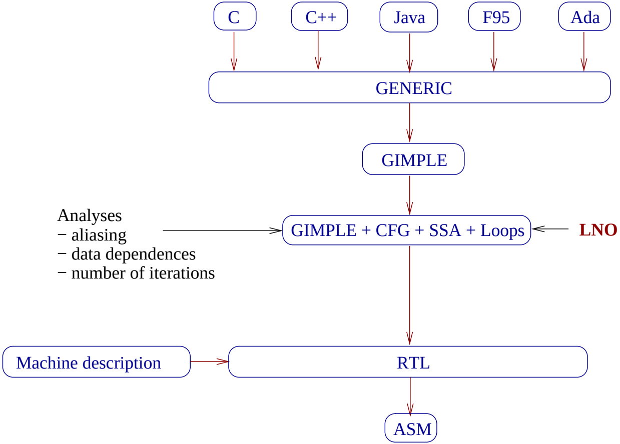
Loop Nest Optimizer of GCC

Sebastian Pop

CRI / Ecole des mines de Paris

Август, 2006

Architecture of GCC and Loop Nest Optimizer



Future Plans for the LNO

1. GRAPHITE: extension of linear transforms
2. parallel code generation (via libgomp)
3. machine models and abstract simulators
4. static profitability analyses
5. hybrid analyses (compress static analysis + dynamic part)

Problems with Classical LNO Transforms

Motivations for GRAPHITE:

- "source to source" modifies the compiled program
- difficult to undo
- order of transforms fixed once for all
- invalidated data deps: ad-hoc correction or rebuild
- difficult to compose

Problems with Classical LNO Transforms

Motivations for GRAPHITE:

- “source to source” modifies the compiled program
- difficult to undo
- order of transforms fixed once for all
- invalidated data deps: ad-hoc correction or rebuild
- difficult to compose

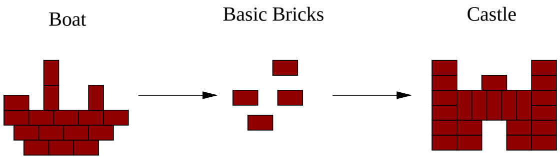
solved in WRaP-IT (from 2002 at INRIA on ORC/Open64)

GRAPHITE = WRaP-IT for GCC

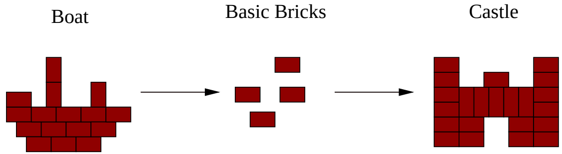
GRAPHITE: Intuitive Idea

GRAPHITE: Intuitive Idea

GRAPHITE: Intuitive Idea



GRAPHITE: Intuitive Idea



C,C++,F95,... → GIMPLE → GRAPHITE

(programming languages) (basic imperative language) (geometrical language)

Statements + parametric affine inequalities

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

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

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

$i \geq 0$
$-i + m \geq -1$
$j \geq 5$
$-j + n \geq -1$

Statements + parametric affine inequalities

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

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

$$\left[\begin{array}{ccccc} i & j & m & n & cst \\ \hline 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{array}\right] \quad \begin{array}{c} 2 * i \\ j + 1 \end{array}$$

GRAPHITE : Representation on Top of Gimple-SSA

Statements + parametric affine inequalities

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

GRAPHITE$(1, 2, 3)$ extends LAMBDA$(1, 2)$

GRAPHITE: Gimple Represented As Polyhedra

(with interchangeable envelopes)

GRAPHITE versus LAMBDA

- common part: unimodular transform data and iteration order
- transform regions: extended from loops to SCoP

  "static control parts": sequences, affine conditions and loops
- GRAPHITE knows about the sequence!

  enables more loop transforms: fusion, fission, tiling, software pipelining, scheduling

Compose Transforms

Small set of primitives (basic operations on matrices)

1. motion
2. interchange
3. strip-mine
4. insert, delete
5. shift
6. skew, reversal, reindexing
7. privatize

Composed transforms

- fission, fusion: 1
- tiling: 2 + 3

Optimal Transform?

Find sequences of transforms based on

- size of loops
- cache misses
- simulation

Automatic selection of transforms

- amounts to choosing a point in a vector space
- hard part (open questions)
- WRaP-IT uses directives

Results From WRaP-IT on Top of PathScale EKOPath

swim from SPEC CPU2000

- 32% speedup on AthlonXP wrt. peak EKOPath (V2.1)
- 38% speedup for Athlon64 wrt. peak EKOPath (V2.1)
- principal SCoP: 421 lines of code
- apply 30 transforms to principal SCoP

  fusion, tiling, peeling, unrolling, interchange, strip-mining
- result 2267 LOC
- 39 sec source to assembly on AthlonXP 2.08GHz
- 22 sec in the backend
- 12 sec polyhedral data deps
- 4 sec polyhedral code gen

Static Estimation of Runtime Properties

How hard is it to simulate a processor?

- DSP: almost deterministic
- superscalar: hard to predict processor transforms
- VLIW: hard to predict compilers future decisions

Need to simulate exact behavior?

Static Estimation of Runtime Properties

How hard is it to simulate a processor?

- DSP: almost deterministic
- superscalar: hard to predict processor transforms
- VLIW: hard to predict compilers future decisions

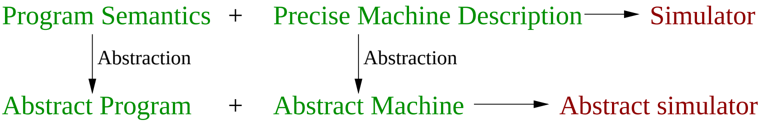
Need to simulate exact behavior? No!

Idea: abstract simulation.

Abstract Simulation

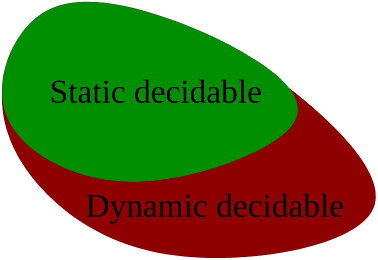
Program Semantics + Precise Machine Description ⟶ Simulator

Abstract Simulation



Hybrid Analyses (Static + Dynamic)

Properties for validating a transform:

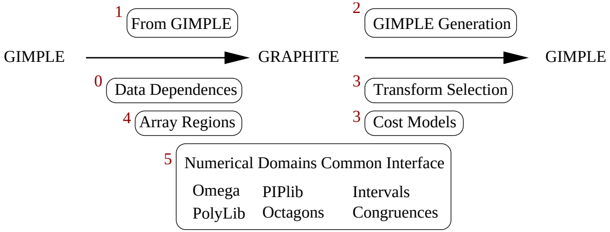


When static analysis fails,

- collect failed static problems
- symbolically compress
- instrument code (instantiate at run time)
- code generation problems (code size + completing static analysis overhead)

GRAPHITE: Road Map

1. select SCoPs filter out difficult codes (Alexandru Plesco)
2. extend LAMBDA build schedule functions, GLooG
3. cost models more static analyzers, and transform selection
4. array regions improve data deps in interproc mode
5. lib integration PolyLib, PiPLib, Omega, lib-APRON



Questions?

limit computation complexity = restrict expressivity

use coarser representations