.

**Q3: How long does the feature calculation take compared to the actual runtime of the generated program?** For each new schedule, whose runtime should be predicted, the set of features described in Section 4.1 must be computed. In order to gain benefit over runtime measurement, the calculation time of the

features has to be less than the measured runtime of the generated programs. Otherwise, the measured runtime of the generated program can be used instead of a prediction.

**Q4: Do the calculated feature values correlate with the measured data of the generated program?** The calculated feature values ought to represent performance key factors of a loop program. In order to get an accurate performance prediction function, the correlation between the feature values and the measured data of the testing programs must be high.

**Q5: How accurate is the predicted runtime of the machine-learned performance prediction function?** The performance prediction function is machine-learned, as described in Section 4.2. The input data is the measured runtime and the calculated feature values of a huge number of different schedules of several benchmark programs. The point of interest is how the predicted runtime correlates with the measured runtime and if the learned prediction function can be generalized to arbitrary programs.

## 5.3 Experimental Setup

### 5.3.1 Hardware

For runtime measurement of the schedules we use the 10-core Intel Xeon E5-2690 v2 CPU, which can compute the parallel loops by its 10 native cores. The benchmarks' problem sizes are chosen in a way, such that the total used data memory does not fit into the L3 cache. The cache specifications of the CPU can be retrieved from Table A.1. As a second platform, the Intel i5-4570 CPU is used to evaluate the geometric schedule sampling method.

### 5.3.2 Software

The geometric sampling method and the features are implemented in the programming language Scala in version 2.11 [6]. The Scala code is compiled to Java Byte code which is executed with OpenJDK 1.8. There is one utility class for schedule sampling and one for computing the values of the features, which uses the internal schedule tree implementation of the Polyite project. Most of the functionalities rely on polyhedral operations. For polyhedron manipulation we use the *Integer Set Library*[1] by Verdoolaege [63] and for counting the number of points inside a polytope we use the implementation of Barvinok's algorithm in version 0.39, also provided by Verdoolaege. As described at the beginning of Chapter 4, Polly is used as the code generator and for further optimization steps, e.g. tiling. We use Polly[2] within the *llvm* environment in version *3.9.0svn*. The compiler *gcc* is operates in version 5.0.4 and the *libpapi* for cache hit measurement is used in version 5.4.3. The Intel Xeon machine for benchmarking the schedules runs Ubuntu 16.04 as the operating

[1] git version: *cfebc0c65aed630a30c6d7925dcf7817e802fd3f*
[2] git version: *2b618e01a6bf30d813bcd39cd22cf60df3d84f17*

system. For machine learning the performance prediction function we make use of the data mining and machine learning framework WEKA [68] in version 3.8.0.

### 5.3.3 Benchmarks

The benchmark programs are chosen among 29 of the 30 programs of the Polybench Suite in version 4.1 [5]. Each of the Polybench benchmarks compromises exactly one SCoP. Polly cannot automatically detect the SCoP of the benchmark program *nussinov* without manually annotating the internal function call as side-effect free. The Polybench Suite allows to measure the cache hit rate of the kernel function using PAPI [46].

We selected two sets of programs. First, for evaluation of the geometric sampling method all 29 benchmark programs are used. The second set consists of the six programs shown in the Table A.2 and is used to evaluate the normalization steps, the features and the learned prediction function. Among these programs there are some variations of matrix multiplication from the so-called Basic Linear Algebra Subprograms (BLAS) [41] (*gemm*, *syr2k*, *syrk*, *trmm*), a matrix decomposition as a linear algebra solver (*cholesky*) and a 9-point stencil (*seidel-2d*).

We execute and measure the runtime of each generated program five times and use the minimum execution time as the result in order to minimize noise from the system or external influences, e.g., the CPU temperature.

## 5.4 Experiments

This section describes different experiments that are necessary to answers the research questions.

### 5.4.1 E1: Schedule sampling

This experiment targets question Q1 and provides data, whether geometric schedule sampling is efficient enough to obtain a sufficient number of different schedule for the input data of the machine learning algorithms. Sampling a schedule, as described in Section 3.2, is reduced to uniformly sampling a point from the search space polytopes of each schedule dimension. The performance of geometrically sampling a point from a $\mathbb{Z}$-polytope is highly related to the runtime of the point counting oracle. This thesis uses an implementation of Barvinok's algorithm [13]. Without the sampling optimization described by Pak [51], the Barvinok algorithm is called $\mathcal{O}(n^2 \cdot L^2)$ times in the worst case during the sampling process. The parameter $n$ is defined by the dimensionality of the polytope and $L$ is the bit size of the dimensions' values. Since our search space polytopes are limited to $([-3, 3] \cap \mathbb{Z})^n$, each dimension can only have seven different values and, therefore, $L$ is equal to 3. The number of dimensions $n$ of the search space polytopes of the considered 29 benchmark programs varies between 4 and 93.

Dependencies between statements are expressed with linear inequality constraints in the search space polytope. The runtime of Barvinok's algorithm increases with the number of constraints of the polytope and the number of dimensions that are

involved in one inequality. Therefore, the runtime for sampling a schedule depends on the number of dependencies of the SCoP, too.

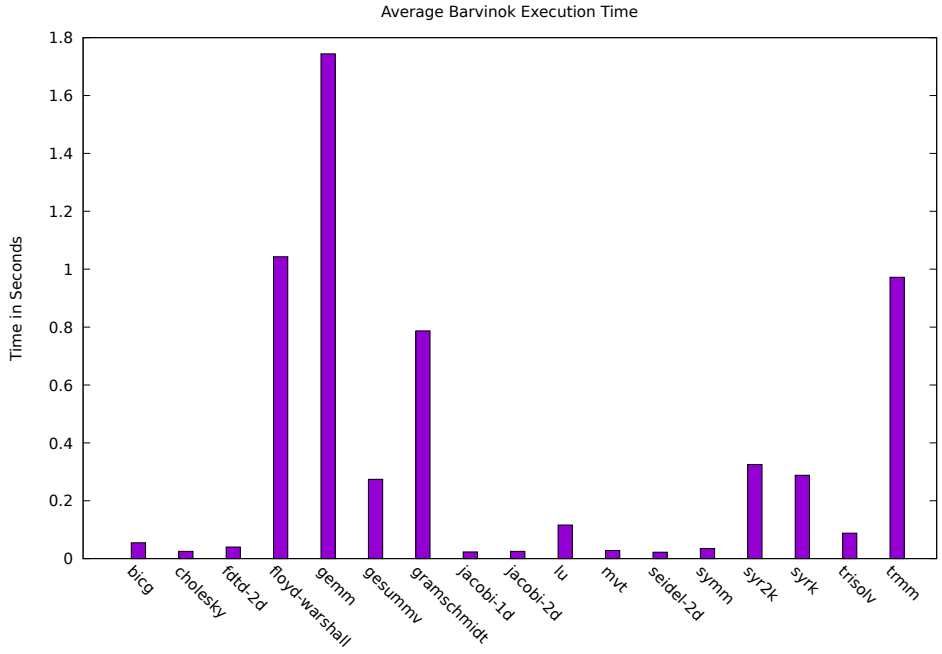



**Figure 5.1:** *We generated two search space regions for each benchmark program and measured the average Barvinok execution time on the Intel i5-4570 CPU.*

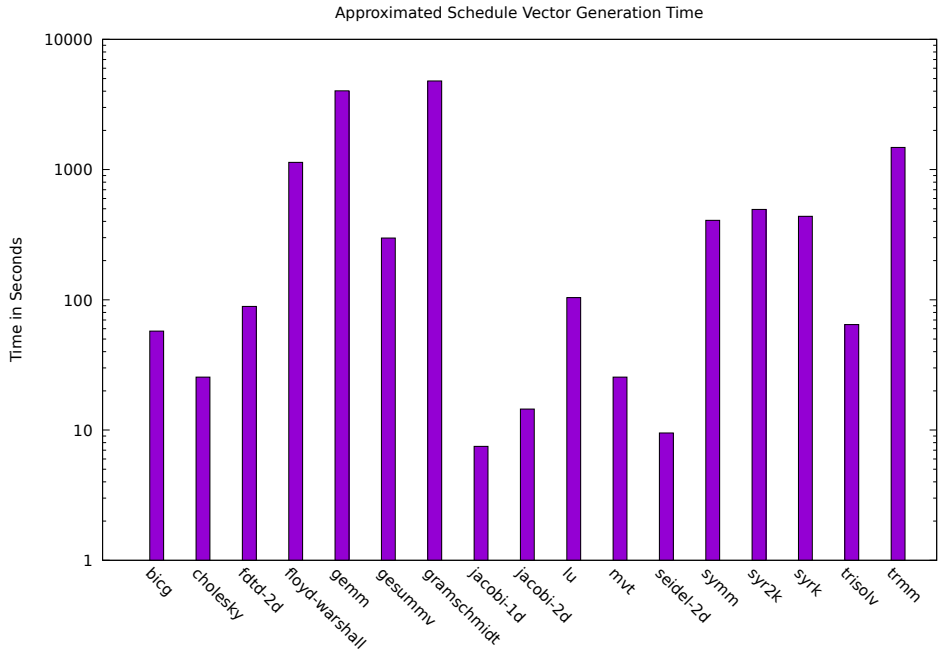



**Figure 5.2:** *The worst case sampling time of one schedule vector from a search space polytope is computed with $\mathcal{O}(n^2 \cdot L^2)$ multiplied by the corresponding Barvinok execution times shown in Figure 5.1. The y-axis of this histogram is in logarithmic scale.*

In order to evaluate the runtime of Barvinok's algorithm on our search space polytopes, we have generated two different regions of the search space for 29 programs of the Polybench Suite. Counting the number of integer points inside those polytopes performs in less than 5 seconds on the Intel i5-4570 CPU with 17 out of 29 benchmark programs. The results for the 17 fastest Barvinok execution times is shown in Figure 5.1. In 13 cases the runtime is less than 0.5 seconds and in 10 cases it is even lower than 0.2 seconds. Multiplying the Barvinok execution times with the worst case number of oracle calls, the sampling of one matrix row vector approximately takes less than two minutes for 9 of the benchmark programs. Figure 5.2 shows the worst case sampling time for one schedule vector out of one search space polytope. Note, the y-axis in this histogram is in logarithmic scale.

In reality, we expect the sampling runtime to be much lower, because each recursion step of the geometric sampling algorithm fixes one parameter and, hence, the dimensionality decreases and complex constraints of the polytope are simplified. We have sampled one thousand schedules for the six chosen testing benchmark programs from 50 regions of the search space. The average sampling time per schedule (including all schedule dimensions) is only 69.32 seconds, which is clearly less than the worst case computed above. Figure 5.3 shows the measured averaged schedule sampling time of the six benchmark programs, that are selected for the evaluation of the normalization, features and prediction function. As expected, according to the Barvinok runtime in Figure 5.1, the sampling time for schedules of the *trmm* and *gemm* benchmark program is higher.

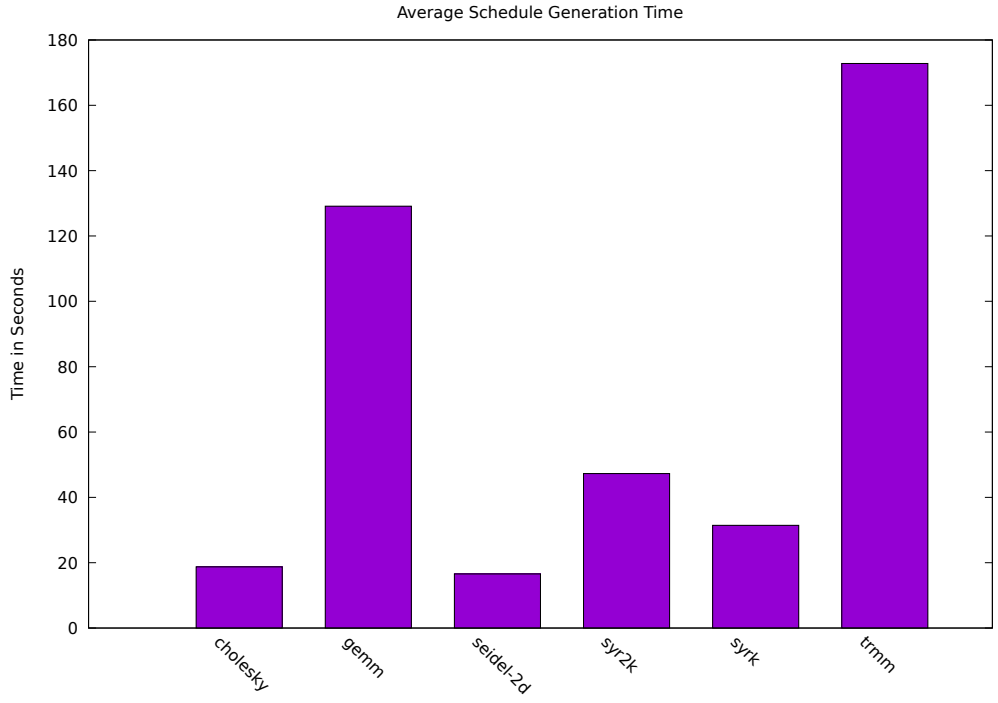



**Figure 5.3:** *Detailed view on the schedule sampling time of the six selected benchmark programs.*

### 5.4.2 E2: Enhancing schedule comparison by using the normalization steps

This experiment helps to answer question Q2, how normalization can improve schedule comparison. Therefore, we have sampled one thousand schedule of each of the six chosen benchmark programs from Table A.2. Polly can generate correct running code for 90.3 % of all six thousand schedules in a reasonable time (< 5 minutes). Without normalization there are no schedules with identical matrices. After applying the normalization steps, we can detect up to 516 duplicated schedules of one benchmark program. The result of this experiment is shown in Table 5.4. Duplicates are found only for schedules of the *cholesky* and *seidel-2d* benchmarks, because both span small search space regions containing fewer legal schedules than with the other programs. All duplicate schedules are later filtered out from the input data set for the machine learning algorithms, such that only unique, normalized schedules are used for training the performance prediction function.

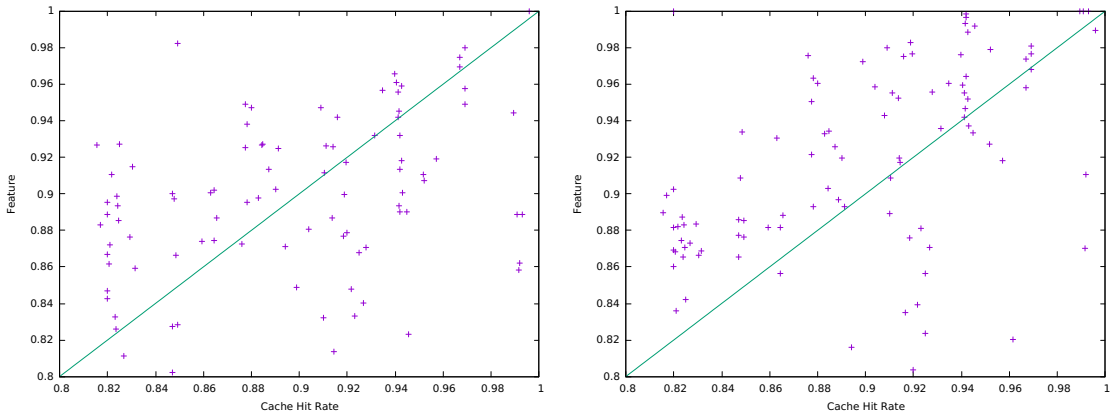
| benchmark | generated and run | duplicates(matrix) | duplicates (normalized) |
|---|---|---|---|
| cholesky | 940 | 0 | 516 |
| gemm | 902 | 0 | 0 |
| seidel-2d | 894 | 0 | 193 |
| syrk | 930 | 0 | 0 |
| syr2k | 795 | 0 | 0 |
| trmm | 958 | 0 | 0 |

**Table 5.4:** *Impact of normalization steps on comparing schedules.*

### 5.4.3 E3: Correlation and performance of the cache feature

This experiment refers to question Q3 and Q4 and focuses on the cache feature, which can be configured with the number of dependency samples that are used for cache hit approximation.

In order to evaluate the accuracy and the runtime of the cache hit approximation, we run the cache feature on 100 different schedules of the *gemm* benchmark. The runtime of the generated programs is in average 40 seconds. We expect the accuracy of the cache feature to increase with the number of evaluated dependency instances. Figure 5.5 shows the cache hit approximation against the measured cache hit rate for two different numbers of dependency samples. The x-axis specifies the measured cache hit rate, whereas the y-axis specifies the approximated one by the cache feature. In part *(a)* we use 27 instances per dependency and the feature calculation time is in average 33 seconds per schedule, which is lower than the average of the measured execution times. The correlation value between the measured cache hit rate and the approximated one is 0.23. That means, the cache hit rate prediction is poorly accurate using only 27 dependency instances. By increasing this number to 64 instances per dependency, as shown in Figure 5.5*(b)*, the accuracy of the predicted cache hit rate is getting better (with a correlation value of 0.48), but is still not reliable. Since the approximation time exceeds the measured runtime of the generated program with this configuration, simple cache hit measurement, for example with PAPI [46], would be faster and exact.




*(a) 27 dependency instances per dependency; correlation: 0.23; runtime: 33 seconds*

*(b) 64 dependency instances per dependency; correlation: 0.48; runtime: 76 seconds*

**Figure 5.5:** *100 different schedules of the gemm benchmark program; the x-axis specifies the measured cache hit rate and the y-axis specifies the predicted cache hit rate by the cache feature described in Section 4.1.2.*

### 5.4.4 E4: Calculation time of the features

This experiment refers to question Q3 and considers all features. Apart from the cache feature, only simple tree walks and static metric calculation is performed. We expect that the runtime of all features besides the cache feature is clearly lower than the execution time of the generated program.

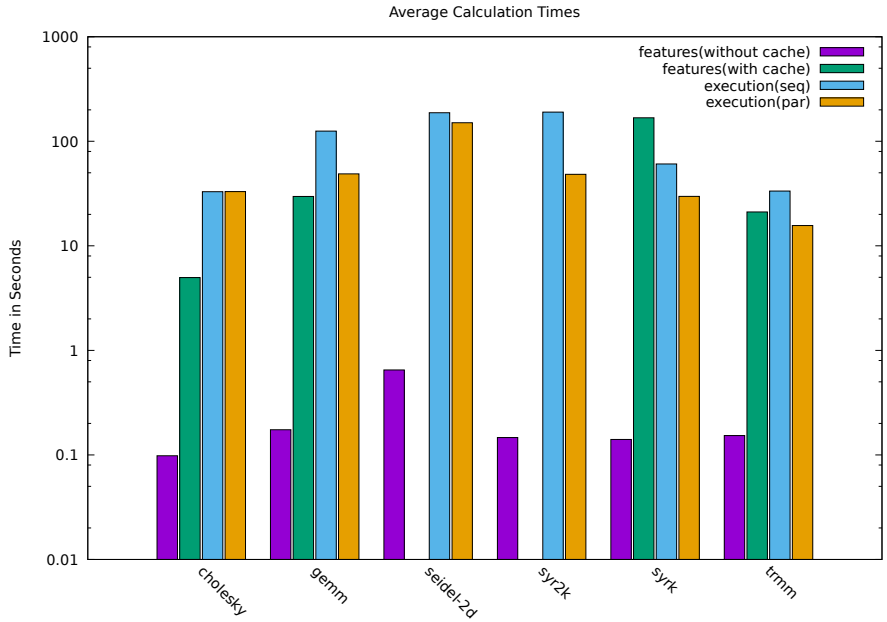



**Figure 5.6:** *Execution time and feature calculation time of the six selected benchmark programs. The y-axis is in logarithm scale.*

We measure the runtime and the feature calculation time of one thousand generated programs of the six chosen testing benchmarks. Figure 5.6 shows the average sequential and parallel execution time of the generated programs along with the feature calculation time, once including the cache hit approximation and once without. There are no cache feature calculation times for the two benchmarks *seidel-2d* and *syr2k*, because the cache feature runtime is too huge. For the other benchmarks the cache feature is configured with 27 instances per dependency. Note, that the y-axis in this histogram is in logarithmic scale.

The average runtime of all features apart from the cache feature is clearly lower than the average sequential and parallel execution times of the benchmark programs. The feature calculation takes on average only 0.5 % of the parallel execution time of the six benchmarks.

### 5.4.5 E5: Correlation of the feature values

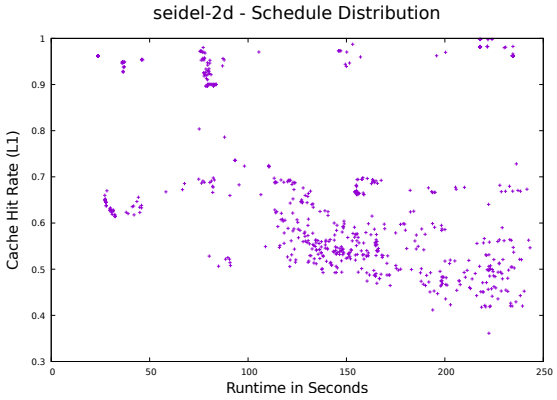
In order to help answering question Q4, this experiment measures the runtime and cache hit rate of the generated programs of one thousand sampled schedules from 50 different regions of the search space for the six chosen benchmarks. The correlation between the feature values and the measured data is shown exemplary with the best correlating benchmark program (*seidel-2d*) and one with poor correlation (*cholesky*).

The *seidel-2d* is a 9-point stencil, which iteratively updates each point of a two-dimensional array with the average value of all the eight neighboring cells and the old cell value. The detected SCoP of this benchmark contains one single statement, which is surrounded by a loop nest of depth three. The *cholesky* benchmark computes the decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and conjugate transpose.

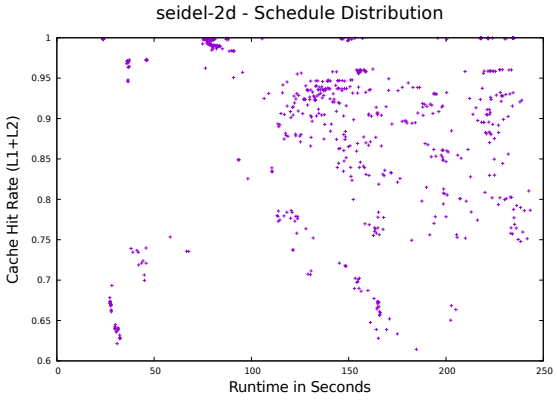
We measured the runtime and cache hit rate of all cache levels for one thousand schedules of the two benchmarks *seidel-2d* and *cholesky*. Remember, that only 894 of the sampled 1000 schedules produce correct executable code for the *seidel-2d* benchmark and only 940 schedules for the *cholesky* benchmark respectively. Removing the duplicates there are 701 schedules left for *seidel-2d* and 424 for *cholesky* respectively. Figure 5.7 shows the distribution of the schedules according to their runtime and the cache hit rate in different cache levels. Part *(a)/(d)* shows the L1 cache hit rate, part *(b)/(e)* show the L1 + L2 cache hit rate and, finally, the overall cache hit rate of all three cache levels is shown in part *(c)/(f)*. For both programs, there are several schedules that have a good cache hit rate (independent of the cache level), but also a high runtime. Normally, we expect a schedule with higher cache hit rate to perform faster. On the other hand, there are also few schedules with worse cache hit rate that in contrast perform very well.

In the following we evaluate the correlation of each single feature. All the following scatter plots specify the measured runtime of the generated program at the x-axis (in seconds) and the measured total cache hit rate of the program at the y-axis. The color, in which each schedule is drawn in the scatter plot, defines the value of the depicted feature.

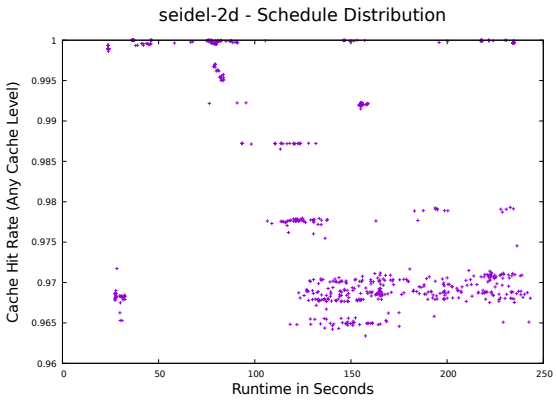
In addition to the scatter plots of the two exemplary benchmark programs in this section, scatter plots of the other four benchmark programs *gemm*, *syrk*, *syr2k* and *trmm* are attached in Appendix B.



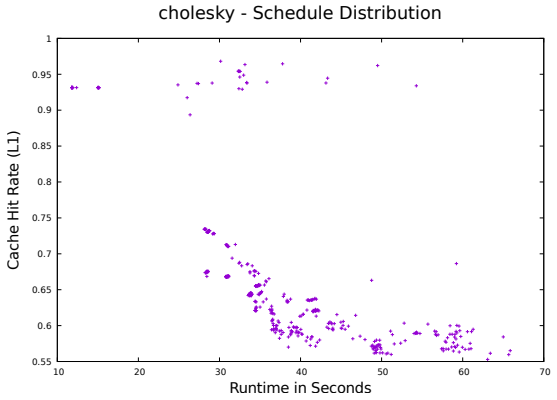

*(a) The cache hit rate of L1 cache.*



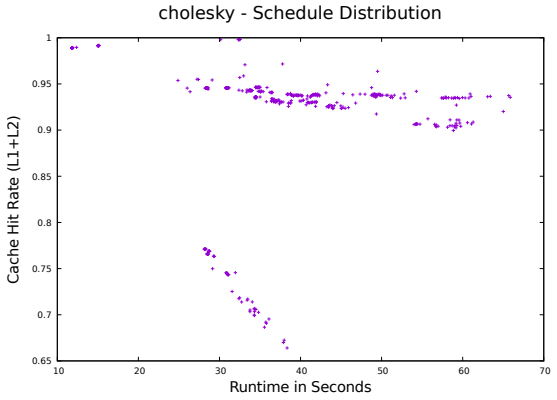

*(b) The cache hit rate of L1 and L2 cache.*



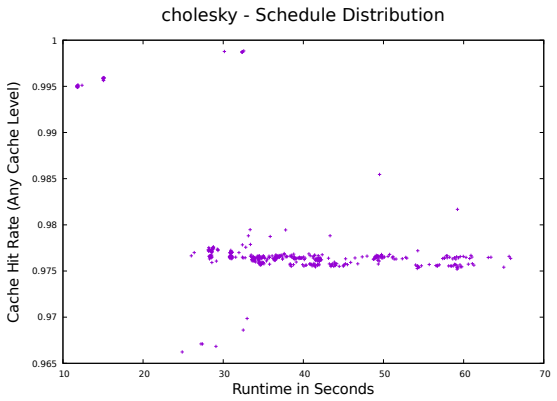

*(c) The cache hit rate of any cache level.*




*(d) The cache hit rate of L1 cache.*




*(e) The cache hit rate of L1 and L2 cache.*




*(f) The cache hit rate of any cache level.*

**Figure 5.7:** *Schedule distribution over the measured runtime and cache hit rate of different cache levels for the seidel-2d stencil (a)-(c) and the cholesky benchmark program (d)-(f).*

**Parallelization Feature.** We expect from the parallelization feature that schedules with a high value perform faster, because the program can make use of the 10 cores of the Intel Xeon CPU.

The color in the scatter plot of Figure 5.8 describes the value of the parallelization feature of the *seidel-2d* benchmark. Its SCoP has only one statement surrounded by three nested loops and, therefore, the parallelization feature outcome can be 1.0

in case the outermost loop is parallel, 0.9 in case the second loop is parallel or 0.0 in case no loop is parallel. The innermost loop is never thread-parallelized by Polly. None of the sampled schedules is capable of parallelizing the outermost loop due to dependency limitations and, hence, the output of the parallelization feature is either 0.9 or 0.0 for our set of schedules. From the plot, we can retrieve that all schedules that perform faster than 68 seconds have parallel loops and are detected by the parallelization feature.

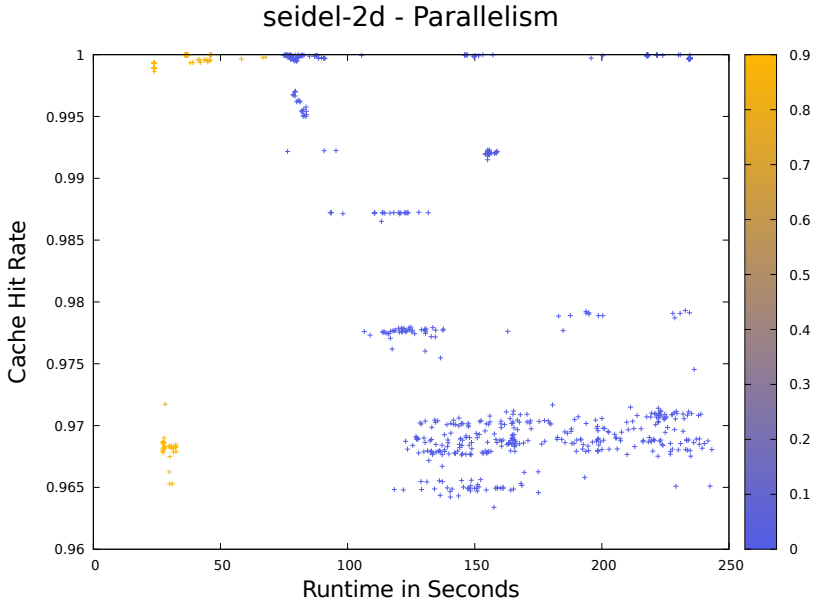



**Figure 5.8:** *Result of the parallelization feature applied to the schedules of the seidel-2d benchmark.*

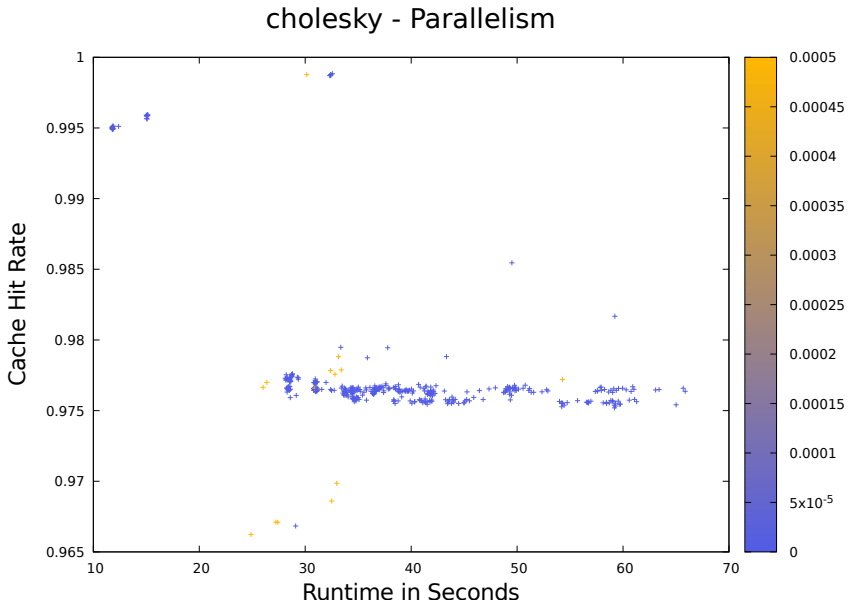



**Figure 5.9:** *Result of the parallelization feature applied to the schedules of the cholesky benchmark.*

Applied to the schedules of the *cholesky* benchmark program, the parallelization feature yields the results depicted in Figure 5.9. Only a few schedules are capable of loop-parallelization, which is detected by the parallelization feature. The maximum feature value is only 0.0005, which means that either the overhead for the parallel loops is too high or the number of statement instances that can be computed in parallel is negligible. The fastest schedules, with runtimes lower than 20 seconds, are computed only sequentially and achieve a very good overall cache hit rate.

**Tiling Feature.** The tiling feature detects schedules that produce loops on which Polly can apply the tiling transformation during code generation process. We expect those schedules to achieve a good cache hit rate.

The tiling feature results of the *seidel-2d* schedules are shown in Figure 5.10. A lot of the schedules with an overall cache hit rate near 100% achieve the cache hit rate by tiling loops and all detections are correct. Since the overall cache hit rate of a schedule is not directly related to its runtime, this feature is possibly less relevant for the performance prediction function.

For the *choleksy* benchmark, the schedules with the corresponding tiling feature values are depicted in Figure 5.11. In contrast to *seidel-2d*, the feature cannot detect the schedules with the best cache hit rates of this benchmark. Nearly all schedules achieve a very good cache hit rate above 97.5 % and most of the generated nested loops are not tilable. Polly can apply tiling only on 14 out of 424 different schedules.

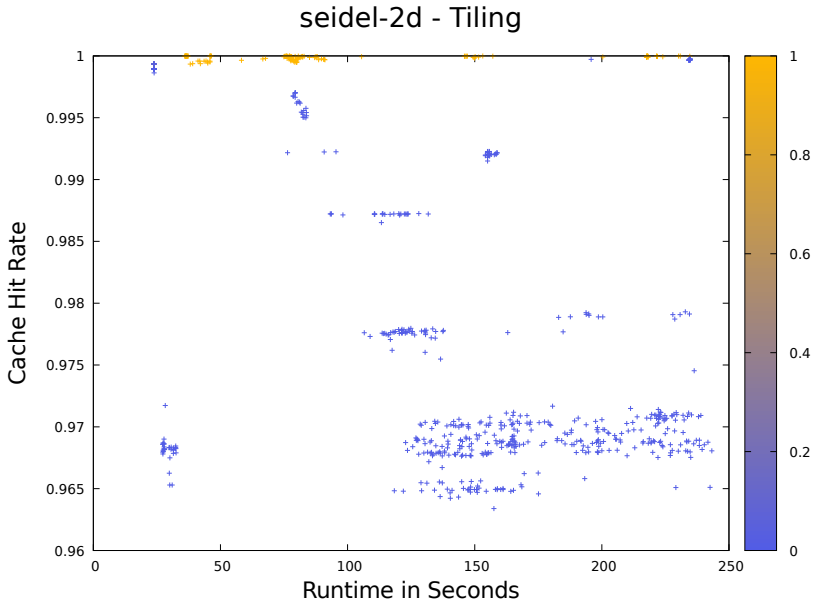



**Figure 5.10:** *Result of the tiling feature applied to the schedules of the seidel-2d benchmark.*

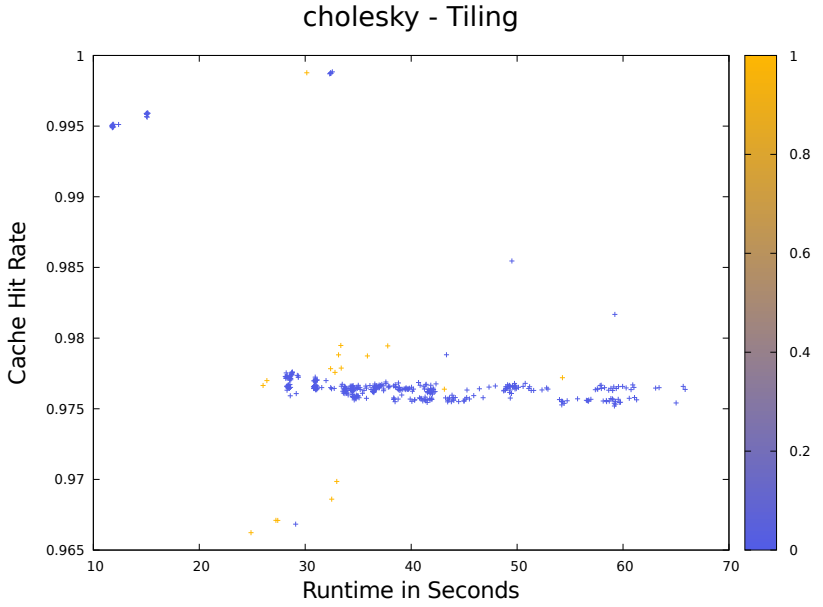



**Figure 5.11:** *Result of the tiling feature applied to the schedules of the cholesky benchmark.*

**Cache Feature** As already discussed in E3, we expect schedules with a high cache hit rate, equivalent to a good cache feature value, to perform faster. We did not compute the cache feature for the *seidel-2d* benchmark, because of performance issues.

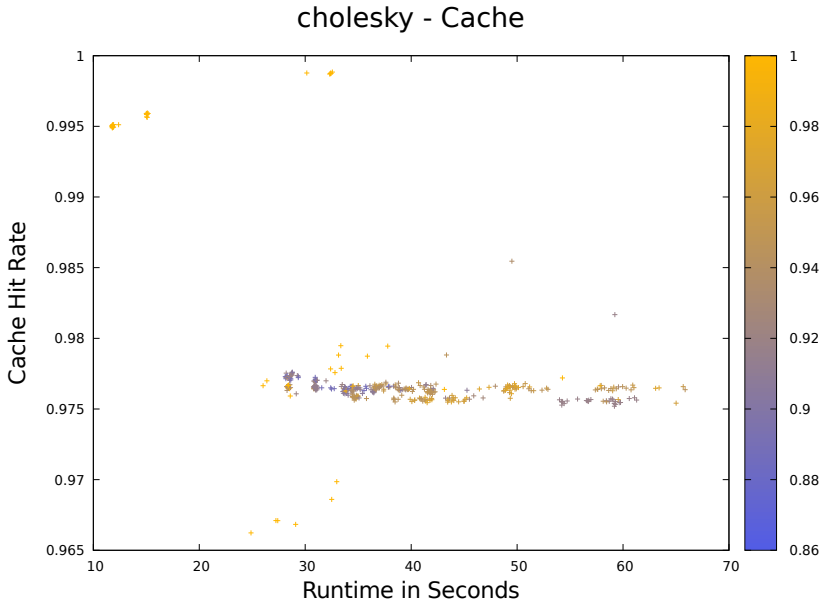



**Figure 5.12:** *Result of the cache feature applied to the schedules of the cholesky benchmark.*

The accuracy of the cache feature for the *cholesky* schedules is shown in Figure 5.12. As already shown in E3, the cache feature is poorly accurate for the generated programs of the *gemm* benchmark. In almost the same manner, the cache feature

value of the *cholesky* benchmark hardly correlates with the measured cache hit rates of the *cholesky* benchmark.

**Out-of-Order Feature** We expect schedules that can execute iterations of the innermost loop out-of-order to perform faster than schedules whose innermost loop carries dependencies.

Figure 5.13 shows the out-of-order feature result for the *seidel-2d* benchmark. As we already know, the schedules at the left side of the plot, with a runtime faster


**Figure 5.13:** *Result of the out-of-order feature applied to the schedules of the seidel-2d benchmark.*

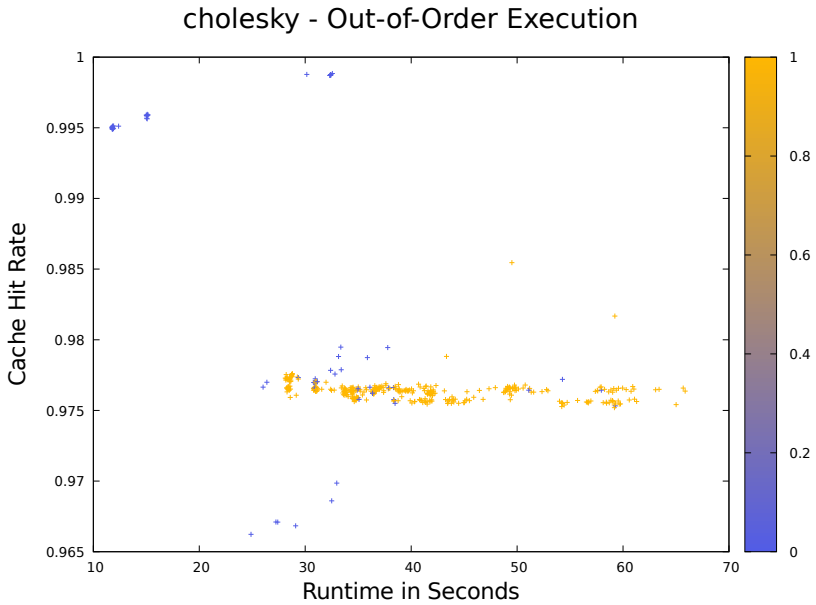



**Figure 5.14:** *Result of the out-of-order feature applied to the schedules of the cholesky benchmark.*

than 68 seconds, produce thread-parallel loops. Among those schedules out-of-order execution of the innermost loops is not possible. The remaining schedules are mainly split into two parts. First, the schedules located at the right top of the chart, whose innermost loop cannot be computed out-of-order, and achieve a good cache hit rate at a slow runtime. The second part consists of all schedules with the capability of out-of-order execution. From the plot, we can infer a relation between the runtime and the cache hit rate of those schedules.

The result of the same feature applied to the schedules of the *cholesky* benchmark is shown in Figure 5.14. Nearly no schedule has thread-parallel loops according to the parallelization feature, but the schedules that allow out-of-order execution have almost the same cache hit rate with a varying runtime. The result is contrary to what we expect, because the schedules without out-of-order execution of the innermost loop perform better.

**Conditional Overhead Feature** We expect schedules with less conditional overhead (equal to a high feature value) to require less runtime. The conditional overhead feature is only applicable on SCoPs with more than one statement. The detected SCoP of the *seidel-2d* benchmark has only one statement and the conditional overhead feature will always returns 1.0.

The conditional overhead feature applied to the schedules of the *cholesky* benchmark creates the result shown in Figure 5.15. Most of the schedules do not need if-instructions inside their loop bodies and get the highest feature value 1.0. The schedules whose generated code contains additional *if*-instructions are equally distributed in the ranges of the measured runtime and the cache hit rate. Therefore, no direct correlation can be identified for this feature.

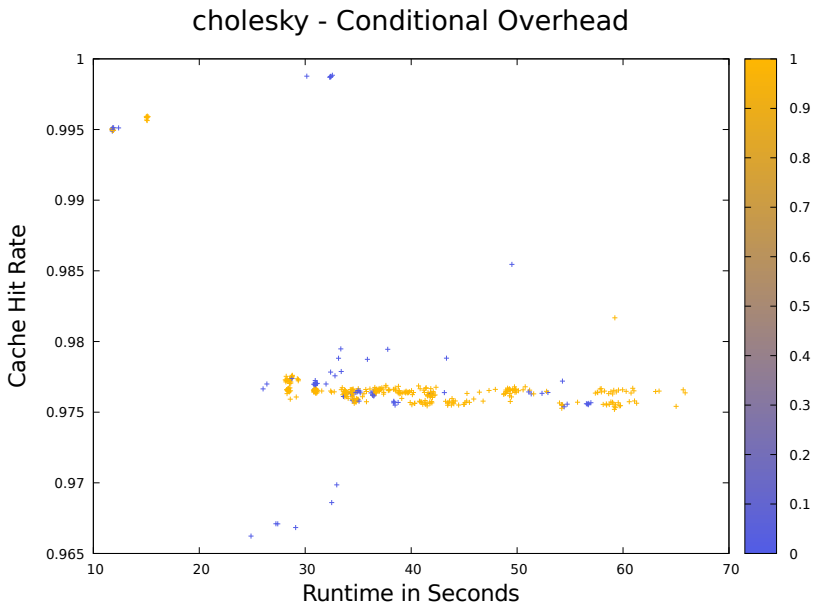



**Figure 5.15:** *Result of the conditional overhead feature applied to the schedules of the cholesky benchmark.*

**Skewing Overhead Features** We expect schedules with a high skewing overhead value to perform better than the others, because a high feature result indicates less extra loop boundary calculations.

Figure 5.16 shows the skewing overhead feature applied to the *seidel-2d* benchmark schedules. The schedules with a good skewing overhead feature value are distributed over the whole range of runtime and cache hit rate. The only thing we can extract from this plot is that there are slightly more schedules with a higher skewing feature value at the upper cache hit rate level, whereas with low cache hit rate, the high skewing feature values are more rare.

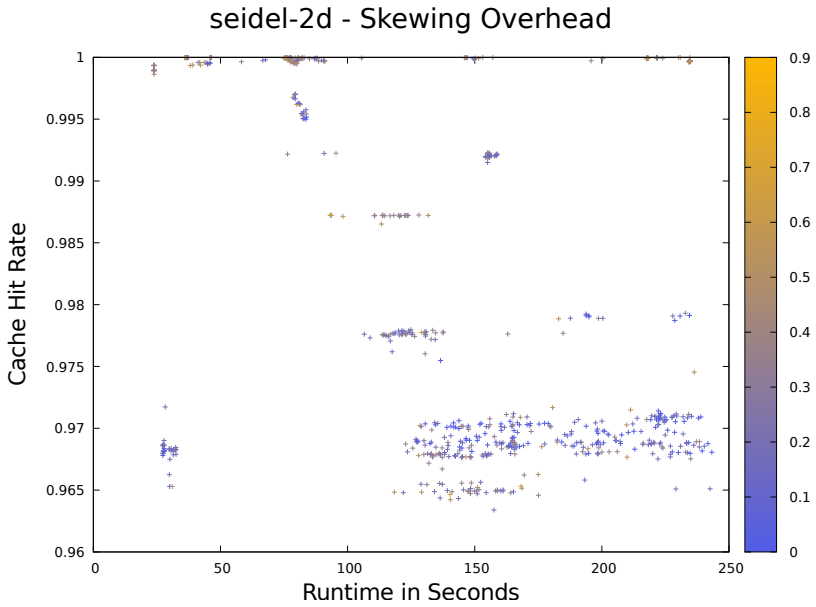



**Figure 5.16:** *Result of the skewing overhead feature applied to the schedules of the seidel-2d benchmark.*

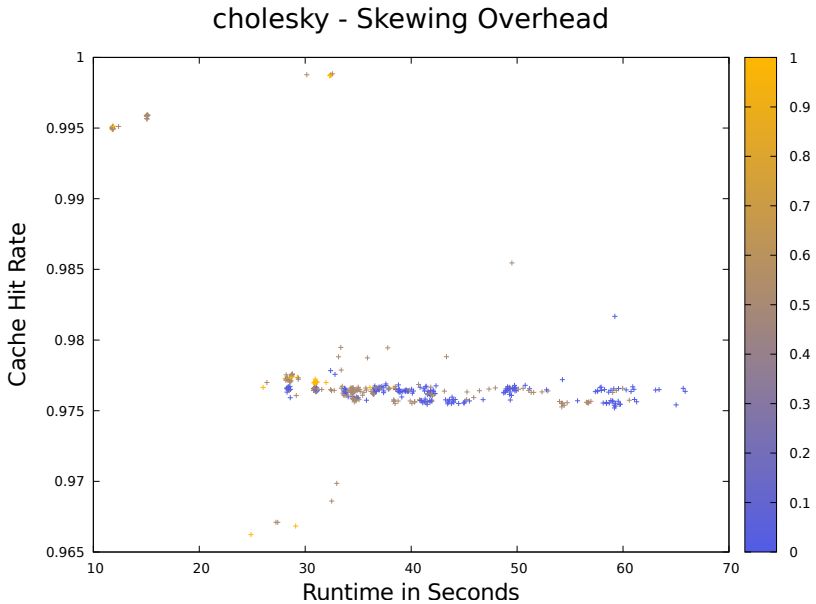



**Figure 5.17:** *Result of the skewing overhead feature applied to the schedules of the cholesky benchmark.*

As can be seen from Figure 5.17, the distribution of the good skewing feature values is entirely different form the *cholesky* benchmark. Most of the schedules with a good skewing overhead value achieve a faster runtime than the other schedules. As against to the *seidel-2d* benchmark, we cannot detect a correlation between the skewing feature and the cache hit rate.

### 5.4.6 E6: Machine learning a performance prediction function

This experiment provides answers to question Q5 using linear regression and the k-nearest neighbor algorithm for the performance prediction function. We sample and run one thousand schedules from the six chosen benchmark programs in order to use them as the input data of the machine learning algorithm. Removing all duplicated schedules from the input set of the machine learning algorithms reduces the noise of the data. For both machine learning algorithms, we learn ten different prediction models using all of the schedules of three benchmark programs as the learning data set and the remaining three benchmark programs as the testing set. The schedules of the *syr2k* benchmark are always part of the testing set for independent validation purpose. Furthermore, a 10-fold cross validation model is trained on the schedules of the three benchmarks from the learning data set. This model is used to evaluate the feature similarity of the schedules of the three different benchmark programs.

| syrk | trmm | cholesky | seidel-2d | gemm | syr2k | 10-fold |
|---|---|---|---|---|---|---|
| x | x | x | 0.5227 | 0.3135 | 0.1668 | 0.6710 |
| x | x | 0.3379 | x | 0.1705 | 0.0747 | 0.7506 |
| x | x | 0.5390 | 0.3274 | x | 0.1654 | 0.3838 |
| x | 0.5281 | x | x | 0.3386 | 0.1553 | 0.7623 |
| x | 0.5633 | x | 0.4936 | x | 0.1621 | 0.6900 |
| x | 0.5288 | 0.4204 | x | x | 0.0971 | 0.7422 |
| -0.0526 | x | x | x | 0.2083 | 0.1573 | 0.7990 |
| 0.0864 | x | x | 0.6090 | x | 0.1871 | 0.6883 |
| 0.0194 | x | 0.4197 | x | x | 0.1465 | 0.7626 |
| 0.0494 | 0.5443 | x | x | x | 0.1429 | 0.7785 |

**Table 5.18:** *Correlation values of the learned function using linear regression. In each row, the three benchmark programs that are used as the training set are marked with x. The values at the other benchmark programs represent the correlation factor between the measured and the predicted runtime of the model. Additionally, the last column reveals the correlation value of a prediction model that is learned by a 10-fold-cross-validation run on the trainings set.*

For linear regression, the correlation values between the predicted runtime and the measured runtime is shown in Table 5.18. Each row represents one prediction model that is trained on the benchmarks that are marked with a 'x' and is validated with the remaining benchmarks, for which the correlation factors are shown. The last column contains the correlation values of the 10-fold cross validation run of the three training benchmark programs. Any prediction model that is not trained on the *syrk* benchmark cannot predict the performance of the schedules of the *syrk*

benchmark at all, which is shown by the correlation factor close to zero. Similar applies for the *syr2k* benchmark program. The maximum correlation values in the model validation phase are reached by the prediction model that is trained on *syrk*, *cholesky*, and *gemm* in row 5. The correlation values are given with 0.5633 for *trmm* and 0.4936 for *seidel-2d*. These correlation factors are too low to get a reliable performance prediction. From the correlation value of the 10-fold cross validation run on the training benchmarks we can see that training a prediction model that is validated against itself is just as unreliable.

| syrk | trmm | cholesky | seidel-2d | gemm | syr2k | 10-fold |
|---|---|---|---|---|---|---|
| x | x | x | 0.5127 | 0.5218 | 0.1464 | 0.8459 |
| x | x | 0.2507 | x | 0.5241 | 0.1576 | 0.8967 |
| x | x | 0.4763 | 0.4814 | x | 0.2956 | 0.6310 |
| x | 0.6855 | x | x | 0.4230 | 0.0769 | 0.8610 |
| x | 0.7072 | x | 0.4290 | x | 0.2390 | 0.7955 |
| x | 0.6718 | 0.0520 | x | x | 0.2576 | 0.8626 |
| -0.0608 | x | x | x | 0.5234 | 0.1443 | 0.8858 |
| -0.1476 | x | x | 0.5597 | x | 0.2842 | 0.8149 |
| -0.1245 | x | 0.2375 | x | x | 0.2934 | 0.8837 |
| -0.0506 | 0.7219 | x | x | x | 0.2361 | 0.8414 |

**Table 5.19:** *Correlation values of the learned function using KNN-algorithm. In each row, the three benchmark programs that are used as the training set are marked with x. The values at the other benchmark programs represent the correlation factor between the measured and the predicted runtime of the model. Additionally, the last column reveals the correlation value of a prediction model that is learned by a 10-fold-cross-validation run on the trainings set.*

Table 5.19 shows the correlation values using k-nearest neighbor as the machine learning algorithm. The parameter $k$ is set to 20 and the predicting value is the unweighted average of the 20 neighbors. Analogous to linear regression, the machine-learned models that contain *syrk* or *syr2k* in the training set will not produce meaningful prediction values, since their correlation factors are close to or even less than zero. The prediction model that is trained on *syrk*, *trmm*, and *cholesky* performs best with the 20-nearest neighbor algorithm, but the performance prediction is still too inaccurate.

### 5.4.7 E7: Additional Idea - Iterative Learning.

This experiment follows the idea of learning the prediction model towards one SCoP using only the schedules of the same benchmark program. This can, for instance, be done during iterative compilation of the genetic algorithm. We expect the feature values of schedules of the same SCoP to behave more similar as with different programs.

We learned a prediction model only on the schedules of the *seidel-2d* benchmark using 10-fold cross validation and the resulting correlation value is 0.83. Figure 5.20 shows the measured runtime plotted against the predicted one. The same method

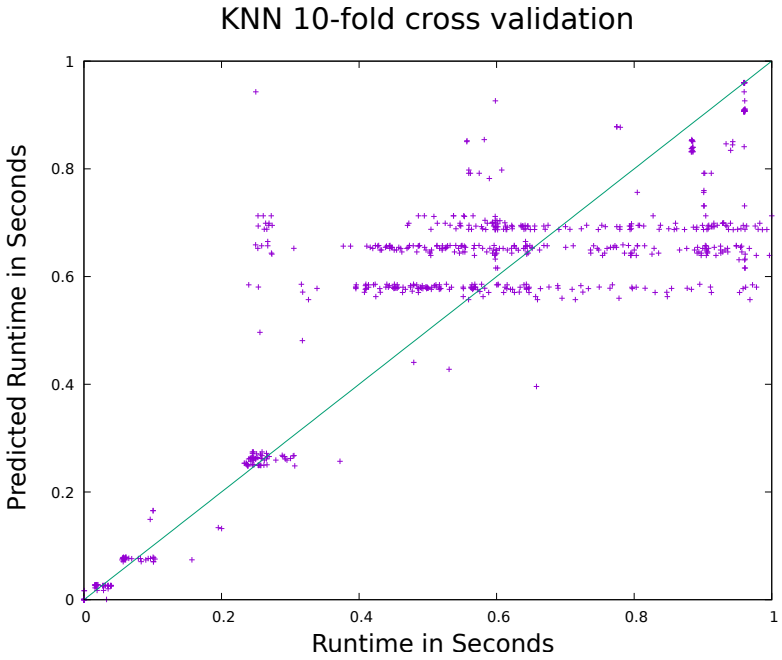



**Figure 5.20:** *10-fold cross validation on the schedules of the seidel-2d benchmark achieve a correlation value of 0.83.*

applied to the *syrk* benchmark reveals a correlation value of only 0.40, hence this idea is not practical in general.

## 5.5 Discussion

This section discusses the research questions using the result of the experiments.

**Q1: Is geometric schedule sampling efficient enough to obtain a huge number of schedules in reasonable time?** As the result in experiment E1 states, the average sampling time over the six benchmarks, that are chosen for training and validating the prediction function, is below 70 seconds. Figure 5.21 shows a more detailed view on the average sampling time per benchmark in comparison to the sequential and parallel average execution time. As can be seen, the sampling time exceeds the execution time of the benchmarks *gemm* and *trmm*. Sampling is at least faster than the sequential execution time of the other benchmarks. In total of all six benchmarks, the average sampling time (~70 seconds) is lower than the average execution times (sequential and parallel), which is given by 75 seconds. Since the generated programs are actually executed five times for better measurement accuracy, the average sampling time is clearly lower than the average execution time.

The proposed optimization by Pak [51] can further improve the sampling runtime by calling the oracle worst case only $\mathcal{O}(n^2 \cdot \log L)$ times, but is not implemented in this thesis. We believe that this optimization can reduce the sampling time such that sampling a schedule is in average faster than the sequential execution for all of the six benchmarks.

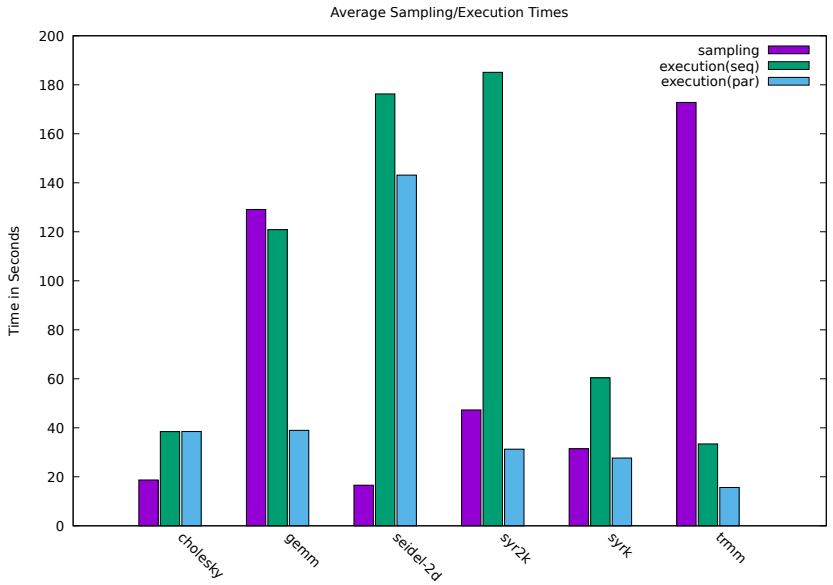



**Figure 5.21:** *This plot shows the average sampling time of the schedules against the average sequential and parallel execution times.*

Arbitrary SCoPs in *real* programs can produce search space polytopes with much higher dimensionality and, hence, worse sampling time, but the geometric sampling strategy is only needed in the learning phase of the performance prediction function and is efficient enough on parts of the benchmark programs from the Polybench Suite. In comparison to the execution time of the schedules, the time needed for sampling a schedule is in a practical range for the six chosen benchmark programs for learning the prediction function.

**Q2: How can normalization improve schedule comparison?** We showed with experiment E2 that it is possible to detect duplicate schedules by transforming them to a normalized schedule representation. The normalization steps described in this thesis are not complete and the normalization procedure can be further improved. Duplicates of the sampled schedules are only identified from two of the six benchmark programs in the experiment above, because their search space regions are small enough, such that duplicate schedules are actually sampled by the sampling strategy. This schedule normalization is a first step towards total schedule comparison of different schedules of one SCoP, although future work has to be done.

As mentioned in Section 3.3, the normalization steps can help to improve the runtime of the generated program, e.g. by enabling the tiling transformation. The influence of normalization on the runtime of a schedule is also a topic for future work.

For this thesis, the normalization steps are mainly used to filter duplicates from the input data of the machine learning algorithm in order to reduce noise. Furthermore, the normalization is needed for correctness of some of the feature calculations, e.g. the loop overhead feature.

**Q3: How long does the feature calculation take compared to the actual runtime of the generated program?** Since all feature values have to be computed for each new schedule that must be classified, the calculation time should be significantly lower than the expected runtime of the generated program. Otherwise, the performance approximation with the prediction function will take longer than the actual runtime measurement, which is exact.

All features except the cache hit approximation perform simple tree walks or metrics calculation on static known information that can be retrieved from the schedule tree. Experiment E4 has shown that their total calculation time lasts in average only 0.5 % of the measured parallel execution time for the six chosen benchmarks for prediction model training.

The runtime of the cache feature strongly depends on the execution time of Barvinok's algorithm in order to count the number of different memory accesses between two references to the same memory cell. The cache feature is the only feature that can be configured by the number of dependency instances that are used to approximate the cache hit rate of one dependency. The task is to find a configuration, which leads to a accurate cache hit approximation but only needs a small time to calculate. Experiment E3 showed, that there exists no such a configuration for the *gemm* benchmark. The result of experiment E4 reveals that the cache feature calculation with a small number of dependency instances is much too slow in comparison to the other feature calculations. In average, the calculation takes 387 times longer than all the other features for the four benchmark programs *cholesky*, *gemm*, *syrk* and *trmm*. The calculation time even exceeds the measured parallel and sequential runtime of the generated programs for some of those benchmarks. With more complex SCoPs and associated schedules we expect the calculation time of the cache hit approximation to further increase. Because of the high calculation time and the inaccuracy, the cache feature is not included in the input data for the machine learned performance prediction function.

**Q4: Do the calculated feature values correlate with the measured data of the generated program?** The cache feature is the only feature that can be directly compared to the measured cache hit rate of the generated benchmark programs. All the other features influence somehow the runtime of the generated program, therefore it is hard to rate the correlation.

The cache feature is configurable regarding the number of instances that are tested per dependency. The data in experiment E4 show that the cache feature is inaccurate with a small number of dependency instance for the *gemm* benchmark. The accuracy can be improved in a certain range by increasing the number of instances per dependency, but it is still not reliable for usage as a performance indicator. The experiment further showed, that increasing the number of dependency instances leads to a longer feature calculation time, which exceeds the measured execution time of the *gemm* benchmark. As a result, there is a upper limit of adjusting the accuracy of the feature, such that the calculation time does not exceed the execution time of the program, which is a precondition of a practical performance prediction function. From experiment E5 we can conclude that the cache feature does not produce precise cache hit approximation for the *cholesky* benchmark as well. Therefore, we will not use this feature for training and learning the

prediction function. A very interesting observation of experiment E5 is, that a good overall cache hit rate does not imply a fast execution time of the program. That also applies to the data that is attached in Appendix B. There are a lot of schedules across the benchmark programs that reach a good overall cache hit rate, but their execution takes longer in contrast to the other schedules with possibly worse cache hit rate.

All the other features are expected to correlate with the measured runtime of the schedules directly. For the two exemplary benchmark programs in experiment E5, each feature value correlates totally different with the measured data of the generated programs. The parallelization feature can detect the fast running schedules of the *seidel-2d* benchmark as expected, but in case of absence of parallelism, as it is nearly the case with the *cholesky* benchmark, the output of this feature is unusable. Most of the schedules of the benchmark programs in Appendix B produce parallel loops and reveal similar parallelization feature values. This leads to a worse correlation as well. The tiling feature correlates well with the overall cache hit rate for five of the six benchmark programs in experiment E5 and the Appendix B, but for the *cholesky* benchmark no correlation can be identified. The out-of-order feature correlates only for the *seidel-2d* benchmark. This can be due to the fact that the SCoP only contains one statement and there are a lot of dependencies between instances of this statement. From the conditional overhead feature of all benchmark programs we cannot extract some direct correlation to the runtime by inspecting the plots. Nearly no schedule needs *if*-instructions inside the loop body according to the conditional overhead feature. The influence of this feature will probably increase with SCoPs that contain more than two statements. The skewing overhead feature correlates slightly with the measured execution time and/or the measured cache hit rate for all of the six benchmark programs shown in experiment E5 and Appendix B.

**Q5: How accurate is the predicted runtime of the machine-learned performance prediction function?** As described in Section 4.2, we use two different machine learning algorithms for the performance prediction function. Experiment E6 shows that it is impossible to learn an accurate performance prediction function based on the proposed features neither with linear regression nor with k-nearest neighbor algorithm. Each of the prediction models is trained on three of the six benchmarks and validated on the remaining three benchmark programs. We observe that in many cases the resulting prediction model is over-fitted to the trainings set, which means that an accurate prediction is only possible on schedules of the trainings benchmarks or very similar programs. In contrast, there are benchmark programs, like the *syrk* benchmark, that must be part of the trainings set, because otherwise the correlation between the predicted and the measured value drops to zero.

In the further experiment E7, we tried to train the prediction model towards only one of the benchmark programs. We expect the prediction function to be more accurate. The correlation value of such a 10-fold cross validated prediction model is good enough for reasonable predictions for the *seidel-2d* benchmark. But the *syrk* benchmark prediction model reveals a correlation value of only 0.4, which corresponds to a worse prediction. This leads us to the conclusion that the proposed features are not suitable for performance indication, at least for the *syrk* benchmark.

## 5.6 Threats to Validity

Certain threats to validity affects our results and the ability to generalize them to arbitrary SCoPs.

Internal threats include the design and selection of the proposed features. Some of the features described in Section 4.1 make assumptions on the program behavior that are possible to strong or maybe do not completely apply to the reality. It is also possible that the features are dealing with a to general view on the performance indicators. Furthermore, there are potentially more, better correlating features beyond the scope of this thesis.

Another internal threat concerns the selection of the machine learning algorithm. The input is always a normalized feature vector, but the accuracy can vary with different machine learning algorithms. In this thesis we used two simple algorithms to get a first insight, whether a performance prediction function can be learned based on the proposed features.

The performance of machine learning algorithms depends on the input data. An internal threat can regard the schedules sampling strategy. We think that by choosing uniformly distributed schedule vectors from the search space polytopes of a search space region, we obtain a well distributed set of schedules. The internal threat of measurement errors is reduced by using the fastest of five execution time measurements of each generated program.

An external threat is that the learned performance prediction function cannot be applied to general programs, since our training and testing set consists of benchmark programs, whose detected SCoP only contains one or two statements. We do not know how the features and, hence, the performance prediction function perform on SCoPs with more than two statements.

# Chapter 6

# Conclusion

In this thesis, we proposed a methodology to obtain good distributed schedules from regions of the search space in a reasonable time with the geometric sampling approach. We were able to sample a huge number of schedules of the chosen benchmark programs for machine learning the performance prediction function.

For schedule comparison and simplification, we described the normalization steps that are used in the Polyite project and provided a mathematical proof of correctness. Furthermore, we enhanced the normalization with an additional step, which is necessary for correctness of the features calculations. The normalization is still not complete and requires further research.

This thesis proposes several features concerning parallelization, cache hit rate and overhead computations in Section 4.1. Most of them can be calculated in reasonable time. Only the performance of the cache feature, which is described in Section 4.1.2, exceeds the measured runtime of the generated programs from the schedules. According to our experiments, the correlation of the feature values depend on the SCoP, on which the feature is applied. The feature values behave different on various SCoPs. We further observed that a good cache hit rate (independent of the cache level) does not imply a low execution time.

We trained a performance prediction function with a huge number of schedules from different benchmark programs, including the *cholesky*, *gemm*, *seidel-2d*, *syr2k*, *syrk* and *trmm* benchmark from the Polybench Suite. The predicted execution time achieves only a weak correlation with the measured execution times depending on the trainings and testing schedules set. The trained prediction models are over-fitted to the trainings set.

**Future Work** The normalization steps from the Polyite project and this thesis can be further improved to obtain a complete normalization procedure, which enables total comparison of schedules of the same SCoP. A further research topic is to figure out, whether schedule normalization has any influence on the performance of the generated programs.

Further investigations in new or more accurate features can lead to a better correlating performance prediction function. As already mentioned in Section 4.1.6, a vectorization feature or a feature regarding the code size of the generated program can be introduced.

# Appendix A

# Tables

This Appendix contains tables that are used multiply times in this thesis. The first Table A.1 contains the cache specification of the benchmark CPU Intel Xeon E5 2600 v2. The second Table A.2 consists of the six benchmark programs from the Polybench Suite [5] that are used in most of the experiments, e.g., for training and validating the performance prediction function. The table shows, per benchmark, the number of statements and the overall number of data dependencies that are detected by Polly.

| Intel Xeon E5 2690 v2 | L1 Cache | | L2 Cache | L3 Cache |
|---|---|---|---|---|
| Implementation | instruction | data | unified | unified |
| Hierarchy | - | - | non-inclusive | inclusive |
| Access | private | private | private | shared |
| Associativity | 8-way | 8-way | 8-way | 20-way |
| Cache Size | 32KB | 32KB | 256KB | 25MB |
| Cache Line Size | 64B | 64B | 64B | 64B |

**Table A.1:** *Cache data of Intel E5 2690 v2 according to the manual [1, 2], programmers guide [3] or extracted with Linux.*

| benchmarks | #statements | #dependencies |
|---|---|---|
| cholesky | 2 | 4 |
| gemm | 2 | 2 |
| seidel-2d | 1 | 6 |
| syr2k | 2 | 2 |
| syrk | 2 | 2 |
| trmm | 2 | 4 |

**Table A.2:** *The six benchmark programs that are used for evaluation of the normalization steps, the features and the performance prediction function. Each row shows the benchmark name along with the number of statements and dependencies between them.*

# Appendix B

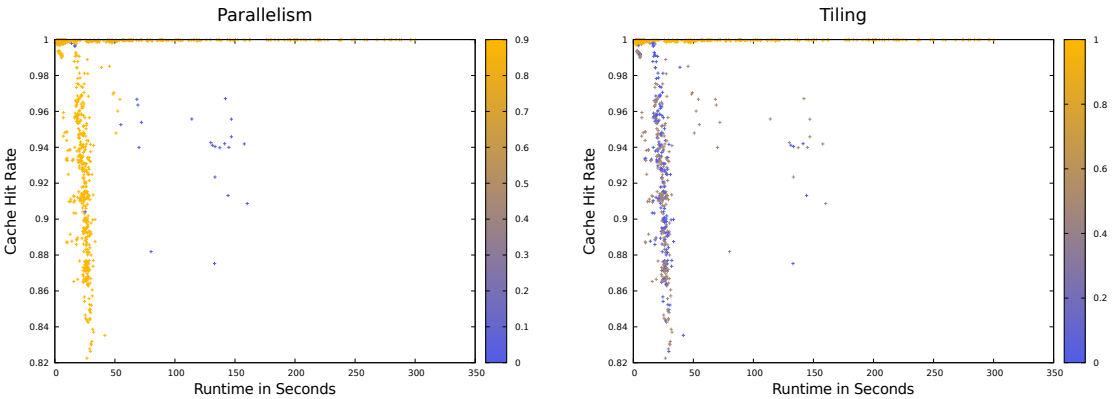
# Evaluation results

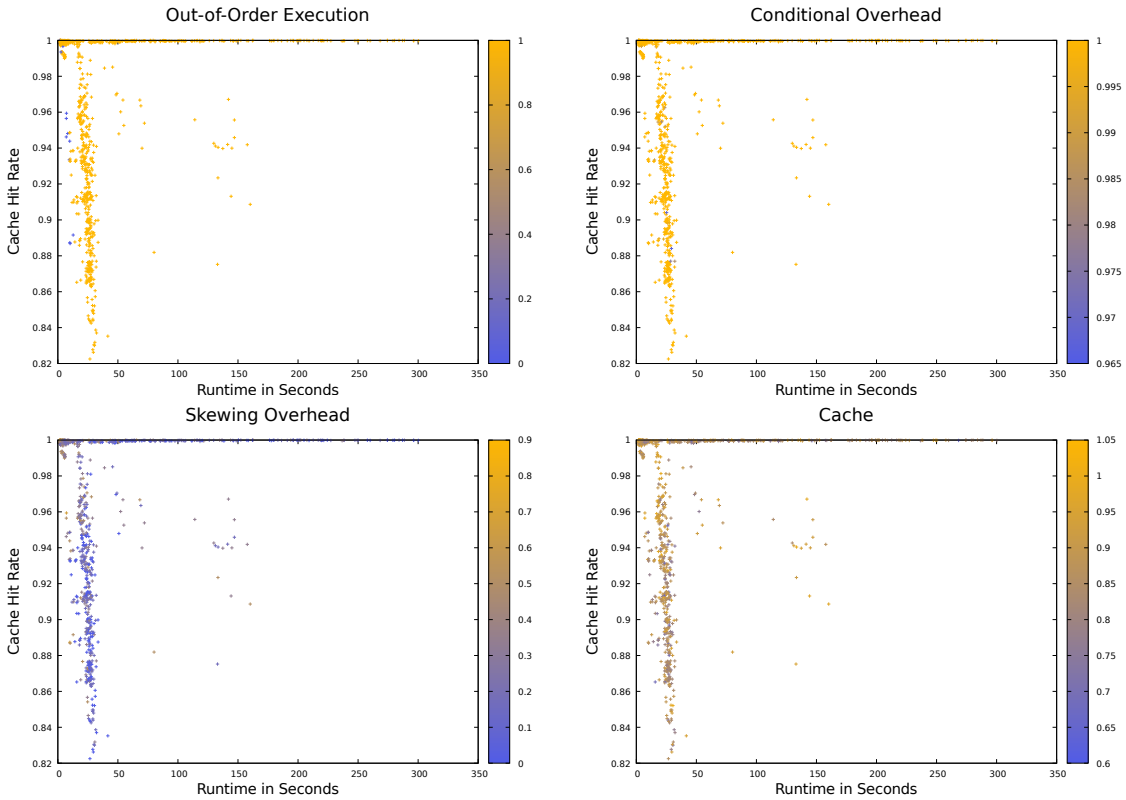
This Appendix contains several plots that show the result of the different feature applied to the benchmark programs *gemm*, *syr2k*, *syrk* and *trmm*. The x-axis of these diagrams specifies the measured parallel runtime and the y-axis specifies the measured overall cache hit rate. The color in the diagrams indicates the depicted feature value.

For all benchmarks, the cache feature is configured to use 27 instances per dependency. For the *syr2k* benchmark we have not measured the runtime of the cache feature, since it takes a too long calculation time.

## B.1 Benchmark: gemm

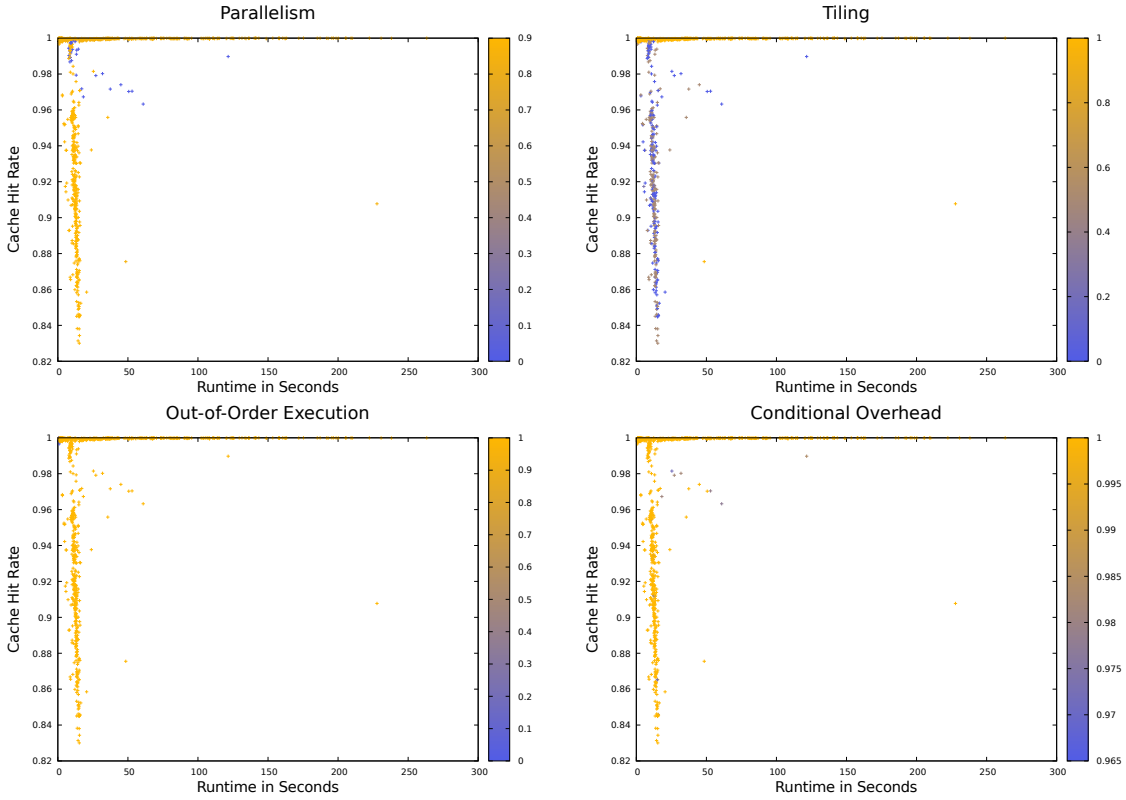
The *gemm* benchmark is a general matrix multiplication and part of the Basic Linear Algebra Subprograms (BLAS) [41]. Nearly all sampled schedules are capable of loop parallelization according to the parallelization feature, but not all of theses schedules perform fast. The tiling feature detects the programs with the best cache hit rate near 100 % very well. Neither the out-of-order execution feature, nor the conditional overhead feature detect any significant difference between the schedules. The left upper part of the plot regarding the skewing overhead feature contains more schedules with a better skewing feature value. As already shown in experiment E3 in Section 5.4.3, the cache feature with only 27 instances per produces no reliable output.

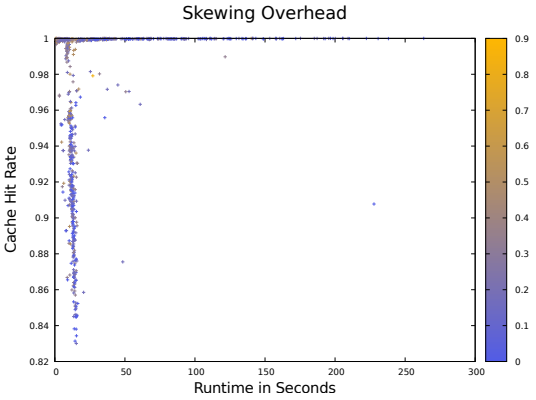


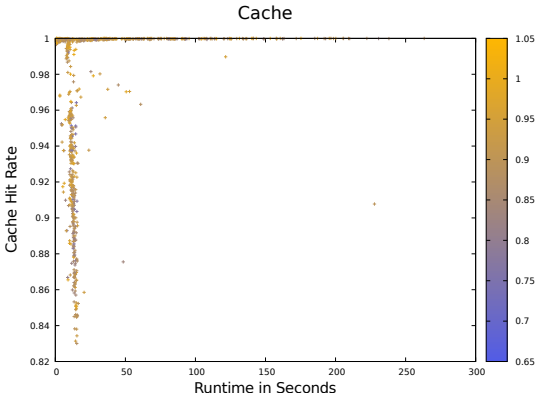



## B.2 Benchmark: syrk

The *syrk* benchmark is an symmetric rank-k update operation and also part of BLAS. The distribution of the schedules, the feature values look pretty similar to the *gemm* benchmark.

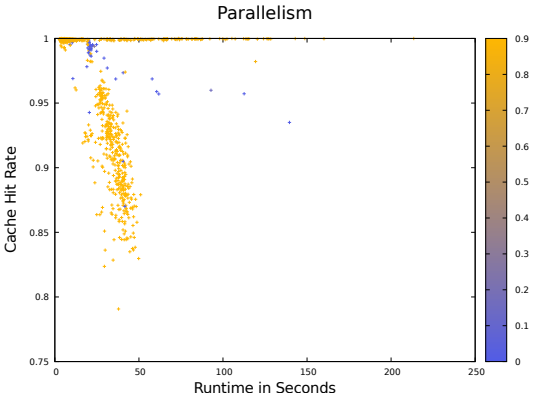


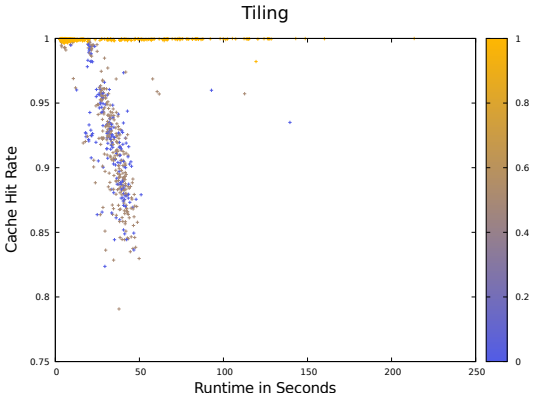


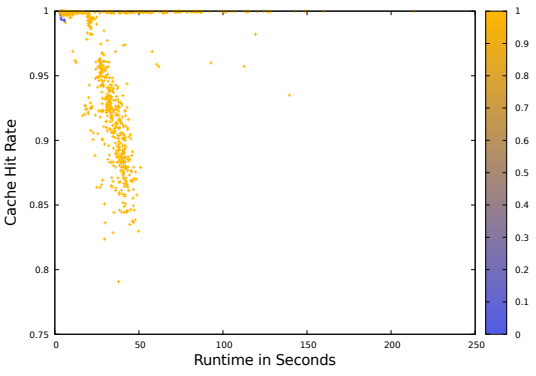



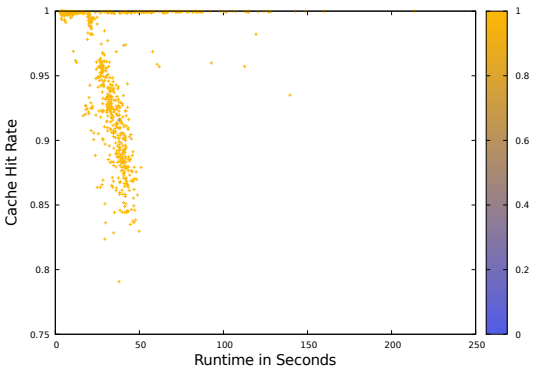

## B.3 Benchmark: syr2k

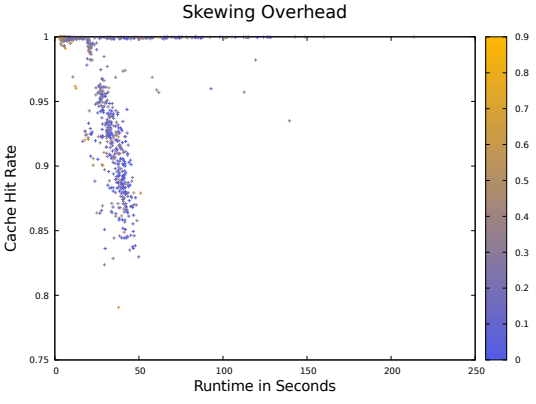
The *syr2k* benchmark is an symmetric rank-2k update operation and also part of BLAS. The distribution of the schedules looks slightly different from the previous benchmark programs, but the feature values are similar again.



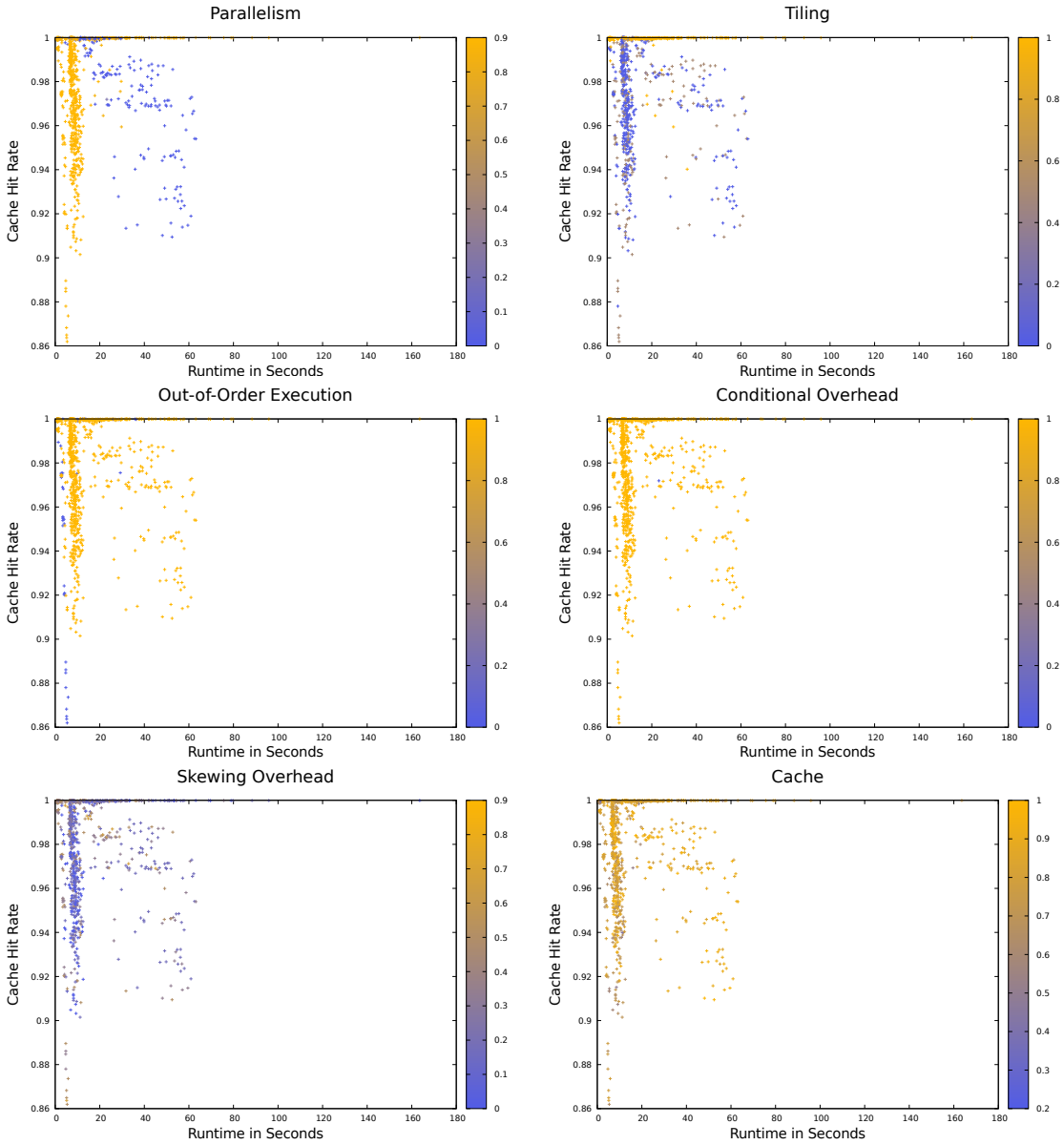




The cache feature is not calculated for this benchmark, because of performance issues.

## B.4 Benchmark: trmm

The *trmm* benchmark is a triangular matrix-matrix multiplication and part of BLAS. Most of the sampled schedules have parallel loops, but the schedules that cannot utilize parallelism perform worse than most of the other programs. The tiling feature, as with the other benchmarks, detects the schedules with a good overall cache hit rate. Similar to the previous benchmarks, the out-of-order feature and the conditional overhead feature values are equal for nearly all of the schedules. The left upper part of the plot regarding the skewing overhead feature contains more schedules with a good feature value than the rest of the plot. From the cache feature values it is not possible to draw any conclusions.



# Bibliography


[1] *Intel® Xeon® Processor E5-1600/2600/4600: Datasheet Vol. 1*, 2012.

[2] *Intel® Xeon® Processor E5 v2 Product Family*, 2012.

[3] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 2016.

[4] *Polly Project: LLVM Framework for High-Level Loop and Data-Locality Optimizations*, last visited February 2017. URL `http://polly.llvm.org/`.

[5] *Polybench/C the Polyhedral Benchmark Suite*, last visited February 2017. URL `http://web.cs.ucla.edu/~pouchet/software/polybench/`.

[6] *Scala Language*, last visited February 2017. URL `http://www.scala-lang.org/`.

[7] David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. In *Machine learning*, volume 6, pages 37–66. Springer, 1991.

[8] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling Imperfectly-nested Loop Nests. In *In Proc. of SC 2000*, page 31, 2000.

[9] Randy Allen and Ken Kennedy. Automatic Translation of Fortran Programs to Vector Form. volume 9, no. 4, pages 491–542. ACM, 1987.

[10] George Almási, Călin Caşcaval, and David A Padua. Calculating Stack Distances Efficiently. In *ACM SIGPLAN Notices*, volume 38, pages 37–43. ACM, 2002.

[11] Utpal Banerjee. A Theory of Loop Permutations. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 54–74. Pitman Publishing, 1990.

[12] Utpal Banerjee. *Unimodular Transformations of Double Loops*. University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1990.

[13] Alexander Barvinok and James E Pommersheim. An Algorithmic Theory of Lattice Points. volume 38, page 91. Cambridge University Press, 1999.

[14] Cedric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16. IEEE Computer Society, 2004.

[15] Stephen Baumert, Archis Ghate, Seksan Kiatsupaibul, Yanfang Shen, Robert L Smith, and Zelda B Zabinsky. Discrete Hit-and-Run for Sampling Points from Arbitrary Distributions over Subsets of Integer Hyperrectangles. volume 57, pages 727–739. INFORMS, 2009.

[16] Mohamed-Walid Benabderrahmane, Cédric Bastoul, Louis-Noël Pouchet, and Albert Cohen. A Conservative Approach to Handle Full Functions. Technical Report 6814, INRIA, 2008.

[17] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Automatic Transformations for Communication-minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction*, pages 132–146. Springer, 2008.

[18] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *ACM SIGPLAN Notices*, volume 43, pages 101–113. ACM, 2008.

[19] Călin Caşcaval and David A. Padua. Estimating Cache Misses and Locality Using Stack Distances. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 150–159. ACM, 2003.

[20] Călin Caşcaval, Luiz DeRose, David A Padua, and Daniel A Reed. Compile-time Based Performance Prediction. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 365–379. Springer, 1999.

[21] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. Exact Analysis of the Cache Behavior of Nested Loops. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 286–297. ACM, 2001.

[22] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. In *ACM Trans. Program. Lang. Syst.*, volume 23, pages 603–625. ACM, 2001.

[23] G. De Concini and D. De Martino. Overelaxed hit-and-run Monte Carlo for the uniform sampling of convex bodies with applications in metabolic network analysis. 2014.

[24] D. De Martino and Valerio Parisi. Monte Carlo uniform sampling of highdimensional convex polytopes: reducing the condition number with applications in metabolic network analysis. 2013.

[25] Pradip S. Devan and R.K. Kamat. A Review-LOOP Dependence Analysis for Parallelizing Compiler, 2014.

[26] Persi Diaconis, Gilles Lebeau, and Laurent Michel. Gibbs/Metropolis algorithms on a convex polytope. In *Mathematische Zeitschrift*, volume 272, pages 109–129. Springer, 2012.

[27] Paul Feautrier. Parametric Integer Programming. In *RAIRO Recherche Op'erationnelle*, volume 22, 1988.

[28] Paul Feautrier. Dataflow Analysis of Array and Scalar References. In *International Journal of Parallel Programming*, volume 20, pages 23–53. Springer, 1991.

[29] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. In *International journal of parallel programming*, volume 21, pages 389–420. Springer, 1992.

[30] Paul Feautrier and Christian Lengauer. *Encyclopedia of Parallel Computing*, chapter Polyhedron Model, pages 1581–1592. Springer US, 2011.

[31] Martin Griebl. Automatic Parallelization of Loop Programs for Distributed Memory Architectures, 2004.

[32] Martin Griebl and Christian Lengauer. Introducing Non-linear Parameters to the Polyhedron Model. In *Proc. 11th Workshop on Compilers for Parallel Computers (CPC 2004), Research Report Series*, pages 1–12, 2004.

[33] Martin Griebl, Paul Feautrier, and Christian Lengauer. Index Set Splitting. In *International Journal of Parallel Programming*, volume 28, pages 607–631, 1999.

[34] Armin Größlinger. Scanning Index Sets with Polynomial Bounds Using Cylindrical Algebraic Decomposition. *Number MIP-0803*, 2008.

[35] Armin Größlinger. Some Experiments on Tiling Loop Programs for Shared-Memory Multicore Architectures. In *Programming Models for Ubiquitous Parallelism*, number 07361 in Dagstuhl Seminar Proceedings, 2008.

[36] François Irigoin. *Encyclopedia of Parallel Computing*, chapter Tiling, pages 2040–2049. Springer US, 2011.

[37] Norman P Jouppi and David W Wall. *Available instruction-level parallelism for superscalar and superpipelined machines*, volume 17. ACM, 1989.

[38] Robert M Keller. Look-ahead processors. In *ACM Computing Surveys (CSUR)*, volume 7, pages 177–195. ACM, 1975.

[39] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Stanford, CA, 1995.

[40] David L Kuck. *Structure of Computers and Computations*. John Wiley & Sons, Inc., 1978.

[41] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. In *ACM Transactions on Mathematical Software (TOMS)*, volume 5, pages 308–323. ACM, 1979.

[42] Christian Lengauer. Loop Parallelization in the Polytope Model. In *CONCUR '93, Lecture Notes in Computer Science 715*, pages 398–416. Springer-Verlag, 1993.

[43] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. In *IBM Systems journal*, volume 9, pages 78–117. IBM Corp., 1970.

[44] Huseyin Onur Mete and Zelda B. Zabinsky. Pattern Hit-and-Run for sampling efficiently on polytopes. In *Operations Research Letters*, volume 40, pages 6–11. Elsevier, 2012.

[45] Huseyin Onur Mete, Yanfang Shen, Zelda B. Zabinsky, Seksan Kiatsupaibul, and Robert L. Smith. Pattern Discrete and Mixed Hit-and-Run for Global Optimization. In *Journal of Global Optimization*, volume 50, pages 597–627. Springer, 2011.

[46] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[47] Thomas Müller-Gronbach, Erich Novak, and Klaus Ritter. *Monte Carlo-Algorithmen*. Springer-Verlag, 2012.

[48] Andy Nisbet. GAPS: A Compiler Framework for Genetic Algorithm (GA) Optimised Parallelisation. In *High-Performance Computing and Networking*, pages 987–989. Springer, 1998.

[49] Andy Nisbet. GAPS: Genetic Algorithm Optimised Parallelization. In *Proc. Workshop on Profile and Feedback Directed Compilation*, 1998.

[50] Andy P Nisbet. Towards Retargettable Compilers-Feedback Directed Compilation Using Genetic Algorithms. In *of the 9th International Workshop on Compilers for Parallel Computers CPC2001*, 2001.

[51] Igor Pak. On Sampling Integer Points in Polyhedra. *Foundations of Computational Mathematics (Proceedings of SMALEFEST 2000)*, pages 319–324, 2002.

[52] Louis-Noël Pouchet. *Interative Optimization in the Polyhedral Model*. PhD thesis, University of Paris-Sud 11, 2010.

[53] Louis-Noël Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative Optimization in the Polyhedral Model: Part I, One-dimensional Time. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 144–156. IEEE, 2007.

[54] Louis-Noël Pouchet, Cédric Bastoul, John Cavazos, and Albert Cohen. A Note on the Performance Distribution of Affine Schedules. In *2nd Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion (SMART 08), Göteborg, Sweden*, 2008.

[55] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *ACM SIGPLAN Notices*, volume 43, pages 90–100. ACM, 2008.

[56] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.

[57] Fabien Quilleré, Sanjay V. Rajopadhye, and Doran Wilde. Generation of Efficient Nested Loops from Polyhedra. In *International Journal of Parallel Programming*, volume 28, pages 469–498, 2000.

[58] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1998.

[59] Andreas Simbürger, Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Armin Größlinger, and Louis-Noël Pouchet. Polly- Polyhedral optimization in LLVM.

[60] Kevin Stock, Louis-Noël Pouchet, and P Sadayappan. Using machine learning to improve automatic vectorization. In *ACM Transactions on Architecture and Code Optimization (TACO)*, volume 8, page 50. ACM, 2012.

[61] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of research and Development*, volume 11, pages 25–33. IBM, 1967.

[62] Wing N. Toy and Benjamin Zee. *Computer Hardware-Software Architecture*. Prentice Hall Professional Technical Reference, 1986.

[63] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.

[64] Sven Verdoolaege. Presburger Formulas and Polyhedral Compilation, 2016.

[65] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. Schedule Trees. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques. Vienna, Austria*, 2014.

[66] Frédéric Vivien. On the Optimality of Feautrier's Scheduling Algorithm. In *Concurrency and Computation: Practice and Experience*, volume 15, pages 1047–1068. Wiley Online Library, 2003.

[67] David W Wall. *Limits of instruction-level parallelism*, volume 19. ACM, 1991.

[68] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[69] Michael Wolfe. Loops Skewing: The Wavefront Method Revisited. In *International Journal of Parallel Programming*, volume 15, pages 279–293. Springer, 1986.

[70] Jingling Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.

# Statement of Authorship

I, Dominik Karl Danner, hereby certify that this master's thesis has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. All references and verbatim extracts have been quoted, and all sources of information have been specifically acknowledged. It has not been accepted in any previous application for a degree.

Passau, February 28, 2017

(Dominik Danner)