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

\[
\begin{align*}
e & \quad \downarrow \\
a & \quad \rightarrow \quad x \\
\uparrow & \\
b &
\end{align*}
\]

maybe SCoP
Step 1: accept natural loops

Natural loop

\[ e \xrightarrow{\downarrow} a \xrightarrow{x} b \]

maybe SCoP

Nested loops

\[ e \xrightarrow{\downarrow} b \xrightarrow{d} c \]

\[ a \xrightarrow{\downarrow} x \]

\[ a \xrightarrow{\downarrow} b \xrightarrow{\uparrow} c \]

maybe SCoP
Step 1: accept natural loops

Natural loop
\[
\begin{array}{c}
e \\
\downarrow \\
a \rightarrow x \\
\uparrow \\
b
\end{array}
\]
maybe SCoP

Nested loops
\[
\begin{array}{c}
e \\
\downarrow \\
a \rightarrow x \\
\downarrow \\
b \\
\downarrow \\
d \rightarrow c \\
\uparrow \\
b \\
\uparrow \\
c
\end{array}
\]
maybe SCoP

Irreducible
\[
\begin{array}{c}
c \quad e \\
\downarrow \\
b \leftrightarrow a \rightarrow x
\end{array}
\]
not a SCoP
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;
    }
}
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

\[ i_0 = \text{phi}_l(0, i_1) \]
\[ // i_0 = \{0, +, 1\}_l1 \]
\[ i_1 = i_0 + 1 \]
\[ // i_1 = \{1, +, 1\}_l1 \]

maybe SCoP

Non-linear

\[ j_2 = \text{phi}_l(3, j_3) \]
\[ j_3 = j_2 + i_1 \]
\[ // j_3 = \{3, +, \{1, +, 1\}_l1\}_l1 \]

not an ACL: polynomial of degree 2

Non-linear

\[ k_4 = \text{phi}_l(4, k_5) \]
\[ k_5 = k_4 \times 2 \]
\[ // k_5 = \{4, \times, 2\}_l2 \]

not an ACL: exponential

analyzed expressions ▶ branch conditions ▶ memory accesses
Step 3: affine scalar evolutions

Linear

\( i_0 = \text{phi}_l1(0, i1) \)
// \( i_0 = \{0, +, 1\}_l1 \)
\( i_1 = i_0 + 1 \)
// \( i_1 = \{1, +, 1\}_l1 \)

maybe SCoP

Non-linear

\( j_2 = \text{phi}_l1(3, j3) \)
\( j_3 = j_2 + i_1 \)
// \( j_2 = \{3, +, \{1, +, 1\}_l1\}_l1 \)

not an ACL: polynomial of degree 2
Step 3: affine scalar evolutions

Linear

\[ i_0 = \phi_{l1}(0, i_1) \]
// \[ i_0 = \{0,+,{1}_{l1}\}_{l1} \]
\[ i_1 = i_0 + 1 \]
// \[ i_1 = \{1,+,{1}_{l1}\}_{l1} \]

maybe SCoP

Non-linear

\[ j_2 = \phi_{l1}(3, j_3) \]
\[ j_3 = j_2 + i_1 \]
// \[ j_2 = \{3,+,{1}_{l1}\}_{l1} \]

not an ACL: polynomial of degree 2
Step 3: affine scalar evolutions

Linear

\[ i_0 = \phi_{l1}(0, i_1) \]
// \( i_0 = \{0, +, 1\}_{l1} \)

\[ i_1 = i_0 + 1 \]
// \( i_1 = \{1, +, 1\}_{l1} \)

maybe SCoP

Non-linear

\[ j_2 = \phi_{l1}(3, j_3) \]
\[ j_3 = j_2 + i_1 \]
// \( j_2 = \{3, +, \{1, +, 1\}_{l1}\}_{l1} \)

not an ACL: polynomial of degree 2

Non-linear

\[ k_4 = \phi_{l2}(4, k_5) \]
\[ k_5 = k_4 \times 2 \]
// \( k_4 = \{4, *, 2\}_{l2} \)

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

- Natural loops
  - no side-effects?
    - affine branch conditions?
      - affine memory accesses?
        - SCoP
Overall picture: SCoP detection

Natural loops

no side-effects?

affine branch conditions?

affine memory accesses?

SCoP

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

- **Function**
- **Loop**
- **Loop**
- **Loop**
- **next**
Example: Induction on Natural Loops Tree

Function

inner

Loop\textsubscript{1}

inner

Loop\textsubscript{2}

next

Loop\textsubscript{3}
Example: Induction on Natural Loops Tree

```
Function
    inner
    Loop1
        inner
        Loop2
            next
            Loop3
```

1. **Loop1**
2. **Loop2**
3. **Loop3**

Inner loops are connected by **next**.
Example: Induction on Natural Loops Tree

- Function
- Loop\(_1\)
- Loop\(_2\)
- Loop\(_3\)

Edges:
- inner
- next
Example: Induction on Natural Loops Tree

Function

inner

Loop$_1$

inner

Loop$_2$

next

Loop$_3$
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
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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Old %</th>
<th>New %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polybench</td>
<td>1.4</td>
<td>1.9</td>
</tr>
<tr>
<td>Tramp3d-v4</td>
<td>7.0</td>
<td>0.3</td>
</tr>
<tr>
<td>GCC 6.0</td>
<td>0.24</td>
<td>0.01</td>
</tr>
</tbody>
</table>

SCoP Metrics on Polybench

<table>
<thead>
<tr>
<th>SCoP Metric</th>
<th>Old</th>
<th>New</th>
<th>Polly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loops/SCoP</td>
<td>2.59</td>
<td>6.09</td>
<td>5.17</td>
</tr>
</tbody>
</table>
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