PolyLingual: a Programmable Polyhedral Scheduler

Tom Hammer and Vincent Loechner
University of Strasbourg and Inria
France

Abstract

In this short paper, we present *PolyLingual*, a DSL and compiler to C source code, for easy generation and fast prototyping of new polyhedral schedulers.

1 Introduction

The current landscape of polyhedral scheduling is shaped by various tools [5, 6, 8–12] that propose implementations of schedulers, optimizing one or several factors to achieve different levels of parallelism, data locality, or other optimization criteria. These implementations often propose different options and parameters allowing to modulate the resulting schedule within a narrow range. They include tiling heuristic, shape and size, or other transformations such as loop unrolling or fusion/fission.

As a consequence when optimizing applications using these tools, the quality of the produced schedule related to the user's desired optimizations is often dependent on the choice of the scheduling algorithm and its parameters. Furthermore, depending on the hardware architecture of the execution machine, the existing tools may not be able to produce satisfying schedules without adaptation or modification of their core algorithm to fit those specific needs. However, it is hard to modify and develop new polyhedral schedulers.

The difficulty of developing a polyhedral scheduler is twofold. Firstly, and this cannot be avoided, the basic mathematical knowledge around polyhedra theory is necessary in order to be able to specify schedule constraints, objective functions and ILP formulations, all related to the scheduling process. Secondly, and stemming from the nature of the model, there are a lot of moving parts when describing a classical scheduling process. Namely, conversion to/from polyhedral representation, dependency analysis, and integer linear programming are all necessary elements to polyhedral schedulers. Separate and specific research is engaged in these aspects and often provides tools or libraries that implement them. When developing polyhedral schedulers, these libraries and tools have to be understood and assembled even before the core scheduling algorithm can be. This creates a steep learning curve for people who want to theorize and develop new scheduling algorithms.

The multiplicity of necessary notions related to polyhedral optimization creates a convoluted developing process when implementing new polyhedral schedulers. Although functional knowledge of those sub-fields is necessary in order to comprehend and utilize them in the scope of polyhedral scheduling, the concrete technical aspects (data structures, function/object names) should not hinder their development.

Therefore, in order to facilitate the process, we emitted the idea of developing a tool that allows schedulers to be specified closer to their theorization from an algorithmic standpoint. This paper presents PolyLingual, a DSL and a compiler that generates C source code from a polyhedral scheduler high-level description.

2 PolyLingual

2.1 Overview

PolyLingual is a language and compiler for specifying polyhedral schedulers. It aims to provide an interface abstracting the libraries and data architectures related to scheduling. Through this tool, polyhedral schedulers can be specified using a simple, algorithmic-like syntax while integrating mathematical formulas for schedule constraints and objective functions.

Such a specification allows for the end-user to design and test different polyhedral scheduling algorithms while not requiring any knowledge about library usage and language specific constraints (memory allocation, data structures, ...). Instead of manipulating arrays, structures, or lists, the variables natively available in the PolyLingual DSL represent the program structure (dependencies, statements, dependency graph), and subsets can be derived from them as variables through selection operations without knowing the underlying data structure. For every variable, native attributes are available directly: dependence satisfaction, number of scalar dimensions, etc. On top of that, the user can assign new attributes to set members, for which the data structures are handled by our front-end compiler. One could view this mechanism as adding new object members dynamically.

On one hand, this architectural choice guarantees the simplicity and intelligibility of the programming language, allowing for the implementation of polyhedral schedulers with minimal prior knowledge. On the other hand, it allows for developers with more extensive knowledge about the model to develop new libraries or add-ons implemented in the back-end to propose new functionalities or implement the latest findings for their usage in the PolyLingual language itself.

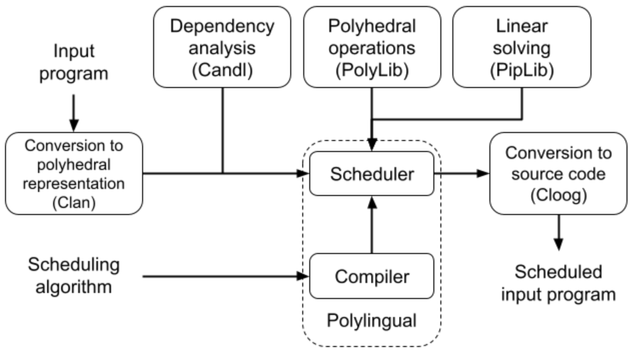
Input program
Dependency analysis (Candl)
Polyhedral operations (PolyLib)
Linear solving (PipLib)
Conversion to polyhedral representation (Clan)
Scheduler
Conversion to source code (Cloog)
Scheduling algorithm
Compiler
Polylingual
Scheduled input program

Figure 1. Visualization of the scheduling process

2.2 Architecture

PolyLingual is built from two parts:

1. a back-end skeleton, handling library interfacing, data structures, and providing some functions for specific tasks (*e.g.*, generation of the orthogonal space from the already found dimensions of a schedule);
2. a front-end compiler generating a C source code, from the specified high-level algorithm. This code is then inserted in the skeleton to generate a compilable scheduler.

The skeleton consists of a dynamic data layout, representing the input program's data, a set of library functions usable from within the programming language, and internal functions (not available from the front-end) used to make the interface with the different used libraries.

As shown in Figure 1 the libraries that we used in the current implementation are all part of the PeriScop toolchain. The input SCoP is parsed and analyzed before the scheduling process through the use of Clan and Candl [2, 3] using the OpenScop representation [4]. At the end of the scheduling process, the data structures representing the schedule are transmitted to Cloog for generation of the output source code. Once initialized, the input program's data is not modified until the scheduling process ends, user defined subsets are implemented as lists of pointers to the elements in that fixed skeleton.

Solving of the ILP formulations is handled with PipLib [1], while coefficient elimination is handled with PolyLib [7] projections. Although specific libraries are used to perform the various tasks relative to scheduling, this does not mean that PolyLingual is incompatible with other implementations. The inner representation and overall architecture allow for potential integration with those implementations by writing simple interfacing functions.

2.3 PolyLingual Syntax

Figure 2 presents an example of code conforming to PolyLingual high-level syntax. It is an implementation of Pluto's

```
SCHEDULE c_i * IT_VEC + c_c * CONS_VEC
S_CONS legality = (T_SCHED - S_SCHED >= 0)
S_CONS volume_bounding = (u * PAR_VEC + w *
    CONS_VEC - (T_SCHED - S_SCHED) >= 0)
O_FUNC positivity = (c_i >= 0, SUM(c_i) >=
    1)
// c_i >= 0 is a shortcut notation
    equivalent to: for all c_i, c_i >= 0

void main(S, D, DDG){
  forall d in D{
    System c1 = apply_to(legality, d)
    System c2 = apply_to(volume_bounding, d)
    System c3 = aggregate(c1, c2)
    d.cons = c3
  }
  repeat{
    ILP problem = (u:global, w:global, c_i,
        c_c)
    forall d in D{
      add_to_ILP(d.cons, problem)
    }
    forall s in S{
      System c = apply_to(positivity, s)
      add_to_ILP(c, problem)
    }
    Solution sol = solve(problem)
    Boolean solution_found = sol.found
    while sol.found{
      store_schedule(sol, S)
      forall s in S{
        System orth = orth_schedule_space(s)
        add_to_ILP(orth)
      }
      sol = solve(problem)
    }
    if !solution_found{
      Graph DAG = gen_DAG(DDG)
      forall n in DAG{
        forall s in n.statements{
          insert_scalar_dim(n.order, s)
        }
      }
    }
    update()
  }until (subset(D, solved == True).length
      == 0) or (orth_schedule_space(max(D,
      dim)) == NULL)
}
```

Figure 2. PolyLingual source code for the Pluto nofuse algorithm.

Type	Description
S_CONS	Specify schedule constraints, these constraints involve the schedule of the pair of statements in a dependence. The Farkas lemma is necessary in order to generate their related solution spaces
O_FUNC	Specify other constraints that do not require the use of the Farkas lemma as they relate to simpler constraints that do not involve relations on the schedule coefficients, such as positivity
O_VAR	Specify variables that are to be used in the ILP formulation to orient the solver in a certain direction. For instance, such a variable can represent the sum of uncarried constraints that is to be minimized in Feautrier's algorithm

Table 1. Available specifications.

nofuse algorithm [5]. A PolyLingual program consists of two parts, described hereunder.

2.3.1 Specification. First, it contains a mathematical specification of the necessary constraints, objective functions and the form of the desired schedule. In addition to the schedule form, denoted by *SCHEDULE*, three main specification types are available, as given in Tab. 1. acyclicl

The keywords *IT_VEC*, *PAR_VEC*, and *CONS_VEC* respectively represent the schedule coefficient vector, the parameter vector, and the constant. They are used to link coefficients to their respective dimensions. Note that those vectors are only used when specifying new coefficients and do not appear later in the program.

The keywords *T_SCHED* and *S_SCHED* are to be used when specifying schedule constraints. *S_SCHED* and *T_SCHED* respectively represent the source and target statement's schedule, as specified with the *SCHEDULE* directive.

2.3.2 Algorithm. Second, a PolyLingual program contains the definition of the scheduling algorithm. Functions can be defined and have fixed parameters that must be one or more among:

- **S**: a set of statements,
- **D**: a set of dependencies,
- **DDG**: a directed dependence graph (redundant with S and D).

The interface provides several datatypes given in Tab. 2.

At any point in the algorithm, the user can declare new attributes to the datatypes by simply assigning them: this is done for example at the beginning of the main function as "`d.cons = c3`" (l.12) in the above example. The underlying structural implementation is handled by the compiler by extending the data structures of the skeleton. Some attributes,

Datatype	Description
Statement	A statement
Dependency	A dependence
Graph	A graph (the type of DDG)
S_Set	A set of statements (the type of S)
D_Set	A set of dependencies (the type of D)
System	A polyhedral relation, internally represented by a PolyLib Matrix or Polyhedron
Solution	A special kind of polyhedral relation, representing the one-line matrix encoding the solution found by the solver to an ILP formulation.
ILP	A special kind of polyhedral relation, used to specify the desired ILP formulation, on which systems can be added to generate the solution space to input in the solver.

Table 2. Datatypes available in PolyLingual.

Scope	Description
global	The variable or coefficient appears once in the ILP formulation (in the Pluto example, this specifies that u and w are common to all statements)
dep	There is one instance of the coefficient for each dependency in the input program (useful for example when specifying the e variable in Feautrier's algorithm)
stmt	There is one instance of the coefficient for each statement, this is the default scope of the schedule coefficients when declaring the schedule form

Table 3. Scope of variables in an ILP problem.

such as dependency satisfaction or the number of elements in a subset are natively implemented and accessible from their respective datatypes.

Sets are generated from the input program's data, stored in S, D or DDG and are initialized through subsetting operations or element extraction. This is shown with the functions *subset* and *max* (l.42) at the end of the repeat loop of the above example. The function *gen_DAG* (l.34) generates a directed acyclic graph corresponding to the input graph, while also calculating the topological sort of all nodes.

2.3.3 ILP formulation. The proposed implementation of ILP formulations is based on the specification of ILP formulations for PPCG [13]. The definition of an ILP formulation involves the order of the coefficients and other variables, as well as a scope for each of them. The scope defines how constraints are assembled, and can be of three types as described in Tab. 3.

```
Matrix *positivity(int dim){
  int i;
  Matrix *out;
  int count = 0;
  out = Matrix_Alloc(1+ (dim*1), (dim*1) +
      2);
  for(i = 0; i < out->NbRows; i++){
    value_assign(out->p[i][0], 1);
  }
  for(i = 0; i < dim; i++){
    value_assign(out->p[count+i][i + (dim*0)
         + 1],+1);
  }
  count += dim;
  for(i = 0; i < dim; i++){
    value_assign(out->p[count][i + (dim*0) +
         1],+1);
  }
  value_assign(out->p[count][out->NbColumns
      - 1],-1);
  count++;
  return out;
}
```

Figure 3. Example C code generated by PolyLingual.

The end-user will also be able to generate scopes from subsets, allowing, for instance, constraint grouping.

2.4 Sample compiler output

The following constraint compiles into the C function given in Fig. 3:

```
O_FUNC positivity = \
    (c_i >= 0, SUM(c_i >= 0))
```

3 Conclusion

We conceived PolyLingual, a DSL for specifying polyhedral schedulers, in the hope that someone, at least our direct colleagues, will find it useful.

Our goal is to soften the steep learning curve to develop new polyhedral scheduling algorithms. The user of PolyLingual has two possible entry points: (1) a novice user can write a PolyLingual DSL code that will automatically be converted into a scheduler as a fully functional back-end C code; (2) a more skilled user may modify the skeleton, adding new library functions, and render those accessible through the DSL.

The DSL and the PolyLingual compiler are currently still under development. Any suggestion regarding their features and future extensions is very welcome.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments.

The authors are grateful to Émilien Bauer who developed early ideas on the structure of PolyLingual during his Master's internship from 2019 to 2021.

References

[1] 2009. PIP - The Parametric Integer Programming's Home. http://www.piplib.org/.

[2] 2014. Candl - Data Dependence Analysis Tool in the Polyhedral Model. http://icps.u-strasbg.fr/people/bastoul/public_html/development/candl/.

[3] 2014. CLAN - A Polyhedral Representation Extraction Tool for C-Based High Level Languages. http://icps.u-strasbg.fr/~bastoul/development/clan/index.html.

[4] 2014. OpenScop - A Specification and a Library for Data Exchange in Polyhedral Compilation Tools. http://icps.u-strasbg.fr/~bastoul/development/openscop/index.html.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[6] Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Process. Lett.* 22, 4 (2012). https://doi.org/10.1142/S0129626412500107

[7] Vincent Loechner. 1999. PolyLib: A Library for Manipulating Parameterized Polyhedra. https://icps.u-strasbg.fr/polylib/.

[8] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. 2011. *R-Stream Compiler.* Springer US, Boston, MA, 1756–1765. https://doi.org/10.1007/978-0-387-09766-4_515

[9] Marek Palkowski, Tomasz Klimek, and Wlodzimierz Bielecki. 2015. TRACO: An automatic loop nest parallelizer for numerical applications. In *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 681–686. https://doi.org/10.15439/2015F34

[10] Louis-Noël Pouchet. 2012. PoCC - The Polyhedral Compiler Collection. http://web.cs.ucla.edu/~pouchet/software/pocc/.

[11] Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. Citeseer, 1.

[12] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. 9, 4, Article 54 (jan 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[13] Sven Verdoolaege and Gerda Janssens. 2017. *Scheduling for PPCG.* Technical Report. Department of Computer Science, KU Leuven. https://doi.org/10.13140/RG.2.2.28998.68169