The Polyhedral Model Is More Widely Applicable Than You Think

Mohamed-Walid Benabderrahmane [1] Louis-Noël Pouchet [1,2]
Albert Cohen [1] Cédric Bastoul [1]

[1] **ALCHEMY group, INRIA Saclay / University of Paris-Sud 11, France**
[2] **The Ohio State University, USA**

March 26, 2010

Motivation: High Level Optimization

Complex program *transformations*

- To exhibit and to exploit *parallelism*

Type	implicit/explicit	Extraction
Instruction pipeline	implicit	hardware + compiler
Superscalar	implicit	hardware + compiler
VLIW-EPIC	explicit	compiler
Vector	explicit	compiler
Multithreading	explicit	compiler + system

- To benefit from *data locality*

Type	implicit/explicit	Extraction
Temporal locality	implicit (except on local memories)	compiler
Spatial locality	implicit (except on some DSPs)	compiler

Finding & Applying Transformations

Very hard in general

- Which transformations, in which order?
- Is the semantics preserved?
- Is it profitable (performance, energy...)?

Much easier *within the scope* of the polyhedral model

- Complex sequences of optimizations in a single step
- Exact data dependence analysis
- Many existing optimizing algorithms
- But restricted to static control codes

Contributions:

- Extending the polyhedral model to handle full functions
- Revisiting the framework to support these extensions
- Demonstrate that codes with data-dependent control flow may benefit from existing techniques, even with conservative dependence approximations

Outline

Polyhedral Representation

For each program statement, capture its *control array access* semantics through *parametrized* affine (in)equalities:

1. A *domain* $\mathcal{D} : A\vec{x} + \vec{a} \geq \vec{0}$
   The bounds of the enclosing loops
2. A list of *access functions* $f(\vec{x}) = F\vec{x} + \vec{f}$
   To describe array references
3. A *schedule* $\theta(\vec{x}) = T\vec{x} + \vec{t}$
   An affine function assigning logical dates to iterations

```
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    if (i <= n-j+2)
S1:   M[2*i+1][i-j+n] = 0;
```

$$\mathcal{D}_{S_1} : \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \\ n+2 \end{pmatrix} \geq \vec{0}$$

Iteration Domain of S_1

Polyhedral Representation

For each program statement, capture its *control array access* semantics through *parametrized* affine (in)equalities:

1. A *domain* $\mathcal{D} : A\vec{x} + \vec{a} \geq \vec{0}$
   The bounds of the enclosing loops
2. A list of *access functions* $f(\vec{x}) = F\vec{x} + \vec{f}$
   To describe array references
3. A *schedule* $\theta(\vec{x}) = T\vec{x} + \vec{t}$
   An affine function assigning logical dates to iterations

```
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    if (i <= n-j+2)
S1:   M[2*i+1][i-j+n] = 0;
```

$$f_{S_1,M}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & -1 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 1 \\ n \end{pmatrix}$$

Subscript Function of M[$f(\vec{x})$]

Polyhedral Representation

For each program statement, capture its *control array access* semantics through *parametrized* affine (in)equalities:

1. A *domain* $\mathcal{D} : A\vec{x} + \vec{a} \geq \vec{0}$
   The bounds of the enclosing loops
2. A list of *access functions* $f(\vec{x}) = F\vec{x} + \vec{f}$
   To describe array references
3. A *schedule* $\theta(\vec{x}) = T\vec{x} + \vec{t}$
   An affine function assigning logical dates to iterations

```
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    if (i <= n-j+2)
S1:    M[2*i+1][i-j+n] = 0;
```

$$\theta_{S_1}\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Identity Schedule

Polyhedral Model Constraints

Strict control constraints to be eligible: *static control*

- Affine bounds (`for`)
- Affine conditions (`if`)

Does it mean that more general codes cannot benefit from a polyhedral compilation framework?

Motivating *Transformation*: Loop Fusion

```
// 2strings: count occurences of two words in the same string
nb1 = 0;
for(i=0; i < size_string - size_word1; i++){
   match1 = 0;
   while(word1[match1] == string[i+match1] && match1 <= size_word1)
       match1++;
   if (match1 == size_word1)
       nb1++;
}
nb2 = 0;
for(i=0; i < size_string - size_word2; i++) {
   match2 = 0;
   while(word2[match2] == string[i+match2] && match2 <= size_word2)
       match2++;
   if (match2 == size_word2)
       nb2++;
}
```

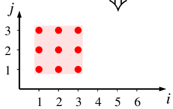
- Loop fusion would improve data locality
- Tough by hand
- Trivial transformation if expressed in the polyhedral domain
- But `while` loops and non-static `if` conditions here...

Revisiting The Polyhedral Framework

```
for (i = 1; i <= 3; i++)
  for (j = 1; j <= 3; j++)
    A[i+j] = ...
```

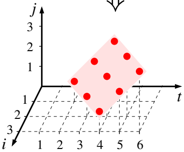
1 Program analysis

⇓



2 Affine transformation

⇓



3 Code generation

⇓

```
for (t = 2; t <= 6; t++)
  for (i = max(1,t-3); i <= min(t-1,3); i++)
    A[t] = ...
```

Extension to `while` Loops

- Extend iteration domain to support predication tags
- (Virtually) Convert `while` loops into infinite `for` loops
- Tag statement iteration domains with *exit predicates*

```
while (condition)
  S();
```

(a) Original Code

```
for (i = 0;; i++) {
  ep = condition;
  if (ep)
    S();
  else
    break;
}
```

(b) Equivalent Code

$$\begin{cases} i \geq 0 \\ (ep = condition) \end{cases}$$

(c) Iteration Domain of S

Extension to Non-Static `if` Conditionals

- Extend iteration domain to support predication tags
- Tag statement iteration domains with *control predicates*

```
for (i = 0; i < N; i++)
  if (condition)
    S();
```

(a) Original Code

```
for (i = 0; i < N; i++)
  cp = condition;
  if (cp)
    S();
```

(b) Equivalent Code

$$\begin{cases} i \geq 0 \\ i < N \\ (cp = condition) \end{cases}$$

(c) Iteration Domain of S

A Conservative Approach

Problem: exact data dependence analysis is not always possible
Conservative escape: it is safe to consider extra dependences

- Non-static control is *over-approximated* (predicates considered always true)
- Non-static references are *over-approximated* (e.g. arrays are considered as single variables)
- Predicate evaluations are considered as plain statements
- Predicated statements depend on their predicate definitions

► OK for data dependence analysis but not sufficient for some more evolved analyses (see paper)

A Conservative Approach: Example (Outer Product Kernel)

Original Kernel

```
for (i = 0; i < N; i++) {
  if (x[i] == 0) {
    for (j = 0; j < M; j++) {
      A[i][j] = 0;
    }
  }
  else {
    for (j = 0; j < M; j++) {
      A[i][j] = x[i] * y[j];
    }
  }
}
```

A Conservative Approach: Example (Outer Product Kernel)

Control Predication

```
for (i = 0; i < N; i++) {
  cp = (x[i] == 0);
  for (j = 0; j < M; j++) {
    if (cp) {
      A[i][j] = 0;
    }
  }
  for (j = 0; j < M; j++) {
    if (!cp) {
      A[i][j] = x[i] * y[j];
    }
  }
}
```

A Conservative Approach: Example (Outer Product Kernel)

Abstract Program for Data Dependence Analysis

```
for (i = 0; i < N; i++) {
  S0: Write = {cp}, Read = {x[i]}
  for (j = 0; j < M; j++) {
    S1: Write = {A[i][j]}, Read = {cp}
  }
  for (j = 0; j < M; j++) {
    S2: Write = {A[i][j]}, Read = {x[i], y[j], cp}
  }
}
```

Transformation Expressiveness Recovery

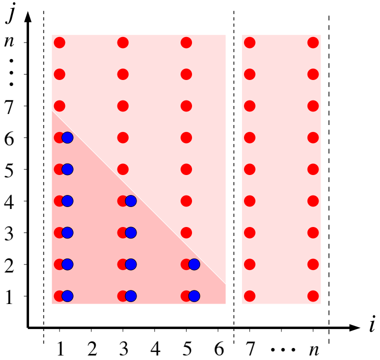
Problem: manipulating unbounded domains is not easy
(how to distribute `while` loops with one-dimensional schedule?)

Solution: an artificial parameter, *"w"* (meaning ω, or *while*)

- The upper bounds of all unbounded loops is w
- w is strictly greater than all upper bounds
- w is only used in affine transformations
- w is removed during the code generation process
- *w allows any existing polyhedral transformation technique to be used in the extended model*

Quilleré-Rajopadhye-Wilde Algorithm

- Direct use of polyhedral operations [Quilleré et al. IJPP00]
- Depth recursion with direct optimization of conditionals:
  - Projection onto outer dimensions
  - Separation into disjoint polyhedra




```
for(i = 1; i <= 6; i += 2)
  for (j = 1; j <= 7-i; j++) {
    S1(i, j);
    S2(i, j);
  }
  for (j = 8-i; j <= n; j++)
    S1(i, j);
}
for (i = 7; i <= n; i += 2)
  for (j = 1; j <= n; j++)
    S1(i, j);
```

Predicate Post-Processing

- Usual QRW code generation for predicated domains
- Exit and control predicates are post-processed
  - The target code is modified according to the situation
  - Post-pass insertion of predicate evaluations

▶ First scenario: same exit predicates

```
for (i = 0; i < w; i++) {
  S1(); {ep1}
  S2(); {ep1}
}
```

(a) Intermediate Code

```
while (ep1) {
  S1();
  S2();
}
```

(b) Post-Processed Code

Predicate Post-Processing

- Usual QRW code generation for predicated domains
- Exit and control predicates are post-processed
  - The target code is modified according to the situation
  - Post-pass insertion of predicate evaluations

► Second scenario: different exit predicates

```
for (i = 0; i < w; i++) {
  S1(); {ep1}
  S2(); {ep2}
}
```

(a) Intermediate Code

```
while (ep1 && ep2) {
  S1();
  S2();
}
while (ep1)
  S1();
while (ep2)
  S2();
```

(b) Post-Processed Code

Predicate Post-Processing

- Usual QRW code generation for predicated domains
- Exit and control predicates are post-processed
  - The target code is modified according to the situation
  - Post-pass insertion of predicate evaluations

▶ Third scenario: exit predicate inside a regular loop

```
for (i = 0; i < w; i++) {
  S1();
  S2(); {ep1}
}
```

(a) Intermediate Code

```
stop1 = 0;
for (i = 0; i < N; i++) {
  S1();
  if (ep1 && !stop1)
    S2();
  else
    stop1 = 1;
}
```

(b) Post-Processed Code

Predicate Post-Processing

- Usual QRW code generation for predicated domains
- Exit and control predicates are post-processed
  - The target code is modified according to the situation
  - Post-pass insertion of predicate evaluations
- Additional optimizations
  - Hoisting predicate evaluations
  - Privatization of predicate variables

Experimental Results

State-of-the-art polyhedral optimization techniques applied to (partially) irregular programs

- LeTSeE [Pouchet et al. PLDI08]
- Pluto [Bondhugula et al. PLDI08]

	Speedup regular		Speedup extended		Compilation time penalty	
	LetSee	Pluto	LetSee	Pluto	LetSee	Pluto
2strings	N/A	N/A	1.18×	1×	N/A	N/A
Sat-add	1×	1.08×	1.51×	1.61×	1.22×	1.35×
QR	1.04×	1.09×	1.04×	8.66×	9.56×	2.10×
ShortPath	N/A	N/A	1.53×	5.88×	N/A	N/A
TransClos	N/A	N/A	1.43×	2.27×	N/A	N/A
Givens	1×	1×	1.03×	7.02×	21.23×	15.39×
Dither	N/A	N/A	1×	5.42×	N/A	N/A
Svdvar	1×	3.54×	1×	3.82×	1.93×	1.33×
Svbksb	1×	1×	1×	1.96×	2×	1.66×
Gauss-J	1×	1.46×	1×	1.77×	2.51×	1.22×
PtIncluded	1×	1×	1×	1.44×	10.12×	1.44×

Setup: Intel Core 2 Quad Q6600

Backend compiler (and baseline): ICC 11.0 `icc -fast -parallel -openmp`

Conclusion

The limitation to static control programs is mostly artificial

- *Slight and natural extension* to consider irregular codes
  - Infinite loops plus exit and control predication
  - w parameter to preserve affine schedule expressiveness
  - Code generation with predicate support
- Benefit from *unmodified* existing techniques for both analysis and optimization
- Currently rely on a conservative dependence analysis

New extensions should be investigated

- Minimizing the conservative aspects (inspection and speculation for control dependences)
- Designing optimizations in the context of full functions (algorithmic complexity issues)