Design of Graphite and the Polyhedral Compilation Package

Jan Sjödin Sebastian Pop Harsha Jagasia
Open Source Compiler Engineering, Advanced Micro Devices, Inc., Austin, Texas, USA
jan.sjodin@amd.com, sebastian.pop@amd.com, harsha.jagasia@amd.com

Tobias Grosser
University of Passau, Passau, Germany
grosser@fim.uni-passau.de

Antoniu Pop
MINES ParisTech, Centre de Recherche en Informatique, Mathématiques et Systèmes, Paris, France
antoniu.pop@mines-paristech.fr

Abstract

Graphite is the loop transformation framework that was introduced in GCC 4.4. This paper gives a detailed description of the design and future directions of this infrastructure. Graphite uses the polyhedral model as the internal representation (GPOLY). The plan is to create a polyhedral compilation package (PCP) that will provide loop optimization and analysis capabilities to GCC. This package will be separated from GIMPLE via an interface language that is restricted to express only what GPOLY can represent. The interface language is a set of data structures that encodes the control flow and memory accesses of a code region. A syntax for the language is also defined to facilitate debugging and testing.

1 Introduction

The polyhedral compilation package (PCP) is an optimization package that uses the polyhedral model as the internal representation to perform program analysis and transformations. Our goal is to define an optimization framework with clear interfaces to simplify testing and integration with GCC.

The polyhedral model can represent structured code containing sequences, linear conditions, well behaved loops, and affine memory accesses. The compilation unit is a static control part (SCoP), which does not have any side effects and all data accesses are statically determined to be linear. Array subscripts are limited to affine expressions of induction variables and constants: this restricts the data dependences to be regular, such that the data flow can be represented by unions of convex polyhedra. Scalar identifiers defined outside a SCoP are called *parameters*. Parameters cannot be modified in a SCoP. Parameters and arrays that are read inside a SCoP are inputs. The output of a SCoP are the arrays that have been modified and that are used after the SCoP.

In this paper, we will discuss the components of PCP and how they interact. Figure 1 shows an overview of PCP. GIMPLE is translated to the PCP language, which is in turn translated to the polyhedral representation GPOLY. The PCP optimizers, guided by a set of heuristics, exclusively work on the polyhedral representation to transform the code. These heuristics are based on information about the architecture, which must be provided by GCC in the form of a machine description.

To integrate PCP with GCC, there are four interfaces to consider:

- Language interface - defines a small imperative language used to represent a compilation unit.
- Polyhedral library interface - must be implemented to provide basic operations on polyhedra. Several polyhedral libraries exist and it is desirable to be able to use different libraries.
- Machine description interface - specifies the system the code should be optimized for.
- Transformation interface - allows GCC to specify specific transformations.

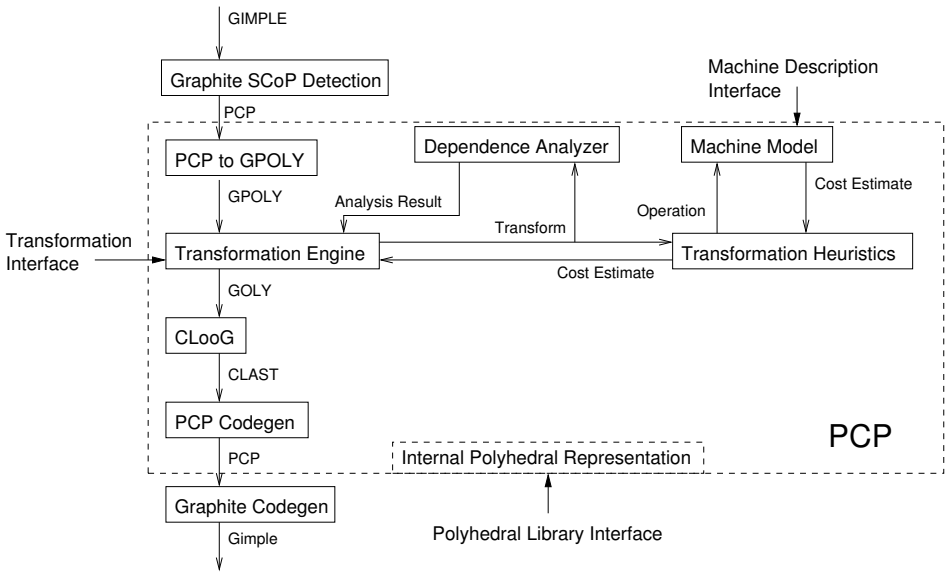

Figure 1: Architecture of PCP and Graphite.

Finally, there are two more aspects of the framework: PCP constructs can encode auxiliary information, and PCP has an infrastructure for testing.

2 Language interface

The PCP language[1] hides the internal representation (GPOLY) such that it can evolve without breaking backward compatibility with the translation to and from GIMPLE. The PCP interface language is restricted to express only what the polyhedral model can represent. PCP is a simple imperative language that only expresses communication between statements through array accesses and does not specify computations. Therefore the number of types and control flow constructs are fewer than in a general purpose programming language. Computations are encoded as "black boxes," or user statements, that are parameterized with the reads and writes to arrays. The control flow constructs are structured loops and conditionals. PCP separates the identification of the structured regions of a GIMPLE program from the translation to GPOLY and clearly defines the information passed between GCC and GPOLY. The language constructs are relatively close to GIMPLE, which means the translation becomes fairly straight forward once a region of code has been identified. In addition to the data structures to represent the language, we have also defined a syntax to parse and emit PCP code.

[1] We refer to both the package and the language as PCP.

In this section we describe the PCP language. The examples use the syntax that has been defined in the language specification[2]. The reason for having a textual language interface is to simplify testing and debugging. If there is no simple way to read and understand a piece of code, the debugging becomes a lot harder.

The syntax for the external language should be easy to read and write by humans and should not contain ambiguities. The expressiveness of the external language must not only be able to express all legal constructs, but also allow illegal constructs for negative tests.

Annotations and tests can be encoded in the language through optional arguments. Optional arguments encode extra information that is not needed to express the meaning of the program, but that is needed for other reasons. By specifying a standard syntax to allow parsing optional arguments, the parser can provide an AST for them. To eliminate ambiguities, such as operator precedence, and allow for a simple syntax for annotations, we use a functional (prefix) form for all language components.

2.1 Types

There is only one scalar type: arbitrary precision integer[3] is used for array indexing, loop bounds, and linear conditions. The type is implicit and there is no syntax

[2] `http://gcc.gnu.org/wiki/PCP`
[3] `http://gmplib.org/`

for it. The only types that must be specified are array types. Arrays types are defined by a list of constants or parameters that define the size of each dimension. If the list is empty, the type stands for a scalar type, for example:

```
// Types:
myType <- array(10, 10)
myScalarType <- array()
```

2.2 Expressions

An expression is a linear combination of constants, parameters, and loop induction variables. Parameters are declared as inputs to a SCoP and never written inside the SCoP.

```
// Parameter example:
myParameter <- parameter()

//Expression example:
+(*(4, N), *(2,i), *(4,4))
```

2.3 Array accesses

`def` and `use` define memory writes and reads. Each `def`/`use` takes a base array and a list of linear expression subscripts. A `maydef` encodes a possible write of a memory location, which may be used if there is control flow inside a user statement:

```
// Array access example:
use(A, i, j)
def(B, +(i,j), k)
maydef(C, i)
```

2.4 Statements

Statements are the constructs that modify the machine state, either control flow or memory. User statements define computations that read and write arrays, but have no other side effects. The user statement consists of a unique name. The arguments to a statement completely define the memory operations done by the statement. The order of the arguments is maintained throughout the compilation. The access functions of `uses` and `defs` may be rewritten during the PCP transformations:

```
// User statement example:
mystmt(def(B, i, j), use(A, -(i,1), -(j,1)))
```

The `copy` statement copies data from source to destination. The `copy` statement is a separate construct from the user statement, which allows PCP to introduce these non-computational memory operations. This construct may be used for "fan-out" communication patterns, as in array privatization.

```
// Copy statement example:
copy(def(B, i, j), use(A, j, i))
```

The guard statement executes the body if the condition evaluates to true. There are two kinds of comparison operators: `eq` (equality) and `ge` (greater than or equal)

```
// Guard example:
guard(eq(i, N))
{
  // Body
}
```

The `loop` statement takes four arguments. First a variable declaration for the induction variable. Second, an expression that defines the initial value of the induction variables. Third, a boolean expression that determines when the loop exits. Fourth, the stride (increment) of the induction variable after each iteration. The `loop` implicitly defines the induction variable. The induction variable can only be accessed inside the `loop` body.

```
// Loop example:
loop(i <- iv(), 1, ge(N, i), 1)
{
  // Body
}
```

2.5 SCoPs

A SCoP is the compilation unit. It has a set of inputs and outputs. The inputs are scalar values (parameters), which are invariant in the SCoP, and arrays, which can be modified. Outputs are arrays that have been modified and will be used after the SCoP.

```
// SCoP example:
scop(inputs(B, C), outputs(A),  parameters(N))
{
  // Scop body
}
```

2.6 A complete example

Below is a small fragment of C code. Assume that the arrays A, B, and C have type double[1000][1000], and that N is a parameter.

```
for (int i = 0; i < N; i++)
{
  A[i][0] = 0;
  for (int j = 1; j < 100; j++)
    A[i][j] = A[i][j-1] + B[j][i] + C[j-1][i-1];
}
```

The C code corresponds to the following PCP code:

```
N <- parameter()
arrayType <- array(1000, 1000)
A <- variable(arrayType)
B <- variable(arrayType)
C <- variable(arrayType)
scop(inputs(B, C), outputs(A),  parameters(N))
{
  loop(i <- iv(), 0, ge(N,i), 1)
  {
    stmt1(def(A, i, 0))
    loop(j <- iv(), 1, ge(100, j), 1)
    {
      // userStmt maps to the add and assignment
      stmt2(def(A, i, j),
            use (A, i, -(j, 1)),
            use(B, j, i),
            use(C, -(j, 1), -(i, 1)))
    }
  }
}
```

2.7 Annotations

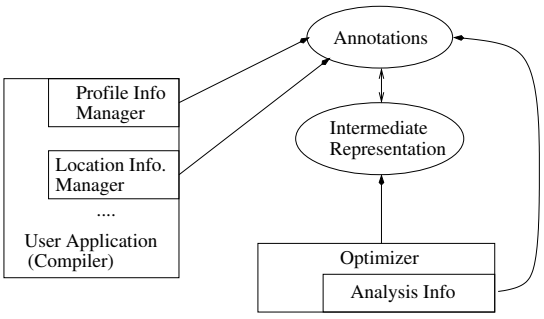

Figure 2: Annotation framework.

Annotations are used to represent auxiliary information that is needed for the compilation process. These can be added to any object in the language. Annotations should be handled by a generic framework, which will allow a compiler to track the information as the code is transformed. During the code generation, the annotations are added to the generated AST. Figure 2 shows the communication among the different components.

Annotations consist of a tag and a list of annotation arguments. An annotation argument can be a scalar value, an identifier, a string, or an annotation.

```
// Annotation example:
A <- variable(array() | myannotation())
```

2.8 Test framework

The test infrastructure, as illustrated in Figure 3, takes text files containing PCP code as input. The input is parsed and dispatched to different components to perform the tests. Tests are specified either using annotations in the code or as a flags to the tester. For example, assume that the tester contains a test that checks if two statements in a loop can be distributed. This kind of test would test the dependence analysis. Assume that the associated annotation with the test is called `distributable`. This is an example of how a test case could be specified:

```
for(i <- iv(), 1, ge(N, i), 1, 1
    | distributable(stmt1, stmt2))
{
  stmt1(def(A, i), use(B,i))
  stmt2(def(C, i), use(D,i))
}
```

Another example would be a check for loop fusion:

```
loop1 <- for(i <- iv(), 1, ge(N, i), 1, 1)
{
  stmt1(def(A, i), use(B,i))
}

loop2 <- for(j <- iv(), 1, ge(N, i), 1, 1
             | fusable(loop1))
{
  stmt2(def(C, j), use(D,j))
}
```

If a test fails, the file name and line number where the annotation occurred is reported along with any diagnostic why it failed.

It is undesirable to use C or FORTRAN source code for unit testing since GCC is unlikely to be capable of producing all possible test cases. In addition, the test cases become unreliable because any of the passes before PCP may change and therefore modify the input

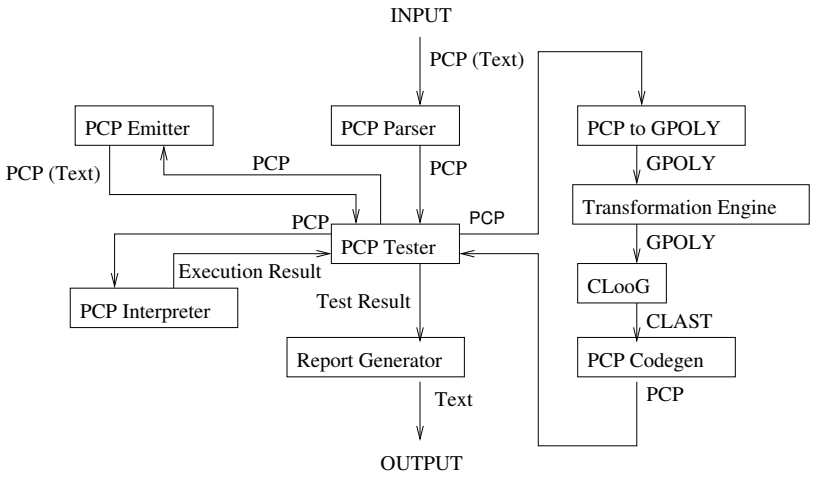

Figure 3: Test infrastructure.

to PCP. Test cases can be automatically extracted from C/Fortran code by using the PCP emitter to dump the SCoPs that are identified by Graphite. The emitter can also be used during debugging to produce reduced test cases that later can be added to the test suite.

The kinds of tests that are needed are both syntactic and semantic. Syntactic tests use simple string compare to check against the expected output. Semantic checks can be done both statically by analysis or dynamically by using an interpreter to execute the code. Since the actual computation is not represented in PCP, the result of the execution is the trace of memory accesses. Execution tests would mostly be used to verify the correctness of a transform by interpreting the code before and after the transform and comparing the trace results.

3 Translation of GIMPLE to PCP and back

Translating a region of GIMPLE to PCP requires the following steps:

1. Identify single-entry/single-exit (SESE) control regions in the CFG that can be represented in PCP and, thus, in the polytope model. This includes analyzing the control flow, loop structures, and induction variables and checking that all expressions for loop bounds, if-conditions, and array indexes are linear. A SCoP is defined in a context and is composed of a set of statements.

2. Detect relations between the parameters.

3. Detect natural loops based on the CFG or on the SESE structured program tree[4].

4. Identify the GIMPLE statements that will map to user statements. The statements that compose the SCoP are also called black boxes. A black box is a SESE region of the SCoP that describes a calculation. As we saw in the previous sections defining the PCP language, the only part exposed to PCP are the data references contained in the black box. As the name suggests, the scalar computations contained in a black box are hidden. A black box can contain a large set of statements, function calls, or irregular control flow, as long as the black box does not have side effects that are escaping the memory definitions and uses. A black box can be defined to encapsulate a part of the program that should not be transformed by PCP. Therefore, for efficiency reasons, one may want to use this mechanism to turn a part of a PCP program into a black box whenever the complexity of the polyhedral code generation is too high. Currently a black box is a basic block.

5. Construct PCP code.

3.1 Translation of PCP to GPOLY

The translation of PCP to the polyhedral model requires computing the iteration domain and the schedule for each statement in a SCoP.

[4] `http://gcc.gnu.org/ml/gcc-patches/2005-09/msg01860.html`

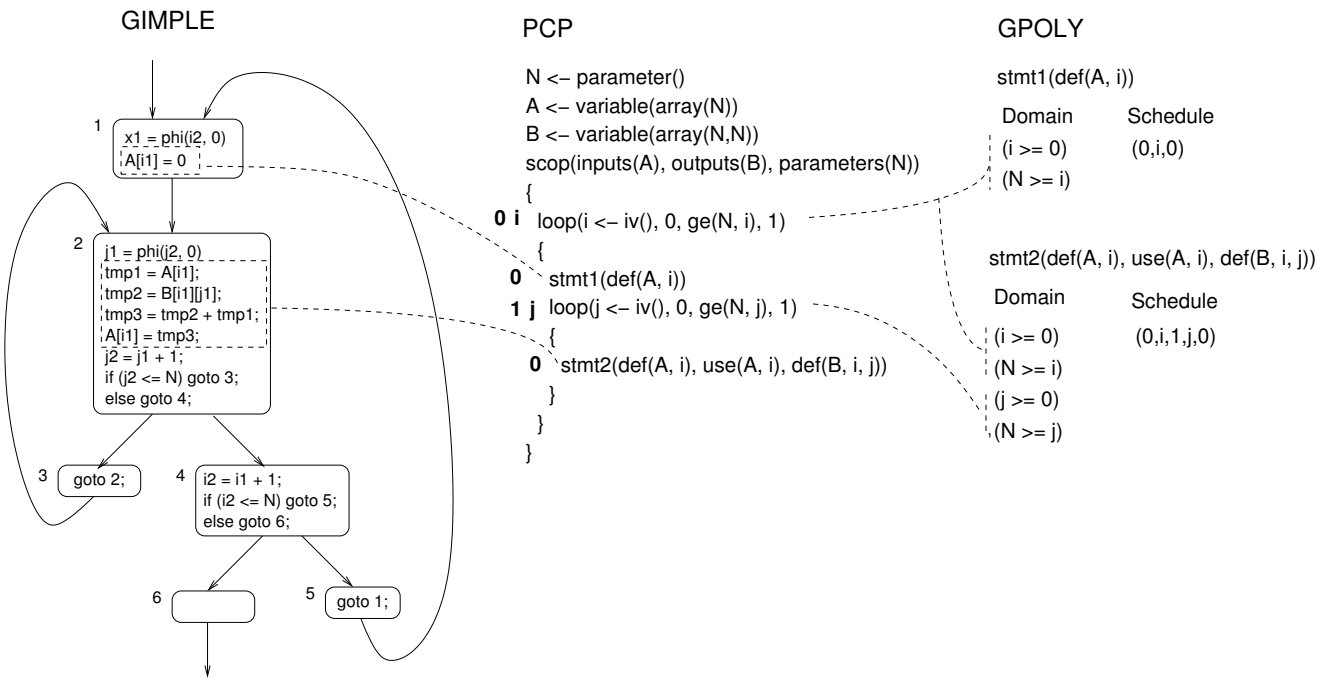

Figure 4: GIMPLE to PCP to GPOLY example.

A canonicalization pass is used to transform all expressions to a uniform format that makes it easy to generate constraints. The polyhedral library interface defines a linear expression as a vector of coefficients in which the position determines the variable or parameter the coefficient is multiplied with. The length of the vectors must be identical for all linear expressions in a constraint. For example:

```
and(ge(i, N), or(eq(j, 5), ge(j, N)))
```

is translated to:

```
or(and(ge(+(*(-1,i), *( 0,j), *(1,N), 0), +(0)),
       eq(+(*( 0,i), *(-1,j), *(0,N), 5), +(0))),
   and(ge(+(*(-1,i), *( 0,j), *(1,N), 0), +(0)),
       ge(+(*( 0,i), *(-1,j), *(1,N), 0), +(0))))
```

The resulting constraint system consists of a union of two polyhedra shown in matrix form:

$$\begin{pmatrix} 1 & -1 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 & 5 \end{pmatrix}$$

$$\begin{pmatrix} 1 & -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 1 & 0 \end{pmatrix}$$

The first column in the matrix encodes if the constraint is an equality ($= 0$) or inequality ($>= 0$).

Uses and defs in the user statements are translated into linear expressions of the polyhedral library. The canonicalization has transformed the subscripts so they can be traversed and the coefficients can be extracted easily, which makes the translation straightforward.

The schedule of a statement is the time at which the statement is executed. There are two components to the execution time of a statement: the static time is the order in which a statement is executed in the sequence that is defined by the PCP abstract syntax tree. To define the static schedule, we use a Dewey numbering of the PCP abstract syntax tree. The dynamic schedule is represented by the iteration domain. Producing the schedule for all the statements is done by a traversal of the PCP abstract syntax tree.

The iteration domain is extracted syntactically from the PCP loop and guard constructs. In a PCP abstract syntax tree, each statement is contained in a set of loops and guards. Each surrounding loop defines a dimension in the iteration domain of the statement. The iteration domain for a statement defines the boundaries for the available induction variables. The guards define extra constraints and relations on the induction variables.

Figure 4 shows an example translating GIMPLE code to GPOLY via PCP. The nested loops in the GIMPLE code maps to two PCP loops. In basic block 1 there is an ini-

tialization of A, which maps to `stmt1` in PCP. Basic block 2 contains a computation consisting of four GIMPLE statements that map to `stmt2`. The translation from PCP to GPOLY builds the iteration domains and schedule for each statement. The domains are defined by the loop bounds. The schedule is created by traversing the PCP code. For `stmt1` the schedule is $(0,i,0)$, which means it is the first statement at the top level and the first statement inside the `i` loop. The schedule of `stmt2` is $(0,i,1,j,0)$, which is the first statement at the top level, the second statement inside the `i` loop, and the first statement inside the `j` loop.

3.2 Translation of GPOLY to PCP

The translation from GPOLY back to an imperative program is done by CLooG, which takes the iteration domains and schedule and produces an AST containing loops and guards.

Translating the CLAST to PCP is simple since both languages have the same constructs. In addition, CLAST gives a mapping for each statement that maps old the induction variables to expressions using new induction variables. All expressions in PCP are rewritten using this mapping.

3.3 Translation of PCP to GIMPLE

Translating PCP to GIMPLE is done by traversing the PCP structure and building the GIMPLE loop and conditions top-down. When a loop is encountered, a new loop structure is created in GIMPLE with a new variable that is the corresponding variable to the PCP induction variable. Each PCP induction variable is mapped to a new GIMPLE variable. When a user statement is found, the array accesses are translated and replace the old accesses in the original GIMPLE code.

4 GPOLY interface

The polyhedral representation of a PCP program is based on the following data structures:

- iteration domains
- scattering polyhedra
- data references
- data dependences

All these data structures can be accessed in read-only mode. The GPOLY transformations interface creates new scattering polyhedra from the original scattering. The original scattering represents the identity transform. The legality check for the transformed scattering is performed based on the original scattering.

4.1 Black box

The black box $B = (domain, drs, scattering)$ is defined by the iteration domain *domain*, a set of data references *drs*, and the *scattering* polyhedra.

4.2 Iteration domains

Each black box has an iteration domain represented with a union of convex polyhedra of dimension d, where d is the loop nesting depth where the black box occurs. The iteration domain describes the set of iterations on which the black box is executed. The iteration domain does not describe the order in which the iterations are executed. The execution order, or dynamic time, is defined by the scattering dimensions of the scattering polyhedra.

4.3 Scattering polyhedra

A transform in the polyhedral model is a function that maps, for each statement, the original dynamic and static time to a new execution order. These transformation functions are also called scattering polyhedra, and are used to define an execution order, which provides the constrains necessary to produce an imperative code back from the polyhedral representation. The scattering polyhedra are expressive enough to represent all the loop and code motion transforms that are allowed in the polyhedral representation. They are composed of the following dimensions[5]:

- *scattering dimensions* represent the loops to be generated,

[5] In this paper we will always use the name of the dimensions, and we will not define a mapping order for the dimensions. The reader can find examples of CLooG scattering polyhedra on `http://gcc.gnu.org/wiki/Graphite/Scattering_polyhedron`. Additional information about scattering polyhedra can be found in the CLooG documentation `http://www.bastoul.net/cloog/manual.php\#SEC8`.

- *original iteration domain* are the dimensions of the original loop nest,
- *parameter dimensions* correspond to the variable names used in the program that are not varying in the current SCoP; parameters can be considered as induction variables of loops around the SCoP, and
- *inhomogeneous term* or *constant dimension*.

The scattering dimensions are a function of the original iteration domain, of the parameters and of the inhomogeneous term.

4.4 Data references

A data reference $DR = (aliasset, subscripts, type)$ is defined by the *alias* set of the data reference. Every alias set is mapped to an unique value. If the array is part of more than one alias set, every array cell is mapped to one point for every alias set the array is part of. Then, each dimension of *subscripts* represents a subscript of the data reference. Scalar values are handled like arrays of dimension 0. The *type* of a data reference can either be read, write, or may-write. Read means a data reference reads or may read any of the values marked in accesses. Write means a data reference must overwrite all the values marked in accesses. May-write means that the values marked in accesses can be, but do not need to be, overwritten.

4.5 Legality and heuristics

The transformation engine in PCP will determine if a given loop transformation is legal based on the information obtained from the dependence analyzer and check if the transformation is profitable based on the information obtained from the transformation heuristics. To check if a transformation is profitable, the transformation engine will model the transformation by modelling the individual operations and comparing them with the machine characteritics provided by GCC in the form of machine descriptions. Based on the cost estimate, the transformation engine will decide the code generation and optimization. In some cases, the transformation engine may not be able to acurately determine the cost of the transformation because the passes after the transformation engine, like CLooG, can make further decisions to manipulate the code and have better knowledge of generated code characteristics like code size. In that case, a second profitability check will be done during PCP code generation.

Each user statement has costs associated with it: an execution time estimate and code size estimate. The execution time estimate is needed to determine the scheduling. The code size estimate is used to avoid code explosion when duplicating code, which could result in poor i-cache locality. The loop optimizations primarily focus on memory reuse, vectorization, and parallelization. Therefore the machine description must contain information about the memory hierarchy, vector instructions, and the configuration of the parallel system (*e.g.*, number of cores and processors) and the latencies for communication.

To generate vector code, PCP annotates loops that are vectorizable (as independent), which can optionally be translated by the compiler in vector code. The compiler can encourage generation of vectorizable loops by giving lower costs to independent inner loops.

4.6 Polyhedral transform interface

Some of the operations in the polyhedral model have been discussed[6] in [1, 2]. These operations are basic transformations of the scattering functions of statements. A similar interface will be provided in GPOLY, but only applies transforms to the scattering polyhedra.

This polyhedral interface is internal to the PCP library and is not exposed outside the polyhedral framework. A classical loop transform interface can be used to annotate transforms on the PCP abstract syntax trees and can be used to direct the transformations performed by PCP.

5 Loop transformation interface

PCP exposes a classical loop transform interface that can be used to drive the transformations that PCP applies. The interface is based on annotations that are set on the PCP trees:

- `loop1 (... | fuse (loop2))`
  Appends the code of `loop1` to the end of `loop2`.

[6] `http://www.lri.fr/~girbal/site_wrapit/`

- `stmt1 (... | move (stmt2))`
  Moves `stmt1` after `stmt2`. This can be used to distribute loops or partially fuse loops.

- `loop1 (... | skew (factor))`
  Multiplies the stride of `loop1` by `factor`.

- `loop1 (... | shift (offset))`
  Adds `offset` to the initial value of the main induction variable of `loop1`.

- `loop1 (... | interchange (loop2))`
  Interchanges `loop1` with `loop2`.

- `loop1 (... | stripMine (factor))`
  Splits the iteration domain of `loop1` into two loops, the outer iterating with strides of `factor`, the inner iterating with the strides of `loop1`. Loop blocking is a composition of `stripMine` and `interchange`.

- `loop1 (... | unroll (factor))`
  Unrolls `loop1` by `factor`.

- `loop1 (... | reverse)`
  Reverses the iteration order of `loop1`.

- `loop1 (... | parallelize)`
  Annotates `loop1` with the parallel flag if `loop1` is parallel.

A single operation can be applied per statement and the composition of the loop transforms is supported only by successive cycles of PCP code generation. This limitation is specific to the classical imperative loop transform interface, and it does not apply to the PCP internal polyhedral transforms.

6 PCP language extensions

The PCP language will evolve over time and there are several important extensions that are needed to make it more complete and capture dependencies and constraints more precisely. In the following sections we describe some extensions that are likely to be included in future versions of PCP.

6.1 Invariants

Invariants is an extension to define further restrictions on scalar values. The compiler may have information about parameters or induction variables (for example, the type of a variable in the original program may restrict the range). The extension is an annotation that can be attached to an object or an expression. For example:

```
// N < 256
N <- parameter(|invariant(ge(255, N)))

// Indexing restriction
stmt(use(A, i | invariant(ge(100, i))))
```

6.2 Reductions

The copy statement can be used to expand (duplicate) data, but there is currently no way to express compression (reduction) other than a regular user statement. The problem with using a user statement for a reduction is that it induces a loop-carried dependence that cannot be parallelized or transformed. For example:

```
loop(i <- iv(), 0, ge(N, i), 1)
{
  userStmt(def(A), use(A), use(B, i))
}
```

The `use(A)` and `def(A)` encode the reduction. The dependencies between two successive iterations of the loop are fixing the evaluation order; it would be illegal to parallelize or to perform some loop transforms.

To solve this problem, we introduce a reduction statement that provides extra information about the associativity and commutativity of a binary reduction operation. The reduction statement takes as a first operand the destination, and the second and third operands are the sources. With this extension, the previous example would be written as:

```
loop(i <- iv(), 0, ge(N, i), 1)
{
  reduction(def(A), use(A), use(B, i))
}
```

This would now allow the loop to be marked as parallel:

```
loop(i <- iv(), 0, ge(N, i), 1 | parallel)
{
  reduction(def(A), use(A), use(B, i))
}
```

6.3 User statements accessing induction variables

Currently a user statement may only access arrays. However, some computations may use the induction variables, which means a user statement must be able to directly access an induction variable. For example:

```
loop(i <- iv(), 0, ge(N, i), 1)
{
   stmt(use(i))
}
```

The assumption with this construct is that the use of an induction variable does not contain any dependencies that PCP must consider.

6.4 While loops

In some cases, the iteration domain may not be known before a loop starts to execute. To handle this case, we must introduce while loops. A while loop takes two arguments, an induction variable and a scalar variable that represents both the predicate and side effect of updating `p1` in every iteration. The proposed syntax would be:

```
// While loop example
while(i <- iv(),p1)
{
  // Body
}
```

6.5 Range specification for data accesses

Since a user statement can represent a larger control flow structure such as loop, it is possible that each invocation can read or write more than a single element of an array. To represent this in PCP, we must be able to specify a range for a subscript that is read or written by a statement. One proposal for specifying a range would be:

```
stmt(use(A, range(0, i), j))
```

6.6 The mayuse annotations

The PCP language only defines the data accesses necessary to define correct semantics for the polyhedral transforms: these are `def`, `use`, and `maydef`. A `mayuse` could be used as a hint for the optimizers to detect locality properties of statements.

7 Conclusion

This paper provides a detailed description of the design and future directions of the Graphite and PCP infrastructures. PCP provides a language and a transform interface to represent and optimize data communications through array operations. The expressiveness of the PCP language is that of the polyhedral model: PCP programs can be translated in the polyhedral model and back to their imperative PCP format. The benefits of the PCP infrastructure are modularity, ease of debugging, and testing of the polyhedral transforms and analyses.

The paper provides technical details of the translation of PCP to the polyhedral representation GPOLY and back. The GPOLY interface provides data structures for a classical polyhedral representation, together with a set of transformations operating on GPOLY. An imperative loop transform interface is defined as annotations on PCP constructs. Finally we discussed extensions of the PCP language to capture a larger set of programs, for providing more precise information to the data dependence analysis, and hints for the cost models.

References

[1] Cedric Bastoul, Albert Cohen, Silvain Girbal, S. Sharma, and Olivier Temam. Putting polyhedral loop transformation to work. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, 2003.

[2] Albert Cohen, Silvain Girbal, and Olivier Temam. A polyhedral approach to ease the composition of program transformations. In *In Euro-Par'04*, 2004.