GRAPHITE: Towards a Declarative Polyhedral Representation

Sebastian Pop
sebastian.pop@{ensmp.fr,inria.fr,amd.com}

CRI Ecole des mines de Paris - Fontainebleau, France
INRIA Futurs - Orsay, France
AMD - Austin, Texas

GREPS workshop
Brașov, România
September 16, 2007
Architecture of GCC and Loop Nest Optimizer

(programming languages) C C++ Java F95 Ada

GENERIC

(basic imperative language) GIMPLE

GIMPLE + CFG + SSA + Loops

Analyses
- data dependences
- number of iterations

(declarative language) GRAPHITE

GIMPLE

Machine description

RTL

ASM

Sebastian Pop

GRAPHITE: Towards a Declarative Polyhedral Representation
• polyhedral representation based on matrices
• static control parts (SCoPs)
• parameters
• iteration domains
• memory access functions
• data dependence relations
• statement scheduling
From a given point (static sequence + iteration), RD is the last point where a variable was written.
Reaching Definitions (Imp)

From a given point (static sequence + iteration), RD is the last point where a variable was written.

- for scalar variables:
  - RD is statically computable
  - SSA statically translates RDs to declarations
From a given point (static sequence + iteration), RD is the last point where a variable was written.

- **for scalar variables:**
  - RD is statically computable
  - SSA statically translates RDs to declarations

- **for arrays:**
  - RD statically uncomputable (undecidable address expressions)
  - RD approximations: array regions $\subset$ data dependence relations
  - exact static RD: insert disambiguation copies = privatization
For two memory operations in static sequence

\[ A; \ldots; B \]

executed at iteration points \( k_A \) and \( k_B \), and
accessing memory locations \( f_A(k_A) \) and \( f_B(k_B) \), the
dependence relation between \( A \) and \( B \) is the set of

iterations \( (k_A, k_B) \) satisfying

\[
\begin{cases} 
  k_A \in \text{iterDomain}(A) \\
  k_B \in \text{iterDomain}(B) \\
  f_A(k_A) = f_B(k_B)
\end{cases}
\]
Data Dependence Relations (Imp)

- DDRs \((k_A, k_B)\) track only iterations in RDs need to keep on the side the sequence \(A; \ldots; B\)
- redundant with the SSA for scalar variables
aggregate GIMPLE operations
reconstruct higher level FORTRAN array ops
keep the PR in SSA form
Purely Declarative Polyhedral Representation?

- no statements in the PR
- no schedule in the PR
- generate schedules and statements in out-of-PR
Questions?
Motivations for GRAPHITE:
- difficult to undo loop transforms
- order of transforms fixed once for all
- difficult to compose
- invalidated data deps: ad-hoc correction or rebuild
- transform applied to loop bodies
Limitations of GRAPHITE

SCoP (Static Control Parts) and their limits
- working on dominator trees
- loops containing a harmful basic block are split
- basic blocks with side effect statements are split

Harmful statements:
- function calls
- side effects (volatile, asm, etc.)
- non supported inductions (exp, wrap, etc.)

Selecting interesting SCoPs
Example: building SCoPs

- SCoPs built on top of the CFG:

basic blocks of the SCoP
  - contain only affine constructs, no side effects
Example: building SCoPs

SCoPs built on top of the CFG:

basic blocks of the SCoP
- contain only affine constructs, no side effects
- dominated by entry, postdominated by exit of the SCoP
SCoPs built on top of the CFG:

splitting basic blocks: split_block

basic blocks = containers
split basic blocks for:

- smaller code chunks
- reduce number of dependences
- moving parts of code around
SCoP Parameters

- **parameters** = variables varying outside a SCoP
  - function parameters
  - variables varying in outer loops

- **context** = constraints on parameters
  - use IPA info for bounding function parameters
  - use VRP’s propagation engine
GRAPHITE built on top of:

- **scalar evolutions**: number of iterations, access functions
- array and pointer analyses
- data dependence analysis
- **scalar range estimations**: undefined signed overflow, undefined access over statically allocated data, etc.
Statements + parametric affine inequalities

a domain = bounds of enclosing loops

```
for (i=0; i<m; i++)
  for (j=5; j<n; j++)
    A[2*i][j+1] = ...
```

```
\[
\begin{bmatrix}
i & j & m & n & cst \\
1 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 & -1 \\
0 & 1 & 0 & 0 & 5 \\
0 & -1 & 0 & 1 & -1 \\
\end{bmatrix}
\]
```

\[
i \geq 0 \\
-1 + m \geq -1 \\
j \geq 5 \\
-j + n \geq -1
\]
Statements + parametric affine inequalities

1. A **domain** = bounds of enclosing loops
2. A list of **access functions**

for (i=0; i<m; i++)
  for (j=5; j<n; j++)
    A[2*i][j+1] = ...

\[
\begin{bmatrix}
i & j & m & n & cst \\
2 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
2 * i \\
j + 1 \\
\end{bmatrix}
\]
Statements + parametric affine inequalities

1. **a domain** = bounds of enclosing loops
2. **a list of access functions**
3. **a schedule** = execution time (static + dynamic)

- sequence \([s_1; s_2]\): 
  
  \[
  S[s_1] = t \\
  S[s_2] = t + 1
  \]

- loop \([\text{loop}_1 s \text{ end}_1]\): \(i_1\) indexes \(\text{loop}_1\) iterations: dynamic time
  
  \[
  S[\text{loop}_1] = t \\
  S[s] = (t, i_1, 0)
  \]
CLooG: Code Generation

1. fill CLooG program structures: context, statement domains, scheduling functions
2. cloog_program_generate
3. cloog_clast_create
4. walk the AST, recompose a GIMPLE program:
   - modify loop structure
   - create_iv
   - reuse parts of lambda-code.c
- select SCoPs
- lib integration PolyLib, CLooG, PiPLib, Omega
- extend lambda-code.c interface with CLooG
- cost models more static analyzers, and transform selection
- array regions improve data depts in interproc mode