# Graphite: Towards a Declarative Polyhedral Representation

Sebastian Pop

Centre de recherche en informatique - Ecole des mines de Paris

sebastian.pop@cri.ensmp.fr

## Abstract

Classical polyhedral representations of imperative languages entangle untranslated scalar imperative operations to the declarative descriptions of the polyhedral model. This representation can handle high level array operations, but is more difficult to work on programs that split the array computations in smaller chunks involving scalar temporary variables.

The aim of the current paper is to provide an overview of the algorithms implemented in Graphite, and to convey the intuitive ideas behind the changes needed to the classical polyhedral representations to accommodate to an SSA three address representation. These changes tend to shift the polyhedral representation to a purely declarative language.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—compilers

***General Terms*** Languages

***Keywords*** polyhedral model, static single assignment, SSA

## 1. Introduction

In [3] we proposed to implement Graphite, a classical framework for polyhedral representations in GCC based on GIMPLE. In this effort, we adapted the existing algorithms to the low level representations of GCC. Indeed, it is the first time that a polyhedral representation is built on a three address SSA form. This work also raises several questions that remain unanswered: what is the relation between the polyhedral model and the SSA, why is it so hard to deal with scalar operations and scalar dependences in the polyhedral model?

We present an adaptation of the classical polyhedral model to GIMPLE in Section 2, then we present a set of changes to integrate the SSA representation in the polyhedral model in Section 3.



## 2. Classical Polyhedral Model on GIMPLE

In the classical polyhedral model [4, 2], imperative language constructs are translated to a stand alone representation based on matrices. After this translation, the original representation of the program can be discarded. The matrix representation can then be transformed without having to impact the transformations on the original code, until the last phase of the polyhedral framework, that generates imperative language constructs that can then be passed to the reminder of the compiler.

In this section we summarize the different aspects collected in the polyhedral representation: the SCoP, a basic contiguous set of operations representable with polyhedra, the iteration domains, the memory accesses, data dependence relations, and statement scheduling functions.

### 2.1 SCoP: Static Control Part

The largest part of code that can be represented in the polyhedral model is a region of code that does not contain irregular memory accesses, irregular control flow, statements with side effects, non pure function calls, etc. These regions of code are called Static Control Parts, or SCoPs.

In the previous publications defining the notion of Static Control Parts, the SCoPs are detected on a FORTRAN like syntax tree [1]. As this definition cannot be used on lower level representations that do not contain loops as syntactic constructs, we propose another

definition of SCoPs based on the properties of the Control Flow Graph (CFG). The notion of loops appears in the CFG as strongly connected components, i.e. regions of the CFG where the control might return following a back-edge of the CFG.

The construction of SCoPs rely on the following properties: a SCoP stops before a statement that cannot be represented in the polyhedral model, and the SCoP starts on a basic block that dominates the block that ends the SCoP.

The SCoP construction algorithm walks the dominator tree until a basic block containing a difficult construct is found, in which case that basic block delimits the end of a SCoP. The starting basic block of this new SCoP is then determined as either the basic block that ends the previous SCoP, or the basic block that starts the body of the loop in which the SCoP end block belongs. This algorithm is illustrated on an example in Figure 1.

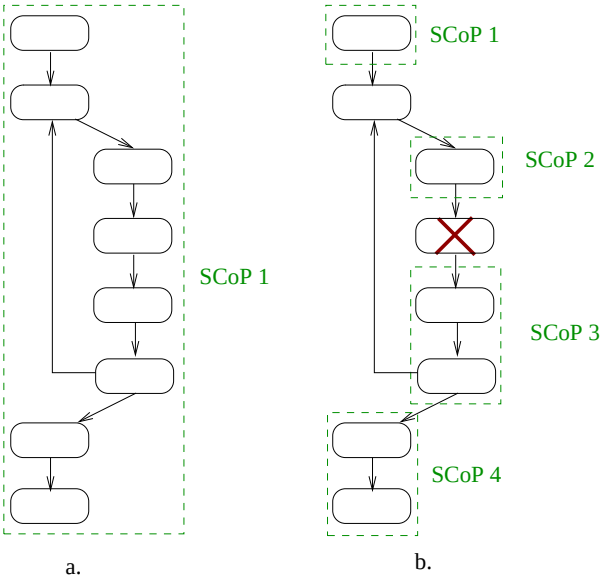


**Figure 1.** Example of SCoP construction. In Figure a, the whole program can be represented in the polyhedral model, and a single SCoP is created. In Figure b, the basic block marked with an X marks the end of a SCoP, and the CFG is partitioned in several SCoPs.

## 2.2 Parameters of a SCoP

A parameter of a SCoP is a variable whose value is not known at compile time, and whose value does not vary during the execution of the SCoP. Thus, are considered parameters all the non volatile variables defined outside the bounds of the SCoP, or defined in a non strongly connected component of the SCoP. The set of parameters and the relations between the parameters is called the context of the SCoP. The context is set up as a linear constraint system before the construction of the rest of the polyhedral representation of the SCoP. The context is used to simplify the linear constraint systems by reducing their dimension.

A correspondence table between the name of the variable and the number of the column representing the variable is kept on the side of the representation. This correspondence table is used in particular during the construction of the polyhedral representation, and during the code generation back to the imperative representation, as parameters can appear in the linear constraints of loop bounds, memory access functions, etc.

## 2.3 Iteration Domains

The low level representation of the CFG and of the natural loops is translated to a higher level representation containing synthesized informations such as the number of iterations, or approximations of the number of iterations. These informations are stored under the form of a linear constraint system: the iterations of each loop are represented by a variable whose values index the iteration of the loop, i.e. start at zero and bounded by the number of iterations in the loop. The number of iterations of an inner loop can be represented by a linear function of scalar variables whose evolutions are either linear functions of the outer loops indexes, or parameter variables.

## 2.4 Memory Access Functions and Dependence Relations

Regular accesses to memory either by arrays or by pointers are translated to a matrix form that represents the linear memory accesses as functions of the loop indexes and SCoP parameters. However, an important part of the source imperative language is missing: the order in which the memory accesses are executed is not captured by this representation.

The partial order of the memory accesses is added on the side under the form of dependence relations between the memory accesses [4, 5, 6]. For each pair of memory accesses, a constraint system represents the elements accessed twice by both accesses for different iterations or in the sequence. This is illustrated in Figure 2.

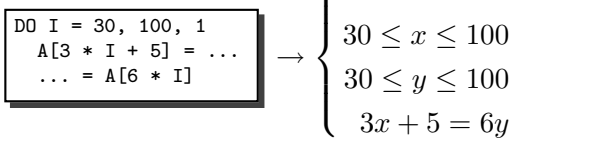


**Figure 2.** Example of a dependence relation equation for an imperative program.

One of the reasons to not include the scalar dependences in the data dependence relations is that the exact same information is contained in the SSA representation. However, because of this separation, classical algorithms, such as the loop distribution, that work on a single representation of data dependences have to be adapted, or alternatively, the two representations have to be mixed again as in the reduced dependence graph used in the loop distribution.

### 2.5 Statement Scheduling Functions

Scheduling functions describe the moment at which a statement has to be executed. There are two components to a schedule, a static time corresponding to the place in the sequence of statements where the statement has to be executed, and the dynamic time corresponding to one of the loops iteration time.

Several schedules can be valid with respect to the dependence relations. As we have mentioned, the dependence relations provide a partial order on the execution time of the memory operations. So, among all the schedules some are feasible if they respect the initial order of computations, others schedules are not valid.

The schedule representation tries to decouple the sequence and iteration space from the sequence and loop constructs. The resulting representation is closer to a declarative language, than to the source imperative language. However we outline several inconsistencies that have to be addressed in order to obtain a purely declarative language that would integrate the informations available in the SSA.

## 3. Towards a Declarative Polyhedral Representation

It is possible to remark that most of the definitions of the classical polyhedral model can be adapted to a three address code form after the higher level informations were synthesized from the low level information. The main exception is the SSA representation, that is difficult to integrate in the polyhedral representation, due to the fact that the informations of the SSA are in part contained in the data dependence information and another part in the byte-code that is handled as is by the code generator.

We propose three different approaches to deal with the SSA form in the polyhedral representation: the first approach consists in removing as much SSA constructs as possible by removing scalar dependences due to temporary variables used in the three address code, the second approach proposes an update of the code generation pass (out of polyhedral model) to handle statements containing SSA constructs, and finally the last proposition intends to move the polyhedral representation towards a purely declarative language, avoiding the need of a representation of statements on the side.

### 3.1 Reconstruction of computations on arrays

The first temporary solution is to aggregate computations in larger chunks, and avoid as much as possible the operations on temporary scalar variables. These operations are not handled in a flexible way by the polyhedral representation. This is a temporary fix that is intended to transform the GIMPLE-SSA representation to a more FORTRAN-like language on which the polyhedral model was proposed.

A further improvement in the same direction would be to propagate high level constructs from the FORTRAN front-end down to the GIMPLE level without having to pass through the scalar GIMPLE representation.

### 3.2 SSA aware code generation

Another temporary solution is to improve the code generation pass out of the polyhedral model, to generate correct SSA constructs. This also intends to make the underlying polyhedral representation aware of the SSA constructs contained in the statements.

### 3.3 Translation of array operations in a dependence declarative language

Finally a more elegant solution would be to avoid the notion of statement in the polyhedral representation, and to introduce it only in the code generation part. In this case, the notion of statement schedule is also introduced only by the code generation pass that becomes a translator from a declarative language back to an imperative language.

## 4. Conclusion

We looked at the construction of the polyhedral model from a different perspective than what is generally described in the literature: instead of starting from a high level FORTRAN-like representation, the information is synthesized from the basic bricks of the three address GIMPLE-SSA representation. We have seen several adaptations to GIMPLE-SSA of the classical algorithms used for building the polyhedral representation, and we proposed a slight change in the polyhedral representation for also capturing the informations handled by the SSA.

## References


[1] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In *Euro-Par'04*, number 3149 in LNCS, pages 292–303, Pisa, Italy, Aug. 2004. Springer-Verlag.

[2] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, Dec. 1992. See also Part I, one dimensional time, 21(5):315–348.

[3] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache. GRAPHITE: Polyhedral Analyses and Optimizations for GCC. In *Proceedings of the 2006 GCC Developers Summit*, 2006. `http://www.gccsummit.org/2006`.

[4] W. Pugh. Uniform techniques for loop optimization. In *ICS '91: Proceedings of the 5th international conference on Supercomputing*, pages 341–352, New York, NY, USA, 1991. ACM Press.

[5] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, Aug. 1992.

[6] D. Wonnacott and W. Pugh. Nonlinear array dependence analysis. In *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995. Troy, New York.