Characterization of Legal Transformation Sequences

Nicolas Vasilache
Albert Cohen
Sylvain Girbal
David Parello
Presentation Outline

- Context
- URUK representation - Examples
- Code Generation
- Dependence Analysis
- Violated Dependence Correction
- Future work
Non Perfect Loop Nest Transformations

Static Control Parts, affine parametric bounds

Array Accesses with affine parametric subscripts

Previous work on Polyhedral Model (Feautrier, Vivien, Darte …)
Flaws of Syntactic Loop Transformations

- Code size explosion
- Dependences evolve with transformations
- Rigid transformation sequences
Flaws of Syntactic Loop Transformations

- Code size explosion
- Dependences evolve with transformations
- Rigid transformation sequences

Need a representation where composition laws have a simple algebraic structure:
  - Giving (at least) the same expressiveness as classical semantics
  - Without conversion to/from syntactic form
Polyhedral Representation Goals

- Fully capture control and array access semantics of every statement
- Through affine (in)equality on iteration vectors
- Allow full parameterization by symbolic constants
- Allow classical loop transformations (and more)

URUK (Unified Representation Unified Kernel)
Represent static control code in terms of independent matrices

**Domain matrices**: ranges of possible values for iterators

```plaintext
for(i=1; i<= N; i++)
    for(j=1; j<=N; j++)
        S1(i,j)
```

```
i  j  N  1
1  0  0  0
-1  0  1  0
0  1  0  0
0  -1  1  0
```

\(i \geq 0\)
\(-i + N \geq 0\)
\(j \geq 0\)
\(-j + N \geq 0\)
Represent static control code in terms of **independent** matrices

**Domain matrices**: ranges of possible values for iterators

\[
\begin{align*}
\text{for}(i=1; i\leq N; i++) & \quad B(S_1) = [0, 0] \\
\text{for}(j=1; j\leq N; j++) & \quad B(S_2) = [0, 1, 0] \\
\text{for}(i=1; i\leq N; i++) & \quad B(S_3) = [0, 1, 1]
\end{align*}
\]

**Beta vectors**: syntactical interleaving of statements
URUK and Polyhedral Representation

- Represent static control code in terms of **independent** matrices
- **Domain matrices**: ranges of possible values for iterators
  
  ```
  for(i=1; i<= N; i++)
    for(j=1; j<=i;j++)
      S1(i,j)
  B(S1)  = [0, 0]
  B(S2)  = [0, 1, 0]
  B(S3)  = [0, 1, 1]
  ``

  ```
  i >= 0
  -i + N >= 0
  j >= 0
  -j + N >= 0
  ```

- **Beta vectors**: syntactical interleaving of statements
  
  ```
  for(i=1; i<= N; i++)
    S1(i)
    for(j=1; j<=i;j++)
      S2(i,j)
      S3(i,j)
  ```

  ```
  for(t0=1; t0<= N; t0++)
    for(t1=1; t1<=t0;t1++)
      S2(t0,t1)
      S3(t0,t1)
  ```

  ```
  for(t0=1; t0<= N; t0++)
    S1(t0)
  ```

  ```
  B(S1)  = [1, 0]
  B(S2)  = [0, 0, 0]
  B(S3)  = [0, 0, 1]
  ```
**URUK and Polyhedral Representation**

- Represent static control code in terms of **independent** matrices
- **Domain matrices**: ranges of possible values for iterators

```
for(i=1; i<= N; i++)
  for(j=1; j<=N;j++)
    S1(i,j)
```

```
for(i=1; i<= N; i++)
  S2(i,j)
S3(i,j)
```

```
for(i=1; i<= N; i++)
  S1(i)
```

```
for(i=1; i<= N; i++)
  for(j=1; j<=N;j++)
    S1(i,j)
```

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>N</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Beta vectors**: syntactical interleaving of statements

```
B(S1) = [1, 0]
B(S2) = [0, 0, 0]
B(S3) = [0, 0, 1]
```

```
for(t0=1; t0<= N; t0++)
  for(t1=1; t1<=N;t1++)
    S1(t0,t1)
```

```
B(S1) = [1, 0]
B(S2) = [0, 1, 0]
B(S3) = [0, 0, 0]
```

```
for(t0=1; t0<= N; t0++)
  for(t1=1; t1<=N;t1++)
    S2(t0,t1)
```

```
for(t0=1; t0<= N; t0++)
  S1(t0)
```
**URUK and Polyhedral Representation**

- **Gamma matrices**: Express constant delay of a statement respectively to another on a given dimension (shifting)

```plaintext
for(i=1; i<= N; i++)
    S1(i)
for(j=1; j<=N;j++)
    S2(i,j)
    S3(i,j)
G(S1) : 0 0
G(S2) : 0 0
G(S3) : 0 0
```
**Gamma matrices**: Express constant delay of a statement respectively to another on a given dimension (shifting)

\[
\begin{align*}
\text{for}(i=1; i<= N; i++) & \quad \text{G}(S1) : \begin{array}{cc}
0 & 0 \\
\end{array} & \quad \text{G}(S1) : \begin{array}{cc}
-1 & 0 \\
\end{array} \\
\text{S1}(i) & \quad \text{G}(S2) : \begin{array}{cc}
0 & 0 \\
0 & 0 \\
\end{array} & \quad \text{G}(S2) : \begin{array}{cc}
0 & 0 \\
0 & 2 \\
\end{array} \\
\text{for}(j=1; j<=N;j++) & \quad \text{S2}(i,j) & \quad \text{G}(S3) : \begin{array}{cc}
0 & 0 \\
\end{array} & \quad \text{G}(S3) : \begin{array}{cc}
0 & 0 \\
0 & 0 \\
\end{array} \\
\text{S3}(i,j) & \quad \end{align*}
\]
**URUK and Polyhedral Representation**

- **Gamma matrices**: Express constant delay of a statement respectively to another on a given dimension (shifting)

```plaintext
for(i=1; i<= N; i++)
  S1(i)
for(j=1; j<=N;j++)
  S2(i,j) S3(i,j)
```

<table>
<thead>
<tr>
<th>for(t0=1- N;t0&lt;= 0; t0++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1(t0+N)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>for(t0=1; t0&lt;= N; t0++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1(t0); S3(t0,2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>for(t1=1; t1&lt;=N-2;t1++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2(t0,t1) S3(t0,t1+2) S2(t0,N-1); S2(t0,N)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G(S1)</th>
<th>N 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G(S2)</th>
<th>N 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G(S3)</th>
<th>N 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>
```
**URUK and Polyhedral Representation**

- **Gamma matrices**: Express constant delay of a statement respectively to another on a given dimension (shifting)

  
  \[
  \begin{align*}
  &\text{for(}i=1; \ i\leq N; \ i++\text{)} \\
  &\quad \begin{array}{l}
  S1(i) \\
  \text{for(}j=1; \ j\leq N; j++\text{)} \\
  S2(i,j) \\
  S3(i,j)
  \end{array}
  \\
  \begin{array}{c}
  \text{G}(S1) \quad \begin{pmatrix} 0 & 0 \end{pmatrix} \\
  \text{G}(S2) \quad \begin{pmatrix} 0 & 0 \end{pmatrix} \\
  \text{G}(S3) \quad \begin{pmatrix} 0 & 0 \end{pmatrix}
  \end{array}
  \end{align*}
  \]

- **for(}t0=1- N; \ t0\leq 0; \ t0++\text{)} \\
  \begin{array}{l}
  S1(t0+N) \\
  \text{for(}t0=1; t0\leq N; t0++\text{)} \\
  S3(t0,1); S3(t0,2) \\
  \text{for(}t1=1; t1\leq N-2; t1++\text{)} \\
  S2(t0,1); S3(t0,1+2) \\
  S2(t0,N-1); S2(t0,N)
  \end{array}
  \\
  \begin{array}{c}
  \text{G}(S1) \quad \begin{pmatrix} -1 & 0 \end{pmatrix} \\
  \text{G}(S2) \quad \begin{pmatrix} 0 & 0 \end{pmatrix} \\
  \text{G}(S3) \quad \begin{pmatrix} 0 & 0 \end{pmatrix}
  \end{array}
  \]

- **Alpha matrices**: Express speed of a statement’s in terms of linear combination of domain iterators

  \[
  \begin{align*}
  &\text{for(}i=1; \ i\leq N; i++\text{)} \\
  &\quad \begin{array}{l}
  S1(i,j) \\
  \text{for(}j=1; j\leq M; j++\text{)} \\
  S1(i,j)
  \end{array}
  \\
  \begin{array}{c}
  \text{A}(S1) \quad \begin{pmatrix} 1 & 0 \\
  0 & 1 \end{pmatrix} \\
  \text{A}(S1) \quad \begin{pmatrix} 0 & 1 \\
  1 & 0 \end{pmatrix}
  \end{array}
  \end{align*}
  \]
**URUK and Polyhedral Representation**

- **Gamma matrices**: Express constant delay of a statement respectively to another on a given dimension (shifting)

\[
\begin{align*}
\text{for}(i=1; i<= N; i++) & \quad \text{G(S1)}: 0 \quad 0 \\
\text{for}(j=1; j<=N;j++) & \quad \text{G(S2)}: 0 \quad 0 \\
\text{S1(i)} & \quad \text{G(S3)}: 0 \quad 0 \\
\text{S2(i,j)} & \\
\text{S3(i,j)} & \\
\end{align*}
\]

- **Alpha matrices**: Express speed of a statement’s in terms of linear combination of domain iterators

\[
\begin{align*}
\text{for}(i=1; i<= N; i++) & \quad \text{A(S1)}: 1 \quad 0 \\
\text{for}(j=1; j<=M;j++) & \quad 0 \quad 1 \\
\text{S1(i,j)} & \\
\end{align*}
\]
Polyhedral Code Generation

- Transformations are commutative, easy to define, compose, decoupled from syntactic representation
Polyhedral Code Generation

- Transformations are commutative, easy to define, compose, decoupled from syntactic representation
- Combining A, B and G $\rightarrow$ Affine Schedule Function
- Define a multidimensional time (with lexicographic order) and map domain iterators on time dimensions
Polyhedral Code Generation

- Transformations are commutative, easy to define, compose, decoupled from syntactic representation
- Combining A, B and G → Affine Schedule Function
- Define a multidimensional time (with lexicographic order) and map domain iterators on time dimensions
- Need to iterate on "time" to execute instructions
- Code Generation deals with retranscription into for loops
Polyhedral Code Generation

- Transformations are commutative, easy to define, compose, decoupled from syntactic representation
- Combining A, B and G → Affine Schedule Function
- Define a multidimensional time (with lexicographic order) and map domain iterators on time dimensions
- Need to iterate on "time" to execute instructions
- Code Generation deals with retranscription into \texttt{for} loops

\[
\Theta(i, q) = \begin{bmatrix}
0 & \ldots & 0 & 0 & \ldots & 0 & \beta_0 \\
\alpha_{1,1} & \ldots & \alpha_{1,d} & \gamma_{1,1} & \ldots & \gamma_{1,g} & \gamma_{1,g+1} \\
0 & \ldots & 0 & 0 & \ldots & 0 & \beta_1 \\
\alpha_{2,1} & \ldots & \alpha_{2,d} & \gamma_{2,1} & \ldots & \gamma_{2,g} & \gamma_{2,g+1} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\alpha_{d,1} & \ldots & \alpha_{d,d} & \gamma_{d,1} & \ldots & \gamma_{d,g} & \gamma_{d,g+1} \\
0 & \ldots & 0 & 0 & \ldots & 0 & \beta_d
\end{bmatrix}
\begin{bmatrix}
i_1 \\
i_d \\
q_1 \\
\vdots \\
q_g \\
i_1 \\
\vdots \\
q_g \end{bmatrix}
\text{ with }
\begin{bmatrix}
i_1 \\
i_d \\
q_1 \\
\vdots \\
q_g \\
i_1 \\
\vdots \\
q_g \end{bmatrix}
\text{ in } D
Project Synthesis

- Based on ORC’s IR, LNO phase short-circuited
- Compatible with EKOPath
- Back-end generates code for x86-32, x86-64, IA64
Polyhedral Dependence Analysis

- Dependence $S_1 \rightarrow S_2$ at depth $k$:
  - Iterators $I(S_1)$ in $D(S_1)$, $I(S_2)$ in $D(S_2)$
  - $\text{Beta.prefix}(k)(S_1) == \text{Beta.prefix}(k)(S_2)$
  - $\Theta(S_1)[0\ldots k-1] == \Theta(S_2)[0\ldots k-1]$
  - $\Theta(S_1)[k] < \Theta(S_2)[k]$
  - $\text{Access}(S_1)[I(S_1)] == \text{Access}(S_2)[I(S_2)]$
Polyhedral Dependence Analysis

- Dependence $S_1 \rightarrow S_2$ at depth $k$:
  - Iterators $I(S_1)$ in $D(S_1)$, $I(S_2)$ in $D(S_2)$
  - $\beta_{\text{prefix}}(k)(S_1) == \beta_{\text{prefix}}(k)(S_2)$
  - $\Theta(S_1)[0…k-1] == \Theta(S_2)[0…k-1]$
  - $\Theta(S_1)[k] < \Theta(S_2)[k]$
  - $\text{Access}(S_1)[I(S_1)] == \text{Access}(S_2)[I(S_2)]$

- Adding all these constraints yields exact $(I(S_1), I(S_2))$ in $D_1 \times D_2$ corresponding to a dependence at depth $k$.

- Dependence computation is done only once and captures exact order of all instructions
  - Transformations don’t modify Dependency Graph
Polyhedral Dependence Analysis

- Dependence \( S_1 \) -> \( S_2 \) at depth \( k \):
  - Iterators \( I(S_1) \) in \( D(S_1) \), \( I(S_2) \) in \( D(S_2) \)
  - \( \beta_{\text{prefix}}(k)(S_1) = \beta_{\text{prefix}}(k)(S_2) \)
  - \( \Theta(S_1)[0…k-1] = \Theta(S_2)[0…k-1] \)
  - \( \Theta(S_1)[k] < \Theta(S_2)[k] \)
  - \( \text{Access}(S_1)[I(S_1)] = \text{Access}(S_2)[I(S_2)] \)

- Adding all these constraints yields exact \( (I(S_1), I(S_2)) \) in \( D_1 \times D_2 \) corresponding to a dependence at depth \( k \).

- Dependence computation is done only once and captures exact order of all instructions
  - \( \Rightarrow \) Transformations don’t modify Dependency Graph

- Polyhedron empty \( \Rightarrow \) No dependence
Dependence Analysis - Example

for(i=0; i< N; i++)
for(j=0; j<i; j++)
    C[i][j] += A[i][i];
    C[i+j][j] += C[i][j-1] + C[i-1][j];
Dependence Analysis - Example

for(i=0; i<N; i++)
    for(j=0; j<i; j++)
        C[i][j] += A[i][i];
        C[i+j][j] += C[i][j-1] + C[i-1][j];

for(i=0; i<N; i++)
    for(j=0; j<i; j++)
        C[i][j] += A[i][i];
        C[i+j][j] += C[i][j-1] + C[i-1][j];

OPR_ISTORE #22_ID=5 --> OPR_ISTORE #20_ID=4
OPR_ISTORE #22_ID=5 --> OPR_ISTORE #20_ID=4

raw

C --> CDepth: 1

After simplification

Access Schedule

D(S1)

D(S2)
Consider a complex transformation on matrices

Legality ensured iff for each previously built dependence:
- The polyhedron augmented with the constraint:
  \[ \text{New}_\theta(\text{Source}) - \text{New}_\theta(\text{Sink}) > 0 \]
  is empty

Otherwise the former dependence yields violated dependences
Consider a complex transformation on matrices.
Legality ensured iff for each previously built dependence:
- The polyhedron augmented with the constraint:
  \[
  \text{New}_\text{Theta(Source)} - \text{New}_\text{Theta(Sink)} > 0
  \]
  is empty
Otherwise the former dependence yields violated dependences.

Legality check is not done on a single transformation.
Series of illegal transformations can be legal.
Consider a complex transformation on matrices
Legality ensured iff for each previously built dependence:
  - The polyhedron augmented with the constraint:
    \[
    \text{New}_\text{Theta}(\text{Source}) - \text{New}_\text{Theta}(\text{Sink}) > 0
    \]
    is empty
Otherwise the former dependence yields violated dependences

Legality check is not done on a single transformation
Series of illegal transformations can be legal

Fixup (enabling) transformations can be generated
Violated Dependence Analysis - Example

```plaintext
for(i=0; i< N; i++)
    for(j=0; j<i; j++)
        C[i][j] += A[i][i]; C[i+j][j] += C[i][j-1] + C[i-1][j];

for(t0=1; t0<=N-2; t0++)
    for(t1=0; t1<=t0-1; t1++)
        C[t0+1+t1][t1] += C[t0+1][t1-1] + C[t0][t1];
```

Swap(S1,S2)
Shift(S2) by -1 wrt i

```plaintext
for(i=0; i< N; i++)
    for(j=0; j<i; j++)
        C[i][j] += A[i][j];
        C[i+j][j] += C[i][j-1] + C[i-1][j];

for(t0=1; t0<=N-2; t0++)
    for(t1=0; t1<=t0-1; t1++)
        C[t0+1+t1][t1] += C[t0+1][t1-1] + C[t0][t1];
        C[t0][t1] += A[t0][t0];
        C[2*t0+1][t0] += C[t0+1][t0-1] + C[t0][t0];

for(t1=0; t1<=N-2; t1++)
    C[N-1][t1] += A[N-1][N-1];
```
Violated Dependence Analysis - Example

for(i=0; i< N; i++)
  for(j=0; j<i; j++)
    C[i][j] += A[i][i];
    C[i+j][j] += C[i][j-1] + C[i-1][j];

for(t0=1; t0<=N-2; t0++)
  for(t1=0; t1<=t0-1; t1++)
    C[t0+1+t1][t1] += C[t0+1][t1-1] + C[t0][t1];
    C[t0][t1] += A[t0][t0];
    C[2*t0+1][t0] += C[t0+1][t0-1] + C[t0][t0];
  for(t1=0; t1<=N-2; t1++)
    C[N-1][t1] += A[N-1][N-1];

ILLEGAL
Enabling Transformations

- Systematic strategy to correct violated dependences uses Feautrier’s Pip (Parametric Simplex Solver)
Enabling Transformations

- Systematic strategy to correct violated dependences uses Feautrier’s Pip (Parametric Simplex Solver)

- Given a violated dependence at depth level $k$:
  
  - What is the minimal amount of shifting at level $k$ to correct it?
  - Use Pip to solve $\text{MAX}(\text{New}_\Theta(\text{sink}) - \text{New}_\Theta(\text{source}))$
Enabling Transformations

- Systematic strategy to correct violated dependences uses Feautrier’s Pip (Parametric Simplex Solver)

- Given a violated dependence at depth level k:
  - What is the minimal amount of shifting at level k to correct it?
  - Use Pip to solve $\max(\text{New}_\Theta(\text{sink}) - \text{New}_\Theta(\text{source}))$

- Pip may find different shifting values corresponding to a subdivision of cases, use cases for index set splitting
Enabling Transformations

- Systematic strategy to correct violated dependences uses Feautrier’s Pip (Parametric Simplex Solver)

- Given a violated dependence at depth level k:
  - What is the minimal amount of shifting at level k to correct it?
  - Use Pip to solve \( \text{MAX}(\text{New}_\text{Theta}(\text{sink}) - \text{New}_\text{Theta}(\text{source})) \)

- Pip may find different shifting values corresponding to a subdivision of cases, use cases for index set splitting

- Systematic method used on SPECFP2000’s swim
  - 38% speedup on AMD64
  - Fusion + Blocking
  - Code Generation Bound
Conclusions and Future Works

- Framework that allows complex transformations
- Exact dependence analysis and enabling transformations
- Acceptable overhead for analysis and code generation
Conclusions and Future Works

● Framework that allows complex transformations
● Exact dependence analysis and enabling transformations
● Acceptable overhead for analysis and code generation

Next :
- Automatize the generation of enabling transformations
- ADA + array expansion to focus only on RAW dependences
- Drive the selection of transformations without being blocked by legality issues
- Extend model to more general loops
- Cost models