SCoP Detection: A Fast Algorithm for Industrial Compilers

Sebastian Pop and Aditya Kumar

SARC: Samsung Austin R&D Center

Jan 19, 2016

Polyhedral compilation in industrial compilers

- Goal: enable isl scheduler in GCC at -O3

Polyhedral compilation in industrial compilers

- Goal: enable isl scheduler in GCC at -O3

- search loops that can benefit from polyhedral compilation
- minimal overhead: search as fast as possible
- only use existing analysis information
- use the right abstract representation

What is a SCoP?

Regions of code that can be represented in the Polyhedral Model.

- SCoPs = Static Control Parts

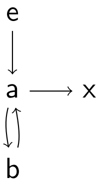
What is a SCoP?

Regions of code that can be represented in the Polyhedral Model.

- SCoPs = Static Control Parts
- ACLs = Affine Control Loops
- PWACs = Parts With Affine Control

Step 1: accept natural loops

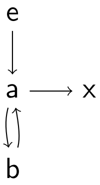
Natural loop



maybe SCoP

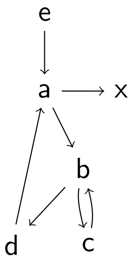
Step 1: accept natural loops

Natural loop



maybe SCoP

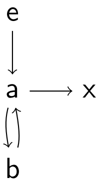
Nested loops



maybe SCoP

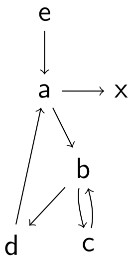
Step 1: accept natural loops

Natural loop



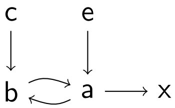
maybe SCoP

Nested loops



maybe SCoP

Irreducible



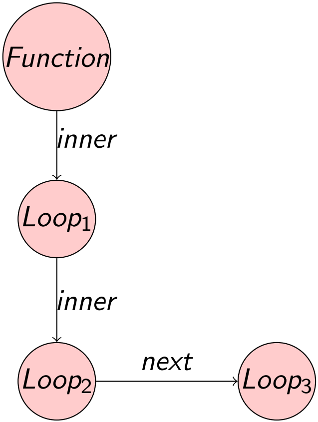
not a SCoP

Natural Loop Tree

```
int foo(int N)
{
 int i, j, k;
 for(i=0; i<N; ++i){//Loop1
  stmt1;
  for (j=0; j<N; ++j)//Loop2
   stmt2;
  for (k=0; k<N; ++k)//Loop3
   stmt3;
 }
}
```

Natural Loop Tree

```
int foo(int N)
{
 int i, j, k;
 for(i=0; i<N; ++i){//Loop1
  stmt1;
  for (j=0; j<N; ++j)//Loop2
   stmt2;
  for (k=0; k<N; ++k)//Loop3
   stmt3;
 }
}
```



Step 2: check for side-effects

- function calls
- inline assembly
- volatile operations

Step 3: affine scalar evolutions

Linear

```
i0 = phi_l1(0, i1)
// i0={0,+,1}_l1
i1 = i0 + 1
// i1={1,+,1}_l1
```

maybe SCoP

Step 3: affine scalar evolutions

Linear

```
i0 = phi_l1(0, i1)
// i0={0,+,1}_l1
i1 = i0 + 1
// i1={1,+,1}_l1
```

maybe SCoP

Non-linear

```
j2 = phi_l1(3, j3)
j3 = j2 + i1
// j2={3,+,{1,+,1}_l1}_l1
```

not an ACL: polynomial of degree 2

Step 3: affine scalar evolutions

Linear

```
i0 = phi_l1(0, i1)
// i0={0,+,1}_l1
i1 = i0 + 1
// i1={1,+,1}_l1
```

maybe SCoP

Non-linear

```
j2 = phi_l1(3, j3)
j3 = j2 + i1
// j2={3,+,{1,+,1}_l1}_l1
```

not an ACL: polynomial of degree 2

Non-linear

```
k4 = phi_l2(4, k5)
k5 = k4 * 2
// k4={4,*,2}_l2
```

not an ACL: exponential

Step 3: affine scalar evolutions

Linear

```
i0 = phi_l1(0, i1)
// i0={0,+,1}_l1
i1 = i0 + 1
// i1={1,+,1}_l1
```

maybe SCoP

Non-linear

```
j2 = phi_l1(3, j3)
j3 = j2 + i1
// j2={3,+,{1,+,1}_l1}_l1
```

not an ACL: polynomial of degree 2

Non-linear

```
k4 = phi_l2(4, k5)
k5 = k4 * 2
// k4={4,*,2}_l2
```

not an ACL: exponential

analyzed expressions

- branch conditions
- memory accesses

Step 4: delinearize memory access functions

Linear access functions

```
A[100*i + 400*j]
B[i][j]
```

can represent in isl

Step 4: delinearize memory access functions

Linear access functions

```
A[100*i + 400*j]
B[i][j]
```

can represent in isl

Non-linear access functions

```
C[i*i]
D[4*N*M*i + 4*M*j + 4*k]
E[4*i*N + 4*j]
```

cannot represent in isl

Step 4: delinearize memory access functions

Linear access functions

```
A[100*i + 400*j]
B[i][j]
```

can represent in isl

Non-linear access functions

```
C[i*i]
D[4*N*M*i + 4*M*j + 4*k]
E[4*i*N + 4*j]
```

cannot represent in isl

delinearization

- recognize array multi-dimensions
- compute linear access functions

Step 4: delinearize memory access functions

Linear access functions

```
A[100*i + 400*j]
B[i][j]
```

can represent in isl

Non-linear access functions

```
C[i*i]
D[4*N*M*i + 4*M*j + 4*k]
E[4*i*N + 4*j]
```

cannot represent in isl

delinearization

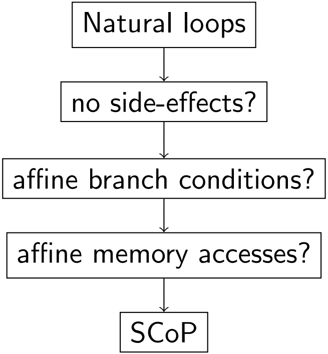
- recognize array multi-dimensions
- compute linear access functions

delinearized access functions

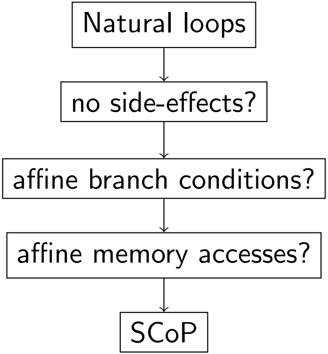
```
int D[][N][M];
D[i][j][k]

int E[][N];
E[i][j]
```

can represent in isl

Overall picture: SCoP detection



Overall picture: SCoP detection



Required analyses:

- natural loops tree
- (post-)dominators tree
- alias analysis
- scalar evolution analysis

Detecting SCoPs by induction on Natural Loops Tree

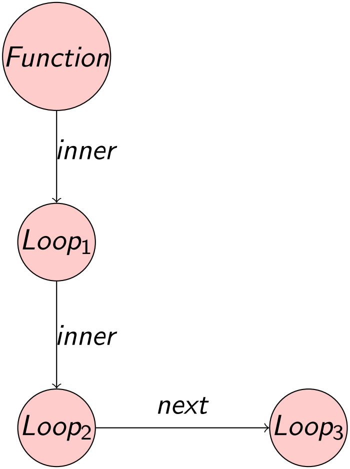
- Start with a loop in the natural loops tree rather than the root of the CFG

Detecting SCoPs by induction on Natural Loops Tree

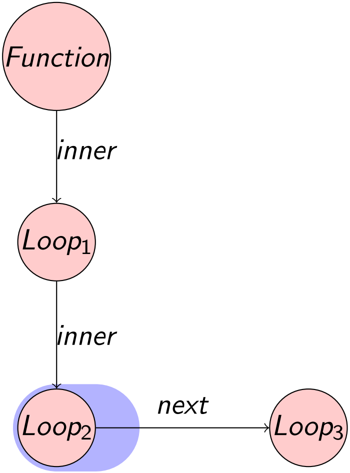
- Start with a loop in the natural loops tree
  rather than the root of the CFG

- Focus on structure of natural loops
  before the validity of each statement

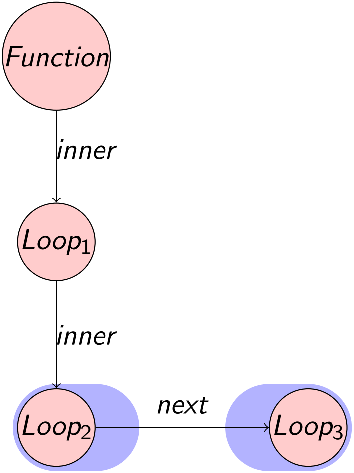
Example: Induction on Natural Loops Tree



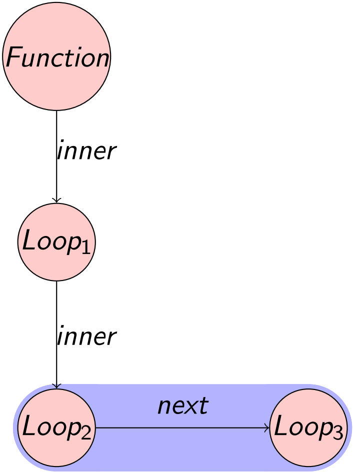
Example: Induction on Natural Loops Tree



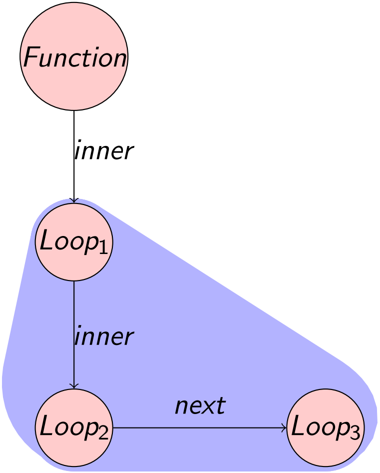
Example: Induction on Natural Loops Tree



Example: Induction on Natural Loops Tree



Example: Induction on Natural Loops Tree



Other implementations of SCoP Detection

- Previous graphite SCoP detection based on CFG and DOM (misses the structure of loops)

Other implementations of SCoP Detection

- Previous graphite SCoP detection based on CFG and DOM (misses the structure of loops)
- Polly's SCoP detection based on structure of SESE regions (full function body analysis even without interesting loops)

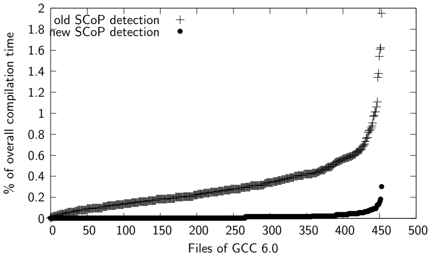
Other implementations of SCoP Detection

- Previous graphite SCoP detection based on CFG and DOM (misses the structure of loops)
- Polly's SCoP detection based on structure of SESE regions (full function body analysis even without interesting loops)
- Pet, Rose, other source-to-source compilers: SCoP detection based on the AST of a specific programming language

Experimental Results

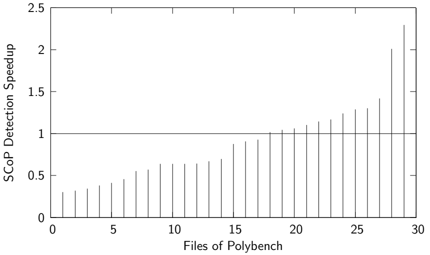
Compilation time overhead

Benchmark	Old %	New %
Polybench	1.4	1.9
Tramp3d-v4	7.0	0.3
GCC 6.0	0.24	0.01



SCoP Metrics on Polybench

SCoP Metric	Old	New	Polly
Loops/SCoP	2.59	6.09	5.17



Conclusion and Future work

Conclusion

- New faster algorithm for SCoP detection
- Enable polyhedral optimization in industrial compilers

Future Work

- SCoP detection to drive polyhedral optimization (avoid maximal SCoPs)
- Use profile data to guide and select polyhedral transforms