The Polyhedral Compilation Package (PCP)

Jan Sjödin, Harsha Jagasia, Tobias Grosser, Sebastian Pop

AMD - Austin, Texas

June 8, 2009

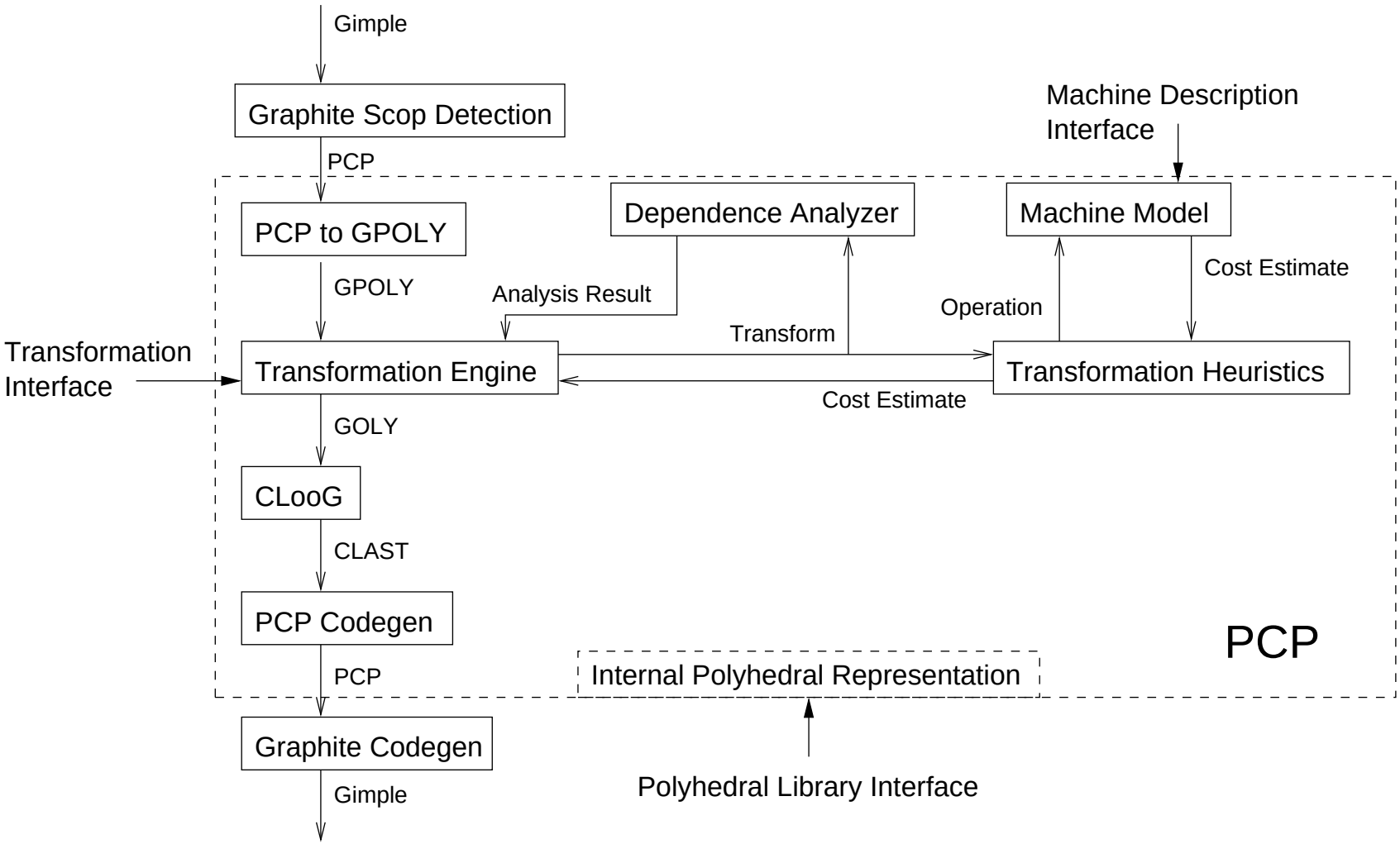
Outline

- Motivation
- PCP framework
- PCP language
- Annotations
- Test infrastructure
- Loop interface
- Machine model and heuristics

Motivation for PCP

- Graphite was created to improve loop optimizations in GCC
- PCP defines a set of interfaces to make the code more modular by separating GIMPLE and the polyhedral representation (GPOLY)
- Well defined interfaces that are easy to use and understand
- Ability to evolve the framework independently of GIMPLE
- Test the framework without the need for a front end
- Source-to-source translation is desirable to allow iterative optimization

Graphite/PCP Overview



Design Criteria for the PCP Language

- Easy to understand
  - Imperative language constructs that easily map from and to GIMPLE.
- Small
  - Design a language that is only as expressive as the polyhedral model.
  - Cannot represent all possible operators/statements that exist: use black boxes and only expose array reads/writes. *Mapping IR to a black box is not trivial in some cases.*
- Easy to integrate and extend
  - Allow auxiliary information to be passed to and from PCP.
- Testable and Debuggable
  - Test framework
  - Define syntax for the PCP language

PCP Language Overview

- Imperative language that expresses data communication via array accesses.
- Compilation unit is a program fragment of structured code that can be expressed with simple loops, conditional statements and array accesses. This is called a SCoP (Static Control Part).
- Computations are expressed as black boxes, only the array accesses are represented.
- Expressions for array accesses and conditions must be linear combinations of induction variables and parameters.

Language Components

- Type: `array(Dimension0, ..., DimensionN)`
- Variable: `variable(ArrayType)`
- Parameter: `parameter()`
- Linear expressions: `i, N, 42, +(*(4, j), 17)`
- Boolean expressions: `ge(N, i),`
  `and(ge(j,5) ge(i, *(2, j))),`
- Array access:
  `use/def(Array, Subscript0, ..., SubscriptN)`
- Statements:
  - `copy(Def, Use)`
  - `stmt(Access0, ..., AccessN)`
  - `guard(Condition) { Body }`
  - `loop(Iv, Start, Condition, Stride) { Body }`
- SCoP: `scop(Inputs, Outputs, Parameters) { Body }`

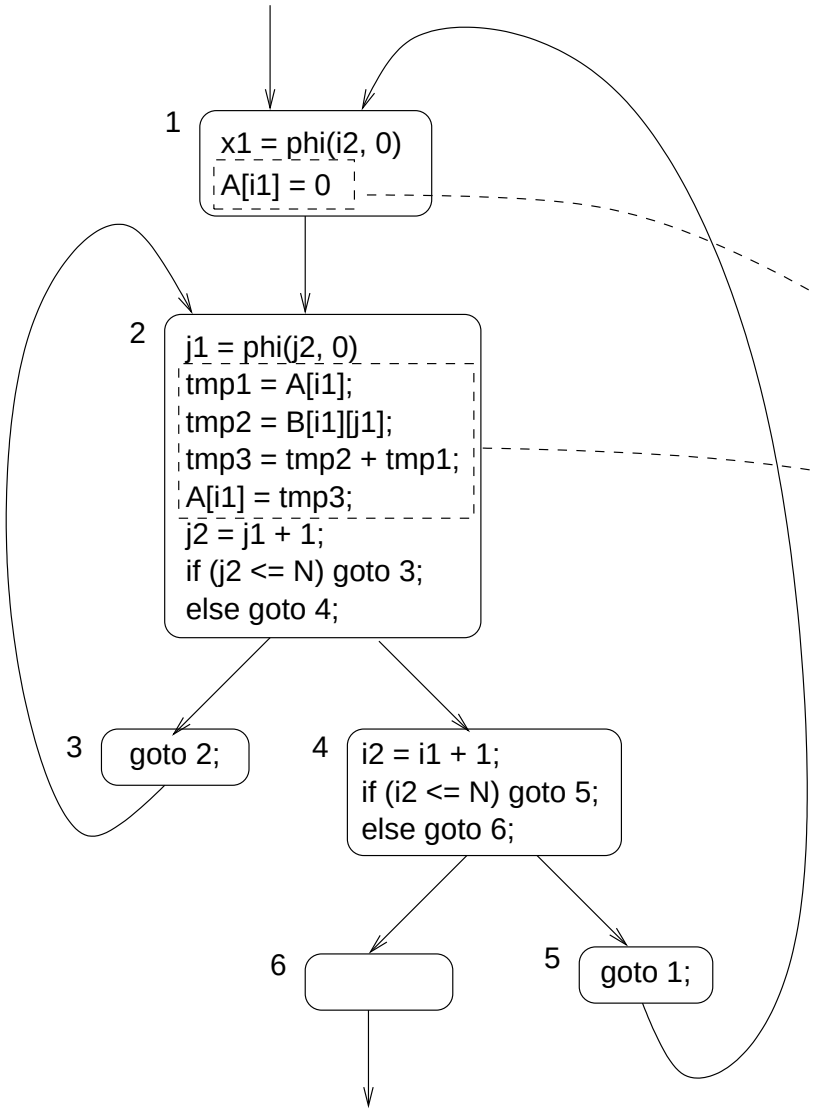
Example

```
C Code (assume a/b/c[1000][1000])
-------------------------------------------------
for (int i = 0; i < n; i++)
  for (int j = 0; j < 100; j++)
      A[i][j] = B[j][i] + C[i+1][i+j];

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

Gimple to PCP

GIMPLE



PCP

```
N <− parameter()
A <− variable(array(N))
B <− variable(array(N,N))
scop(inputs(A), outputs(B), parameters(N))
{
  loop(i <− iv(), 0, ge(N, i), 1)
  {
    stmt1(def(A, i))
    loop(j <− iv(), 0, ge(N, j), 1)
    {
      stmt2(def(A, i), use(A, i), use(B, i, j))
    }
  }
}
```

PCP to GPOLY

PCP

```
N <− parameter()
A <− variable(array(N))
B <− variable(array(N,N))
scop(inputs(A), outputs(B), parameters(N))
{
0 i  loop(i <− iv(), 0, ge(N, i), 1)
     {
 0     stmt1(def(A, i))
 1 j   loop(j <− iv(), 0, ge(N, j), 1)
       {
  0      stmt2(def(A, i), use(A, i), use(B, i, j))
       }
     }
}
```

GPOLY

stmt1(def(A, i))

Domain	Schedule
(i >= 0)	(0,i,0)
(N >= i)	

stmt2(def(A, i), use(A, i), use(B, i, j))

Domain	Schedule
(i >= 0)	(0,i,1,j,0)
(N >= i)	
(j >= 0)	
(N >= j)	

Annotations

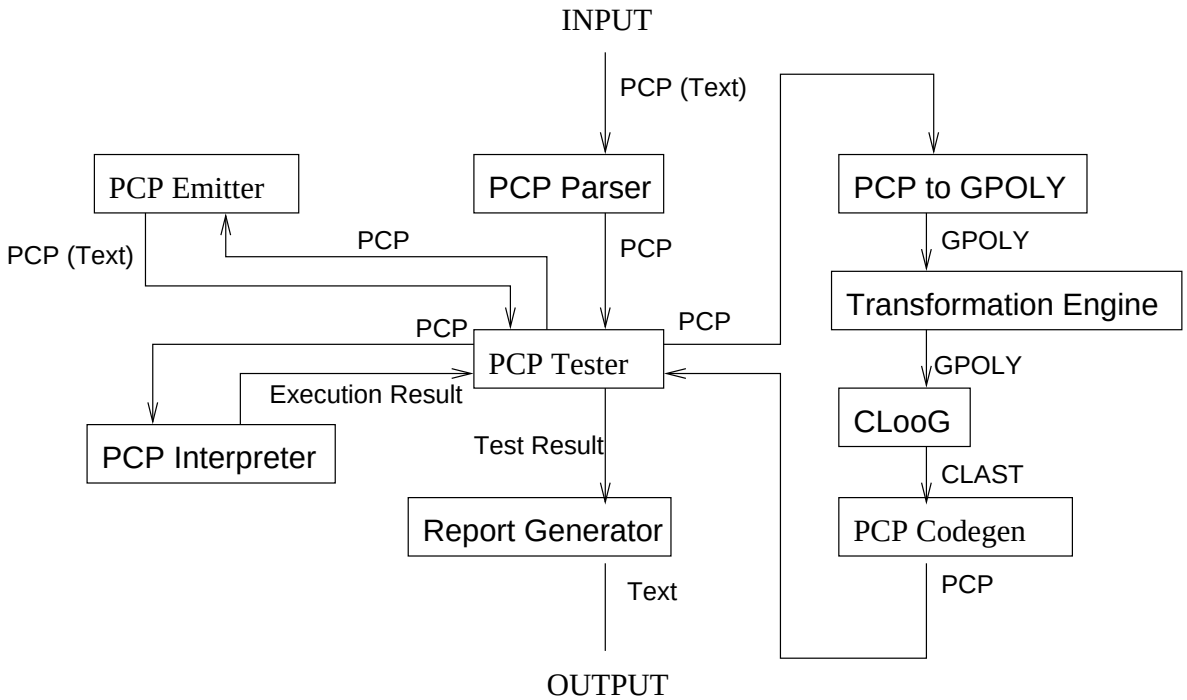
- Annotate any object in the PCP ASTs to express extra information
- Does not encode information that is needed to ensure correctness
- Parser and emitter can handle any annotation since they are purely syntactic

Examples:

```
copy(def(A, i), use(B, i) | lineinfo("test.pcp", 17))
stmt(def(X, i, j), use(Y, i, j) | costs(size(11),
                                        time(120)));

loop(i <- iv(), 0, ge(N, i), 1 | parallel())
```

Test Infrastructure



- Test cases in the form of PCP code with annotations
- Easy to dump out a SCoP to a file from any compiler
- Syntactic tests done by diffing with expected result.
- Semantic tests are either static tests using analyzes, or dynamic tests that will use an interpreter.
- Interpreter output will be a trace of memory accesses.

Test Case Example: Loop fusion

```
loop1 <- loop(i <- iv(), 0, ge(N, i), 1))
{
    stmt1(def(B, i, j), use(A, i, j));
}

loop(j <- iv(), 0, ge(N, j), 1 | fusable(loop1))
{
    stmt2(def(C, i, j), use(B, i, j));
}
```

- If a test fails the file name and line number of the PCP file can be reported.
- For generic testing the annotations can be ignored and a test can instead be specified via a flag to the tester.
- May still need a fancier pretty printer that can dump HTML with links/scripts for fast debugging.

Classical Loop Transformations 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))`
- `stmt1 (... | move (stmt2))`
- `loop1 (... | skew (factor))`
- `loop1 (... | shift (offset))`
- `loop1 (... | interchange (loop2))`
- `loop1 (... | stripMine (factor))`
- `loop1 (... | unroll (factor))`
- `loop1 (... | reverse)`
- `loop1 (... | parallelize)`

Machine description and costs for heuristics

Work still in progress but we know we need the following:

- Description of the machine/system that the code should be optimized for:
  - Memory hierarchy (caches, latencies, bandwidth)
  - Number of processors/cores
  - Vector operation costs
- Annotations from GCC to estimate code size and execution time.

Conclusion

PCP provides

- a language for an imperative language restricted to the constraints of the polyhedral model
- a classical loop transform interface
- a test infrastructure
- modularity and ease of debugging

A polyhedral representation GPOLY is contained in PCP