Iterative Schedule Optimization for Parallelization in the Polyhedron Model

STEFAN GANSER, University of Passau
ARMIN GRÖSSLINGER, University of Passau
NORBERT SIEGMUND, Bauhaus-University, Weimar
SVEN APEL, University of Passau
CHRISTIAN LENGAUER, University of Passau

The polyhedron model is a powerful model to identify and apply systematically loop transformations that improve data locality (e.g., via tiling) and enable parallelization. In the polyhedron model, a loop transformation is, essentially, represented as an affine function. Well established algorithms for the discovery of promising transformations are based on performance models. These algorithms have the drawback of not being easily adaptable to the characteristics of a specific program or target hardware. An iterative search for promising loop transformations is more easily adaptable and can help to learn better models. We present an iterative optimization method in the polyhedron model that targets tiling and parallelization. The method enables either a sampling of the search space of legal loop transformations at random or a more directed search via a genetic algorithm. For the latter, we propose a set of novel, tailored reproduction operators. We evaluate our approach against existing iterative and model-driven optimization strategies. We compare the convergence rate of our genetic algorithm to that of random exploration. Our approach of iterative optimization outperforms existing optimization techniques in that it finds loop transformations that yield significantly higher performance. If well configured, random exploration turns out to be very effective and reduces the need for a genetic algorithm.

CCS Concepts: • **Computing methodologies** → **Genetic programming**; • **Software and its engineering** → **Massively parallel systems**;

Additional Key Words and Phrases: Automatic loop optimization, genetic algorithm, OpenMP, parallelization, polyhedron model, tiling

ACM Reference format:
Stefan Ganser, Armin Größlinger, Norbert Siegmund, Sven Apel, and Christian Lengauer. 2017. Iterative Schedule Optimization for Parallelization in the Polyhedron Model. *ACM Transactions on Architecture and Code Optimization* 1, 1, Article 1 (July 2017), 26 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This work was partially supported by the German Research Foundation grant no. AP 206/4 and AP 206/6.
Authors' addresses: Stefan Ganser, Armin Größlinger, Sven Apel and Christian Lengauer, Faculty of Computer Science and Mathematics, University of Passau; Norbert Siegmund, Faculty of Media, Bauhaus-University, Weimar.

XXXX-XXXX/2017/07-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern computer architectures are designed for parallel execution. Yet, writing efficient parallel code by hand is cumbersome. Also, efficient parallel code is often platform-dependent [30]. As an alternative, compilers should be able to automatically parallelize and tune sequential code written in standard programming languages to exploit a given multi-core architecture. The polyhedron model [12] supports this goal by providing scheduling algorithms that infuse parallelism and other types of optimizations into loops. LLVM [22] with Polly [14] and GCC with Graphite [37] are compilers that can perform polyhedral program optimizations in a model-driven way.

To improve the model-driven approaches, one must empower the underlying model to identify profitable loop transformations. *Iterative optimization* is a way of traversing the search space of possible transformations and finding more profitable program transformations. Furthermore, iterative optimization can help to learn about the distribution of the quality of individual transformations in the search space of possible program transformations. Finally, search-based optimization can adapt to a given hardware and is, therefore, better suited to find an optimal transformation for a given architecture. The downside of iterative optimization is that, in most cases, it is more time consuming than model-based optimization and the search space may be too large for exhaustive exploration. Thus, it is often not practically applicable.

As it turns out, with its model-driven optimization, the speedup reached by Polly for 9 of the 30 benchmarks in PolyBench 4.1 [35] is only less than or equal to 1.05 in our experiments[1]. Such limitations in the optimization of program transformations motivate our approach of iterative polyhedral optimization to find loop transformations that yield higher performance and to get a better understanding of the characteristics of profitable transformations. Specifically, we enable the sampling of the search space of legal transformations at random and a more directed search via a tailored genetic algorithm, to identify faster schedules in a reduced amount of time. We aim particularly at transformations that enable tiling and parallelization.

We base our iterative optimization on the approach by Pouchet et al. [32, 33] of iteratively optimizing the sequential execution time of programs that can be represented in the polyhedron model. Pouchet's approach, which addresses only legal transformations, consists of an algorithm that narrows the search space of all legal loop transformations to a specific subset. Furthermore, Pouchet employs a decoupling heuristic for sampling the narrowed space at random with reasonable effort. The decoupling heuristic is based on experience. To reduce optimization time, Pouchet et al. present a genetic algorithm with custom genetic operators.

Pouchet's approach is promising but does not support the detection of transformations that enable tiling and parallelization. The main reason for narrowing the search space to a specific region is the huge number of legal transformations that would need to be evaluated to find a fast one. Later, Pouchet et al. extended their approach to iteratively explore the space of possibilities for loop distribution and fusion and subsequently optimize each loop nest in a model-driven way [34].

The nature of the purely iterative approach causes a dilemma. To optimize for tiling and parallelization, we must remove the restrictions narrowing the search space. Yet, the space is too large to be modeled and traversed entirely. Its exponential size results from the numerous orders in which data dependences may be carried [33]. We overcome the dilemma by dividing the search space into regions and by sampling these regions. The selection of the regions is driven by chance, but it can be configured such that outer parallel loops are likely to occur. To this end, we alter the

[1]For baseline measurements, we canonicalize the LLVM IR with `-polly-canonicalize` and then optimize the target SCoP with `-march=native -basicaa -scev-aa -polly-opt-isl -polly-codegen -polly-parallel=true -polly-tiling=true -polly-vectorizer=none -polly-default-tile-size=64`. Subsequently, we run `-O3 -march=native`. We use the Intel Xeon E-5 2650 v2 CPU @ 2.6GHz with eight physical cores for benchmarking.

existing algorithm for search space construction to sampling regions of the search space of legal transformations.

Pouchet's decoupling heuristic for sampling assumes a sequential execution. Consequently, to sample search space regions, we propose a novel strategy, relying on the Farkas-Minkowski-Weyl Theorem and Chernikova's algorithm (see Section 3). Here, it is important that our algorithm does not rely on performance models but makes solely use of the search space regions' geometric structure.

Finally, we present a set of novel genetic operators for finding profitable transformations, because the operators presented by Pouchet et al. [33] would not be able to traverse our unbounded search space of all legal transformations of a program.

Our evaluation on PolyBench 4.1 reveals that our approach outperforms existing model-driven and iterative techniques in that it finds loop optimizations that yield significantly higher performance of the resulting code. However, to our surprise, the transformations found by a well configured random exploration are very profitable and narrow the advantage of the genetic algorithm. From this, we conclude that profitable loop transformations can be detected more easily than expected. Future research must decide whether the superior transformations found by our iterative optimization share characteristics that can be incorporated by model-driven optimizers. All evaluation results and our implementation are available on a supplementary Web site[2].

In summary, we contribute the following:

- We extend Pouchet's algorithm for search space construction to enable the optimization of transformations that incorporate parallel execution with tiling.
- We propose a novel method to sample the search space of loop transformations. It employs the Farkas-Minkowski-Weyl Theorem and Chernikova's algorithm.
- We propose custom genetic operators to sample the search space of loop transformations with more guidance than at random.
- We provide the tool Polyite, which relies on LLVM's polyhedral code optimizer Polly to model programs in the polyhedron model, apply tiling, and generate optimized code. Polyite is written in Scala [29].
- We reimplemented the search space construction of Pouchet et al. [33] and combined it with our sampling strategy.
- We evaluate our genetic algorithm against a contemporary variant of PLuTo [5], a polyhedral scheduling algorithm that optimizes specifically for tiling and data locality. Furthermore, we compare our genetic algorithm against random exploration with our own search space construction and sampling strategy, and against random exploration with our reimplementation of Pouchet's approach.
- We evaluate the execution time of our search space construction.

2 POLYHEDRON MODEL

Our proposed schedule optimization relies on several concepts of the polyhedron model and its mathematical foundation. We introduce them in this section and borrow from Schrijver [36][Chap. 7, 8].

The *polyhedron model* [12] is a mathematical abstraction of loop programs. Program transformations can be applied in the model, and source code can be generated from the model. The model's power lies in its ability to express a potentially complex sequence of individual loop transformations, such as loop distribution/fusion [19], skewing [40], tiling [17], and index set splitting [13], in a single affine function.

[2] https://stganser.bitbucket.io/taco2017/

```
for (i = 0; i < ni; ++i)
 for (j = 0; j < nj; ++j) {
    C[i][j] = C[i][j] * beta; // Statement R
    for (k = 0; k < nk; ++k)
      C[i][j] = C[i][j] + alpha
          * A[i][k] * B[k][j]; // Statement S
 }
```

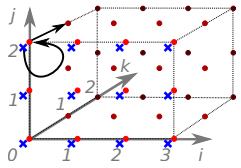


Fig. 1. On the left is a program with static control. On the right are the program's iteration domains together with data dependences. The crosses are the instances of statement R, the circles those of statement S. Of each of the two dependence vectors, only the respective instance originating at the upper left front corner is shown. The instances replicate throughout the entire domain.

Static Control. The polyhedron model puts restrictions on the kind of programs that can be optimized. To be able to model a code region, it must have *static control*. A code region with static control is called a *static-control part* or *SCoP*, for short. Loops should be `for` loops. A loop's limits must depend only on structure parameters, numerical constants, and iteration variables of the enclosing loops. *Structure parameters* are integer variables whose values remain unchanged inside a SCoP. The statements in a SCoP may operate on linear data structures (typically arrays). Loop limits and array indices must be affine functions of the iteration variables and structure parameters. Figure 1 shows a matrix multiplication as an example of a static-control program. A more restrictive definition of a SCoP would require the stride of all loops to be one [9]. Wider definitions of static control exist, too [3].

Iteration Domain. In the polyhedron model, each iteration or *instance* of a statement is represented by a point in a vector space whose dimensionality equals the number of nested loops. The coordinates reflect the values of the iteration variables. We refer to the coordinate vectors as the statement instances' *iteration vectors*. In case of a legal SCoP, the set of all instances of a statement S is a polyhedron $\{\vec{x} \mid A \cdot \vec{x} \leq \vec{b}\}$. The constraints that bound the set correspond to the original loop bounds. Figure 1 depicts a program together with its iteration domains.

Convex Polyhedra. One representation of polyhedra is the *constraint representation*, which was used in the previous paragraph. It is dual to the *generator representation* (Farkas-Minkowski-Weyl Theorem), in which each point inside polyhedron P is expressed as a convex combination of a set of points $\{\vec{v}_1, \ldots, \vec{v}_m\}$, each lying on a different minimal face of P, plus a conical combination of the rays $\{\vec{r}_1, \ldots, \vec{r}_n\}$ pointing in directions in which the polyhedron is unbounded: $P = \{\vec{v} + \vec{w} \mid \vec{v} \in \text{conv}\{\vec{v}_1, \ldots, \vec{v}_m\} \wedge \vec{w} \in \text{cone}\{\vec{r}_1, \ldots, \vec{r}_n\}\}$.

A *line* or bidirectional ray [23] combines two rays pointing in opposite directions. In the absence of lines, $\vec{v}_1, \ldots, \vec{v}_m$ are the vertices of P. In the remainder of the paper, for simplicity of nomenclature, we call the points $\vec{v}_1, \ldots, \vec{v}_m$ vertices, irrespective of their true nature. We call the vertices, rays, and lines *generators* of P. Chernikova's algorithm [36][Part III] allows us to switch between both representations. We reimplemented the extension of Le Verge [23].

Schedule. So far, we did not account for the execution order of the statement instances in a SCoP. Iteration domains alone cannot express execution order, since they only contain the statement instances' iteration vectors. Particularly, they do not express the textual order. An additional schedule is required. A *statement schedule* is a (multi-dimensional) affine function that assigns an execution date to each of a statement's instances. Given two instances of statements S and T with iteration vectors $\vec{i}$ and $\vec{j}$, the instance of S is executed before the instance of R, iff, for the one-dimensional statement schedules Θ_S and Θ_R, $\Theta_S(\vec{i}) < \Theta_R(\vec{j})$ applies. In the case of a multi-dimensional schedule (we may assume that all statement schedules have the same dimensionality),

the lexicographic order ($\prec$) of the result vectors specifies the execution order. Given two vectors $\vec{v} = (v_1, v_2, \ldots, v_n), \vec{v'} = (v'_1, v'_2, \ldots, v'_n)$ we have $\vec{v} \prec \vec{v'} \Leftrightarrow \exists i : 1 \leq i \leq n \wedge (\forall j : 1 \leq j < i \Rightarrow v_j = v'_j) \wedge v_i < v'_i$.

To express an n-dimensional schedule Θ_S of statement S, we use *schedule matrices*. Let $\vec{p}$ be the vector of all structure parameters and $\vec{i}_S$ the vector of the iteration variables of the loops that enclose S. Θ_S is expressible by the matrix $M_{\Theta_S} = (\vec{m}^1_{\Theta_S}, \ldots, \vec{m}^n_{\Theta_S})^T$ with row $\vec{m}^d_{\Theta_S} = (\vec{\lambda}^d_S, \vec{\mu}^d_S, \nu^d_S)$. $\vec{\lambda}^d_S$ is the coefficient vector of $\vec{i}_S$; $\vec{\mu}^d_S$ is the coefficient vector of $\vec{p}$; ν^d_S is the coefficient of the value 1.

Data Dependences. Other than by the iteration domain and schedule, a SCoP is characterized by the memory accesses of its statements. A *data dependence* from one statement instance, the source, to another, the target, exists if both access the same memory location and the schedule prescribes that the former statement instance is executed before the latter. A dependence is relevant iff one of the accesses is a write. Let statement O be the origin and T the target of a dependence. *Dependence polyhedron* $D_{O,T} \subseteq D_O \times D_T$ contains all pairs of statement instances of O and T for which the dependence exists. In some cases, this polyhedron over-approximates the true set of dependent statement instances. In the remainder of the paper, we treat the terms dependence and dependence polyhedron as synonyms. If we speak of a single instance of a dependence (i.e., a pair of statement instances), we always do so explicitly. Figure 1 depicts the data dependences in our example as arrows from source to target.

Data dependences make many schedules illegal. A dependence's source must be executed before its target. Therefore, we introduce terminology regarding the legality of schedules. We start with one-dimensional schedules. Dependence $D_{O,T}$ is *weakly satisfied* iff $\forall(\vec{i}, \vec{j})^T \in D_{O,T} : \Theta_T(\vec{j}) - \Theta_O(\vec{i}) \geq 0$ applies. $D_{R,S}$ is *strongly satisfied* iff $\forall(\vec{i}, \vec{j})^T \in D_{O,T} : \Theta_T(\vec{j}) - \Theta_O(\vec{i}) > 0$ applies. A set of one-dimensional statement schedules is *legal* if it strongly satisfies all data dependences that exist in the SCoP.

A set of multi-dimensional statement schedules of a SCoP is legal iff $\forall(\vec{i}, \vec{j})^T \in D_{O,T} : \Theta_T(\vec{j}) > \Theta_O(\vec{i})$ holds for each dependence $D_{O,T}$. The first schedule dimension, starting from dimension 1, that strongly satisfies a dependence *carries* the dependence. Geometrically seen, the directional vector of a dependence points forward in the direction of a dimension that strongly satisfies the dependence. It either points forward or is orthogonal to a dimension that weakly satisfies the dependence and points backward in the direction of a violating dimension.

Tiling. The loop transformation *tiling* improves data locality by grouping statement instances that are geometrically close to each other into so-called *tiles*. Grouping the statement instances changes their execution order. Tiling doubles the number of loops in the nest. The new outer loops enumerate the tiles. The inner loops iterate inside the tiles. A rectangular tiling of a sequence of schedule dimensions is legal iff the elements of the sequence are *permutable*: All dimensions must weakly satisfy the same data dependences. Such a sequence is called a *tilable band*.

Often, tilable bands cannot be found directly. Then, skewing [40] may help as an enabling transformation. *Skewing* shifts a loop's limits by a multiple of an enclosing loop's iteration variable. This changes the directional vectors of dependences and can make (rectangular) tiling of a loop nest legal.

3 SEARCH SPACE CONSTRUCTION

In contrast to Pouchet et al. [33], we sample the entire search space of legal affine schedule functions. We divide the search space into regions, such that one region contains all schedules carrying the data dependences in a specific order. Since the number of regions is exponential, we select only

a sample. Then, we can sample schedules from each selected region. We adapt Pouchet's search space construction accordingly.

We first outline Pouchet's algorithm for search space construction. Subsequently, we present our extensions for sampling different regions of the search space.

Schedule Representation. Schedules are represented as schedule matrices (see Section 2). Since statement schedules cannot be chosen independently, an evident approach is to aggregate them to one overall schedule of the entire SCoP. An n-dimensional schedule Θ of a SCoP with statements $S_1, \ldots, S_k$ is represented by the schedule matrix $M_\Theta = (\vec{m}^1_\Theta, \vec{m}^2_\Theta, \ldots, \vec{m}^n_\Theta)^T$ with $\vec{m}^d_\Theta = (\vec{\lambda}^d_{S_1}, \ldots, \vec{\lambda}^d_{S_k}, \vec{\mu}^d_{S_1}, \ldots, \vec{\mu}^d_{S_k}, \nu^d_{S_1}, \ldots, \nu^d_{S_k})$. We call the rows $\vec{m}^1_\Theta, \ldots, \vec{m}^n_\Theta$ of M_Θ *schedule coefficient vectors*. Per statement, each row $\vec{m}^d_\Theta$ contains a constant rational coefficient $\vec{\lambda}^d_{S_j}$ for each of the iteration variables of the loops surrounding the statement, $\vec{\mu}^d_{S_j}$ for each structure parameter, and $\nu^d_{S_j}$ for each constant. We must use rational coefficients due to the nature of our sampling strategy (see Section 4.2) and some of our genetic operators (see Section 4.3). Rational schedule coefficient vectors can be transformed into equivalent integer vectors (see Section 4.2).

Legal Schedules. Since the number of illegal schedules grows exponentially faster with the program size than the number of legal schedules [32], Pouchet et al. constrain the search space to legal schedules. The constraints originate from the data dependences that exist in the SCoP. For each data dependence $D_{O,T}$, Pouchet defines two sets of schedule coefficient vectors. Set $W_{O,T}$ contains all coefficient vectors that satisfy $D_{O,T}$ at least weakly. Set $S_{O,T}$ contains all coefficient vectors that satisfy $D_{O,T}$ strongly. Both sets are polyhedra. The constraints that bound them can be computed from $D_{O,T}$ with the affine form of Farkas Lemma [10]. The set of legal schedules is never empty, as it contains the original schedule.

Size of the Search Space. Apart from restrictions that originate from the specific set of dependences (e.g., it may not be possibly to carry two specific dependences in the same schedule dimension), a dependence can be carried by any schedule dimension, assuming that all preceding dimensions satisfy the dependence weakly. Depending on the choices made, the number of schedule dimensions and the legality constraints of the dimensions vary. This creates a combinatorial problem. To be precise, the size of the search space is exponential in the number of dependences. To reduce the search effort, Pouchet considers only schedules that carry the data dependences in a specific order. To achieve this, he sorts the dependences by a heuristic that involves primarily the volume of data communicated between the dependent statements. The following paragraph describes how the ordered list of dependences is processed.

Modeling the Schedule Search Space. The algorithm builds a model of the search space from a list G of all data dependences in the SCoP. G is ordered according to the aforementioned heuristic. The model consists of a list of polyhedra, with the i^{th} polyhedron containing the possible schedule coefficient vectors for schedule dimension i. As long as G is not empty, the algorithm adds another schedule dimension d. The algorithm creates a polyhedron P_d that fills the entire schedule coefficient space. Constraints that bound each dimension to the interval $[-1, 1]$ are added to P_d. Then, the algorithm intersects P_d with the sets $W_{O,T}$ associated with all $D_{O,T} \in G$. Now, P_d contains only schedules that weakly satisfy the dependences in G. Subsequently, the algorithm iterates through G and intersects P_d with the set $S_{O,T}$ of each dependence. If, for a dependence, the intersection is not empty, P_d is updated to the result of the intersection and the dependence is removed from G.

Extending Pouchet's Algorithm. Algorithm 1 outlines the extended search space construction. Differences from the original algorithm are highlighted.

Pouchet's algorithm for search space construction, described in this section, models a specific set of schedules as a list of polyhedra. All data dependences are carried by the outer dimensions of

all of these schedules. In other words, Pouchet's algorithm permits only inner parallelism, which is unacceptable if one optimizes for parallelization. Further, with parallelization and tiling in mind, skewing is a promising loop transformation (see Section 2). Following Pouchet, and limiting the value range of schedule coefficients to $[-1, 1]$, we would eliminate schedules with non-unit skewing from the search space. Also, many opportunities for loop distribution are lost. Yet, loop distribution can expose parallelism [19, 30].

Example 3.1. To better understand the limitations of Pouchet's algorithm, consider the SCoP from Figure 1. The schedule $\Theta_R(i,j) = (0,i,j,0); \Theta_S(i,j,k) = (1,i,j,k)$ is profitable, since it yields code where each statement is nested in a separate loop nest and where the outermost loop around each statement can be parallelized. Yet, Pouchet's approach would presumably produce $\Theta_S(i,j,k) = (1,k,i,j)$ (or an equivalent schedule), since the k loop around S carries dependence $D_{S,S}$. ◁

As mentioned, the full search space of legal schedules for a SCoP is dividable into regions, each containing the schedules that carry the data dependences in one order. As we cannot model the complete search space, we modify Pouchet's search space construction to sample the regions. Further, to allow for outer parallel loops, we must permit outer schedule dimensions that do not carry dependences.

We propose the following adaptations: We start from a set G of dependences instead of an ordered list. When constraining polyhedron P_d to schedules that carry data dependences, we consider only dependences from a randomly chosen set $G_d \subseteq G$. In our experiments we set the probability that G_d is not empty to 0.4. Subsequently, we remove all dependences from G that are effectively carried by all schedules in P_d. Leaving G_d empty permits P_d to contain schedules that encode parallel loops. To support non-unit skewing and easier loop distribution, we do not limit the value range of the schedule coefficients.

ALGORITHM 1: Search space construction

Input: G Set of dependence polyhedra for dependences between pairs of statements R and S
U (Universe) A polyhedron that fills the entire schedule coefficient space of the SCoP.
Output: For each schedule dimension d, a triple $C_d = (V_d, R_d, L_d)$ of generators that span the polyhedron P_d containing the allowed coefficient vectors $\vec{m}_\Theta^d$ for d
Parameters: p Probability that a schedule dimension carries, at least, one dependence

1 $d \leftarrow 0$; `// current schedule dimension`
2 **while** $G \neq \emptyset$ **do**
3 $\quad d \leftarrow d + 1$; $P_d \leftarrow U$;
4 $\quad$ **foreach** $D_{O,T} \in G$ **do**
5 $\quad\quad W_{O,T} \leftarrow \{\vec{m}_\Theta \mid \vec{m}_\Theta \in U \wedge \forall D_{O,T} \in G : \vec{m}_\Theta$ weakly satisfies $D_{O,T}\}$;
6 $\quad\quad P_d \leftarrow P_d \cap W_{O,T}$;
7 $\quad G_d \leftarrow \emptyset$; `// dependences to carry in schedule dimension` d
8 $\quad$ **if** `rand([0,1])` $\leq$ p **then**
9 $\quad\quad G_d \leftarrow$ `rand`$(2^G \setminus \{\emptyset\})$;
10 $\quad$ **foreach** $D_{O,T} \in G_d$ **do**
11 $\quad\quad S_{O,T} \leftarrow \{\vec{m}_\Theta \mid \vec{m}_\Theta \in U \wedge \forall D_{O,T} \in G : \vec{m}_\Theta$ strongly satisfies $D_{O,T}\}$;
12 $\quad\quad$ **if** $P_d \cap S_{O,T} \neq \emptyset$ **then**
13 $\quad\quad\quad P_d \leftarrow P_d \cap S_{O,T}$;
14 $\quad G \leftarrow G \setminus \{D_{O,T} | D_{O,T} \in G \wedge \forall \vec{m}_\Theta^d \in P_d : \vec{m}_\Theta^d$ strongly satisfies $D_{O,T}\}$;
15 $\quad C_d \leftarrow$ `chernikova`(P_d);

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

Final Representation of a Search Space Region. Originally, the polyhedra P_d are modeled in the constraint representation. The constraint representation is convenient for intersecting polyhedra, as done in Algorithm 1. To sample schedule coefficient vectors from these polyhedra, we convert them to their generator representation. This representation is convenient for sampling schedules, since P_d can be sampled by choosing a random combination of its generators. Specifically, each polyhedron P_d is finally represented by a triple $C_d = (V_d, R_d, L_d)$ of the vertices (V_d), rays (R_d), and lines (L_d) that span P_d. We compute the generators using Chernikova's algorithm (see Section 2).

4 SEARCH SPACE EXPLORATION

So far, we have described how we sample regions of the schedule search space. Now, we present a novel method of generating random samples from the generator representation of a search space region.

In addition to using random exploration, we adapt the genetic algorithm by Pouchet et al. [33]. The general schema can be reused, but, as already mentioned, we require new mutation and crossover operators. The existing operators are not applicable in our setting, since they do not enable a traversal of the unrestricted search space. Similar to Pouchet, we weaken the changes made by our mutation operators over time to stay inside promising search space regions at the end of the optimization process.

4.1 Outline of the Genetic Algorithm

A *genetic algorithm* [26] (GA) is an iterative procedure starting from a random population of candidate solutions. In our case, the candidate solutions are schedule matrices. GAs perform a directed random walk and, thus, help to find good solutions faster. The iterations of the GA are the algorithm's *generations*.

Pouchet's genetic algorithm starts from a random population of legal schedules. In each iteration, the GA evaluates the current population, generating code from each schedule and measuring the execution time of the code as the fitness function. The fitter half of the population survives. To reach the full population size again, the schedules that survived from the parent population are mutated and crossed. Using an annealing factor, Pouchet weakens the changes made by mutation operators with time.

Pouchet's genetic operators are not customized for specific loop transformations, but based solely on the geometric structure of the search space. His narrowed search space is closed under the mutation and crossover operators.

We follow the idea of not aiming for specific loop transformations, since it agrees with the philosophy of polyhedral optimization not to apply a sequence of distinct transformations, but to map each statement instance individually to an execution date. Our new genetic operators can move from one search space region to another to reach schedules that are not in the search space regions visited before. The set of legal schedules remains closed under the operators.

Further, we adapt the combination of the GA with the idea of simulated annealing [7, 16]. The idea is to cover many different regions of the search space early in the optimization process. Later, the changes to the schedules must diminish as the reachable optimum is approached. Yet, to avoid being trapped at local optima, a small number of random schedules is introduced in each generation.

4.2 Generating Random Schedules

We continue from the list of triples $[C_1, ..., C_n]$ produced by Algorithm 1. Triple C_d consists of the generators that span the polyhedron P_d, which matches a set of legal schedule coefficients for schedule dimension d. Algorithm 2 takes the list of triples $[C_1, ..., C_n]$ and builds a random schedule matrix $M_\Theta = (\vec{m}_\Theta^1, \vec{m}_\Theta^2, ..., \vec{m}_\Theta^n)^T$. Row $\vec{m}_\Theta^d$ of M_Θ is a linear combination of the generators

of C_d. The choice of generator coefficients is independent for each schedule dimension. The linear combinations meet the criteria described in Section 2. Therefore, for each d, coefficient vector $\vec{m}_\Theta^d$ lies inside polyhedron P_d, and M_Θ represents a legal schedule of the SCoP.

ALGORITHM 2: Construction of random schedules

Input: For each schedule dimension d, $d = 1, \ldots, n$, a triple (V_d, R_d, L_d) containing the vertices, rays, and lines that span the polyhedron containing the legal schedule coefficients d

Output: A schedule matrix $M_\Theta = \left(\vec{m}_\Theta^1, \vec{m}_\Theta^2, \ldots, \vec{m}_\Theta^n\right)^T$

Parameters: rRange, lRange: Bounds for the coefficients of rays and lines

1 **for** $d \leftarrow 1$ **to** n **do**
2 $\quad \vec{m}_\Theta^d \leftarrow$ `rand`(V_d); `// Choose one vertex`
3 $\quad R_\Theta^d \leftarrow$ `rand`(2^{R_d}); `// Choose random rays from` R_d
4 $\quad (\eta_1^d, \ldots, \eta_{|R_\Theta^d|}^d) \leftarrow$ `rand`$(([1, \text{rRange}] \cap \mathbb{N})^{|R_\Theta^d|})$; `// Choose coefficients for selected rays`
5 $\quad \vec{m}_\Theta^d \leftarrow \vec{m}_\Theta^d + \sum_{\vec{r}_i \in R_\Theta^d} \eta_i^d \cdot \vec{r}_i$;
6 $\quad L_\Theta^d \leftarrow$ `rand`(2^{L_d}); `// Choose random lines from` L_d
7 $\quad (\vartheta_1^d, \ldots, \vartheta_{|L_\Theta^d|}^d) \leftarrow$ `rand`$((([-\text{lRange}, \text{lRange}] \cap \mathbb{N}) \setminus \{0\})^{|L_\Theta^d|})$; `// Choose line coefficients`
8 $\quad \vec{m}_\Theta^d \leftarrow \vec{m}_\Theta^d + \sum_{\vec{l}_i^d \in L_\Theta^d} \vartheta_i^d \cdot \vec{l}_i^d$;
9 $M_\Theta \leftarrow \left(\vec{m}_\Theta^1, \vec{m}_\Theta^2, \ldots, \vec{m}_\Theta^n\right)^T$;

Function `rand` selects randomly an element from a given set. Elements are chosen with uniform probability.

The use of integer coefficients for the generators may lead to unreachable points inside the polyhedra P_d. Without further effort, admitting rational coefficients for the generators would result in rational schedule coefficients. Generally, this is not a problem since a schedule coefficient vector with rational elements can be transformed to a vector with only integer elements by multiplying it with the lowest common denominator (LCD) of its elements. Both impose the same relative order on the statement instances. But multiplying with the LCD may lead to very large schedule coefficients, which may trigger integer overflows in the generated code. Therefore, generator coefficients must be small integers. As Example 4.1 illustrates, this requirement prevents Algorithm 2 from being able to reach some schedules.

Example 4.1. The polyhedron $P = \{(i, j) \mid -i + j + 1 \geq 0 \wedge 2 \cdot i + j - 5 \geq 0\}$ is spanned by vertex $(2, 1)^T$ and rays $(-1, 2)^T, (1, 1)^T$. Point $(2, 2)^T$ corresponds to the combination $(2, 1)^T + 1/3 \cdot (-1, 2)^T + 1/3 \cdot (1, 1)^T$. Rational coefficients are required for the rays to reach this point. Any multiple of $(2, 2)$ lies inside P as well. But the following equation cannot be solved as an integer linear program: $\alpha \cdot (2, 2)^T = (2, 1)^T + \beta \cdot (-1, 2) + \gamma \cdot (1, 1)^T, \alpha \in \mathbb{Z}, \beta, \gamma \in \mathbb{N}$. So, in the context of scheduling, there is no equivalent coefficient vector that can be computed only from the P's generators and integer coefficients. ◂

Some of the genetic operators described later on still introduce rational generator coefficients to reach otherwise unreachable areas of the search space.

From experiments with the PolyBench 4.1 benchmarks, we know that vertices with rational coordinates are rare (see Table 1). The sum of all vertices' coefficients in the linear combination forming a schedule coefficient vector must be 1. Since integer schedule coefficients are preferable, we use a single vertex as the basis of a randomly generated schedule coefficient vector. Then, we add a uniformly distributed number of rays and lines. The maximum number of rays and lines can be bounded to avoid a too high density of schedule matrices. The coefficients of the rays and lines are chosen randomly from $(0, \text{rRange}] \cap \mathbb{N}$ and $([-\text{lRange}, \text{lRange}] \cap \mathbb{N}) \setminus \{0\}$, respectively.

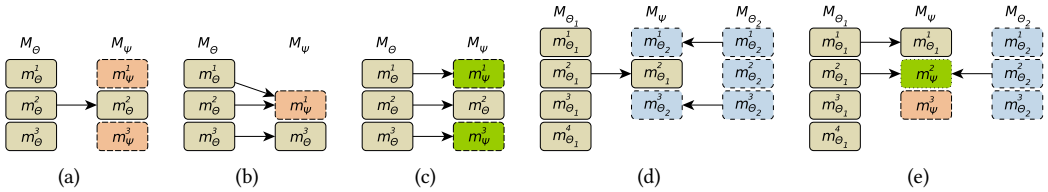


Fig. 2. Illustration of schedule mutation and crossover operators: (a) dimension replacement; (b) prefix replacement; (c) generator coefficient replacement; (d) row-based crossover; (e) geometric crossover

Permitting arbitrarily large coefficients seems to be not beneficial and, as already mentioned, may lead to erroneous programs. Still, the intervals must not be too narrow, as this would exclude profitable opportunities for loop distribution and non-unit skewing (shifting the limits of a loop by a multiple of the iteration variable of an enclosing loop): again, we know from experiments with PolyBench that the components of rays and lines are mostly from $\{-1, 0, 1\}$ (see Table 1). To distribute a larger number of statements across many loops without having to insert many constant dimensions into the schedule or to skew with a bigger factor, we need an accordingly wide range for the generators' coefficients. We use $[1, 3]$ for ray coefficients and $[-3, 3] \setminus \{0\}$ for line coefficients.

Schedule Completion. The schedules that result from our sampling strategy, which consists of Algorithm 1 and Algorithm 2, may not be injective. Entire loops may not be encoded explicitly. Polyhedral code generators, such as isl [15], will choose one possible order and generate code accordingly. Yet, without coding each loop into the schedules, the iterative optimization would not have control over inner loops. Especially, we could not tile these loops. Thus, we append rows to each randomly generated schedule matrix until we cannot find, for any of the statements, another row that is linearly independent with the previous rows in the iteration variable coefficients. This way, we ensure that each loop of the generated code is encoded explicitly in the schedule. We call this process *schedule completion*. Be aware that our process differs from the schedule completion described by Pouchet et al. [33].

The randomly generated schedule matrices contain degenerate rows. For instance, rows occur that encode loops with a single iteration. Also, different schedule matrices can encode the same execution order. Consequently, before analyzing schedules or further transforming them, the schedules should be simplified. Section 5 describes the details of schedule simplification. We do not alter the original schedule matrices on which the GA operates, except for removal of rows that do nothing but assign the same constant offset to all statements. We also do not eliminate pairs of equivalent schedules from the population, because their identification is computationally expensive.

4.3 Schedule Mutation and Crossover

The design of suitable mutation and crossover operators is an essential challenge of crafting a genetic algorithm [7]. In our case, it is especially important to be able to reach any region of the search space of legal schedules.

The goal of mutation and crossover is to generate an offspring from one or two previous schedules. The new schedule should retain some properties of its ancestors but differ in others. The operators must be able to produce target schedules that satisfy the dependences in a different order than the source schedules. This reaches schedules from search space regions that have not been visited before. The operators must also be able to visit schedules that cannot be produced by Algorithm 2,

due to the algorithm's restriction to integer coefficients for the Chernikova generators. Analogous to random schedule generation, each schedule that results from a mutation or crossover is refined by schedule completion.

Figure 2 depicts our mutation and crossover operators graphically. Let us look at each operator in detail. We start with the mutation operators.

A *mutation operator* maps a schedule matrix M_Θ to a new matrix M_Ψ with the same number of columns, but possibly a different number of rows.

Dimension Replacement. The operator replaces randomly chosen rows of a schedule matrix with randomly generated ones (see Figure 2(a)). The idea is to eliminate inefficient rows from a generally profitable schedule.

The procedure iterates across the rows of the original schedule matrix and either copies an existing row to the new matrix or generates a new row. For replacing row i, we perform one iteration of Algorithm 1. Here, G is the set of dependences that are not carried by rows $0, \ldots, i-1$ of M_Ψ. G_d is fixed to the dependences that are carried by the row to be replaced, except for those carried by rows $0, \ldots, i-1$ of M_Ψ. From the resulting polyhedron, we sample a new coefficient vector with an iteration of Algorithm 2. Here, Algorithm 2 is configured as during the generating of the initial population. If rows $0, \ldots, i-1$ of M_Ψ carry all dependences already, we generate the new row analogously to schedule completion (see Paragraph "Schedule Completion"). The set of legal schedules is closed under dimension replacement by the definition of the operator.

Simulated annealing controls the number of replaced schedule dimensions.

Schedule Prefix Replacement. This operator replaces a prefix of a schedule matrix, that is, the first rows (see Figure 2(b)). The prefix length is chosen randomly. To generate the new prefix, we run the familiar sequence of Algorithms 1 and 2, but modify Algorithm 1 such that it chooses the sets G_d as subsets of the set of dependences carried by the original schedule prefix. The modified algorithm terminates, as soon as all of these dependences are carried. The new schedule prefix carries all dependences that are carried by the replaced prefix. Thus, the new schedule is legal. By coincidence, the new prefix may carry more dependences than the original prefix. The new schedule may be located in a different region of the search space. Schedule prefix replacement can particularly alter the partition of statements into loop nests.

Schedule Suffix Replacement. To generate a new suffix of a schedule matrix, we drop rows from the bottom of the matrix and generate the new suffix by running Algorithms 1 and 2. Algorithm 1 starts from the set of dependences that are not carried by the remaining schedule dimensions.

In case of schedule prefix replacement and schedule suffix replacement, simulated annealing controls the length of the replaced prefix or suffix.

Generator Coefficient Replacement. The idea is to replace a row of a schedule matrix by a very similar row (see Figure 2(c)). This operator can, for instance, reverse a loop by negating an iteration variable's coefficient. The operator is similar to dimension replacement, but its effects are more fine-grained.

Each row of a schedule matrix is stored together with the linear combination of the generators that form it. The generators originate from a set of generators that span a set P_d of schedule coefficient vectors. All coefficient vectors in P_d weakly or strongly satisfy defined sets of dependences. Replacing coefficients of the generators in accordance with the constraints given in Section 2 yields a new coefficient vector that again satisfies these dependences. Dependences that the original row carries by coincidence may not be carried by the new row. Replacing generator coefficients with rational numbers allows us to reach schedule coefficient vectors that cannot be reached by

Algorithm 2. The denominators of the new generator coefficients are small. Otherwise, coefficients of an equivalent schedule with integer coefficients will be too large.

Interactions with other genetic operators and the way in which random schedules are generated, may cause this operator to produce illegal rows. In this case, the suffix of the new schedule matrix must be replaced starting from the first illegal row.

Simulated annealing controls the number of mutated rows and for each mutated row, the portion of generator coefficients that change.

Next, we describe our crossover operators. A *crossover operator* takes two schedule matrices M_{Θ_1} and M_{Θ_2} and produces a new schedule matrix M_Ψ.

Row-Based Crossover. As illustrated by Figure 2(d), row-based crossover recombines the rows of M_{Θ_1} and M_{Θ_2} to produce M_Ψ. The idea is to inject new rows that are produced by other operators into more already profitable schedules. For each dimension, the operator chooses a row either from M_{Θ_1} or from M_{Θ_2}. The process starts from the first row and continues for the subsequent dimensions. The iteration terminates as soon as the new schedule carries all dependences. Due to the choices already made for rows $1, \ldots, i-1$, it may be illegal to choose row i from M_{Θ_1} or M_{Θ_2}, as that row may not weakly satisfy one of the dependences that are not carried by rows $1, \ldots, i-1$ of M_Ψ. In this case, we have to step back to row $i-1$ and make a different choice there. If all choices for row $i-1$ have been found to be illegal, we have to step back further. The idea for a uniform, row-based crossover originates from Pouchet et al. [33].

Geometric Crossover. The idea of geometric crossover is to take row d from both M_{Θ_1} and M_{Θ_2} and construct a new row $\vec{m}_d^{\,\Psi}$ as a linear convex combination of the two existing rows (see Figure 2(e)). This can reach schedule coefficient vectors that cannot be reached by a random exploration with Algorithm 2.

We rely on the fact that a linear convex combination of two schedule coefficient vectors weakly satisfies all dependences that are weakly satisfied by both of the original schedule coefficient vectors. This kind of crossover is legal under the following condition. Let $U^d_{\Theta_1}$ be the set of schedules that are not carried by dimensions $1, \ldots, d-1$ of Θ_1. $U^d_{\Theta_2}$ is defined analogously. If $U^d_{\Theta_1} \subseteq U^d_{\Theta_2}$, then $(\vec{m}^1_{\Theta_1}, \ldots, \vec{m}^{d-1}_{\Theta_1}, \vec{m}^d_\Psi)^T$ is a legal schedule prefix for a new schedule matrix M_Ψ, since $\vec{m}^d_\Psi$ weakly carries all dependences in $U^d_{\Theta_1}$. We must complete M_Ψ with a new suffix. Generating the new suffix is analogous to schedule suffix replacement after the removal of the old suffix.

Generator coefficient replacement and geometric crossover are able to reach schedules that cannot be reached by Algorithm 2. On the other hand, they may produce schedule matrices with rational coefficients. To actually generate code from these schedules, the rows must be replaced by equivalent rows with integer coefficients. As outlined in Section 2, we do this by multiplying each row with the lowest common denominator of its components before passing a schedule to the polyhedral code generator.

5 SCHEDULE TREE GENERATION AND SIMPLIFICATION

Schedule matrices are convenient for sampling the schedule search space but less suitable for transformation and analysis of schedules. To apply tiling, one considers each loop nest individually. This requires the identification of the partial order induced by a schedule dimension on the set of statements. Unlike schedule matrices, schedule trees [15] can express this property directly. Furthermore, since our schedule matrices are randomly generated, they contain coefficients that influence neither the execution order of statement instances, nor the directional vectors of dependences. Yet, they complicate the identification of tilable schedule bands and should be removed.

We use only some of the several node types of a schedule tree. A *sequence node* has a list of child nodes and specifies that its children must be executed in the given order. Each child is preceded by a *filter node*, which selects the statement instances that are scheduled in the subtree. A *band node* contains a partial (multi-)dimensional schedule for the statement instances that are scheduled in the present (sub-)tree. A band node may be marked as permutable to signal that its schedule dimensions may be tiled.

The schedule tree construction is recursive. Starting from the first schedule dimension, we test whether the current dimension partitions the set of statements into sets with different execution date. We must consider only the order of statement instances that have the same execution date according to previous dimensions. If the partitioning has more than one element, each element corresponds to a subtree of a new sequence node. We split the iteration domain and the schedule matrix accordingly and continue recursively for each partition or child of the sequence node. If a schedule dimension does not result in a sequence node, we create a one-dimensional band node and continue with the next dimension. Partial schedules of band nodes are simplified by removing constant offsets that apply to every statement and dividing the schedule coefficients by their greatest common denominator. Band nodes are obsolete if any two statement instances, that have the same execution date in the previous schedule dimensions, are also assigned the same execution date by the current band node's schedule. An entire subtree can be removed if its ancestors describe an injective schedule of the statement instances scheduled in the subtree. Finally, we group consecutive one-dimensional band nodes into multi-dimensional band nodes if their partial schedules are permutable.

6 EVALUATION

We aim at finding more profitable schedules than can be found by model-based scheduling algorithms such as PLuTo. Furthermore, we expect that our approach also outperforms the existing iterative optimization technique of Pouchet et al. in terms of profitability of the detected loop transformations, since they focus on sequential execution. With our proposed genetic operators, we expect to find more profitable schedules than with random search and find them faster. We define the following research questions to shed light on our expectations and quantify the benefits of our approach:

RQ1. *Does our approach find loop transformations that yield higher performance than the transformations found by state-of-the art approaches to polyhedral schedule optimization?* We evaluate whether our iterative search produces similar or better program schedules than state-of-the art approaches. Answering this question gives us a baseline and demonstrates the practicality of our approach.

RQ2. *Is our augmentation of the schedule search space justified?* In contrast to Pouchet, we explore potentially the entire space of legal program transformations. We evaluate whether this expansion is actually meaningful (in terms of finding better schedules) as it complicates the optimization process considerably. This question relates to Section 3. The question also concerns schedule completion as described in Section 4.2.

RQ3. *Does optimization with our genetic algorithm have an advantage over random sampling?* We evaluate whether a simple random search on the space of legal schedules can find schedules that perform equally well as our genetic algorithm. Again, answering this question provides a comparison to a baseline, which is often used for related problems due to their simplicity. The question relates to Section 4.

RQ4. *Do schedule matrices with high sparsity yield better performance than dense matrices?* This question sheds some light on the decisions made for the configuration of the experiments. We can control the maximum number of rays and lines that Algorithm 1 adds to a vertex to form a schedule

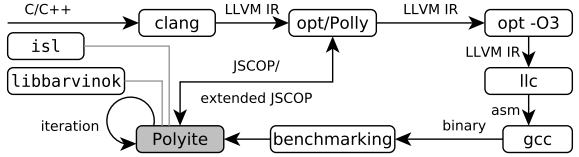


Fig. 3. Our tool chain for iterative schedule optimization

coefficient vector. Here, we explore whether it is better to add only a small number of rays and lines, which yields schedule matrices with high sparsity, or whether we should leave the number unspecified.

RQ5. *Does our iterative approach scale for very large SCoPs?* To evaluate whether our approach scales for realistic settings, we analyze the scaling factors of Algorithms 1 and 2 with respect to the size of SCoPs.

RQ6. *Do similar schedules yield similar performance?* The use of simulated annealing in the design of our GA is based on the assumption that similar schedules yield similar performance (see Section 4.1). So, we evaluate whether this assumption is reasonable.

RQ7. *Is our configuration of the genetic algorithm justified?* The presence of many tuning options is an intrinsic characteristic of iterative optimization tools. Their number makes it hard to find an optimal setup. Hence, we investigate whether the setup that we have chosen for the evaluation of our tool is actually well performing.

To answer these questions, we conducted experiments on PolyBench 4.1. PolyBench is a benchmark suite that is widely used in the polyhedral community. First, we describe our experimental setup. Then, we present the results of the experiments and answer the research questions. Finally, we discuss threats to validity.

6.1 Experimental Setup

We implemented our approach in the tool Polyite [pəˈlīt]. Polyite relies on Polly for SCoP extraction, modeling of SCoPs, tiling, and code generation. We handle polyhedra with the Integer Set Library (isl, commit `cfebc0c6`) [39]. Polyite uses libbarvinok (version 0.39) to mimic the behavior of Pouchet's approach. Precisely, we use Barvinok's counting algorithm to calculate the communication volume of data dependences. Polyite is written in Scala (version 2.11).

Tool chain. Polly is a polyhedral optimizer built on top of LLVM. Polly extracts the model of valid SCoPs from programs. Polly can apply externally generated schedules to SCoPs. The used data format is JSCOP, which is based on isl union maps. Internally, Polly uses the schedule tree representation. Polly is unable to tile imported schedules, though. We extend JSCOP with a schedule tree representation and enable Polly to tile imported schedules. Figure 3 provides an overview of our tool chain.

LLVM supports stepwise compilation. The LLVM C-/C++-front-end clang transforms C into LLVM IR. Optionally, the IR code can be written to a file. LLVM's opt tool allows us to apply individual transformations and analyses to LLVM IR. We extract the polyhedral model of the target SCoP into a JSCOP file using Polly's SCoP detection. Then, Polyite loads the JSCOP file and generates schedules. It generates a binary for each schedule, using Polly/ opt, llc, and gcc. llc transforms LLVM IR into assembly code. Polly tiles the schedules and generates OpenMP-parallelized loops. In our experiments, measured execution time of the optimized SCoP serves as

Table 1. Characteristics of the benchmarks used. The data in columns 6-9 originates from the analysis of P_1 of 1000 search space regions per benchmark. The data in column 10 is from an analysis of the schedule coefficient vectors produced during random exploration in Exp. 3.

benchmark	# stmts	# deps	max. loop depth	# structure params	% unit vectors	avg. # lines	avg. # rays	avg. # vertices	% rational vertices
2mm	4	6	3	7	99.84%	9	19	1	1.00%
3mm	6	10	3	9	99.90%	10	31	1	0.90%
adi	9	64	3	3	83.06%	4	27	4	2.02%
bicg	2	4	2	3	100.00%	4	6	1	0.67%
correlation	13	22	3	3	96.26%	13	40	1	0.54%
covariance	7	12	3	3	99.50%	4	26	1	0.50%
doitgen	3	8	4	5	99.02%	6	1	1	0.16%
fdtd-2d	4	24	3	4	92.08%	5	32	3	10.16%
gemm	2	2	3	5	100.00%	8	7	1	1.78%
gemver	4	6	2	2	99.72%	3	15	1	1.17%
gesummv	3	3	2	2	100.00%	4	7	1	0.56%
heat-3d	2	186	4	2	47.34%	3	15	1	16.69%
jacobi-2d	2	56	3	3	67.64%	4	6	1	12.42%
mvt	2	2	2	2	100.00%	8	2	1	2.14%
seidel-2d	1	59	3	3	85.44%	4	3	1	9.18%
syr2k	2	2	3	4	100.00%	7	7	1	1.94%
syrk	2	2	3	4	100.00%	7	6	1	2.30%
trmm	2	4	3	4	83.95%	6	9	1	2.15%

the fitness of the respective schedules. We use LLVM 3.9 and GCC 5.4. Our extended version of Polly is based on commit `2b618e01` of http://llvm.org/git/polly.git.

Benchmarking of Schedules. We benchmark each binary five times on an Intel Xeon E-5 2650 v2 CPU @ 2.6GHz with eight physical cores and 20MB of L3 cache. We take the shortest measured execution time. Hyperthreading and Intel Turbo Boost are disabled. The OS is Ubuntu 16.04 with Linux kernel 4.4. Schedules yielding invalid computation results are purged. Compilation must not exceed five minutes. The overall benchmarking of a schedule may take 30 minutes. These bounds are estimated from the baseline measurements and preliminary experiments.

Of the 30 programs in PolyBench 4.1, we evaluated 27. Polly does not manage to identify the SCoP in program `nussinov`. Presumably, the functions that are called in the SCoP would have to be declared as free of side-effects. We also excluded the programs `lu` and `ludcmp` from the experiment because of their extraordinarily long startup times. We do not show the results for nine of the remaining benchmarks, since none of the evaluated model-driven and iterative optimization methods produced a speedup during preliminary experiments. We used the largest possible data set that is configurable in PolyBench 4.1. For smaller sizes, the entire data tends to fit into the L3-cache. Table 1 shows characteristics of the PolyBench 4.1 benchmarks that we considered in the evaluation. The relevant compiler flags that we use to generate optimized LLVM IR are `-polly-parallel=true -polly-vectorizer=none -polly-tiling=true -polly-default-tile-size=64 -march=native`.

Adaptation of Pouchet's Algorithm. To compare Polyite against the iterative optimization of Pouchet et al. [33], we reimplemented their search space construction and combined it with our sampling strategy. A direct use of their implementation LeTSeE [31] was not possible, due to the incompatibility of its file format with Polly. Furthermore, to the best of our knowledge, statement iteration domains that can be expressed only as the union of multiple polyhedra cannot be represented with LeTSeE's input format. Polly and Polyite support these. We do not rely on the original sampling strategy, since it is based on assumptions that may not hold in the context of tiling and parallel execution.

To meet the characteristics of Pouchet's approach, we must adapt our sampling strategy. Before calculating the generator representation of the polyhedra P_d, we drop the constraints that bound the schedule coefficients to $\{-1, 0, 1\}$. This makes the polyhedra unbounded. The modified Algorithm 2 uses coefficients from $\{-1, 1\}$ for the generators and rejects schedule coefficient vectors with coefficients that are not from $\{-1, 0, 1\}$.

Configuration. An important configuration option of Polyite is the maximum number of rays and lines that may form a schedule coefficient vector. We use three settings: (1) If at most two rays and two lines may be added to a vertex to form a schedule coefficient vector, the schedule matrix is sparse, because many coefficients are zero. We call this the *sparse* setting. (2) If the allowed number of rays and lines is only limited by their total number, the matrices tend to be dense, which we call the *dense* setting. (3) The *mixed* setting produces sparse and dense matrices. These settings apply to the generating of coefficient vectors by random exploration as well as by genetic operators.

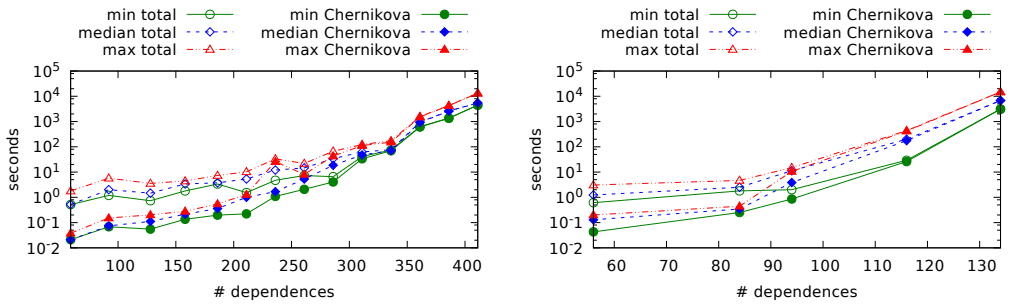
A run of the GA has 40 generations. The initial population is produced randomly using the sparse setting of random exploration. The regular population size is 30 schedules. A larger number of generations or a bigger population size conflicts with our costly fitness function. As, in one generation, we replace half of the population, the GA visits at most 630 schedules. Random exploration visits the same number of schedules. To ensure a good distribution of randomly sampled schedules, we sample just one schedule from the search space region currently visited. Still, random exploration may enter a search space region repeatedly.

The selection probability of the genetic operators is uniform: experimental evaluation did not reveal that some operators are more effective than others. The mutation operators are configured to mutate 10% of a schedule's dimensions at the start of the optimization process, but at least one row. Accordingly, generator coefficient replacement mutates 10% of the line, ray, and vertex coefficients at the beginning. Since, for schedule matrices with less than 10 rows, this always results in one row to be modified, the effect of simulated annealing diminishes for small SCoPs. The annealing function $f(p, g) = p/\sqrt{ln(g + \mathrm{e} - 1)}$ calculates the fraction of schedule dimensions to mutate from the currently generated generation g and the initial fraction p. Geometric crossover produces up to three schedules at once. The other operators produce just one schedule. The special role of geometric crossover is justified, since it produces otherwise unreachable rows. In each generation of the GA, we introduce two randomly generated schedules. The chosen configuration is the result of preliminary experiments. We challenge some of the choices in the research questions.

6.2 Experiments

Exp. 1 (RQ5). *Performance of Search Space Construction.* The execution time of Algorithm 2 for sampling the search space regions is in $O(d \cdot n_g \cdot \dim(\mathit{CoeffSpace}))$, where d is the number of dimensions of the region, n_g is the maximum number of generators that may form a schedule coefficient vector, and $\dim(\mathit{CoeffSpace})$ is the dimensionality of the schedule coefficient space. The algorithm has polynomial execution time, under the assumption that n_g is limited by a constant upper bound like in the sparse setting. Algorithm 1 employs Chernikova's algorithm. The size of a polyhedron's generating system is exponential in the number of its bounding inequalities [10]. This dominates the algorithm's run-time complexity. So, to decide whether our approach scales for large SCoPs, one must evaluate the actual execution time of Algorithm 1 depending on the number of data dependences.

When modeling a search space region, the maximum number of constraints that bound the set of possible schedule coefficient vectors for each schedule dimension grows linearly in the number of data dependences. So, we expect the execution time of Algorithm 1 to be exponential in the number of data dependences. To quantify the run-time complexity of Algorithm 1 empirically, we



(a) `seidel-2d` (b) `jacobi-2d`

Fig. 4. Minimum, maximum, and median of the execution time in seconds of Algorithm 1, depending on the number of data dependences. The y-axes are logarithmic.

had to scale the number of data dependences of a SCoP. This can be done by partially unrolling the SCoP's loops [38]. For the experiment, we used the benchmarks `seidel-2d` and `jacobi-2d` of PolyBench 4.1. Both are two-dimensional stencil codes surrounded by a time loop. We unrolled the outermost loop of each stencil. Per number of unrolled iterations, we ran Algorithm 1 five times and measured the total execution time and the time spent by Chernikova's algorithm. Figure 4 shows the results. The effort grows exponentially (note the logarithmic y-axes) and Algorithm 1 is dominated by Chernikova's algorithm. The measurements were conducted on a different CPU (Intel Xeon E5-2690 v2 CPU @ 3.00 GHz). We used the same Ubuntu 16.04 with an Oracle 8 JDK (version 8u101). Balev et al. [2] evaluated the usefulness of the geometric representation of polyhedra in the context of schedule optimization. Their work gives explanations to the observed domination of Algorithm 1 by Chernikova's algorithm.

Exp. 2 (RQ3, RQ4). *Convergence Rate of Iterative Optimization.* We compare the number of schedules that different variants of iterative optimization have to visit, on average, to find the optimal reachable schedule. We compare the following configurations: (1) random exploration with the adaptation of Pouchet's approach with the sparse setting, (2) random exploration with the sparse setting, (3) random exploration with the dense setting, and (4) our genetic algorithm with the sparse setting. Finally, (5) we evaluated configuration (1) but with schedule completion. We selected the benchmarks `3mm`, `correlation` and `adi`. They belong to different categories of PolyBench. They are promising candidates for this experiment, as the number of generators in the generator representation of their search space regions is high (see Table 1). Thus, an actual difference between sparse and dense setting can be expected. Apart from this, other benchmarks, for instance `gemm` and `jacobi-2d`, are not interesting for a detailed evaluation as, for them, the GA cannot surpass the performance of the initial population. To leverage the effect of an incomplete exploration, we applied each configuration 10 times to each benchmark. To reduce the benchmarking effort, we ran the GA only for 20 iterations. Accordingly, the random exploration runs generated 330 schedules. Figure 5 shows the median speedup over `O3` that was reached by each optimization strategy after evaluating n schedules. The x-axis is n. The y-axis is speedup.

There is no clear difference between the convergence rate of GA and random with the sparse setting. Both significantly outperform random exploration with the adapted approach of Pouchet et al. [33] and random with the dense setting. The GA yields slightly better schedules than sparse random. The effect of schedule completion on Pouchet's approach remains unclear.

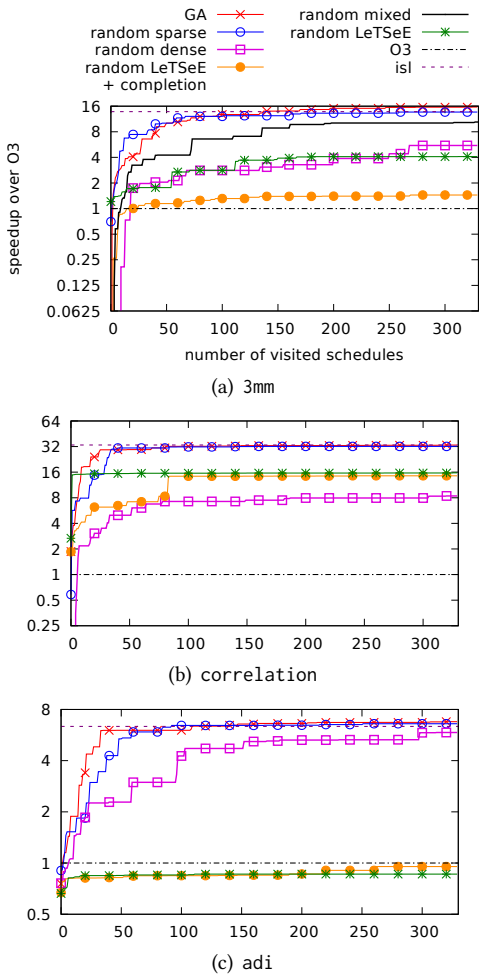


Fig. 5. Convergence rate of different configurations of iterative optimization. The x-axis shows the number of schedules already visited. The y-axis shows speedup over 03. For convenience, also the speedup reached by the ISL-scheduler is shown. Only every 20th data point is plotted.

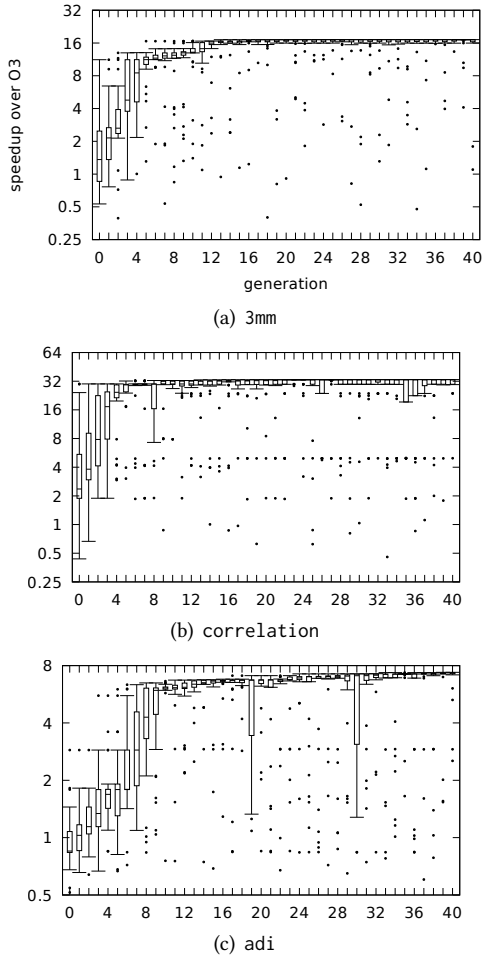


Fig. 6. Box plots showing the distribution of performance within the generations of the GA. The data are from Exp. 3. The length of the whiskers is 1.5 times the interquartile range. The whiskers may have been truncated to remain within the data set's range. The dots are outliers beyond the range of the whiskers.

In addition, we ran random exploration with the mixed setting, which produces both sparse and dense matrices, for 3mm. Its effectiveness is between random dense and random sparse.

Exp. 3 (RQ1, RQ2, RQ3). *Comparison with existing Approaches and Random.* We compared the following configurations to the original code optimized with 03: (1) ISL's adaptation of the PLuTo algorithm, (2) random iterative optimization, (3) random iterative optimization with the adaptation of Pouchet's approach (LeTSeE), (4) LeTSeE combined with schedule completion and (5) GA. The comparison to ISL's scheduler is fairer than comparing to the original PLuTo algorithm would

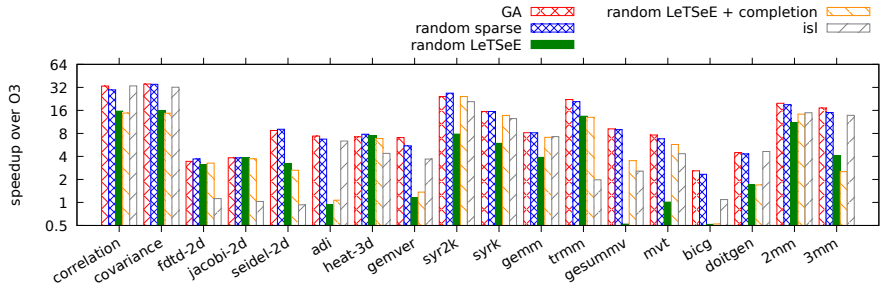


Fig. 7. Speedups over plain 03 reached by different configurations. The baseline is 03.

Table 2. *p*-values obtained from a pair-wise Mann-Whitney U Test quantifying the significance of differences between different configurations of schedule optimization

	GA	isl	random LeTSeE	random LeTSeE + completion
isl	7.60E-05	–	–	–
random LeTSeE	5.70E-05	0.0384	–	–
random LeTSeE + completion	2.50E-05	0.7987	0.3375	–
random sparse	0.2859	0.0014	2.50E-05	2.50E-05

Table 3. Per benchmark, the speedup over O3 of the best schedule found by the GA for 1-thread parallel execution and 8-threads parallel execution.

benchmark	2mm	3mm	adi	bicg	correlation	covariance	doitgen	fdtd-2d	gemm
1-thread parallel (speedup)	2.77	2.46	1.07	2.58	4.92	4.61	1.17	0.95	1.10
8-threads parallel (speedup)	19.79	17.14	7.37	2.58	33.10	35.28	4.18	3.50	8.12

gemver	gesummv	heat-3d	jacobi-2d	mvt	seidel-2d	syr2k	syrk	trmm
0.94	2.32	0.99	0.99	0.97	2.49	3.16	1.99	3.12
4.87	3.54	7.01	3.78	4.67	8.53	22.83	15.46	22.02

be, because the latter excludes some loop transformations that require negative coefficients for iteration variables. The comparison is made solely in terms of the performance yielded by the best found program transformation.

Figure 7 shows, per benchmark and optimization method, the speedup over plain 03 yielded by the best schedule found. In many cases, our novel iterative optimization technique yields more profitable schedules than PLuTo/isl and the adapted approach of Pouchet et al. [33]. Only for `doitgen` and `correlation`, the schedule found by iterative optimization is slightly worse than the schedule found by isl. Our approach to random exploration is never worse than the approach of Pouchet et al.. There is no apparent difference between the speedup reached by random exploration and GA. Table 3 lists the speedup over plain O3 of 8-threads parallel and 1-thread parallel execution of the optimal schedules found by GA. This gives an idea of the influence of the parallelization on the achieved speedups. The achieved speedups are largely due to parallelization.

We performed a paired Mann-Whitney U Test [1] in combination with false discovery rate control [4] to determine the statistical significance of the differences in the optimization results of each configuration pair. Table 2 shows, for each pair of configurations, the probability of erroneously assuming a difference between its optimization results. The GA has significantly better results than all other configurations except sparse random. The test is inconclusive about the effect of schedule completion on Pouchet's approach.

Table 4. Per benchmark, the number and total sizes of optimal schedule equivalence classes in the GA's final population. Further, per benchmark, the number of groups of equivalence classes with larger differences in the resulting code. `heat-3d` and `adi` had to be omitted, as some of the schedules could not be canonicalized in reasonable time.

benchmark	2mm	3mm	bicg	correlation	covariance	doitgen	fdtd-2d	gemm
# equivalence classes	16	13	1	16	12	13	18	10
total # schedules	21	24	22	16	22	26	22	19
# groups by similarity	3	2	1	11	3	4	10	1
	gemver	gesummv	jacobi-2d	mvt	seidel-2d	syr2k	syrk	trmm
	5	7	11	2	16	15	4	13
	23	13	22	22	23	23	26	23
	1	3	1	1	1	1	1	1

Finally, we estimated the number of search space regions visited by the genetic algorithm and random sparse. For all benchmarks, random sparse visits more regions than the GA. On average, the number of regions visited by the GA is 38% of the number of regions visited by random exploration. This was to be expected, since the GA is designed to continue its search in the vicinity of the most profitable schedules known so far. Using Spearman's rank correlation [21][Chap. 4], we found that the number of regions visited by the GA and the number of regions visited by random sparse both strongly correlate with the number of statements in the SCoP (GA: $\rho = 0.92,\ p = 6.249 \cdot 10^{-8}$, random sparse: $\rho = 0.91,\ p = 1.263 \cdot 10^{-7}$; ρ is the correlation coefficient, the p-values result from a two-sided significance test). Correlation with the number of data dependences is weak for random sparse ($\rho = 0.57,\ p = 0.01$) and undecidable for the GA ($\rho = 0.31,\ p = 0.21$).

Exp. 4 (RQ6). *Performance Distribution in the Generations of the GA.* For the benchmarks `3mm`, `correlation`, and `adi`, Figure 6 shows the performance distribution in each population of the GA using box plots. The data is from Exp. 3. The x-axis enumerates the populations. The y-axis shows speedup over `O3`. While the genetic algorithm proceeds, the variance of performance within the populations diminishes. Populations 8 and 35 of `correlation` and populations 19 and 30 of `adi` contain remarkably higher numbers of unprofitable schedules than their predecessors. In case of `adi`, geometric crossover produced more than half of the newly introduced schedules in populations 19 and 30 with a speedup smaller than the first quartile. In population 35 of `correlation`, 5 of the 7 new schedules with a speedup smaller than the first quartile are produced by prefix replacement.

Figure 6 exhibits a notable property of the genetic algorithm: It delivers a set of profitable schedules, while optimizing a SCoP requires just one. We wanted to know, how many semantically different schedules the set contains. To determine the according equivalence classes, we extended our schedule tree simplification to represent loop fusion in a canonical form and to transform the dimensions of band nodes' partial schedules to the canonical form of schedules with minimal coefficients. The latter borrows ideas from PLUTO+ [6]. We grouped the schedules of the GA's final population from Exp. 3 into equivalence classes. We took the execution time of its initial member as representative for an equivalence class. The extended schedule tree simplification is not a complete canonicalization. Therefore, we manually inspected the code that corresponds to each equivalence class. Thereby, we could merge a few equivalence classes. Further, we grouped equivalence classes that yield code which differs only by inverted loops or skewing. Table 4 shows the results.

The complexity of the remaining experiments forced us to limit them to one benchmark. Of the three benchmarks that have already been studied in more detail, we chose `3mm` since its numbers of statements and dependences and its maximum loop depth are closer to average than those of `adi` and `correlation`.

Exp. 5 (RQ7). *Effectiveness of the Genetic Operators.* To evaluate and compare the effectiveness of the genetic operators, we optimized the schedule of `3mm` with six configurations. In each configuration, one operator was disabled. Otherwise, the configurations meet the specification in Section 6.1. We ran each configuration 10 times, starting from one of 10 initial populations. A noticeable effect of disabling any of the operators on the convergence speed or the profitability of the optimal found schedule cannot be observed. Yet, from Exp. 4, we know that the last generation consists mostly of profitable schedules, so, in combination, our genetic operators are effective.

Exp. 6 (RQ7). *Intensity of the Mutations.* We evaluated the effect of varying the GA's mutations' intensity. We set the initial intensity (see Section 6.1) to 10%, and later increased it to 30% and 60%. With each configuration, we optimized `3mm` 10 times. Increasing the mutations' intensity does not affect the convergence speed of the GA or the profitability of the optimal schedule found. This was to be expected: In Exp. 2, we already noticed that the GA with its mutation intensity set to 10% does not converge faster than random sparse. Furthermore, we expect the GA to degenerate to a purely random search as we increase the mutation intensity. So, in our case, increasing the intensity of mutations should not yield faster convergence. Due to the small size of the benchmarks considered, a mutation intensity below 10% should not yield different results, either.

6.3 Discussion

We can answer **RQ1** positively based on the results shown in Figure 7. Iterative optimization detects program transformations yielding a higher performance than those found by the model-driven optimization of ISL. There are many possible reasons. The underlying assumptions of the model-driven approach may not apply in every case. The model-driven algorithm maximizes the size of tilable bands, which our iterative approach does not. Pouchet et al. [34] iteratively search for a good partitioning of a SCoP's statements into classes with different execution date and then schedule the statement instances in each partition in a model-driven way. Their motivation is that cost models for scheduling inside each class are well understood, while those for the partitioning itself are not. Likewise, our iterative approach may profit from this fact by finding a better partitioning. It is likely that, on different hardware, the iterative optimization would identify different optimal schedules. Furthermore, the tile shapes chosen by the PLuTo algorithm may be suboptimal. Iterative optimization may find different ones. In the end, performance optimization is hardware-specific. Naturally, iterative optimization can adapt itself more easily than model-driven algorithms.

Figure 7 reveals that random exploration with the sparse configuration almost always finds schedules that yield better performance than random exploration with the adaptation of LeTSeE. This justifies our augmentation of the schedule search space in combination with schedule completion and, thus, we can answer **RQ2** positively.

The answer to **RQ3** is that the GA and random exploration with the sparse setting converge at almost equal rate (see Figure 5). In Exp. 2, the GA was able to yield slightly more profitable schedules than sparse random. In Exp. 3, the speedup of the transformed code produced by the GA was 4% higher than by sparse random, on average. This difference cannot be tested significant. Figure 8 shows for `3mm` the performance distribution of the schedules visited during Exp. 2. We aggregated the data from all runs of each experiment and removed identical schedules. The GA is capable of finding significantly more schedules with a higher performance than sparse random. Still, the probability that sparse random hits a schedule with good performance is high. This must be attributed to the design and configuration of Algorithms 1 and 2. Studying this finding is an interesting avenue of future research.

RQ4 can also be answered positively. From Figure 8, it is apparent that, for `3mm`, random dense finds fewer profitable schedules than random sparse. The effect is similar for `correlation` and

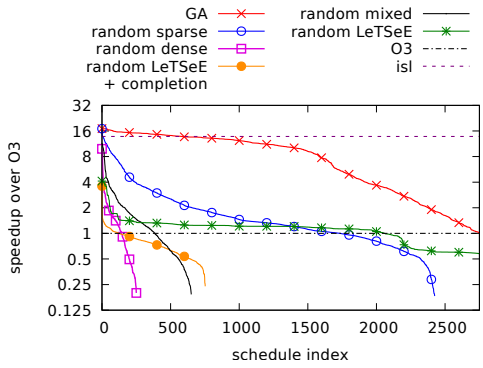


Fig. 8. Performance distribution of the schedules visited in Exp. 2 for 3mm. The x-axis enumerates the schedules. The y-axis is speedup over O3. Note that not every data point is plotted.

less pronounced for adi. We observe that random exploration with the dense setting produces by far more schedules that cannot be executed successfully than other configurations. For 3mm, 91.49% of the schedules visited by the genetic algorithm are healthy, 73.48% by random sparse, 100.00% by LeTSeE random but only 7.61% of random dense. Reasons range from a failure or timeout (5 minutes) of compilation (78.79% of the schedules from random dense), and failure of execution or erroneous computation results, to overall timeout of benchmarking. This conforms to the prediction of Pouchet et al. [33] that picking schedule coefficients close to zero is beneficial. Also, matrices formed from a large number of generators tend to contain larger coefficients, which may trigger integer overflows. The answer to the question of why random exploration with the sparse setting is extremely close to or even better than optimization with the GA is subject of future research.

Figure 4 confirms that the run-time complexity of Algorithm 1 is indeed exponential in the number of data dependences. The execution time does not grow equally fast for different programs. To answer **RQ5**, our search space construction does not scale for SCoPs with a large number of data dependences.

By Figure 6, **RQ6** is answered positively: If a genetic operator produces a new schedule, the probability is high that its performance is similar to that of its parents. Since a schedule's parents and the schedule itself are similar by construction, we can assume that similar schedules have a similar performance. Furthermore, we find that, for most benchmarks, semantically different schedules exist among the best schedules found by the GA. For most benchmarks with more than two statements, these differences are beyond loop inversion and skewing and comprise different loop distribution and fusion or interchange of loops.

In Exp. 5, we could not observe that disabling any genetic operator has a noticeable effect on the convergence speed of the genetic algorithm. Thus, we conclude that it is indeed best to select the operators with uniform probability. In Exp. 6, we checked whether an increased intensity of mutations would improve the convergence speed of the GA. We could not observe that an initial intensity higher than 10% (the default) yields faster convergence. We do not expect a lower value to yield faster convergence, either. In answer of **RQ7**, we conclude, that our choices for the selection probability of the genetic operators and the intensity of mutations are justified.

6.4 Threats to Validity

Threats to Internal Validity. Due to the exponential size of the search space of legal schedules, a full exploration of the entire (reasonable) search space, is computationally infeasible. Instead,

to create the initial population of the GA, we randomly sample search space regions and select one schedule from each selected region. Thus, we perform an incomplete exploration but can at least ensure that we start from a set of schedules that originate from different regions of the search space. Random exploration works accordingly. Still, different runs may find different optima. Although Figure 7 shows results of single runs, there are a number of apparent regularities in the data. Especially, random exploration and the GA often yield very similar performance. In other experiments, we present average results of multiple runs with different random seed to mitigate the effect of incomplete exploration.

Although all schedules visited are theoretically legal, some result in broken binaries or compilation failures. Any mathematically illegal schedules will be caught by Polly and will be reported to Polyite. Polyite will then abort. One of the run-time failures is the presence of overly large schedule coefficients. We are unaware of the other causes of compile-time or run-time failures that occur despite the legality of the schedules. As already mentioned in the discussion, we noticed that dense schedule matrices are likely to cause failure. They are also likely to increase compilation time tremendously. In part, the transformation of schedule matrices to schedule trees and the subsequent schedule tree simplification reduce the problem. The problematic schedules can simply be discarded but, the more schedules are discarded, the longer it takes to find the desired number of healthy schedules. An examination of the schedules from Exp. 2 reveals that, in contrast to random dense, the GA and random sparse discover only few schedules that fail.

To avoid measurement bias, the benchmarking systems had a minimal setup and ran no other workloads in parallel. Intel Turbo Boost and hyper-threading are known to destabilize measurements and, consequently, were disabled. The systems have a NUMA design with two sockets. If threads were distributed arbitrarily across the two CPUs and used non-local memory, time measurements would be unreliable. We pin the threads to the cores of one CPU and use only local memory [20]. We repeated execution time measurements five times and chose the shortest result. Five measurements are hardly enough to reach statistical significance but a higher number of measurements conflicts with the extremely large total number of program versions that had to be evaluated in our experiments. Data regarding the stability of our measurements is available on the supplementary Web site.

Threats to External Validity. We evaluated our approach to iterative schedule optimization on the PolyBench 4.1 benchmark suite. PolyBench is used by many researchers in the polyhedral community. This facilitates the comparability of our results. PolyBench comprises SCoPs in different application domains. This leverages the bias of an artificial benchmark. PolyBench is constantly being developed to adapt to the state-of-the-art in the polyhedral community.

Since PolyBench is particularly designed for benchmarking polyhedral optimization techniques, each benchmark contains one SCoP to be detected and optimized. To optimize SCoPs in real-world programs with Polyite, each SCoP would have to be dissected into a small program and would have to be fed with small (artificial) data.

7 RELATED WORK

Frequently used scheduling algorithms in contemporary polyhedral compilers are model-driven. A parametric integer programming algorithm [8] is used to select lexicographically minimal solution vectors from a polyhedron. Most notable are the algorithm by Feautrier [10, 11] and PLuTo [5]. Feautrier's algorithm reduces latency. Feautrier [10] states the generator representation of a polyhedron together with Chernikova's algorithm as an alternative way of solving the constraint system of his algorithm. PLuTo optimizes for tiling and data locality. The Integer Set Library (isl) by Verdoolaege [39] comprises a generalization of PLuTo. Another generalization is PLUTO+ [6]. Both

generalizations have fewer practical limitations than the original PLuTo algorithm. Particularly, they include transformations that require negative coefficients for iteration variables.

Iterative polyhedral optimizers do not rely on elaborate performance models. They sample the search space of possible schedules and assess each schedule's fitness. Some early approaches put legality constraints aside and consequently suffer from many illegal schedules. All are based on UTF [18]. GAPS [27] uses a GA to find polyhedral schedule transformations. GAPS starts from a random population of potentially illegal transformations that is seeded by one legal transformation. The fitness function either measures execution time of transformed programs or predicts loop and synchronization overhead. Among 20,000 generated schedules, at most 5.5% were legal. ICE [28] is built on top of GAPS. It enables the GA to apply more transformations, including tiling. ICE has new mutation operators in addition to the ones that exist in GAPS. The reproduction (genetic) operators can account for profiling data. The fitness function measures execution time. Long and O'Boyle [25] optimize Java programs. They machine-learn the performance impact of loop transformations that are expressible in UTF. To match unknown programs with learned strategies, program similarity is expressed by a set of features. Long and Fursin [24] also optimize Java code. They separate the search of mappings of the iteration variables (tiling, skewing) and the exploration of the constant part of schedules (loop distribution, fusion). Run time feedback directs the search.

We build on the approach of iterative optimization by Pouchet et al. [32, 33]. We go beyond their approach to optimize for tiling and parallelization in a purely iterative manner. We avoid search space restrictions that are viable only for sequential execution and propose a new sampling strategy. In Section 1, we already mention that later Pouchet et al. [34] combined iterative and model-driven optimization in the polyhedron model. The result is an automatic parallelization framework in which tiling, parallelization, and vectorization can be applied. They iteratively find a legal partitioning of the statements which corresponds to a sequence of loop distribution and fusion operations and then find a schedule for each partition in a model-driven way.

8 CONCLUSION AND FUTURE WORK

We propose an approach of purely iterative schedule optimization for tiling and OpenMP parallelization in the polyhedron model. We find schedules that yield better performance than with the model-driven PLuTo algorithm or our adaption of the iterative approach by Pouchet et al. [33]. We have presented a genetic algorithm to detect profitable schedules with more guidance than at random. Interestingly, a well configured random exploration finds schedules that yield almost equal performance as the schedules found by the genetic algorithm. We would like to address this finding in future research. Yet, the genetic algorithm is able to find more profitable schedules than random exploration. An investigation of the effectiveness of our genetic operators did not show an apparent disparity between the different operators.

ACKNOWLEDGMENTS

The first author is grateful to Stefan Kronawitter for his helpful technological support during the early stages of Polyite. Special thanks go to the reviewers, in particular Reviewer 1, whose impressively detailed debate of our work led to numerous clarifications and improvements.

REFERENCES

[1] T. W. Anderson and J. D. Finn. 1996. *The New Statistical Analysis of Data.* Springer.

[2] S. Balev, P. Quinton, S. V. Rajopadhye, and T. Risset. 1998. Linear Programming Models for Scheduling Systems of Affine Recurrence Equations – A Comparative Study. In *Proc. 10th Ann. ACM Symp. on Parallel Algorithms and Architectures (SPAA).* ACM Press, 250–258.

[3] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. 2010. The Polyhedral Model Is More Widely Applicable Than You Think. In *Compiler Construction (CC) (LNCS 6011)*, Rajiv Gupta (Ed.). Springer, 283–303.

[4] Y. Benjamini and Y. Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *J. Royal Stat. Soc. Series B (Methodological)* 57, 1 (1995), 289–300.

[5] U. Bondhugula et al. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction (CC) (LNCS 4959)*, Laurie Hendren (Ed.). Springer, 132–146.

[6] U. Bondhugula, A. Acharya, and A. Cohen. 2016. The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests. *ACM Trans. Program. Lang. Syst. (TOPLAS)* 38, 3 (May 2016), 12:1–12:32.

[7] J. Clarke et al. 2003. Reformulating Software Engineering as a Search Problem. *IEE Proceedings – Software* 150, 3 (June 2003), 161–175.

[8] P. Feautrier. 1988. Parametric Integer Programming. *RAIRO – Operations Research* 22, 3 (1988), 243–268.

[9] P. Feautrier. 1991. Dataflow Analysis of Array and Scalar References. *Int. J. Par. Prog. (IJPP)* 20, 1 (1991), 23–53.

[10] P. Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-Dimensional Time. *Int. J. Par. Prog. (IJPP)* 21, 5 (1992), 313–347.

[11] P. Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. *Int. J. Par. Prog. (IJPP)* 21, 6 (1992), 389–420.

[12] P. Feautrier and C. Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*, D. A. Padua (Ed.). Vol. 3. Springer, 1581–1591.

[13] M. Griebl, P. Feautrier, and C. Lengauer. 2000. Index Set Splitting. *Int. J. Par. Prog. (IJPP)* 28, 6 (Dec. 2000), 607–631.

[14] T. Grosser, A. Größlinger, and C. Lengauer. 2012. Polly – Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Par. Proc. Lett. (PPL)* 22, 4 (2012). Article 1250010, 28 pages.

[15] T. Grosser, S. Verdoolaege, and A. Cohen. 2015. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst. (TOPLAS)* 37, 4 (Aug. 2015), 12:1–12:50.

[16] M. Harman. 2007. The Current State and Future of Search Based Software Engineering. In *Proc. Workshop on the Future of Software Engineering (FOSE)*. IEEE Computer Society, 342–357.

[17] F. Irigoin. 2011. Tiling. In *Encyclopedia of Parallel Computing*, D. A. Padua (Ed.). Vol. 4. Springer, 2040–2049.

[18] W. Kelly and W. Pugh. 1995. A Unifying Framework for Iteration Reordering Transformations. In *Proc. IEEE First Int'l Conf. on Algorithms and Architectures for Parallel Processing (ICAPP)*, Vol. 1. IEEE, 153–162.

[19] K. Kennedy and K. S. McKinley. 1993. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *Languages and Compilers for Parallel Computing (LCPC) (LNCS 768)*, U. Banerjee et al. (Eds.). Springer, 301–320.

[20] A. Kleen. 2004. *An NUMA API for Linux.* Technical Report. SUSE Labs.

[21] P. R. Krishnaiah and P. K. Sen. 1984. *Handbook of Statistics.* Vol. 4. Elsevier.

[22] C. Lattner. 2008. LLVM and Clang: Next Generation Compiler Technology. In *BSDCan: The BSD Conference.*

[23] H. Le Verge. 1994. *A Note on Chernikova's Algorithm.* Res. Report RR-1662. INRIA.

[24] S. Long and G. Fursin. 2009. Systematic Search within an Optimisation Space Based on Unified Transformation Framework. *Int. J. Comput. Sci. Eng. (IJCSE)* 4, 2 (2009), 102–111.

[25] S. Long and M. F. P. O'Boyle. 2004. Adaptive Java Optimisation Using Instance-Based Learning. In *Proc. 18th Ann. Int.l Conf. on Supercomputing (ICS)*. ACM, 237–246.

[26] M. Mitchell. 1998. *An Introduction to Genetic Algorithms.* MIT Press.

[27] A. Nisbet. 1998. GAPS: A Compiler Framework for Genetic Algorithm (GA) Optimised Parallelisation. In *High-Performance Computing and Networking (HPCN Europe)*, P. Sloot, M. Bubak, and B. Hertzberger (Eds.). Springer, 987–989.

[28] A. Nisbet. 2001. Towards Retargettable Compilers – Feedback Directed Compilation Using Genetic Algorithms. In *Proc. 9th Int'l Workshop on Compilers for Parallel Computers (CPC)*. 12 pages. http://www.icsa.informatics.ed.ac.uk/cpc2001/

[29] M. Odersky, L. Spoon, and B. Venners. 2008. *Programming in Scala.* Artima.

[30] D. Padua. 2011. Parallelization, Automatic. In *Encyclopedia of Parallel Computing*, D. A. Padua (Ed.). Vol. 3. Springer, 1442–1450.

[31] L.-N. Pouchet. 2012. LeTSeE – The LEgal Transformation SpacE Explorator. (2012). http://web.cs.ucla.edu/~pouchet/software/letsee/

[32] L.-N. Pouchet et al. 2007. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. In *Proc. Fifth Int. Symp. on Code Generation and Optimization (CGO)*. IEEE Computer Society, 144–156.

[33] L.-N. Pouchet et al. 2008. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *Proc. ACM SIGPLAN 2008 Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 90–100.

[34] L.-N. Pouchet et al. 2010. Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework. In *Proc. ACM/IEEE Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 1–11.

[35] L.-N. Pouchet and T. Yuki. 2015. PolyBench 4.1. (May 2015). http://web.cse.ohio-state.edu/~pouchet/software/polybench/.
[36] A. Schrijver. 1994. *Theory of Linear and Integer Programming*. John Wiley & Sons.
[37] K. Trifunovic et al. 2010. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *Proc. Int'l Workshop on GCC Research Opportunities (GROW)*. 1–13. http://ctuning.org/workshop-grow10.
[38] R. Upadrasta and A. Cohen. 2013. Sub-Polyhedral Scheduling Using (Unit-)Two-Variable-per-Inequality Polyhedra. In *Proc. 40th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*. ACM, 483–496.
[39] S. Verdoolaege. 2010. *isl*: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, K. Fukuda et al. (Eds.). Springer, 299–302.
[40] M. Wolfe. 1986. Loops Skewing: The Wavefront Method Revisited. *Int. J. Par. Prog. (IJPP)* 15, 4 (Aug. 1986), 279–293.