Speeding up Iterative Polyhedral Schedule Optimization with Surrogate Performance Models

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

Iterative program optimization is known to be able to adapt more easily to particular programs and target hardware than model-based approaches. An approach is to generate random program transformations and evaluate their profitability by applying them and benchmarking the transformed program on the target hardware. This procedure's large computational effort impairs its practicality tremendously, though.

To address this limitation, we pursue the guidance of a genetic algorithm for program optimization via feedback from surrogate performance models. We train the models on program transformations that were evaluated during previous iterative optimizations. Our representation of programs and program transformations refers to the polyhedron model. The representation is particularly meaningful for an optimization of loop programs that profit a from coarse-grained parallelization for execution on modern multicore-CPUs. Our evaluation reveals that surrogate performance models can be used to speed up the optimization of loop programs. We demonstrate that we can reduce the benchmarking effort required for an iterative optimization and degrade the resulting speedups by an average of 15%.

CCS Concepts: • **Computing methodologies → Supervised learning by classification**; **Genetic algorithms**; **Genetic programming**; • **Software and its engineering → Massively parallel systems**;

Additional Key Words and Phrases: Automatic loop optimization, genetic algorithm, OpenMP, parallelization, polyhedron model, tiling, CART, random forests, supervised learning

ACM Reference Format:
Stefan Ganser, Armin Größlinger, Norbert Siegmund, Sven Apel, and Christian Lengauer. 2018. Speeding up Iterative Polyhedral Schedule Optimization with Surrogate Performance Models. *ACM Transactions on Architecture and Code Optimization* 1, 1, Article 1 (October 2018), 25 pages. https://doi.org/0000001.0000001

This work was partially supported by the German Research Foundation grant no. AP 206/6.
Authors' addresses: Stefan Ganser, University of Passau, Faculty of Computer Science and Mathematics, Innstraße 33, 94032, Passau, Germany, stefan.ganser@uni-passau.de; Armin Größlinger, University of Passau, Faculty of Computer Science and Mathematics, Innstraße 33, 94032, Passau, Germany, armin.groesslinger@uni-passau.de; Norbert Siegmund, Bauhaus-University, Weimar, Faculty of Media, Bauhausstraße 9a, 99423, Weimar, Germany, norbert.siegmund@uni-weimar.de; Sven Apel, University of Passau, Faculty of Computer Science and Mathematics, Innstraße 33, 94032, Passau, Germany, apel@uni-passau.de; Christian Lengauer, University of Passau, Faculty of Computer Science and Mathematics, Innstraße 33, 94032, Passau, Germany, christian.lengauer@uni-passau.de.

XXXX-XXXX/2018/10-ART1 $15.00
https://doi.org/0000001.0000001

1 INTRODUCTION

The complexity of processor architectures is constantly increasing. The end of Dennard scaling [23], which prohibits further strong increases in processor performance merely by ongoing miniaturization of integrated circuits and rising clock speeds, leads to a wide spread of multi-core architectures that enable further increases in computing performance via parallel execution [38]. Adding levels of caches between memory and processor is supposed to abolish the von-Neumann bottleneck. Vectorization adds a further level of parallelism. The exploitation of the available performance requires ongoing research on optimizing compilers that can, ideally, adapt the target code to different execution platforms. Effective cost models must take many different aspects into account.

In this context, iterative (or search-based) program optimization can help to improve the understanding of the characteristics of profitable program transformations, as it relies on few or no prior assumptions. This technique may also help in deriving more effective cost models for compilers.

The polyhedron model [31] enables a systematic exploration of the set of legal transformations of a given program [35, 56, 57]. Generally, it allows one to unify sequences of different loop transformations, such as distribution/fusion [42], skewing [73], tiling [39], and interchange [2] in one framework. Yet, the model imposes restrictions on the source programs that can be treated.

With this in mind, we proposed the iterative polyhedral optimizer Polyite [35]. Replacing the model-based PLuTo algorithm [16, 19] in LLVM's [45] polyhedral optimizer Polly [36] by Polyite leads to significant speedups when optimizing for tiling and parallelization. Polyite searches at random or by means of its genetic algorithm. The PLuTo algorithm enables tiling and improves data locality. Depending on the PLuTo algorithm's specific variant, parallelism is a byproduct of improving data locality or can be targeted explicitly [71]. Recently, Zinenko et al. [74] improved a variant of the PLuTo algorithm to account explicitly for spatial proximity of memory accesses.

A *genetic algorithm* (*GA*, for short) searches iteratively for a solution that is good in terms of a *fitness function*. It derives new solutions from already existing solutions. Typically, the probability of a solution to reproduce depends on its *fitness* according to the fitness function. A GA starts from an, often randomly generated, set of solutions, its initial *population*. Using the fitness function, it assesses each solution's profitability. Mutation and crossover of solutions from the current population cause a new population or *generation* to arise.

To determine the fitness of a candidate transformation, Polyite applies the transformation and executes the resulting code. As might be expected, this process is time-consuming. One could try to reduce the optimization time by benchmarking with very small data set sizes, but this would reduce the effect of data locality optimizations and also the extent of parallelism. One way out is to replace benchmarking with a less precise but cheaper performance prediction. The prediction can be based on a surrogate model that has been trained on structural features of program transformations and the execution time of the respective transformed code. The training data originates from previous iterative optimizations. A *feature* is a property whose intensity is expressible with a numeric value.

Such a *supervised machine-learning* algorithm deduces a model from a set of training data. The model serves to predict the value of a dependent variable from the values of independent variables. In our case, the independent variables are the structural features of a program transformation and the dependent variable correlates to the speedup in execution time yielded by the transformation.

For practicality, there must be features of program transformations that can be extracted at low cost. Furthermore, the features must permit to learn a performance model from a training set of transformations gathered during the iterative optimization of a set of programs. The model must be transferable, that is, suitable to predict the profitability of transformations for unseen programs.

Our proposal is to reduce the execution time of Polyite's GA by the use of a surrogate performance model that allows a classifier (a predictor that assigns labels) to distinguish profitable

from unprofitable program transformations. We train the model on features and performance data of transformations generated in previous runs of Polyite. The features are meaningful for an optimization of loop programs that can be expected to profit from coarse-grained parallelization and tiling for an execution on modern multi-core CPUs.

In theory, there are infinitely many encodings of one program transformation in the polyhedron model. This complicates feature extraction since a transformation's feature vector may vary, depending on its representation. Unnecessary noise in a program transformation's representation can also complicate further processing and yield unnecessarily complex code. To avoid this, we propose a semantics-preserving simplification of program transformations. We have mentioned this transformation briefly in our previous work [35]. Here, we elaborate on it.

Our evaluation indicates that replacing benchmarking by a classifier based on a surrogate performance model can reduce the time consumption of Polyite's GA by an average of 49%. We lose an average of 15% of speedup for the optimized program. In particular, we can reduce the benchmarking effort on the target hardware. We assume the presence of suitable training data.

It was to be expected and is also the case that a performance model learned from any program cannot succeed in detecting profitable program transformations for another program. While we can show that often this is possible, it would be wrong to claim that it works generally. This message is important for the polyhedral community, as it strengthens the assumption that one model cannot serve to optimize any program. In summary, we contribute the following:

- We devise a new GA for our iterative schedule optimizer Polyite. In the presence of suitable training data that have resulted from previous iterative optimizations we can machine-learn a surrogate performance model and mostly replace benchmarking with classification.
- We propose a set of schedule features that serves to learn a surrogate performance model from sets of program transformations that originate from the iterative optimization of other programs.
- We present a semantics-preserving simplification of the representation of program transformations to make them suitable for feature extraction.
- We evaluate our approach on the PolyBench 4.1 benchmark set [60].

The data of the evaluation is available on the paper's supplementary Web site [34].

2 BACKGROUND

Our approach relies on concepts of the polyhedron model and on machine-learning techniques. We introduce these and recall the current state of Polyite.

2.1 Polyhedron Model

The polyhedron model [31] is a mathematical abstraction of loop programs. It models a program as a union of polyhedra and relations between these polyhedra. Program transformations are represented by multi-dimensional linearly affine functions. They operate on the model and change the execution order of operations. The (transformed) model can be converted to new code.

Static Control. To be expressible in the polyhedron model, a code region must be a *static-control part* (*SCoP*). A SCoP consists of nests of counting loops. The code must operate on linear data structures. Loop bounds and indices of accessed memory locations must be linearly affine expressions of structure parameters (integer variables that remain unchanged during the SCoP's execution) and the values of iteration variables. Wider definitions of static control exist [14]. We use the benchmark `syrk` of PolyBench 4.1, presented in Listing 1, as a running example.

Iteration Domain. The polyhedron model represents programs at the statement-instance level. A *statement instance* is the occurrence of a single execution of a statement. A statement instance is represented as a point in a vector space whose dimensionality is the number of loops that encase

the respective statement. A statement instance's coordinates are called its *iteration vector*. The restriction to static control enables the representation of the set of a statement's instances as union of polyhedra. Their bounds are specified by the SCoP's loop bounds and branch conditions.

Example 2.1. Our example's iteration domain is modeled by two polyhedra, one per statement: $I_R = \{(i,j) \mid 0 \le i < n \ \wedge \ 0 \le j \le i\}, I_R \subset \mathbb{Z}^2$; $I_S = \{(i,k,j) \mid 0 \le i < n \ \wedge \ 0 \le k < m \ \wedge \ 0 \le j \le i\}, I_S \subset \mathbb{Z}^3$.

Listing 1. The code performs a symmetric rank-k update. It computes $C = \alpha \cdot AA^T + \beta \cdot C$.

```
1  for (i = 0; i < n; i++) {
2    for (j = 0; j <= i; j++)
3      C[i][j] *= beta; // R
4    for (k = 0; k < m; k++)
5      for (j = 0; j <= i; j++)
6        C[i][j] += alpha
7          * A[i][k] * A[j][k]; // S
8  }
```

Schedule. A SCoP's iteration domain defines the set of all its statements' instances. But the statement instances' order remains undefined. Execution order is imposed by a multi-dimensional linearly affine function, which is called a *schedule*. A schedule can be represented by a set of functions, one per statement. Each function's domain is the respective statement's iteration domain. The codomain is a vector space. Execution order is defined as the lexicographic order ($\prec$) on the schedule's range. We may assume that, in a SCoP, all statements' schedules have the same codomain.

Example 2.2. Our running example has the following schedule functions: $\Theta_R : I_R \to \mathbb{Z}^4;\ \Theta_R(i,j) = (i,0,j,0)$; $\Theta_S : I_S \to \mathbb{Z}^4;\ \Theta_R(i,k,j) = (i,1,k,j)$.

Program transformation works by replacing the schedule in the SCoP's model. If a schedule is multi-dimensional, we call its one-dimensional components *schedule dimensions*. We call the first dimension the *outermost* one and the last dimension the *innermost* one. To address a schedule dimension, we superscribe the schedule's name Θ with the dimension's index d (Θ^d). Analogously, we denote ranges of schedule dimensions from dimension d to dimension e ($e > d$) by $\Theta^{d..e}$.

Memory Accesses and Data Dependences. A SCoP's model must specify memory accesses. A *data dependence* from one statement instance to another exists if both access the same memory address and the former statement instance is executed prior to the latter. From memory accesses, iteration domain, and schedule, *data dependence polyhedra* as relations between the statements' iteration domains can be computed. Transitive ones are omitted. Dependences of which at least one access is a write may affect the schedule's legality. A schedule is *legal* iff $\left(\forall\, (\vec{i},\vec{j}) \in D_{O,T} : \Theta_O(\vec{i}) \prec \Theta_T(\vec{j})\right)$ for all dependence polyhedra $D_{O,T}$ that involve write accesses [30].

In this context, let us mention two additional concepts that are crucial in the construction of legal schedules. A schedule dimension d *weakly satisfies* a dependence polyhedron $D_{O,T}$ if $\left(\forall(\vec{i},\vec{j}) \in D_{O,T} : \Theta^d_O(\vec{i}) \le \Theta^d_T(\vec{j})\right)$. The dependence polyhedron is said to be *strongly satisfied* if $\left(\forall(\vec{i},\vec{j}) \in D_{O,T} : \Theta^d_O(\vec{i}) < \Theta^d_T(\vec{j})\right)$. The first dimension of a schedule that strongly satisfies a dependence polyhedron is said to *carry* the dependence polyhedron. There are constellations in which the dependences in one dependence polyhedron are carried by different schedule dimensions.

With respect to a one-dimensional schedule Θ, a dependence polyhedron $D_{O,T}$ can have a direction. If for all $(\vec{i},\vec{j}) \in D_{O,T}$ the value of $\mathrm{sgn}(\Theta_T(\vec{j}) - \Theta_O(\vec{i}))$ is 1, then the direction is forward; -1 is backward; 0 is orthogonal. The direction may also vary among a polyhedron's elements.

Example 2.3. The following data dependence polyhedra of `syrk` affect the legality of schedules:

$$D_{R,S} = \{(i,j,i,0,j) \in I_R \times I_S \mid i < n \ \wedge \ 0 \le j \le i\}$$
$$D_{S,S} = \{(i,k,j,i,1+k,j) \in I_S \times I_S \mid m > 0 \ \wedge \ i < n \ \wedge \ 0 \le k \le -2+m \ \wedge \ 0 \le j \le i\}.$$

Tiling. Tiling [39] can improve data locality. The transformation blocks some of a given loop nest's loops and permutes the loops that enumerate the blocks outward in the nest. It is also applicable to imperfect loop nests [74]. In the polyhedron model, rectangular tiling may be applied legally to a sequence of schedule dimensions iff these schedule dimensions are *permutable* (that is, if they all weakly satisfy the same set of data dependences) [19]. Such a sequence is called a *tilable band*.

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

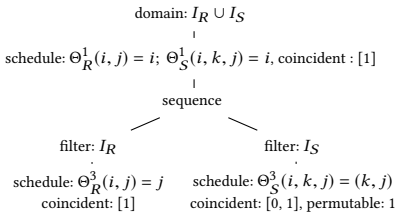

Fig. 1. Schedule matrix (left) and schedule tree (right) of the SCoP in Listing 1.

Representing Schedules. A schedule function Θ_S of a statement S can be represented as a *schedule (coefficient) matrix*. Each of its rows contains a coefficient per iteration variable of the loops encasing S, a coefficient per structure parameter, and a coefficient for the constant 1. Putting statement schedules' matrices side by side, the schedule of the entire SCoP is expressible in one matrix.

Schedule matrices are convenient for sampling schedules since, given a dependence polyhedron $D_{O,T}$, the set of all rows that correspond to one-dimensional schedules that must weakly or strongly satisfy the dependences in $D_{O,T}$ is a polyhedron that can be computed using Farkas's Lemma [29].

Schedule matrices are less convenient when it comes to analysis or transformation of schedules. Schedule trees by Grosser et al. [37] make schedules more tangible. Other than matrices, schedule trees can capture textual execution order uniquely with sequence nodes. This makes them more suitable for transformation and analysis, since many operations, like tiling, process one branch of the tree at a time. We recall the concept, to the extent necessary, using our running example.

Example 2.4. Figure 1 shows a schedule matrix and a schedule tree that both encode the execution order of the source code of our running example `syrk`. The correspondence between the matrix and the schedule functions in Example 2.2 is straightforward. Let us focus on the schedule tree.

The root of a schedule tree is a *domain node*. It declares all statements' iteration domains. In our case, the root's child is a *band node*. Band nodes contain partial schedule functions. The band node underneath the root encodes the outermost loop of our SCoP. Dimensions of a band node's schedule can be marked parallel (*coincident*). The band node's child is a sequence node. *Sequence nodes* partition the iteration domain. Per partition in the partitioning, a schedule is encoded in a separate subtree. The partitions' execution order is given by the subtrees' order. The children of a sequence node are *filter nodes*, which specify the iteration domain's subset that is scheduled in the subtree underneath. In the subtree on the right, we see a band node that is marked *permutable*, which enables it for tiling. Branches are terminated by *leaf nodes*, which we omit in the figure.

2.2 Decision Trees and Random Forests

Classification and regression tree (*CART*) [21] learning is a technique of supervised machine-learning. A CART results from splitting a training set of samples repeatedly into two subsets until each subset is homogeneous in terms of the dependent variable's value or a splitting criterion is reached. Each recursive split yields a node in a binary tree that stores the predicate on the independent variable according to which the split was made. The final (homogeneous) sets of training samples are represented by the tree's leaf nodes, which hold the dependent variable's respective values. Optimal splitting predicates are identified by the use of entropy measures such as GINI [21]. To predict the dependent variable's value from an input vector of values for the independent variables, one must start at the tree's root, evaluate the inner nodes' predicates, and descend further accordingly until a leaf is reached. The prediction is the majority class of the values stored in the leaf node.

Decision trees are susceptible to overfitting. An *overfitted* classifier has a poor predictive power and its model is overly complex since it reflects all details of the training data. Pruning of subtrees is

a profitable technique to reduce overfitting in decision trees. Another technique is to set a minimum number of training samples that are classified in a leaf of the decision tree.

Random forests [20] reduce the overfitting effect of CART. One learns several trees, each on a subset of the training data. Predictions are the average predictions of the individual trees.

2.3 Polyite

Polyite [35] is an open-source tool for iterative schedule optimization in the polyhedron model. It samples schedule matrices from the search space of legal schedules for a given SCoP. Its theoretical foundation is the work by Pouchet et al. [56, 57]. Other than their approach, Polyite practically considers the entire search space. Exploration may proceed either at random or guided by a GA. The GA's schema resembles that of Pouchet et al. [57], but it can traverse our wider search space.

Polyite's GA starts from a randomly generated population of schedules. Strong diversity in this population must be ensured. A population's fitter half survives and is passed to the next generation and reproduces until the full population size has been reached again. Mutations of schedules are strong at the beginning of the search and are gradually dampened by an annealing factor.

There is little locality in the schedule search space in a sense that a very small change, like replacing a single coefficient of an already profitable schedule, is unlikely to produce an even better schedule [57]. Therefore, the initial population's strong diversity, the strong mutations to schedules, and the *elitism* [8], which lets the fittest schedules survive to the next generation, are indispensable.

Polyite relies on Polly [36] to apply schedules to code. Before passing a schedule to Polly, Polyite transforms the schedule matrix to a meaningful simplified but equivalent schedule tree.

Polyite's general sampling strategy has two phases. The first phase is to select randomly a subset of the search space. For this purpose, we relax the multi-dimensional search space construction by Pouchet et al. [57]. In the second phase, we select a schedule from the chosen subset of the search space. Here, we rely on an extension [46] of Chernikova's algorithm to represent polyhedra geometrically. Regarding sampling with Chernikova, the important aspect is that the polyhedra that model subsets of the search space are unbounded except for the preservation of legality. This keeps their number of vertices low [35]. In contrast to the approach by Pouchet et al. [56, 57], the schedules sampled by Polyite encode explicitly all loops of the transformed code.

By avoiding likely unprofitable subsets of the search space and sampling such that schedule matrices tend to be sparse we can strongly increase the profitability of the best schedule found [35].

3 SCHEDULE TREE CONSTRUCTION AND SIMPLIFICATION

In this section, we present our conversion of schedule matrices sampled by Polyite to simplified schedule trees [37], which suit further transformation and feature extraction better. While the Integer Set Library (isl) [67] can convert from schedule trees to schedule matrices, there is no sophisticated reverse transformation. With isl, the only option is presently to store the complete schedule in a single band node [68], which does not improve over schedule matrices' expressiveness.

Some of the schedule matrices that we sample happen to encode loops with only one iteration and uselessly shifted loop bounds. Furthermore, schedule coefficients may be needlessly large. Two different schedule matrices, even matrices with differing numbers of rows, may represent schedules that prescribe the same execution order. Yet, for schedule classification to work, equivalent schedules should have equal feature values. Thus, we convert schedule matrices to simplified schedule trees. We also apply this conversion before passing schedules to code generation. Cohen et al. [25] recognized the need for normalization of representations of schedules and SCoPs.

Basic Construction. We start by describing the conversion of schedule matrices to schedule trees. Let us assume a SCoP with statements $S_1, \ldots, S_m$ and respective iterations domains $I_{S_1}, \ldots, I_{S_m}$.

Let M_Θ be an n-dimensional schedule matrix. M_Θ can be decomposed into statement schedules $\Theta_{S_1}, \ldots, \Theta_{S_m}$. We define a partial order on the statements' iteration domain:
$I_{S_x} <^k I_{S_y} \Leftrightarrow \left(\forall (\vec{i}, \vec{j}) \in I_{S_x} \times I_{S_y} : \Theta_{S_x}^{0..k-1}(\vec{i}) = \Theta_{S_y}^{0..k-1}(\vec{j}) \Rightarrow \Theta_{S_x}^{k}(\vec{i}) < \Theta_{S_y}^{k}(\vec{j})\right)$. Iteration domain I_{S_x} precedes I_{S_y} iff, with respect to schedule dimension k, all iterations of S_x are executed before the iterations of S_y. This is textual execution order of statements. The ISL code generator [37] also determines the statements' textual order. Generally, polyhedral code generators determine textual ordering, but at a finer-grained level that splits iteration domains [12, 37, 66].

There is another way to encode textual order in schedules. Let schedule dimension k encode a loop around statements $S_1, \ldots, S_m$. Let the loop have stride (increment) $\alpha_k \in \mathbb{N}$. If dimension k of the statements' schedules has the form $\alpha_k \cdot i + \beta_{k_1}, \ldots, \alpha_k \cdot i + \beta_{k_m}$ with i being some original iteration variable and $\beta_{k_1}, \ldots, \beta_{k_m} \in [0, \alpha_k - 1]$, then schedule dimension k can be represented by a band node with schedule $\alpha \cdot i$ for each statement, followed by a sequence node that enumerates the statements according to the values of $\beta_{k_1}, \ldots, \beta_{k_m}$ in ascending order. We could replace α by 1, following Bastoul [13] and Grosser et al. [37], who proposed the same simplification, but, in our case, this is taken care of by one of the simplification steps described further on. We denote this partial order by $\widetilde{<}^k$. Grosser et al. call this technique *shifted stride detection*.

ALGORITHM 1: Basic Schedule Tree Construction (`constructTree`)

Input: $I = \{I_{S_1}, \ldots, I_{S_m}\}$: Set of statement iteration domains, $\Theta_{S_1}, \ldots, \Theta_{S_m}$: n-dimensional statement schedules, k: current dimension, n: total number of schedule dimensions
Output: The constructed schedule tree.

Procedure `Partition`$(J, \otimes)$ `// Set of iteration domains, order predicate`
  return $\langle P_1, \ldots, P_l \subseteq J \mid (\forall i, j \in \{1, \ldots, l\} : (i < j) \Rightarrow (P_i \cap P_j = \emptyset \wedge (\forall (I_{S_x}, I_{S_y}) \in P_i \times P_j : I_{S_x} \otimes I_{S_y}))) \wedge (\forall i \in \{1, \ldots, l\} : \neg(\exists Q \subset P_i : Q \neq \emptyset \wedge (\forall I_{S_x} \in Q : (\forall I_{S_y} \in (P_i \setminus Q) : I_{S_x} \otimes I_{S_y}))))\rangle$
if $k > n$ **then return** `LeafNode`(I);
if $\Theta_{S_1}, \ldots, \Theta_{S_m} == 0$ **then return** `constructTree`$(I, \Theta_{S_1}, \ldots, \Theta_{S_m}, k+1, n)$;
children $\leftarrow \langle\rangle$; $\langle P_1, \ldots, P_l \rangle \leftarrow$ `Partition`$(I, <^k)$
if $l > 1$ **then**
  for $i \in [1, l]$ **do**
    $S \leftarrow \{\Theta_{S_x} | I_{S_x} \in P_i\}$; children.append(`FilterNode`$(P_i$, `constructTree`$(P_i, S, k, n))$)
  return `SeqNode`(children)
$\langle P'_1, \ldots, P'_{l'} \rangle \leftarrow$ `Partition`$(I, \widetilde{<}^k)$
if $l' > 1$ **then**
  for $i \in [1, l']$ **do**
    $S \leftarrow \{\Theta_{S_x} | I_{S_x} \in P'_i\}$; children.append(`FilterNode`$(P'_i$, `constructTree`$(P'_i, S, k, n))$)
  return `BandNode`$(\Theta^k_{S_1} - \beta_{k_1}, \ldots, \Theta^k_{S_m} - \beta_{k_m}$, `SeqNode`(children))
return `BandNode`$(\Theta^k_{S_1}, \ldots, \Theta^k_{S_m}$, `constructTree`$(I, \Theta_{S_1}, \ldots, \Theta_{S_m}, k+1, n))$

Algorithm 1 illustrates our recursive construction of a basic, unsimplified schedule tree starting from a set I of statement iteration domains and, per statement, an n-dimensional schedule matrix. Finally, the algorithm requires a counter k to know the next schedule dimension to be processed.

If $k > n$, that is the schedule matrices have been processed, we are done and return a leave node. If schedule dimension k is constant zero, we can ignore it and go on to dimension $k + 1$.

Next, we perform topological partitioning according to either of the partial orderings $<^k$ and $\widetilde{<}^k$ on I. In either case, the result is a partitioning of I. The resulting subsets are totally ordered. If, in one case (the preference is for $<^k$), the partitioning has more than one element, we introduce a sequence node to enumerate the subsets. If the sequence node is yielded by $\widetilde{<}^k$, it is preceded by a band node. The sequence node's children are constructed recursively by Algorithm 1.

If the partitioning according to both, $<^k$ and $\widetilde{<}^k$, has only one element, we construct a band node that stores schedule dimension k and recursively call Algorithm 1 for I and $k + 1$.

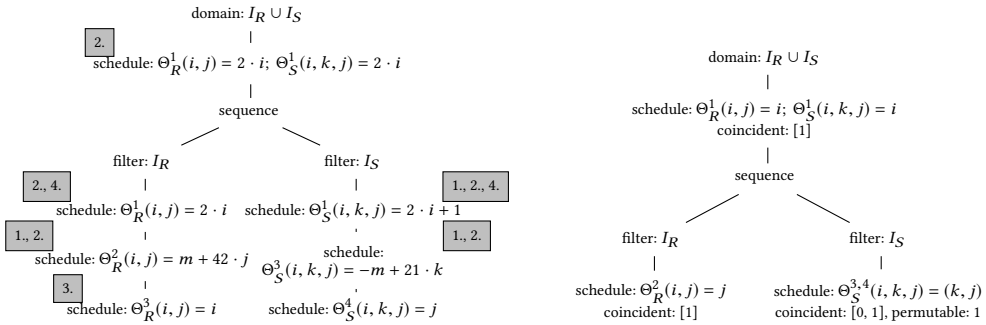

Fig. 2. Schedule tree of the schedule of Example 3.1, before (left) and after (right) simplification. The nodes that are affected by schedule tree simplification are each annotated with the relevant simplification steps.

Example 3.1. We pick up the running example from Section 2. The following (needlessly complicated) schedule prescribes the same order as the schedule in Example 2.2:

$$\Theta_R : I_R \to \mathbb{Z}^4;\ \Theta_R(i,j) = (2 \cdot i, 42 \cdot j + m, i, 0) \quad \Theta_S : I_S \to \mathbb{Z}^4;\ \Theta_S(i,k,j) = (2 \cdot i + 1, 0, 21 \cdot k - m, j).$$

By calling Algorithm 1 for Θ_R, Θ_S, we obtain the schedule tree shown in Figure 2 on the left. The initial call yields the band node and the sequence node directly underneath the domain node, as the first schedule dimension encodes textual order according to $\widetilde{<}^k$. The sequence node separates the two statements. Its subtrees result from calling Algorithm 1 individually for each of the statements and their schedule. Further sequence nodes cannot occur, as there are only two statements.

Simplification. The conversion of the schedule matrix to a schedule tree is followed by a simplification that preserves the tree's semantics and exposes tilable bands and parallelism. At any step during the simplification process, let I be the set of the statements' iteration domains that are scheduled in the subtree that we consider at this step. Let Θ be the original schedule and Θ' the result of simplifying Θ. One may verify that all of the proposed simplification steps comply with the following properties that are necessary for a simplifcation step to be semantics-preserving.

Execution Order. The relative execution order of statement instances is preserved. Formally:

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

Direction of Dependences. Given a data dependence from an instance of statement O with iteration vector $\vec{i}$ to an instance of statement T with iteration vector $\vec{j}$, their execution order must be retained with respect to any dimension k of the transformed iteration space. If $\Theta^k_T(\vec{j}) - \Theta^k$, has been eliminated by simplification, we do not need to consider it further. Otherwise, then let dimension k' of Θ' correspond to dimension k of Θ (the simplification may have changed the index). Prerequisite:

$$\operatorname{sgn}\big(\Theta^k_T(\vec{j}) - \Theta^k_O(\vec{i})\big) \ =\ \operatorname{sgn}\big(\Theta'^{k'}_T(\vec{j}) - \Theta'^{k'}_O(\vec{i})\big)$$

The second criterion is necessary, as the first does not distinguish between two schedules that only differ by a skew of a nested loop. This is particularly relevant because we simplify the schedules before the application of tiling. Essentially, this criterion preserves the forward only data communication in permutable bands. We apply the following steps to simplify schedule trees:

1. Remove the constant offset (which includes multiples of structure parameters) from each band node's one-dimensional schedule that applies to every statement.
2. Divide a band node's schedule's coefficients by their greatest common divisor (GCD).

Step 2 may modify loop strides with the goal of normalization. Polyhedral code generators adjust loop strides to avoid guards in loop bodies. Given the set of all statements in a loop body, Bastoul [13] determines the optimal loop stride as the GCD of the strides required by each of the statements and expressions that correspond to the required lower loop bounds.

3. Replace subtrees whose ancestors encode an injective schedule by leaves.

4. Delete band nodes whose one-dimensional schedule Θ^k does not assign different execution steps to any pair of statement instances that are executed in the same step according to $\Theta^{1,k-1}$. Formally, a band node with schedule Θ^k can be deleted if the following condition holds:

$$(\forall I_X, I_Y \in I : (\forall (\vec{i}, \vec{j}) \in I_X \times I_Y : \Theta_X^{k-1}(\vec{i}) = \Theta_Y^{k-1}(\vec{j}) \Rightarrow \Theta_X^k(\vec{i}) = \Theta_Y^k(\vec{j}))).$$

Vasilache [65] showed the validity of steps 1 and 2. Step 4 and the detection of degenerate loops by Grosser et al. [37] have the same purpose.

As we apply tiling as an additional schedule tree transformation after the simplification of the schedule trees, the modification of loop strides by Step 2 may affect the number of statement instances computed per tile. Since, for strides that exceed the chosen tile size, tiling would have no effect, loop strides should be reduced before code generation.

Minimizing schedule coefficients' absolute values is possible [35] but too expensive.

After simplification, we identify permutable bands and collect all in band nodes, which we mark permutable. Finally, we mark parallel dimensions in band nodes' schedules and, per statement, we identify schedule dimensions that yield loops. The latter is due to the noisiness of iteratively generated schedules and eases feature extraction (see Section 4).

Example 3.2. Figure 2 shows on the right the schedule tree of Example 3.1 after simplification. The nodes of the tree on the left are annotated by the simplification steps that affect them.

4 SCHEDULE TREE FEATURES

Section 3 describes how we prepare schedules for feature extraction. Now, we present schedule tree features that we use to learn a performance model and predict the profitability of schedules.

A feature must adhere to some properties to allow prediction. (1) It must be independent of a SCoP's size or complexity (e.g. the number of statements). Otherwise, models could only be transferred to equally structured SCoPs. (2) Its value can be normalized to the interval $[0, 1]$. Otherwise, different value ranges for the same feature of different schedules would complicate learning a pattern of profitable schedules. (3) Feature calculation must not exceed the transformed code's execution time. The features should express most of schedules' performance aspects.

We expect that the inclusion of program features would, on the one hand, yield more accurate models, but on the other hand, require a much large and comprehensive training set.

Schedule features are either structural, such as schedule matrices' sparsity, without an obvious relation to performance, or performance-related, such as coarse-grained parallelism. To define features, we use the symbols in Table 1. The features' definitions rely on the fact that our schedules are linearly affine and that a statement can only occur in one branch of a schedule tree.

4.1 Structural Features

These features are not obviously performance-related. We discuss their relation to performance.

Schedule Tree Depth (F_{Depth}). This feature is the depth of a schedule tree relative to the maximum depth that any simplified schedule tree produced by Polyite can have. While the true maximum is $\Gamma = |I| + \sum_{i=1}^{|I|} \dim(I_{S_i})$ (excluding domain and filter nodes), we observed that Polyite produces mostly schedule trees that have a depth lower than $\Gamma' = |I| + \max_{i=1}^{|I|} \dim(I_{S_i}) + 1$. Our schedule

trees' depths exceed this bound rarely. With γ denoting the actual depth, the normalized feature is $F_{\text{Depth}} = \gamma/\Gamma'$. The feature's value increases with loop distribution at different dimensions and one-dimensional bands that cannot be grouped in a permutable band. A schedule that, assuming the absence of data dependences in the SCoP, has exactly tree depth $\Gamma' = 5$ is $\Theta_X(i) = (i, 0, 0, 0, 0)^T; \Theta_Y(i) = (0, i, 1, 0, 0)^T; \Theta_Z(i) = (0, 0, 1, i, 1)^T; i \geq 0$.

Table 1. Symbols in features' descriptions

$\dim(I_S)$	Dimensionality of statement S's iteration domain
I	The set of all statements' iteration domains of a SCoP
$\dim(\Theta)$	Dimensionality of the schedule
$\mathcal{S}$	The number of sequence nodes in a schedule tree
$\mathcal{L}$	The number of leaves of a schedule tree
$\mathcal{A}$	Total number of memory access relations in a SCoP
Δ_i	The number of loops surrounding statement S_i, or 1 in the absence of loops ($\max(\dim(I_{S_i}), 1)$)

Let us explain the theoretical bound Γ. Excluding domain node and filter nodes, the maximum depth of any simplified schedule tree is $\Gamma = |I| + \sum_{i=1}^{|I|} \dim(I_{S_i})$. $|I|$ is the depth of a cascade of sequence nodes where, at each level, one statement's iterations become separated from those of the remaining statements. At the tree's bottom are leaf nodes. The second summand in the definition of Γ represents the case that every band node's schedule encodes just one loop around one statement.

Number of Sequence Nodes (F_{Seq}). A schedule tree contains at most $|I| - 1$ sequence nodes. The number is reached if each leaf node corresponds to exactly one statement and each sequence node has two children. Thus, the normalized number of sequence nodes in a schedule tree is $F_{\text{Seq}} = \mathcal{S}/(|I| - 1)$. The feature relates to loop distribution, and to how many loops are split.

Number of Leaves (F_{Leaves}). The number of leaf nodes is another feature that is related to the degree of loop distribution. To normalize this features' value, we rely on the fact that, in our case, a schedule tree has at most $|I|$ leaves. The feature is $F_{\text{Leaves}} = \mathcal{L}/|I|$.

It is not straightforward to relate the aforementioned three features to the profitability of schedules. The balance between loop distribution and loop fusion is a trade-off between data locality, on the one side, and activation of parallelism and tiling, on the other [17]. With this in mind, Pouchet et al. [59] propose to determine the optimal balance between loop distribution and loop fusion using iterative optimization and then schedule each loop nest in a model-driven way.

We could substantiate the widely held conjecture that the sparsity of schedule matrices correlates to efficient transformed code [35]. Thus, we specified two features to express this property.

Sparsity of Iterators' Coefficients (F_{SpIter}). This feature expresses the proportion of iteration variable coefficients that are zero. A high density of the iteration variables' coefficients increases the likelihood of skewing. Skewing shifts a loop's iterations by a multiple of the iteration variable of an encasing loop. While skewing can enable parallelism and tiling, it complicates index computations.

Sparsity of Structure Parameters' Coefficients and the Constant (F_{SpP}). This feature is defined as the proportion of structure parameters' coefficients and the constant's coefficients that are zero.

4.2 Performance-Related Features

Besides structural features, there are a number of features that bear a known relation to specific performance aspects. Since there are many ways of generating optimized code according to a given schedule, performance-related schedule features must be adapted to the code generator that is being used. For instance, a code generator can choose whether to parallelize the innermost parallelizable loop of a loop nest. A parallelism feature should reflect this choice. The following features reflect the behavior of Polly, which is the polyhedral code generator that we use in our experiments.

Parallelism (F_{Par}). Polly parallelizes each loop nest's outermost parallelizable loop. The more statements are encased by a parallelizable loop and the farther outside in the nest each parallelizable loop is, the larger the feature's value ought to be. The feature is $F_{\text{Par}} = \left(\sum_{i=1}^{|I|} \omega_i \cdot \frac{\dim(I_{S_i}) - \delta_i}{\Delta_i}\right) / \left(\sum_{i=1}^{|I|} \omega_i\right)$. Here, δ_i is the index of the outermost dimension in schedule Θ_{S_i} that generates a loop and does not carry data dependences. If the outermost such dimension generates the outermost loop that

encases S_i, we have $\delta_i = 0$. In our implementation, the ω_i are constantly 1 but, to increase the feature's expressiveness, we could use them to weigh statements by their number of instances and their amount of computation. The benefit of parallelism increases with the amount of computation.

Data Locality ($F_{DataLoc}$). CPUs access memory through a hierarchy of cache levels. Data is transferred between memory and CPU in cache lines (that is, blocks of data). Since cache size is limited, a replacement strategy must make space for new cache lines. Thus, to profit from caching, cache lines must be re-accessed with few accesses to other cache lines inbetween. An exact model of cache behavior would be an ideal locality feature. However, its computation seems too complex for practical use [10, 24]. Recent work on analytical modeling of cache behavior [10] can handle multi-level cache hierarchies and shows a promising tendency in observed time complexity.

Instead, we propose a computationally cheaper feature. The larger the volume of data communicated by the dependences in a dependence polyhedron $D_{O,T}$, the more deeply $D_{O,T}$ should be carried in the loop nest. Particularly in the case of a dependence that involves a large number of memory cells, good temporal data locality is important, because otherwise intermediate memory accesses to other memory cells can edge values out of the processor's caches before they can be reused. In this context, read-after-read dependences are relevant, too. To estimate the communicated data volume $\eta_{R,S}$, we consider all pairs of memory accesses of O and T. Per pair, we calculate with Barvinok's counting algorithm [69] the number of memory cells accessed by both the source and the target statement instances in $D_{O,T}$. $\eta_{R,S}$ is then the calculated volumes' sum. To be able to compute the volumes, we need to know the structure parameters' values. Say, the dependences in our SCoP are modeled by k dependence polyhedra and $d_{O,T}$ is the innermost schedule dimension that carries instances of $D_{O,T}$. The feature is $F_{\text{DataLoc}} = \left(\sum_{i=1}^{k} \eta_{O_i,T_i} \cdot d_{O_i,T_i}\right) / \left(\dim(\Theta) \cdot \sum_{i=1}^{k} \eta_{O_i,T_i}\right)$.

Tiling (F_{Tile}). Since tiling improves data locality, we express the extent to which rectangular tiling is applicable to a schedule. The version of Polly used tiles permutable bands whose children are leaves. In our tiling feature, we count, per statement S_i, the number λ_i of loops encoded in the tilable band node that is associated with the statement. The normalized feature is $F_{\text{Tile}} = \sum_{i=1}^{|I|} \frac{\lambda_i}{\Delta_i} / |I|$.

Memory Access Pattern (F_{MemAcc}). As described, a cache miss results not in the load of a single memory cell but of a contiguous cache line into the CPU cache. Therefore, efficient programs do not access memory randomly but, ideally, in a pattern along the innermost array dimension with a small, positive stride. Alternatively, an access may target the same cell for one iteration of the second-innermost loop. After transformation of the SCoP by the schedule, the feature is $F_{\text{MemAcc}} = |\mathcal{A}'|/|\mathcal{A}|$, where $\mathcal{A}'$ is the number of accesses adhering to one of the previous criteria.

4.3 Discussion

Together, our features cover the following aspects of a schedule: loop fusion and distribution, sparsity of the underlying schedule matrix, parallelization, temporal and spatial data locality, and tiling. All features' value ranges are (approximately) normalized to the interval $[0, 1]$. To avoid performance models that are only transferable among programs of very similar structure, no feature relates directly to the structure of the source program. The features addressing textual execution order are only reasonably applicable to schedules of SCoPs with at least two statements.

The proposed features are meaningful for optimizing loop programs that can be expected to profit from tiling and coarse-grained parallelization for execution on multicore-CPUs. We are convinced that an optimization for single-threaded processors, GPUs, or Intel Xeon Phi would require a different set of features. Moreover, to find schedules that permit offloading of computation to GPUs, for instance by using the PPCG compiler [70] as code generator, it would be necessary to restrict the considered set of schedules to ones that yield tiles that can be computed in parallel. Otherwise, one would encounter numerous schedules without this property.

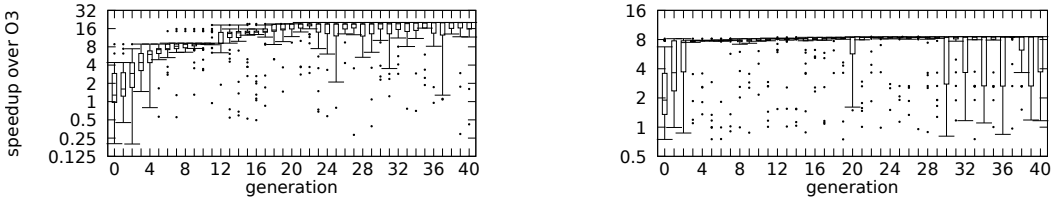

Fig. 3. Distribution of speedups yielded by the schedules in the GA's generations for `3mm` (left) and `adi` (right).

To cover programs that do not profit from coarse-grained parallelism or tiling, we could add program features and would essentially learn a model per class of programs. The code generator's configuration would have to be adapted per class of programs. Furthermore, the features correspond to a specific configuration of Polly as the code generator. Switching on loop transformations that require an enabling schedule, besides rectangular tiling, which we already cover, will make additional features necessary. Among such transformations are strip-mining [72] and diamond tiling [18]. Our features already cover multi-level tiling [43] in the rectangular case. The criterion for rectangular tiling is also sufficient for unroll-and-jam, which was noted by Sarkar [62] for the case of perfect loop nests. Other transformations, like simple loop unrolling, require no extra feature.

Table 2. Feature values of the tree in Figure 2.

F_{Depth}	0.67	F_{Par}	1.00
F_{Seq}	1.00	F_{DataLoc}	0.56
F_{Leaves}	1.00	F_{Tile}	0.58
F_{SpIter}	0.62	F_{MemAcc}	0.83
F_{SpP}	1.00		

Example 4.1. Table 2 shows the feature vector of the schedule tree in Figure 2 for `n=2600`, `m=2000`.

5 LEARNING A CLASSIFIER AND INTEGRATING IT IN THE GENETIC ALGORITHM

In this section, we describe how we learn a surrogate performance model to identify profitable schedules from feature vectors and execution times of previously benchmarked schedules. That is, we learn a function that approximates a schedule's profitability from the schedule's feature vector. We describe how we integrate this function into Polyite as a surrogate for benchmarking.

Initially, we used regression learning with schedule features closely related to performance to predict speedup. We obtained chastening results [27]. Two findings from the evaluation of Polyite's GA with benchmarking (GA_B) led us to devise a new approach based on classification. The first one is that the speedup yielded by the iteratively generated schedules converges surprisingly fast. Figure 3 illustrates this insight exemplary for the benchmarks `3mm` and `adi` of PolyBench 4.1. The GA's population size was 30 schedules and it ran for 40 generations (630 schedules in total). The boxplots in Figure 3 show the distribution of speedups over the sequential version (`-O3`) yielded by the schedules in each population. For `adi`, the optimum is almost reached in the initial population. For `3mm`, 90% of the maximum speedup are reached after 11 generations (195 schedules). The second finding is also apparent in Figure 3: the variance of speedups decreases fast. This is caused by Polyite finding many profitable schedules and the GA's elitism. These findings motivated us to use a classifier that identifies likely profitable schedules as a surrogate for benchmarking.

To learn a model for a classifier, we use CART or random forests. To build the training set for a classifier, we must label each training sample. That is, we must specify whether a certain schedule Θ is profitable for a program P. To this end, we classify Θ as profitable when the transformed code yielded by Θ runs, at least, half as fast as the code that results from the best schedule for P that is in the training set. If none of the schedules for P yields a speedup higher than 1.2, we conclude that P does not profit from optimization with Polyite and we label all schedules for P as unprofitable.

In our evaluation, we trained classifiers from different training sets. The outcome is always a CART or a random forest, but the classifier's structure depends on the training set used. The supplementary Web site [34] shows an illustration of a CART learned from the entire training data.

For running the GA with the classifier (GA_C), Polyite's GA must be altered because in the early generations profitable schedules may be rare. If more than half of the population is considered to be profitable, then a randomly chosen subset of these schedules survives and reproduces. Otherwise, all profitable schedules survive and as many randomly chosen unprofitable ones as are added to reach half the regular population size. All elected schedules can reproduce, but schedules classified as profitable are twice as likely to do so. The GA terminates after the classifier labels 95% of the schedules in the current population as profitable or after a set maximum number of generations.

A conservative option would be to use classification only as a guard for benchmarking.

6 EVALUATION

The overarching goal is to make Polyite's GA time-efficient to the point of practicality. We replace the time-consuming fitness assessment of schedules, which is based on benchmarking, with a classifier that uses a machine-learned surrogate performance model. Thanks to the classifier, only the schedules in the GA's final population require benchmarking. We train the classifier based on feature vectors that we extracted from schedules that result from previous iterative optimizations, for instance, from earlier versions of the same program. The classification of schedules must be computationally cheap to achieve a saving of optimization time. Moreover, the surrogate model must be transferable from the programs used for training to new programs. In the end, the whole optimization process must still obtain sufficient speedups.

We study the following research questions to address these and other properties.

RQ 1 : *Does the classification of schedules scale to large SCoPs?*
For the presented approach to be useful, the classification of schedules must be cheaper than the application of the schedules and the benchmarking of the transformed program. Also, it must scale for large SCoPs. To answer this question, we evaluate the computational cost of the schedule tree transformation (Section 3), feature extraction (Section 4), and the classification itself.

RQ 2 : *Are our schedule features sensitive to profitability of schedules?*
Not all features of Section 4 may be sensitive to the profitability of schedules. That is, not every feature's value may vary between schedules that yield different execution time of the transformed code. Insensitive features can be excluded from the feature vector.

RQ 3 : *Can our classifiers distinguish reliably between profitable and unprofitable program schedules?*
Is it possible to express the difference between profitable and unprofitable schedules for a program using the features of Section 4 and random forests? Can we learn a model on schedules of a set of programs and then use it to classify schedules of another program correctly?

RQ 4 : *Is our schedule classifier an acceptable surrogate for benchmarking?*
We must compare running GA_C (refer to Section 5) and GA_B with respect to two criteria. The first criterion is the time saved by the use of the classifier. The second one is the optimization's outcome.

6.1 Experiment Setup

Implementation. We extended the iterative polyhedral optimizer Polyite which is written in Scala (version 2.11). For machine learning, we use scikit-learn [55] (version 0.19.2). scikit-learn runs in a Python (version 3.5.3) session that is a child process of Polyite. To handle polyhedra, Polyite relies on Scala bindings for isl (commit `cfebc0c6`) and libbarnvinok (version 0.41).

Tool Chain. For SCoP extraction, tiling, and code generation, Polyite relies on the polyhedral optimizer Polly. For comparability of the results, we use the same modified version of Polly as in our previous study [35] (it is based on commit `2b618e01` of http://llvm.org/git/polly.git). The modification extends Polly's JSCOP format by an embedded schedule tree. Further, we enabled Polly to apply its schedule tree optimizations, such as tiling, to imported schedules.

Table 3. Characteristics of the benchmarks used and parameters of the training set for each benchmark.

benchmark	2mm	3mm	adi	atax	bicg	cholesky	correlation	covariance	deriche	doitgen	durbin	floyd-warshall	fdtd-2d	gemm	gemver	gesummv	gramschmidt	heat-3d	jacobi-1d	jacobi-2d	lu	ludcmp	mvt	nussinov	seidel-2d	symm	syr2k	syrk	trisolv	trmm
# stmts	4	6	9	3	2	4	13	7	11	3	10	4	1	2	4	3	9	2	2	2	3	20	2	8	1	5	2	2	3	2
# dep. polyhedra	6	10	64	4	4	8	19	12	25	8	43	24	18	2	6	3	23	171	16	56	8	89	2	24	59	21	2	2	5	4
max. loop depth	3	3	3	2	2	3	3	3	2	4	2	3	3	3	2	2	3	4	2	3	3	3	2	3	3	3	3	3	2	3
share prof. scheds	0.44	0.28	0.40	0.36	0.54	0.13	0.41	0.51	0.00	0.44	0.00	0.51	0.00	0.62	0.34	0.49	0.49	0.29	0.72	0.30	0.35	0.00	0.47	0.00	0.43	0.00	0.61	0.60	0.32	0.41
train. set size	1112	1060	1166	1184	1175	805	1136	1192	1070	1255	1231	832	590	1222	1165	1172	996	857	1167	1103	907	793	1086	1227	1201	1241	1209	1190	1094	1193

The relevant compiler flags that we use to tile (by a fixed tile size of 64) and generate code are: `-polly-parallel -polly-vectorizer=none -polly-tiling -polly-default-tile-size=64 -march=native`. We transform the program further with `-O3 -march=native`.

Benchmarking. The experiments that involve execution time measurements of transformed code were run on an Intel Xeon E5-2650 v2 CPU @ 2.6GHz with eight physical cores and 20MB of L3 cache. All other experiments ran on the Intel Xeon E5-2690 v2 CPU @ 3.00GHz with ten physical cores and 25MB of L3 cache. Hyperthreading and Intel Turbo Boost were disabled. The operating system was Debian 9.5 with Linux 4.9. We ran Polyite with OpenJDK 8. For **E 2** we used Ubuntu 16.04 with Linux 4.4 and Oracle JDK 8. We executed each transformed program version five times and took the shortest measured execution time. In an additional preceding run, we verified the computation result. We purged any schedule that yielded incorrect computation results. While all schedules are legal in theory, they can, for instance, trigger integer overflows in the transformed programs. A schedule's evaluation may fail for the following reasons: A timeout (five minutes for compilation and 30 minutes overall), failed compilation, and mis-compilation, which yields run-time errors. In previous work, we presented a statistics on these causes of failure.

Benchmark Set. We used the PolyBench 4.1 benchmark set, which contains 30 typical algorithms from application domains for which the polyhedron model is relevant. We excluded the programs `floyd-warshall` and `seidel-2d`, which are the only benchmarks that Polly models with a single statement. Since some of our schedule tree features are not meaningful for SCoPs with a single statement, models learned from the other benchmarks are not applicable to these programs. We use the built-in timer of PolyBench to measure execution time. We set the configurable data set sizes of PolyBench to "extra large" because, for smaller sizes, the arrays tend to fit into the L3 cache, which undermines data locality optimizations. Table 3 shows characteristics of the benchmarks.

Configuration. A schedule matrix's rows originate from a list of polyhedra, each row from another polyhedron. By configuration, a row is the result of choosing a point on one of a polyhedron's minimal faces and adding a linear combination of the polyhedron's bidirectional rays (lines) and a conical combination of its unidirectional rays. At most two rays and lines have non-zero coefficients with absolute values from $\{1, 2, 3\}$. Further, the sampling is configured such that schedules with outer parallelism can occur. We found that these particular settings permit random exploration to compete with GA_B [35]. The GA starts from a random population. In our previous study, we let it run for 40 generations (not counting the initial population) with population size 30 (630 schedules tested). We used these settings again to generate training sets. To assess the reduction in optimization time that results from the use of the surrogate performance model, we needed to know the execution time of GA_B. For most benchmark programs in PolyBench 4.1, the number of 40 generations is too high. We use it to ensure convergence against the reachable maximum speedup. Instead, we analyzed the runs that had generated the training sets and analytically determined a more suitable termination criterion for GA_B. The optimization will terminate if, from one generations to the next, the execution time yielded by the optimal schedule known did not reduce by more than 2% over eight generations. Considering a smaller number of generations

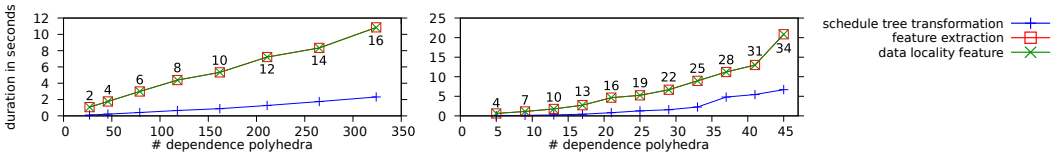

Fig. 4. Avg. duration of schedule tree transformation and feature extraction depending on the number of dependences for `floyd-warshall` (left) and `atax` (right). The data points' annotations are the corresponding numbers of statements. The duration of computing the data locality feature is also shown separately.

would often cause too early termination. To account for the GA's randomness, we used the average duration of five runs of GA_B with the restrictive termination criterion per benchmark.

For GA_C, we increased the population size to 50 schedules to increase the populations diversity. Also, an increased population size should not affect the processing time of GA_C substantially. The GA terminates either after 40 generations (1050 schedules) or as soon as 95% of the schedules have been classified as profitable, followed by benchmarking of the schedules in the final population.

Training Sets. We generated training sets by running GA_B over 40 generations (630 schedules) once per benchmark. We generated another 630 schedules per benchmark with random exploration, merged the two sets of schedules, and filtered for successfully benchmarked schedules. The result is a set that covers reasonably the profitability range of the schedules in which we are interested. We calculated all schedules' feature values and labeled them as profitable or unprofitable according to the rule described in Section 5. Table 3 shows characteristics of the training sets.

6.2 Experiments

E 1 : *Sensitivity of Features* (RQ 2). Given two schedules for a SCoP that yield strongly different execution times of the transformed code, we call a feature *sensitive* to performance if its value differs between the two schedules. The sparsity of parameters' coefficients and the constant appears to be the least sensitive feature. Its value is almost always between 0.75 and 1 and barely differs between schedules with different profitability. Thus, we did not use the feature in the other experiments.

E 2 : *Scalability of Schedule Simplification and Feature Extraction* (RQ 1). To examine how well schedule tree transformation, simplification, and feature extraction scale with the SCoP size, we used a semantics-preserving technique proposed by Upadrasta and Cohen [64] to scale the number of statements and dependences of a SCoP by partially unrolling outer loops. Per benchmark and number of unrolled loop iterations, we generated 100 schedules. Then, per schedule, we measured the duration of each step of the schedule tree construction and simplification, and of feature extraction. Figure 4 shows, per number of dependences, the average duration of the schedule tree construction and simplification, the total duration of feature extraction, and the duration of calculating the data locality feature for the benchmarks `atax` and `floyd-warshall` (modeled with two statements in the unrolled version). Note that, here, an increase of the number of dependences implies an increase of the number of statements and schedule coefficients. One can see from the plot of `floyd-warshall` that, asymptotically, the duration of the schedule tree construction and feature extraction grows faster than linearly. In total, we have evaluated ten benchmarks. The results are on the supplementary Web site [34]. The calculation of the data locality feature always dominates feature extraction. By analyzing the algorithm to calculate the data locality feature, we found that the costliest part is to determine the communicated volumes of data. The cost of this step depends strongly on the number of memory access relations in the SCoP that address the same memory location. This cost can be leveraged partly by caching results.

Table 4. Average results per combination of benchmark and p for the optimal configuration of random forest that we determined in **E 3**. Listed are, per benchmark and p, the average rate of false positives among the unprofitable schedules, the average rate of false negatives among all profitable schedules, among the schedules that yield ≥ 80% of the speedup, and among the schedules that yield ≥ 95% of the speedup. Finally, we show the average rate of miss-predictions per combination of benchmark and p. In the absence of profitable schedules for a benchmark the share false negatives is undefined, which we indicate by dashes.

	p	2mm	3mm	adi	atax	cholesky	correlation	covariance	doitgen	durbin	fdtd-2d	gemm	gemver	gesummv	gramschmidt	heat-3d	jacobi-1d	jacobi-2d	lu	ludcmp	mvt	nussinov	symm	syr2k	syrk	trisolv	trmm	**row avg.**
% false pos.	**1**	5	12	4	5	4	0	6	8	0	0	0	0	11	6	12	0	22	5	1	1	0	59	1	0	10	6	0	9	**7**
% false neg.	**1**	14	5	100	83	94	100	96	54	-	95	-	100	14	93	86	100	72	96	100	99	-	16	-	-	8	60	100	72	**72**
% false neg. ≥ 80%	**1**	4	1	100	95	95	100	98	57	-	100	-	99	15	96	94	100	57	100	100	99	-	29	-	-	5	71	100	77	**74**
% false neg. ≥ 95%	**1**	2	2	100	100	95	100	100	92	-	100	-	99	57	100	95	100	0	100	100	98	-	0	-	-	2	100	100	83	**75**
total % mispred.	**1**	9	10	43	33	53	14	42	31	0	42	0	50	13	36	48	49	37	70	31	35	0	39	1	0	8	39	32	34	**28**
% false pos.	**2**	4	14	4	6	4	1	7	8	0	0	0	0	11	6	11	0	15	3	1	3	0	47	1	0	9	7	2	**7**	**6**
% false neg.	**2**	15	5	97	71	92	100	95	50	-	97	-	100	14	89	85	100	82	97	100	99	-	24	-	-	7	56	99	75	**72**
% false neg. ≥ 80%	**2**	5	1	97	87	92	95	96	52	-	99	-	100	14	92	91	100	84	100	100	99	-	14	-	-	5	65	89	76	**72**
% false neg. ≥ 95%	**2**	3	2	99	88	92	100	100	85	-	99	-	100	60	100	92	100	44	100	100	99	-	0	-	-	2	91	99	81	**76**
total % mispred.	**2**	9	11	42	29	52	14	43	30	0	43	0	51	13	34	47	49	35	70	30	36	0	36	1	0	8	36	33	35	**28**
% false pos.	**3**	4	14	5	6	4	1	7	8	0	0	0	1	12	6	11	0	12	3	1	3	0	42	1	0	10	7	2	6	**6**
% false neg.	**3**	16	5	95	68	90	100	95	49	-	98	-	100	14	87	85	100	85	97	99	99	-	28	-	-	8	55	99	71	**71**
% false neg. ≥ 80%	**3**	6	1	95	85	90	96	96	50	-	99	-	100	14	91	91	100	85	100	99	99	-	13	-	-	5	63	92	68	**71**
% false neg. ≥ 95%	**3**	4	2	99	85	90	100	100	77	-	99	-	100	61	99	91	100	49	100	99	99	-	0	-	-	3	88	98	72	**75**
total % mispred.	**3**	9	11	41	28	51	14	43	29	0	43	0	51	13	33	47	49	33	70	30	37	0	36	1	0	8	36	33	33	**28**
% false pos.	**4**	5	14	6	6	4	1	7	9	0	1	0	1	12	6	11	0	9	3	1	3	0	40	1	0	10	8	2	6	**6**
% false neg.	**4**	17	5	93	67	89	100	95	48	-	98	-	100	14	87	85	100	89	97	99	99	-	30	-	-	8	54	98	71	**71**
% false neg. ≥ 80%	**4**	7	1	94	85	89	96	96	48	-	99	-	100	14	91	90	100	87	100	99	99	-	12	-	-	5	62	94	66	**71**
% false neg. ≥ 95%	**4**	5	2	98	85	89	100	99	70	-	99	-	100	61	98	90	100	55	100	99	99	-	0	-	-	3	88	97	70	**74**
total % mispred.	**4**	10	12	41	28	50	14	43	28	0	43	0	51	13	33	47	49	32	70	30	37	0	35	1	0	8	35	32	32	**28**
% false pos.	**5**	5	15	6	6	5	1	7	9	0	1	0	1	12	6	11	0	7	3	1	4	0	38	1	0	10	8	2	6	**6**
% false neg.	**5**	19	5	92	67	88	99	95	47	-	98	-	100	14	86	84	100	91	97	98	99	-	31	-	-	8	54	98	69	**71**
% false neg. ≥ 80%	**5**	9	1	93	84	88	96	96	46	-	99	-	100	14	90	89	100	89	100	98	99	-	11	-	-	5	62	95	62	**71**
% false neg. ≥ 95%	**5**	6	2	98	84	88	100	99	66	-	99	-	100	61	98	89	100	57	100	98	99	-	0	-	-	3	87	97	65	**74**
total % mispred.	**5**	11	12	41	28	50	14	43	28	0	43	0	51	13	33	46	49	32	70	30	37	0	35	1	0	8	35	32	31	**28**

E 3 : *Leave-p-Out Across Benchmarks* (RQ 2, RQ 3). In this experiment, we determined the extent to which knowledge regarding schedules' profitability can be transferred from one program to another. Further, we tested different configurations of the learning algorithm.

We removed the schedules of p benchmarks from the training set and learned a performance model on the remaining ones. Then, we classified the removed schedules and verified the predictions. This way, we were able to determine the extent to which our performance models are transferable to unseen programs. The schema is known as *leave-p-out*. We scaled p from 1 up to 5 and tested all combinations. Per combination, we investigated the choice between two classifiers – random forests and CART – with respect to classification accuracy. Moreover, since hyper-parameters of machine-learning algorithms can have a substantial influence on the classifiers' accuracy, we tuned the most important ones. We tuned the minimum number of data points that are classified in a leaf node of a tree (`min_samples_leaf`). This parameter controls the tree's depth and, if set properly, prohibits overfitting. We tested the values 10, 15, and 20. With random forest, the number of trees in a forest (`num_trees`) is another parameter that requires tuning. More trees yield more stable predictions, but longer computations. We tested with 100, 200, and 300 trees. Each tree is learned with a randomly chosen subset of the features (`max_features`). We tested the values 4, 6, and 8.

CART yields twice as many false positives at median as random forest. Among the random forest configurations, the following yields the lowest median number of false negatives: `num_trees = 100`, `max_features = 8`, `min_samples_leaf = 20`. We used this configuration in all other experiments.

Table 4 shows results for this optimal configuration of random forest. The average share of false negatives decreases from 72% for $p = 1$ (75% for schedules with ≥ 95% of max. speedup) to 71% for $p = 5$ (74% for schedules with ≥ 95% of max. speedup), while the share of false positives is low. Some benchmarks have an extreme number of false negatives.

To characterize these benchmarks, we computed the features' importance for splitting each benchmark's training set into profitable and unprofitable schedules. The heat map in Figure 5

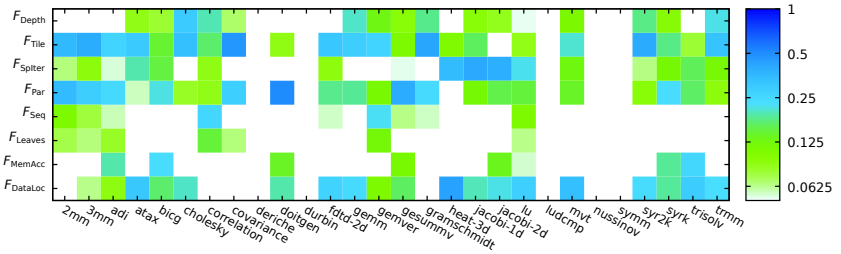

Fig. 5. A heat map that shows, per benchmark, the GINI importance of each feature.

shows the GINI importance [49] per feature and benchmark. The *GINI importance* of a feature F is the sum of the decreases of GINI impurity at a decision tree's nodes whose splitting criterion is based on F. The *impurity decrease* is the difference between the GINI impurity at a node and the weighted sum of the GINI impurity at its children. In case of random forests, the value is normalized by the number of trees. On average, tiling is the most important feature, followed by parallelism and data locality. Next comes the sparsity of iteration variable coefficients. We also calculated GINI importances from all benchmarks' training sets at once. Parallelism and tiling are almost equally important, followed by data locality, and the sparsity of iteration variable coefficients.

To compute the GINI importances, we learned random forests with the following configuration: `min_samples_leaf = 20`, `max_features = 2` and `num_trees = 500`. The large number of trees per random forest and the low number of features that we used to learn each tree in the forest result in a diverse set of trees in each forest. In contrast, deriving the GINI importances from a single CART learned with all features would primarily identify the most important feature per benchmark, but it would not yield strong evidence for the other features.

F_{Par} and F_{Tile} are unimportant or less important for `correlation`, `jacobi-1d`, `jacobi-2d`, `lu`, and `trisolv`. With our transfer-learned classifiers, accurate predictions are not possible for these programs. For `fdtd-2d`, `adi`, `gramschmidt`, and also `correlation` F_{Par}, and F_{Tile} have unusually low values. For `bicg`, F_{Tile} is unimportant and lower values for F_{Par} are better. For `cholesky`, F_{Tile} is important and for most profitable schedules the feature's value is 1, but lower values of F_{Par} are better. In the case of `doitgen`, F_{Tile} is unimportant and the values of F_{Par} are unusually low. The schedules of `gemver` cannot be classified correctly due to a combination of effects.

E 4 : ***k*-Fold Cross-Validation on the Full Training Set (RQ 2, RQ 3).** In addition to **E 3**, we carried out k-fold cross-validation on the full training set. That is, instead of leaving all schedules of p benchmarks out of the training set, we removed a share of $1/k$ ($k = 5, 6, \ldots, 20$) randomly chosen schedules from the full training set. Then, we learned a performance model from the remaining schedules using the configuration that we had elected in **E 3**. Using the model, we classified the removed schedules. Per k, we repeated the experiment until the determined shares of false negatives among profitable and false positives among unprofitable schedules became significant. Table 5 shows exemplary results for $k = 5, 10, 15, 20$. The complete data is on the supplementary Web site [34]. The rate of mispredictions decreased from 9.64% for $k = 5$ to 9.41% for $k = 20$.

E 5 : ***Cost-Benefit-Analysis of Replacing Benchmarking with Classification* (RQ 4).** In this experiment, we evaluated the cost and the benefit of replacing the GA's fitness function based on benchmarking with a classifier using a machine-learned surrogate performance model.

Per benchmark program, we used two measures to assess the classifier's influence on the GA. The first measure is the transformed code's execution time after the application of each schedule compared to the program's execution time after it has been compiled with `-O3`. We can specify the speedup yielded by each schedule. The second measure is the execution time of the GA plus the time needed to benchmark each schedule in the final population. This includes the time needed for

Table 5. Exemplary results from **E 4**. The complete data is on the supplementary Web site [34]. In the absence of profitable schedules a benchmark's share of false negatives is undefined, which we indicate by dashes.

	k	2mm	3mm	adi	atax	bicg	cholesky	correlation	covariance	deriche	doitgen	durbin	fdtd-2d	gemm	gemver	gesummv	gramschmidt	heat-3d	jacobi-1d	jacobi-2d	lu	ludcmp	mvt	mussinov	symm	syr2k	syrk	trisolv	trmm	**total**
% false neg.	**5**	12	5	23	11	10	14	33	19	-	18	-	17	5	14	13	14	48	14	46	38	-	17	-	-	6	10	21	16	**17**
% false pos.	**5**	3	6	8	5	6	2	9	9	0	6	0	5	10	8	11	2	13	4	13	9	0	9	0	0	10	9	10	9	**6**
total % mispred.	**5**	7	6	15	7	8	4	19	15	0	11	0	11	7	10	12	8	23	12	23	19	0	13	0	0	8	10	14	12	**10**
% false neg.	**10**	12	5	23	11	10	14	32	20	-	17	-	16	5	13	13	13	48	15	46	37	-	17	-	-	6	10	21	16	**16**
% false pos.	**10**	4	6	8	5	6	2	9	10	0	6	0	5	10	7	11	2	13	4	14	8	0	9	0	0	9	9	10	7	**6**
total % mispred.	**10**	7	6	14	7	8	3	19	15	0	11	0	11	7	9	12	7	23	12	23	18	0	12	0	0	7	10	13	11	**9**
% false neg.	**15**	11	5	24	11	10	14	32	20	-	17	-	15	5	13	13	13	47	15	45	38	-	17	-	-	6	10	20	16	**16**
% false pos.	**15**	4	6	8	5	6	2	9	10	0	6	0	5	11	7	11	2	13	4	14	8	0	9	0	0	9	9	10	7	**6**
total % mispred.	**15**	7	6	14	7	8	4	19	15	0	11	0	10	7	9	12	7	23	12	23	19	0	13	0	0	7	10	13	11	**9**
% false neg.	**20**	11	5	23	11	10	16	32	20	-	17	-	15	5	14	13	13	46	15	46	37	-	17	-	-	6	10	20	16	**16**
% false pos.	**20**	3	6	8	5	6	2	9	10	0	6	0	5	10	7	12	2	14	4	14	8	0	9	0	0	9	9	10	7	**6**
total % mispred.	**20**	7	6	14	7	8	4	19	15	0	11	0	10	7	9	12	7	23	12	23	18	0	13	0	0	8	10	13	11	**9**

the classification of schedules. We assume that the training data for the classifier is already available. To account for the randomness of the search, we optimized ten times with each configuration.

To generate schedules, we used a single thread. Also, the benchmarking of transformed program versions was carried out on a single CPU. Polyite may classify 20 schedules in parallel.

This experiment has two steps. In the first step, we guided GA_C with a model learned from schedules of the program to be optimized. This allowed us to determine how well GA_C performs when it is guided by a performance model that resembles well the program to be compiled. Here, we labeled any schedule as profitable that yields at least 50% of the speedup yielded by the best schedule in the program's training set. This deviates from the labeling rule described in Section 5. We use the median of the maximum speedups yielded by the ten runs as GA_C's result for a benchmark.

In the second step, we used the leave-one-out schema to evaluate how well transfer-learned performance models direct GA_C. We learned the models from the training sets of all programs except the program to be optimized. We labeled the training data as described in Section 5.

As the baseline, we ran GA_B with the termination criterion described in Section 6.1 five times per program. We used the median of the maximum speedups reached as a baseline for GA_C.

As a second baseline, we generated randomly ten sets of schedules per benchmark. Each set contained 50 schedules, which equals the size of GA_B's final population. If GA_C's final population does not differ from randomly generated schedules with respect to the distribution of the speedups yielded by the schedules, GA_C is nothing but random exploration. As a simple test, the schedules in GA_C's final population should yield higher average speedups than schedules generated randomly.

Table 6 shows the results of both of the experiment's steps and the baseline. We list each program's sequential execution time (-O3), the speedup over sequential execution yielded by Polly and the speedup yielded by a single run of GA_B running for 40 generations (GA_{B40}). Next, we list the median number of generations produced by GA_B and the median duration of the optimization that we measured thereby. It follows the median of the maximum speedups in execution time of the program to be optimized yielded by the best schedules found by GA_B in each run. Next, we report the results from the runs of random exploration. Per benchmark, we show the arithmetic mean of the speedups yielded by all schedules in all of the sets that we generated randomly. Also, we determined the maximum speedup yielded by the schedules in each set and determined the median of the maxima. The last three columns contain the results from running GA_C. We report the median time saved compared to GA_B, across the 10 GA_C runs per benchmark, the median of the maximum speedups in execution time of the program to be optimized across the replicated runs, and the arithmetic mean of the speedups yielded by all schedules in GA_C's final populations.

In the following, the term "average" refers to the geometric mean since we average normalized data across different benchmarks [32]. A comparison of the optimal schedules found by each run

Table 6. Results from **E 5**. The first ten columns show the baseline. On the right side, the first value per cell corresponds to the well-fitting classifier; the second value corresponds to the leave-one-out schema.

benchmark	exec. time O3 (sec.)	Polly speedup	GA_{B40} speedup	GA_B median # gen.	GA_B median # scheds. tested	GA_B median duration (min.)	GA_B median max. speedup	rand. mean speedup	rand median max. speedup	GA_C % time saved	GA_C median max. speedup	GA_C mean speedup
2mm	37.75	13.81	18.88	15	255	340.37	19.88	4.46	15.47	81.17 / 75.75	17.90 / 18.10	10.87 / 9.36
3mm	51.33	13.39	20.20	18	300	783.15	18.83	2.55	11.37	61.82 / 63.44	15.60 / 16.46	7.79 / 6.97
adi	171.91	7.61	8.57	9	165	1139.65	8.41	2.55	7.90	66.52 / 28.18	8.03 / 7.71	5.08 / 1.76
atax	0.01	0.99	3.09	13	225	22.77	3.02	0.77	2.78	43.81 / 48.35	2.48 / 2.59	1.51 / 1.29
bicg	0.01	1.05	2.79	7	135	18.18	2.78	1.10	2.77	49.27 / 34.37	2.78 / 2.76	1.74 / 1.47
cholesky	13.47	2.68	5.81	9	165	3139.32	6.40	1.31	3.32	76.42 / 72.76	5.03 / 2.12	1.52 / 0.90
correlation	53.11	33.36	43.81	10	180	345.82	36.69	7.19	32.27	-14.29 / 20.70	32.04 / 31.23	11.06 / 13.76
covariance	53.09	33.44	42.24	7	135	160.18	36.40	11.35	35.17	44.38 / 7.20	36.22 / 36.17	23.72 / 23.80
deriche	0.86	1.07	1.12	7	135	282.65	1.07	0.67	1.01	63.94 / 34.65	1.02 / 0.83	0.68 / 0.64
doitgen	5.48	4.06	5.63	16	270	202.45	4.86	1.87	4.10	64.86 / 64.16	4.21 / 4.20	2.68 / 3.10
durbin	0.02	0.99	1.18	7	135	24.55	1.00	0.66	1.00	53.90 / -135.23	1.00 / 1.00	0.71 / 0.65
fdtd-2d	26.81	0.92	3.80	9	165	620.13	3.32	0.74	2.67	65.37 / 60.41	2.76 / 2.55	1.39 / 1.34
gemm	8.95	4.23	4.72	7	135	111.83	4.93	2.07	4.39	48.48 / 67.43	4.41 / 4.41	3.05 / 3.19
gemver	0.10	3.78	6.27	15	255	34.37	6.07	0.83	2.75	36.15 / 37.83	4.45 / 4.50	1.42 / 2.53
gesummv	0.03	3.13	8.99	7	135	15.70	9.04	2.74	8.93	30.15 / 18.05	9.00 / 8.82	6.35 / 4.99
gramschmidt	42.20	8.10	11.17	11	195	489.08	8.56	1.94	7.99	65.40 / 53.58	8.27 / 6.55	5.26 / 1.53
heat-3d	37.31	2.68	4.19	13	225	1211.03	3.65	0.82	2.81	67.13 / 61.06	2.60 / 2.94	1.16 / 1.06
jacobi-1d	0.01	0.88	1.37	7	135	21.35	1.37	0.62	1.36	66.00 / 10.23	1.36 / 1.35	1.01 / 0.75
jacobi-2d	32.82	1.13	4.21	7	135	600.45	4.19	0.97	4.11	66.55 / 40.99	4.16 / 2.78	2.15 / 1.16
lu	71.53	5.38	9.86	14	240	5772.47	9.44	2.24	6.99	76.97 / 75.46	7.75 / 6.39	5.14 / 1.70
ludcmp	69.19	0.97	1.03	7	135	3190.17	1.02	0.97	1.01	72.38 / 63.30	1.01 / 1.01	0.98 / 0.97
mvt	0.08	4.65	8.25	8	150	19.23	8.10	2.46	6.44	36.61 / 27.04	6.31 / 5.89	3.49 / 2.36
nussinov	80.98	0.96	0.98	7	135	973.80	0.97	0.96	0.97	52.25 / 71.29	0.97 / 0.97	0.96 / 0.96
symm	27.41	1.01	1.02	7	135	303.42	1.02	1.00	1.01	48.18 / 42.38	1.01 / 1.01	1.00 / 0.98
syr2k	91.40	21.57	25.97	7	135	197.87	25.99	11.72	25.80	68.46 / 74.18	25.91 / 25.90	19.88 / 20.68
syrk	18.36	13.76	17.27	7	135	105.07	16.14	6.95	16.47	56.64 / 63.09	16.17 / 15.54	11.23 / 10.96
trisolv	0.01	0.36	1.06	7	135	15.35	1.00	0.48	1.33	-6.62 / -5.37	1.00 / 1.00	0.83 / 0.75
trmm	18.86	2.02	22.43	10	180	146.17	22.41	6.42	21.38	70.23 / 57.94	21.46 / 19.83	11.56 / 8.76

of GA_C and random shows that often random performs surprisingly well. On average, the optimal speedups yielded by GA_C are higher in case of the well-fitting classifier (factor 1.04) and lower for leave-one-out (factor 0.97) (1). Yet, for most benchmarks, the arithmetic mean of the speedups yielded by the all schedules in GA_C's final populations is higher than the mean of randomly generated ones (2). Consequently, GA_C requires less benchmarking on the target hardware than random under the precondition that suitable training data is available already. Of course, we assume that the target hardware was available for the generating of the training data. Recall that random exploration is configured such that it tends to generate sparse schedule matrices, and outer parallel schedule dimensions are likely to occur. We illustrate the distribution of the speedups, in case of random and GA_C, using box plots on the supplementary Web site [34]. The best schedules found by GA_C are faster by an average factor of 1.51 than the schedules found by Polly in case of the well-fitting classifier (3) and by 1.40 in case of the leave-one-out schema (4). We tested these differences for significance using a paired Mann-Whitney U Test [3] with false discovery rate control [15] ((1) $p = 0.0595$ (well-fitting classifier); $p = 0.16380$ (leave-one-out); (2) $p = 2.8 \cdot 10^{-8}$ (well-fitting classifier); $p = 0.00307$ (leave-one-out); (3) $p = 1.1 \cdot 10^{-5}$; (4) $p = 0.00064$). The significance test is inconclusive regarding (1). The optimization with GA_C is faster by an average factor of 2.39 in case of the well-fitting classifier and by 1.96 in case of the leave-one-out schema. In case of some programs with an execution time below one second of their sequential version, the use of GA_C yields a slowdown in optimization time. Furthermore, GA_C with the well-fitting classifier is slower at median than GA_B for `correlation`. Each run of GA_C had tested 1050 schedules in this case while GA_B had tested only 180 schedules at median. The time needed to generate the 1050 schedules used up the time saved by using the classifier.

The reduction in the time needed for optimization and benchmarking comes at the price of a lower speedup yielded by the optimal schedules found compared to GA_B. On average, the speedup

Table 7. Further results from the evaluation of GA_C in the leave-one-out schema.

benchmark	2mm	3mm	adi	atax	bicg	cholesky	correlation	covariance	deriche	doitgen	durbin	fdtd-2d	gemm	gemver	gesummv	gramschmidt	heat-3d	jacobi-1d	jacobi-2d	lu	ludcmp	mvt	nussinov	symm	syr2k	syrk	trisolv	trmm
median % profitable	82	80	75	78	84	64	94	96	72	68	0	63	96	79	87	0	74	78	57	65	0	93	80	0	96	96	78	71
% early exit	0	0	0	0	0	0	50	80	0	0	0	0	90	0	10	0	0	0	0	0	0	50	0	0	80	60	0	0
median # gen.	40	40	40	40	40	40	27	23.5	40	40	40	40	11.5	40	40	40	40	40	40	40	40	40	37	40	40	20.5	23.5	40	40
median # scheds tested	1050	1050	1050	1050	1050	1050	725	637.5	1050	1050	1050	1050	337.5	1050	1050	1050	1050	1050	1050	1050	1050	1050	975	1050	1050_+	562.5	637.5	1050	1050

yielded by GA_C is slower by the factor 0.91 in the case of the well-fitting classifier, and by 0.85 in case of the leave-one-out schema. Table 7 shows additional results from the leave-one-out schema.

Running GA_B with the new termination criterion rather than for a fixed number of 40 generations (630 schedules) reduces the speedup of the optimal schedule only by the average factor 0.95.

6.3 Discussion

From the result of **E 2**, we conclude that the answer to **RQ 1** is that schedule tree construction / simplification and feature extraction scale to some extent but not endlessly. Empirically, we found that both steps have a time complexity that is worse than linear. Our algorithms rely heavily on integer linear programming (ILP). In theory, the time complexity of ILP solving is exponential but, in practice, chances are good that an integer linear program can be solved in polynomial time [28]. The computation of our data locality feature involves Barvinok's counting algorithm, whose execution time is exponential in the input's dimensionality [11]. Thus, the time needed to compute the data locality feature grows primarily with the number of schedule coefficients.

In **RQ 2**, we asked whether all of the features in Section 4 are useful for prediction. In **E 1**, we found that the sparsity of structure parameters is largely insensitive to speedup and purged it.

In **E 3**, we found that classifiers based on transfer-learned performance models fail to recognize many profitable schedules. By contrast, the models yielded few false positives. An analysis of the features' importance and value distribution per program largely explains why transfer to some programs is complicated. There are several programs, such as `gemm` and `syrk` in PolyBench 4.1, that are transformed best by fully tiling each loop nest and parallelizing its outermost loop. These programs' simplicity may bias our performance models and decrease their applicability to more complex programs. Indeed, in **E 3** we observed that reducing the number of benchmark programs in the training set does not decrease, but slightly increases the classifiers' average accuracy. Our features' approximative nature may add to the classifiers' inability to recognize some profitable schedules: in the presence of partially fused loops, our parallelism feature may be unable to recognize some parallelism. Yet, a precise identification of parallelism is generally impossible due to parametric loop bounds and unknown parameter values. While we could identify partial loop fusion and model it using sequence nodes in schedule trees, we would still be unable to decide for every loop whether it will actually be executed at run time. The findings in **E 3** indicate that a normalization of the feature values may also be helpful to increase the classifiers' precision.

The k-fold cross-validation on the entire training data in **E 4** shows that a comprehensive training set yields accurate predictions. In summary, the answer to **RQ 3** is that, to a good extent, models learned from the results of previous iterative optimization can be used to recognize profitable schedules of new programs. The success depends on the similarity of the programs involved. There are very profitable schedules of many benchmarks that we do not recognize as such. The inclusion of program features in our feature vector would certainly improve our performance models' accuracy, but would also require a larger and comprehensive training set of programs. If learned from a carefully chosen training set, such a model might be widely applicable. Here, we have studied to which extent one can abstain from program features and learn from data that is available.

The answer to **RQ 4** is that a classifier that is based on a transfer-learned surrogate performance model can help to accelerate the iterative optimization of new programs. Sometimes, the profitable schedules cannot be recognized, though, because the characteristics of profitable schedules for the program to be optimized differ too much from those of profitable schedules for the programs in the training set. Also, the schedules found by GA_C are a bit less profitable than the schedules that can be found with GA_B. While GA_B never loses its best schedule found, GA_C is not guaranteed to keep it. Yet, the average performance of the schedules in GA_C's final population is significantly higher than the average performance of schedules generated by an informed random exploration. Regarding the speedup yielded by the optimal schedule found, random exploration often performs surprisingly well, but it tests more ineffective schedules on the target hardware than GA_C. Under the precondition that suitable training data is available this is a benefit.

For GA_B, we set the population size to 30 and let it run for 40 generations (630 schedules). We had increased this size to 50 for GA_C and still let it run for at most 40 generations (1050 schedules). The motivation was to increase the search space coverage by increasing the population size, since our classifiers often miss actually very profitable schedules. To investigate this choice, we let our GA with classification run again, but with a population size of 30 and for at most $\lceil (50/30) \cdot 40 \rceil = 67$ generations (1035 schedules). Again, we optimized each benchmark ten times. On average, GA_C with the reduced population size terminates after 54 generations (840 schedules). Compared to the original configuration, the median share of schedules classified as profitable is 2.46% higher, on average. The paired Mann-Whitney U Test is inconclusive regarding this difference. Although the larger number of generations appears to improve the share of profitable schedules in GA_C's final population, the larger population size of 50 schedules increases the coverage of the search space.

Finally, let us explain the higher performance of the code optimized by Polyite compared to Polly's result exemplary for `trmm` and `jacobi-2d`. Polly yields a speedup of 2.02 over `-O3` for `trmm`, while Polyite yields 22.43 at best. Both, Polly and Polyite segregate `trmm`'s statements into fully separate loop nests. While both schedules enable tiling of both loop nests, Polyite interchanges the loops in the first nest. For `jacobi-2d`, the speedup yielded by Polly is 1.13 while Polyite yields 4.21. Polly fuses the two statements and cannot parallelize while Polyite splits the loop nest inside the time loop and both inner loop nests can be parallelized at their outermost level.

6.4 Threats to Validity

Threats to Internal Validity. (Reproducibility of results) The search space of legal schedules for a SCoP has an enormous size. Conceptually, it is infinite, but even the size of the finite subset of reasonable schedules is thousands to millions strong. Our search space exploration is necessarily incomplete and repeated runs may have different results. To mitigate the effect of randomness on **E 5**, we executed each tested configuration repeatedly. Tables 6 and 7 show averaged results.

To control execution time measurement bias, we took several precautions. We benchmarked every transformed program version five times. In the case of the very short time measurements in **E 2**, we measured repeatedly until the mean result was significant. We disabled hyper-threading and Intel Turbo Boost since they are known to destabilize time measurements. Our benchmarking machines ran no other workloads in parallel. **E 5** ran on machines with a two-socket NUMA design. We pinned Polyite to one socket and the benchmarking of program versions to the other.

Threats to External Validity. (Generality of conclusions drawn) We use the benchmark set PolyBench 4.1. It contains algorithms that occur in application domains for which the polyhedron model is relevant. Yet, the benchmark set's small size limits our conclusions' generality. Furthermore, our approach currently targets primarily programs that profit from coarse-grained parallelism. Its applicability to very short running loop nests that operate on small data sets is limited.

7 RELATED WORK

There is wide consent that machine learning and iterative optimization, or combinations of both, are necessary to increase a compiler's awareness of programs and hardware. Hand-crafted heuristics frequently cannot deliver optimal performance. Therefore, we proposed the iterative schedule optimizer Polyite [35], whose theoretical foundation is the iterative optimizer LeTSeE by Pouchet et al. [56, 57]. LeTSeE optimizes for sequential execution without tiling. The search space is more restricted than the one of Polyite. Notable is the thorough search space sensitivity analysis.

There is also agreement that purely iterative compilation is too inefficient due to the enormous number of configurations or program transformations that must be tested to reach optimality. Machine-learning can help to reduce the effort. Most approaches that we found in literature share their foundation on program characteristics, which rely on either static program features, such as the number of instructions in a loop body, or dynamic features, such as performance counters.

Primarily, there are two ways of using machine-learned models in compilation. One way is to use them to prune the search space of iterative optimization to an acceptable number of options to be tested. Another way is to learn a model to directly select the best option.

Combined Machine-Learning and Iterative Approaches. Park et al. [53] evaluate different modeling techniques for an iterative search for a good compiler phase sequence. They rely on performance counters to characterize programs. Ashouri et al. [6] also rely on dynamic features to optimize the order of compiler phases. That is, from profiling during a single program execution and a set of hand-crafted rules, they deduce the impact on execution time by a specific sequence. Other approaches rely on static program features. These can be features of code [1, 63] or characteristics of hardware and code [40]. Cooper et al. [26] use virtual execution to find an optimal compiler phase sequence. Ashouri et al. [4] use Bayesian networks to prune their exploration space and rely on static and dynamic program features and control-flow graph features.

There are three iterative approaches that are based on the polyhedron model. Park et al. [54] compose schedules from high-level polyhedral transformations (such as loop fusion, tiling, or pre-vectorization). Each high-level transformation needs an enabling transformation that permits to encode it into a schedule in a semantics-preserving way. They characterize programs using performance counters. Long and O'Boyle [47] perform adaptive compilation based on the Unified Transformation Framework (UTF) [41]. To optimize a program, they reuse known profitable transformations for previously optimized similar programs. They express program similarity with static code-features (e.g., the number of arrays used). The profitability of the transformations applied is measured by run-time feedback. GAPS [50] uses a GA to find profitable schedules. 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 estimates loop and synchronization overhead. GAPS suffers from finding mostly illegal schedules.

Machine Learning Approaches. Again, some approaches rely on dynamic features [5, 22, 53] and others use static ones [33, 48]. Park et al. [51, 52] use both kinds of features. There is just one approach based on the polyhedron model. Ruvinskiy and van Beek [61] build on the iterative approach by Park et al. [54]. They learn a model to predict which of two sequences of primitive high-level transformations is better for a given program. This enables them to choose directly a single optimal schedule instead of having to benchmark several schedules.

Polyhedral features can serve other purposes than machine learning: Bao et al. [9] build on the observation that careful reductions in processor frequency do not reduce execution time significantly but can save energy. They employ polyhedral features that predict speedup from parallelization and operational intensity to statically categorize program regions and then select

a frequency. Kronawitter and Lengauer [44] use polyhedral schedule features in order to prune schedules in an iterative schedule optimization for the domain of stencil codes.

Finally, there are improvements over the PLuTo algorithm [16] and its derivatives [19, 71] that rely on static cost functions or limited iterative search. While loop distribution enables parallelism, it reduces data locality. Loop fusion does the opposite. Bondhugula et al. [17] proposed a loop fusion model that is expressible as an ILP. Pouchet et al. [58] iteratively find a good fusion structure and use the PLuTo algorithm to schedule each loop nest. Zinenko et al. [74] improve over the variant of Verdoolaege and Janssens [71] of the PLuTo algorithm to take explicitly spatial proximity of memory accesses into account. Live-range reordering [7] improves the applicability of tiling.

To the best of our knowledge, besides GAPS, ours is the only combined machine-learning and iterative approach that relies on a characterization of transformations instead of programs.

8 CONCLUSION

We propose to accelerate a genetic algorithm to optimize loop programs for data locality and parallelization by mostly replacing its fitness assessment, which is based on benchmarking, with a predictor that relies on a machine-learned performance model. The model's training data originates from previous iterative optimization. We can reduce the algorithm's time consumption by 49% on average. We lose an average of 15% in speedup of the optimized program. Our novel algorithm hits a sweet spot between faster optimization and a strongly reduced benchmarking effort on the one side, and a slight deterioration of the optimization result on the other. In fact, the best utilization of our classifiers may be as a guard for benchmarking.

In contrast to many other approaches, we rely on a characterization of transformations rather than programs. We were able to demonstrate that knowledge about transformations' profitability that has been learned on results of previous iterative optimization can serve to identify profitable transformations of unknown programs efficiently. We use features to characterize transformation. Analyzing each feature's importance for prediction gives us the capability to reason about benchmarks to which we are unable to transfer our models, and to explicate the premises of our approach. In summary, we conclude that our approach can speed up the iterative optimization by Polyite, but the degree of success depends on the similarity of the programs involved. The results strengthen the assumption that a "one-fits-all" heuristic to loop optimization does not exist.

ACKNOWLEDGMENTS

The first author is grateful to Stefan Kronawitter for his support during the early stages of Polyite's implementation. We thank Gustavo Vale for helpful discussions on the paper. Special thanks go to our reviewers for their impressively detailed and constructive feedback.

REFERENCES

[1] F. V. Agakov et al. 2006. Using Machine Learning to Focus Iterative Optimization. In *Proc. Fourth IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*. IEEE Computer Society, 295–305.
[2] J. R. Allen and K. Kennedy. 1984. Automatic Loop Interchange. In *Proc. SIGPLAN Symp. Compiler Constr.* ACM, 233–246.
[3] T. W. Anderson and J. D. Finn. 1996. *The New Statistical Analysis of Data.* Springer.
[4] A. H. Ashouri et al. 2016. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *ACM Trans. Archit. Code Optim. (TACO)* 13, 2 (2016), 21:1–21:25.
[5] A. H. Ashouri et al. 2016. Predictive Modeling Methodology for Compiler Phase-Ordering. In *Proc. PARMA-DITAM.*
[6] A. H. Ashouri et al. 2017. MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. *ACM Trans. Archit. Code Optim. (TACO)* 14, 3 (2017), 29:1–29:28.
[7] R. Baghdadi et al. 2013. Improved Loop Tiling based on the Removal of Spurious False Dependences. *ACM Trans. Archit. Code Optim. (TACO)* 9, 4 (2013), 52:1–52:26.
[8] S. Baluja and R. Caruana. 1995. Removing the Genetics from the Standard Genetic Algorithm. In *Proc. 12th Int. Conf. on Machine Learning (ICML)*. Morgan Kaufmann, 38–46.

[9] W. Bao et al. 2016. Static and Dynamic Frequency Scaling on Multicore CPUs. *ACM Trans. Archit. Code Optim. (TACO)* 13, 4 (2016), 51:1–51:26.

[10] W. Bao et al. 2018. Analytical Modeling of Cache Behavior for Affine Programs. *Proc. of the ACM on Programming Languages (PACMPL)* 2, *POPL* (2018), 32:1–32:26.

[11] A. I. Barvinok. 1994. A Polynomial Time Algorithm for Counting Integral Points in Polyhedra when the Dimension is Fixed. *Math. Oper. Res.* 19, 4 (1994), 769–779.

[12] C. Bastoul. 2004. Code Generation in the Polyhedral Model is Easier than You Think. In *Proc. 13th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, 7–16.

[13] C. Bastoul. 2004. *Improving Data Locality in Static Control Programs.* Ph.D. Dissertation. Univ. Pierre-et-Marie-Curie.

[14] 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)*, R. Gupta (Ed.). Springer, 283–303.

[15] 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.

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

[17] U. Bondhugula et al. 2010. A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler. In *Proc. 19th Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*. 343–352.

[18] U. Bondhugula et al. 2017. Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations. *IEEE Trans. Parallel Distrib. Syst. (TPDS)* 28, 5 (May 2017), 1285–1298.

[19] 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.

[20] L. Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.

[21] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees.* Wadsworth.

[22] J. Cavazos et al. 2007. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *Proc. Fifth Int. Symp. on Code Generation and Optimization (CGO)*. IEEE Computer Society, 185–197.

[23] L. Chang, D. J. Frank, R. K. Montoye, S. J. Koester, B. L. Ji, P. W. Coteus, R. H. Dennard, and W. Haensch. 2010. Practical Strategies for Power-Efficient Computing Technologies. *Proc. of the IEEE* 98, 2 (2010), 215–236.

[24] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. 2001. Exact Analysis of the Cache Behavior of Nested Loops. In *Proc. ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation (PLDI)*. 286–297.

[25] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache. 2005. Facilitating the Search for Compositions of Program Transformations. In *Proc. 19th Int. Conf. on Supercomputing, (ICS)*. ACM, 151–160.

[26] K. D. Cooper et al. 2005. ACME: Adaptive Compilation Made Efficient. In *Proc. 2005 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 69–77.

[27] D. K. Danner. 2017. *A Performance Prediction Function based on the Exploration of a Schedule Search Space in the Polyhedron Model.* Master's thesis. University of Passau.

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

[29] 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.

[30] 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.

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

[32] P. J. Fleming and J. J. Wallace. 1986. How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. *Commun. ACM* 29, 3 (1986), 218–221.

[33] G. Fursin et al. 2011. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *IJPP* 39, 3 (2011), 296–327.

[34] Stefan Ganser et al. 2018. Supplementary Web site. (2018). https://stganser.bitbucket.io/taco2018/

[35] S. Ganser, A. Grösslinger, N. Siegmund, S. Apel, and C. Lengauer. 2017. Iterative Schedule Optimization for Parallelization in the Polyhedron Model. *ACM Trans. Archit. Code Optim. (TACO)* 14, 3, Article 23 (Aug. 2017), 26 pages.

[36] 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.

[37] 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.

[38] G. Hager and G. Wellein. 2011. *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press.

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

[40] V. I. Kelefouras. 2017. A Methodology Pruning the Search Space of Six Compiler Transformations by Addressing them Together as one Problem and by Exploiting the Hardware Architecture Details. *Computing* 99, 9 (Sept. 2017), 865–888.

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

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

[43] D. Kim et al. 2007. Multi-level Tiling: M for the Price of One. In *Proc. Int. Conf. on High Perf. Comp., Netw., Storage and Analysis (SC)*. ACM, 51:1–51:12.

[44] S. Kronawitter and C. Lengauer. 2018. Polyhedral Search Space Exploration in the ExaStencils Code Generator. *ACM Trans. Archit. Code Optim. (TACO)* (2018).

[45] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. 2nd IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*. IEEE Computer Society, 75–88.

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

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

[48] A. Monsifrot et al. 2002. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Artificial Intelligence: Methodology, Systems, and Applications (AIMSA) (LNCS 2443)*, D. Scott (Ed.). Springer, 41–50.

[49] S. Nembrini, I. R. König, and M. N. Wright. 2018. The Revival of the Gini Importance? *OUP Bioinformatics* (2018), 1–8.

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

[51] E. Park, J. Cavazos, and M. A. Alvarez. 2012. Using Graph-Based Program Characterization for Predictive Modeling. In *Proc. 10th Ann. IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*. ACM, 196–206.

[52] E. Park, C. Kartsaklis, and J. Cavazos. 2014. HERCULES: Strong Patterns towards More Intelligent Predictive Modeling. In *Proc. 43rd Int. Conf. on Parallel Processing (ICPP)*. IEEE Computer Society, 172–181.

[53] E. Park, S. Kulkarni, and J. Cavazos. 2011. An Evaluation of Different Modeling Techniques for Iterative Compilation. In *Proc. 14th Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. ACM, 65–74.

[54] E. Park, L.-N. Pouchet, J. Cavazos, A. Cohen, and P. Sadayappan. 2011. Predictive Modeling in a Polyhedral Optimization Space. In *Proc. 9th Int. Symp. on Code Generation and Optimization (CGO)*. IEEE Computer Society, 119–129.

[55] F. Pedregosa et al. 2011. Scikit-learn: Machine Learning in Python. *J. Machine Learning Research* 12 (2011), 2825–2830.

[56] L.-N. Pouchet et al. 2007. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. In *Proc. 5th IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*. IEEE Computer Society, 144–156.

[57] 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.

[58] L.-N. Pouchet et al. 2010. Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework. In *Proc. Int. Conf. on High Perf. Comp., Netw., Storage and Analysis (SC)*. IEEE Computer Society, 1–11.

[59] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, R. Ramanujam, and P. Sadayappan. 2009. *Hybrid Iterative and Model-Driven Optimization in the Polyhedral Model.* Research Report RR-6962. INRIA. https://hal.inria.fr/inria-00419974

[60] L.-N. Pouchet and T. Yuki. [n. d.]. PolyBench 4.1. ([n. d.]). http://web.cse.ohio-state.edu/~pouchet/software/polybench/.

[61] R. Ruvinskiy and P. van Beek. 2015. An Improved Machine Learning Approach for Selecting a Polyhedral Model Transformation. In *Advances in Artificial Intelligence (Canadian AI) (LNAI 9091)*. Springer, 100–113.

[62] V. Sarkar. 2000. Optimized Unrolling of Nested Loops. In *Proc. 14th Int. Conf. on Supercomputing (ICS)*. ACM, 153–166.

[63] K. Stock, L.-N. Pouchet, and P. Sadayappan. 2012. Using Machine Learning to Improve Automatic Vectorization. *ACM Trans. Archit. Code Optim. (TACO)* 8, 4 (2012), 50:1–50:23.

[64] 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.

[65] N. Vasilache. 2007. *Scalable Program Optimization Techniques in the Polyhedral Model.* Ph.D. Dissertation. U. Paris-Sud.

[66] N. Vasilache, C. Bastoul, and A. Cohen. 2006. Polyhedral Code Generation in the Real World. In *Proc. 15th Int. Conf. on Compiler Construction (CC)*. Springer, 185–201.

[67] 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.

[68] S. Verdoolaege. 2018. *Integer Set Library: Manual.* INRIA. Version isl-0.19.

[69] S. Verdoolaege et al. 2007. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica* 48, 1 (June 2007), 37–66.

[70] S. Verdoolaege et al. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM TACO* 9, 4 (Jan. 2013), 54:1–54:23.

[71] S. Verdoolaege and G. Janssens. 2017. *Scheduling for PPCG.* Technical Report CW706. CS Department, KU Leuven.

[72] M. Weiss. 1991. Strip Mining on SIMD Architectures. In *Proc. 5th Int. Conf. on Supercomputing, (ICS)*. ACM, 234–243.

[73] M. Wolfe. 1986. Loops Skewing: The Wavefront Method Revisited. *Int. J. Par. Prog. (IJPP)* 15, 4 (Aug. 1986), 279–293.

[74] O. Zinenko et al. 2018. Modeling the Conflicting Demands of Parallelism and Temporal/Spatial Locality in Affine Scheduling. In *Proc. 27th Int. Conf. on Compiler Construction (CC)*. ACM, 3–13.