Approximations in the polyhedral model

Alain Darte

CNRS, Inria Comsys project-team
Laboratoire de l’Informatique du Parallélisme
École normale supérieure de Lyon

Impact’11, Chamonix, April 3, 2011

Thanks to:
Y. Robert, F. Vivien, F. Irigoin, G.-A. Silber, G. Huard, G. Villard,
R. Schreiber, F. Baray, P. Feautrier, C. Alias, A. Plesco, L. Gonnord
Outline

1. The polyhedral model
2. Scheduling, SURES, and approximated loops
3. Data mapping & communication optimizations
1. The polyhedral model
   - Paul Feautrier’s static control programs
   - Analyses, optimizations, and tools
   - The polyhedral model is... a model

2. Scheduling, SURES, and approximated loops

3. Data mapping & communication optimizations
Affine bounds and affine array access functions

**Fortran DO loops:**

```
DO i=1, N
    DO j=1, N
        a(i,j) = c(i,j-1)
        c(i,j) = a(i,j) + a(i-1,N)
    ENDDO
ENDDO
```

- Nested loops, static control.
- Iteration domain and vector.
- Loop increment = 1.
- Affine bounds of surrounding counters & parameters.
- Multi-dimensional arrays, same restriction for access functions.
Affine bounds and affine array access functions

**Fortran DO loops:**

```fortran
DO i=1, N
  DO j=1, N
    a(i,j) = c(i,j-1)
    c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```

- Nested loops, static control.
- Iteration domain and vector.
- Loop increment = 1.
- Affine bounds of surrounding counters & parameters.
- Multi-dimensional arrays, same restriction for access functions.

**Polyhedral model: the “all-affine” world, with exact analysis**

- Iteration domain = polytope.
- Sequential order $\leq_{seq}$.
- Data = images of polytopes by affine functions.
Affine bounds and affine array access functions

Fortran DO loops:

```
DO i=1, N
  DO j=1, N
    a(i,j) = c(i,j-1)
    c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```

- Nested loops, static control.
- Iteration domain and vector.
- Loop increment = 1.
- Affine bounds of surrounding counters & parameters.
- Multi-dimensional arrays, same restriction for access functions.

🌞 Polyhedral model: the “all-affine” world, with exact analysis

- Iteration domain = polytope.
- Sequential order $\leq_{seq}$.
- Data = images of polytopes by affine functions.

☁️ Typical criticism: such codes do not exist.
(Parametric) analysis, transformations, optimizations

Data-flow array analysis
- Array expansion.
- Single assignment.
- Liveness array analysis.
- Data reuse.

Mapping computations & data
- Systolic arrays design.
- Data distribution.
- Communication opt.

Loop transformations
- Automatic parallelization.
- Transformations framework.
- Code generation (with loops or with automaton).

Counting & Ehrhart polynomials
- Cache misses.
- Memory size computations.
- Latency computations.

And many more...
The polyhedral model is... a model

Many languages fit in the polyhedral model

C for loops:

```c
for (i=1, i<=N, i++) {
    for (j=1, j<=N, j++) {
        a[i][j] = c[i][j-1];
        c[i][j] = a[i][j] + a[i-1][N];
    }
}
```

C while loops:

```c
y = 0; x = 0;
while (x <= N && y <= N) {
    if (?) {
        x=x+1;
        while (y >= 0 && ?) y=y-1;
    }
    y=y+1;
}
```

Uniform recurrence equations

∀(i,j) such that 1 ≤ i,j ≤ N

```latex
\begin{align*}
    a(i,j) &= c(i,j - 1) \\
    b(i,j) &= a(i - 1,j) + b(i,j + 1) \\
    c(i,j) &= a(i,j) + b(i,j)
\end{align*}
```

FAUST: audio processing

```c
random = +(12345) ~ *(1103515);
noise = random/2147483.0;
process = random/2 : @(10);
```

and more: Matlab, Fortran90, StreamIt, HPF, C for HLS, ...
Many tools and a recent revival

**PIP** Parametric integer programming.
**POLYLIB** Polyhedra manipulations.
**FADALIB** Fuzzy array data-flow analysis.
**CLOOG** Code generation, from polytopes to loops.
**EHRHART & BARVINOK** Counting tools.
**CL@K** Critical and admissible lattices.
**PIPS** Automatic parallelizer & code transformation framework.
**PLUTO** Automatic parallelizer & locality optimizer for multicores.
**GRAPHITE** High-level memory optimizations framework in GCC.
**R-STREAM** High-level compiler of Reservoir Labs.

...
But still, how to deal with non-static control programs?

Polyhedral model.
But still, how to deal with non-static control programs?

Polyhedral model.

Real life.
But still, how to deal with non-static control programs?

Polyhedral model.

Real life.

Extensions.

- Non-affine constraints.
- Handling of while loops.
- Recursive programs.
- Beyond induction variables.
But still, how to deal with non-static control programs?

Polyhedral model.

Extensions.
- Non-affine constraints.
- Handling of while loops.
- Recursive programs.
- Beyond induction variables.

Approximations.
- Dependences, lifetime, data & iteration domains, etc.
- Do not assume exact information is available.

Real life.
But still, how to deal with non-static control programs?

Polyhedral model.

Real life.

Extensions.
- Non-affine constraints.
- Handling of while loops.
- Recursive programs.
- Beyond induction variables.

Approximations.
- Dependences, lifetime, data & iteration domains, etc.
- Do not assume exact information is available.

Think conservative!
Is there some **loop parallelism** (i.e., parallel loop iterations) in the following two codes? What is their **degree of parallelism**?

\[
\begin{align*}
\text{DO } & i=1, N \\
\text{DO } & j=1, N \\
& a(i,j) = c(i,j-1) \\
& c(i,j) = a(i,j) + a(i-1,N) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]
Is there some loop parallelism (i.e., parallel loop iterations) in the following two codes? What is their degree of parallelism?

\begin{align*}
\text{DO } i=1, N \\
&\text{DO } j=1, N \\
&\quad a(i,j) = c(i,j-1) \\
&\quad c(i,j) = a(i,j) + a(i-1,N) \\
&\text{ENDDO} \\
&\text{ENDDO}
\end{align*}

\begin{align*}
\text{DO } i=1, N \\
&\text{DO } j=1, N \\
&\quad a(i,j) = c(i,j-1) \\
&\quad c(i,j) = a(i,j) + a(i-1,j) \\
&\text{ENDDO} \\
&\text{ENDDO}
\end{align*}
Does this program **terminate**?
If yes, **how many steps** in the worst case? Useful for **WCET**.

```plaintext
y = 0; x = 0;
while (x <= N && y <= N) {
    if (?) {
        x=x+1;
        while (y >= 0 && ?) y=y-1;
    }
    y=y+1;
}
```

Terminates in at most $N^2 + 3N + 2 = O(N^2)$ steps.

Note: a single while loop can generate quadratic (or more) WCCC. Surprisingly, similar to parallel detection in Fortran DO loops.
Does this program **terminate**?
If yes, **how many steps** in the worst case? Useful for **WCET**.

```c
y = 0; x = 0;
while (x <= N && y <= N) {
    if (?) {
        x=x+1;
        while (y >= 0 && ?) y=y-1;
    }
    y=y+1;
}
```

Terminates in at most $N^2 + 3N + 2 = O(N^2)$ steps.

Note: a single while loop can generate quadratic (or more) WCCC. Surprisingly, similar to parallel detection in Fortran DO loops.
Outline

1. The polyhedral model

2. Scheduling, SURES, and approximated loops
   - System of uniform recurrence equations
   - Multi-dimensional scheduling and parallel loop detection
   - Multi-dimensional ranking and worst-case execution time

3. Data mapping & communication optimizations
SURE: system of uniform recurrence equations (1967)


∀p ∈ P = \{p = (i, j) | 1 ≤ i, j ≤ N\}

\begin{align*}
 a(i, j) &= c(i, j - 1) \\
 b(i, j) &= a(i - 1, j) + b(i, j + 1) \\
 c(i, j) &= a(i, j) + b(i, j)
\end{align*}

Semantics:

- **RDG** (reduced dependence graph) \( G = (V, E, w) \).
- Explicit dependences & iteration domain \( P \), implicit schedule.
- \( e = (u, v) \Leftrightarrow v(p) \) depends on \( u(p - w(e)) \), i.e., must be computed after. If \( p - w(e) \notin P \), it is an input.
- **EDG** (expanded dep. graph): vertices \( V \times P = \) unrolled RDG.
Looking for zero-weight cycles

**Computability:** Can we compute \( a(p) \) in a finite number of steps?

**Scheduling:** If yes, how to find an explicit and “good” schedule?

**Lemma 1**

A SURE is computable for all bounded domains \( \mathcal{P} \) if and only if the RDG has *no cycle* \( C \) with \( w(C) = 0 \).
Looking for zero-weight cycles

**Computability:** Can we compute $a(p)$ in a finite number of steps?

**Scheduling:** If yes, how to find an explicit and “good” schedule?

**Lemma 1**

*An SURE is computable for all bounded domains $\mathcal{P}$ if and only if the RDG has no cycle $C$ with $w(C) = 0$.*

**Key structure:** the subgraph $G'$ induced by all edges that belong to a multi-cycle (i.e., union of cycles) of zero weight.
Key properties

Three elementary key lemmas.

Lemma 2

A zero-weight cycle is a zero-weight multi-cycle.

Look in $G'$ only.
Key properties

Three elementary key lemmas.

**Lemma 2**

* A zero-weight cycle is a zero-weight multi-cycle.
  - Look in $G'$ only.

**Lemma 3**

* A zero-weight cycle belongs to a strongly connected component.
  - Look in each strongly connected component (SCC) separately.
Key properties

Three elementary key lemmas.

**Lemma 2**

*A zero-weight cycle is a zero-weight multi-cycle.*

Look in $G'$ only.

**Lemma 3**

*A zero-weight cycle belongs to a strongly connected component.*

Look in each strongly connected component (SCC) separately.

**Lemma 4**

*If $G'$ is strongly connected, there is a zero-weight cycle.*

Terminating case.
Lemma 5

If $G'$ is strongly connected, there is a zero-weight cycle.

- $\sum_{i} e_{i}$ cycle that visits all vertices.
- $e_{i}$ in multi-cycle $C_{i}$, with $w(C_{i}) = 0$.
- $C_{i} = e_{i} + P_{i} + C'_{i}$.
- Follow the $e_{i}$, then the $P_{i}$ and, on the way, plug the $C'_{i}$. 
Karp, Miller, and Winograd’s decomposition

Boolean \text{KMW}(G):

- Build $G'$ the subgraph of zero-weight multicycles of $G$.
- Compute $G'_1, \ldots, G'_s$, the $s$ SCCs of $G'$.
  - If $s = 0$, $G'$ is empty, return TRUE.
  - If $s = 1$, $G'$ is strongly connected, return FALSE.
  - Otherwise return $\wedge_i \text{KMW}(G'_i)$ (logical AND).

Then, $G$ is computable iff \text{KMW}(G) returns TRUE.
Karp, Miller, and Winograd’s decomposition

Boolean $\text{KMW}(G)$:
- Build $G'$ the subgraph of zero-weight multicycles of $G$.
- Compute $G'_1, \ldots, G'_s$, the $s$ SCCs of $G'$.
  - If $s = 0$, $G'$ is empty, return TRUE.
  - If $s = 1$, $G'$ is strongly connected, return FALSE.
  - Otherwise return $\land_i \text{KMW}(G'_i)$ (logical AND).

Then, $G$ is computable iff $\text{KMW}(G)$ returns TRUE.

**Depth $d$ of the decomposition**
$d = 0$ if $G$ is acyclic, $d = 1$ if all SCCs have an empty $G'$, etc.

**Theorem 1 (Depth of the decomposition)**

If $G$ is computable, $d \leq n$, otherwise, $d \leq n + 1$.

($n$ is the dimension of the problem, i.e., the dimension of $\mathcal{P}$.)
Length of longest dependence path in the EDG

**Theorem 2 (Longest dependence path)**

*If $\mathcal{P}$ contains a $n$-dimensional cube of size $\Omega(N)$, there exists a dependence path of length $\Omega(N^d)$.***

Subtlety: needs to make sure that the path stays in the EDG.
But how to compute $G'$? Primal and dual programs.

$e \in G'$ iff $v_e = 0$ in any optimal solution of the linear program:

$$\min \left\{ \sum_e v_e \mid q \geq 0, \; v \geq 0, \; q + v \geq 1, \; Cq = 0, \; Wq = 0 \right\}$$

A single (rational) linear program.
But how to compute $G'$? Primal and dual programs.

e \in G' \text{ iff } v_e = 0 \text{ in any optimal solution of the linear program:}

$$\min \left\{ \sum_e v_e \mid q \geq 0, \ v \geq 0, \ q + v \geq 1, \ Cq = 0, \ Wq = 0 \right\}$$

A single (rational) linear program.

Always interesting to take a look at the dual program:

$$\max \left\{ \sum_e z_e \mid 0 \leq z \leq 1, \ X.w(e) + \rho_v - \rho_u \geq z_e, \ \forall e = (u, v) \in E \right\}$$

Additional property, for any optimal solution:

- $e \in G' \iff X.w(e) + \rho_v - \rho_u = 0$.
- $e \notin G' \iff X.w(e) + \rho_v - \rho_u \geq 1.$
But how to compute $G'$? Primal and dual programs.

$e \in G'$ iff $v_e = 0$ in any optimal solution of the linear program:

$$\min \left\{ \sum_e v_e \mid q \geq 0, \; v \geq 0, \; q + v \geq 1, \; Cq = 0, \; Wq = 0 \right\}$$

A single (rational) linear program.

Always interesting to take a look at the dual program:

$$\max \left\{ \sum_e z_e \mid 0 \leq z \leq 1, \; X.w(e) + \rho_v - \rho_u \geq z_e, \; \forall e = (u, v) \in E \right\}$$

Additional property, for any optimal solution:

- $e \in G' \iff X.w(e) + \rho_v - \rho_u = 0$.  
- $e \notin G' \iff X.w(e) + \rho_v - \rho_u \geq 1$.

Schedule $\sigma : V \times P \to \mathbb{N}$, with $\sigma(u, p) = X.p + \rho_u$, is valid if:

$$\sigma(v, p) \geq \sigma(u, p - w(e)) + 1$$

$\iff X.p + \rho_v \geq X.(p - w(e)) + \rho_u + 1$

$\iff X.w(e) + \rho_v - \rho_u \geq 1$.
Scheduling: dual of computability.

- \( e \in G' \iff X.w(e) + \rho_v - \rho_u = 0. \)
- \( e \notin G' \iff X.w(e) + \rho_v - \rho_u \geq 1. \)

Multi-dimensional scheduling: hours, minutes, seconds, etc.
- \( e \notin G' \): \( u \) & \( v \) computed at different hours.
  Different iterations of the outer loop = loop-carried.
- \( e \in G' \): \( u \) & \( v \) same hour, constraints pushed to inner dimensions.
  Same iteration of outer loop = loop-independent.

Special form of schedule: affine, same linear part in a SCC of \( G' \).
Scheduling: dual of computability.

- \( e \in G' \iff X \cdot w(e) + \rho_v - \rho_u = 0 \).
- \( e \notin G' \iff X \cdot w(e) + \rho_v - \rho_u \geq 1 \).

Multi-dimensional scheduling: hours, minutes, seconds, etc.

- \( e \notin G' \): \( u \) & \( v \) computed at different hours.
  Different iterations of the outer loop = loop-carried.
- \( e \in G' \): \( u \) & \( v \) same hour, constraints pushed to inner dimensions.
  Same iteration of outer loop = loop-independent.

Special form of schedule: affine, same linear part in a SCC of \( G' \).

\[
\begin{align*}
X_1(0,1) &= 0 \\
X_1(1,1) &\geq 2
\end{align*}
\] 
\[
\Rightarrow \left\{ \begin{array}{l}
X_1 = (2,0), \quad \rho_a = 1 \\
\rho_b = 0, \quad \rho_c = 1
\end{array} \right.
\]

Final schedule
\[
\begin{align*}
\sigma_a(i,j) &= (2i + 1, 2j) \\
\sigma_b(i,j) &= (2i, -j) \\
\sigma_c(i,j) &= (2i + 1, 2j + 1)
\end{align*}
\]
Performances of schedules for computable equations

Theorem 3 (Optimality of multi-dimensional schedules)

*If \( P \) contains a \( n \)-dim. cube of size \( \Theta(N) \), there is a dependence path of length \( \Omega(N^d) \) and a schedule of latency \( O(N^d) \).*

Theorem 4 (Case of one-dimensional schedules)

*If \( d = 1 \), the best affine schedule is \( \sim \lambda N \), for some \( \lambda > 0 \), and so is the maximal dependence length.*

Theorem 5 (Case of a single equation)

*For one equation, \( d = 0 \) or \( d = 1 \). Moreover, if \( d = 1 \), the best linear schedule is optimal up to a constant.*

Theorem 6 (Link with tiling)

*The maximal number of permutable loops is linked to the dimension of the vector space \( \text{Vect}\{w(C) \mid C \text{ cycle of } G'\} \).*
1. The polyhedral model

2. Scheduling, SURES, and approximated loops
   - System of uniform recurrence equations
   - Multi-dimensional scheduling and parallel loop detection
   - Multi-dimensional ranking and worst-case execution time

3. Data mapping & communication optimizations
Loop terminology

Fortran DO loops:

```
DO i=1, N
  DO j=1, N
    a(i,j) = c(i,j-1)
    c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```

- Nested loops, static control.
- Iteration domain and vector.
- Sequential order \( \leq seq \).
- Dependences:
  - R/W, W/R, W/R.

\[
S(I) <_{seq} T(J) \iff (I|_d <_{lex} J|_d) \text{ or } (I|_d = J|_d \text{ and } S <_{txt} J)
\]

- EDG: dependence graph between operations \( S(I) \Rightarrow T(J) \).
- RDG: dependence graph between statements \( S \rightarrow T \).
- ADG: over-approximation, if \( S(I) \Rightarrow T(J) \), then \( S \rightarrow T \).
Representation of dependences

- **Pair set** (exact dependences): \( R_{S,T} = \{(I, J) \mid S(I) \Rightarrow T(J)\} \), in particular affine dependence \( I = f(J) \) if possible.

- **Distance set**: \( E_{S,T} = \{(J - I) \mid S(I) \Rightarrow T(J)\} \).

- **Over-approximations** \( E'_{S,T} \) such that \( E_{S,T} \subseteq E'_{S,T} \).

**Distance set:** 
\[
E = \left\{ \begin{pmatrix} i - j \\ j - i \end{pmatrix} \mid i - j \geq 1, \ 1 \leq i, j \leq N \right\}
\]

**Polyhedral approximation:** 
\[
E' = \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ -1 \end{pmatrix} \mid \lambda \geq 0 \right\}
\]

**Direction vectors:** 
\[
E' = \begin{pmatrix} + \\ - \end{pmatrix} = \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \mu \begin{pmatrix} 0 \\ -1 \end{pmatrix} \mid \lambda, \mu \geq 0 \right\}
\]

**Level:** 
\[
E' = \begin{pmatrix} + \\ * \end{pmatrix} = \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \mu \begin{pmatrix} 0 \\ 1 \end{pmatrix} \mid \lambda \geq 0 \right\}
\]
Uniformization of dependences: example

\[\begin{align*}
\text{DO } & i=1, N \\
\text{DO } & j=1, N \\
& a(i,j) = c(i,j-1) \\
& c(i,j) = a(i,j) + a(i-1,N)
\end{align*}\]

\[\text{Dep. distance } (1, j - N).\]
Uniformization of dependences: example

\[
\begin{align*}
&\text{DO } i=1, N \\
&\quad \text{DO } j=1, N \\
&\quad \quad a(i,j) = c(i,j-1) \\
&\quad \quad c(i,j) = a(i,j) + a(i-1,N) \\
&\quad \text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]

\[
a(i,j) \Rightarrow a(i-1,N) \\
\text{Dep. distance } (1, j-N).
\]

Direction vector \((1, 0−) = (1, 0) + k(0, -1), k \geq 0.\)

Also \(X.(1, 0−) \geq 1 \Rightarrow X.(1, 0) \geq 1 \text{ and } X.(0, -1) \geq 0.\)  

\(\text{SURE!}\)

No parallelism \((d = 2).\) Code appears (here it is) purely sequential.
Emulation of dependence polyhedra

For a (self) dependence polyhedron $\mathcal{P}$, with vertex $v$ and ray $r$:

$$\forall p \in \mathcal{P} \quad X.p \geq 1 \iff \forall \lambda \geq 0 \quad X.(v + \lambda r) \geq 1 \iff X.v \geq 1 \text{ and } X.r \geq 0$$

- Emulate vertices, rays, and lines.

Example with direction vectors:

```fortran
DO i = 1, N
  DO j = 1, N
    DO k = 1, j
      a(i,j,k) = c(i,j,k-1) + 1
      b(i,j,k) = a(i-1,j+i,k) + b(i,j-1,k)
      c(i,j,k+1) = c(i,j,k) + b(i,j-1,k+i) + a(i,j-k,k+1)
    ENDDO
  ENDDO
ENDDO
```

```
1 0 1
0 0 2
-1 1 0
```

```
S1 0 0 0
S2 0 0 1
S3 0 2 0
```
Second example: dependence graphs

Initial RDG.

Uniformized RDG.
Second example: $G$ and $G'$

Uniformized RDG.


$(2i, j)$ for $S_2$, $(2i + 1, 2k)$ for $S_1$, and $(2i + 1, 2k + 3)$ for $S_3$. 
Second example: parallel code generation

DOSEQ i=1, n
   DOSEQ j=1, n /* scheduling (2i, j) */
      DOPAR k=1, j
         b(i,j,k) = a(i-1,j+i,k) + b(i,j-1,k)
      ENDDOPAR
   ENDDOSEQ
DOSEQ k = 1, n+1
   IF (k ≤ n) THEN /* scheduling (2i+1, 2k) */
      DOPAR j=k, n
         a(i,j,k) = c(i,j,k-1) + 1
      ENDDOPAR
   IF (k ≥ 2) THEN /* scheduling (2i+1, 2k+3) */
      DOPAR j=k-1, n
         c(i,j,k) = c(i,j,k-1) + b(i,j-1,k+i-1) + a(i,j-k+1,k)
      ENDDOPAR
ENDDOSEQ
ENDDOSEQ
Loop parallelization: optimality w.r.t. dep. abstraction

- **Lamport (1974)**: hyperplane method = skew + interchange.
- **Allen-Kennedy (1987)**: loop distribution, optimal for levels.
- **Wolf-Lam (1991)**: unimodular, optimal for direction vectors and one statement. Based on finding permutable loops.
- **Darte-Vivien (1997)**: unimodular + shifting + distribution, optimal for polyhedral abstraction and perfectly nested loops. Finds permutable loops, too.
- **Feautrier (1992)**: general affine scheduling, complete for affine dependences and affine transformations, but not optimal.
- **Lim-Lam (1998)**: extension to coarse-grain parallelism, vague.
- **Ramanujam-Sadayappan (2009)**: second (more sound) extension to permutable loops, with locality optimization.
Outline

1. The polyhedral model

2. Scheduling, SURES, and approximated loops
   - System of uniform recurrence equations
   - Multi-dimensional scheduling and parallel loop detection
   - Multi-dimensional ranking and worst-case execution time

3. Data mapping & communication optimizations
Yet another application of SUREs: understand “iterations”

Fortran DO loops:

```fortran
DO i=1, N
    DO j=1, N
        a(i,j) = c(i,j-1)
        c(i,j) = a(i,j) + a(i-1,N)
    ENDDO
ENDDO
```

C for and while loops:

```c
y = 0; x = 0;
while (x <= N && y <= N) {
    if (?) {
        x=x+1;
        while (y >= 0 && ?) y=y-1;
    }
    y=y+1;
}
```

Uniform recurrence equations:

\[ \forall p \in \{ p = (i,j) \mid 1 \leq i,j \leq N \} \]

\[
\begin{align*}
    a(i,j) &= c(i,j - 1) \\
    b(i,j) &= a(i-1,j) + b(i,j + 1) \\
    c(i,j) &= a(i,j) + b(i,j)
\end{align*}
\]
Example of GCD of 2 polynomials

```c
// expression expr, array A, r>0 integer.
da = 2r; db = 2r;
while (da >= r) {
    cond = (da >= db || A[expr] == 0);
    if (!cond) {
        tmp = db; db = da; da = tmp - 1;
    } else da = da - 1;
}
```

Hard to optimize for HLS tools:
- No loop unrolling possible.
- Limited software pipelining.
- No nested-loops optimization.
- No information for coarse-grain scheduling/pipelining.

Need to bound the number of iterations. When feasible, proves program termination as by-product.
Phase 1: build an integer interpreted automaton

Identify relevant variables:
- vector $\vec{x} \in \mathbb{Z}^n, n =$ problem dimension.

Build RDG:
- control-flow graph and conditional transitions.
- express evolution of $\vec{x}$ with affine relations, a bit more general than affine dependences.

Refine automaton (if desired):
- analysis of Booleans: better accuracy, higher complexity.
- simple-path compression: reduces complexity.
- multiple-paths summary: better accuracy, impacts complexity.

Sequential automaton similar to affine recurrence equations, with a different semantics: different relations express non-determinism.
Phase 2: abstract interpretation to get “invariants”

Explicit dependences and schedule, but implicit iteration domains!

Here, we need to prove \( db \geq r \). Use abstract interpretation.

```c
// expression expr, array A,
// r > 0 integer.
da = 2r; db = 2r;
while (da >= r) {
    cond = (da >= db || A[expr] == 0);
    if (!cond) {
        tmp = db; db = da;
da = tmp - 1;
    } else da = da - 1;
}
```

- **Invariant** = integer points in a polyhedron \( \mathcal{P}_k \): conservative approximation of reachable values for each control point \( k \).
- Possibly infinite, parameterized by program inputs.
Phase 3: ranking function to prove termination

Ranking function  Mapping $\sigma : \mathcal{K} \times \mathbb{Z}^n \rightarrow (\mathcal{W}, \preceq)$, decreasing on each transition, where $(\mathcal{W}, \preceq)$ is a well-founded set.

Multi-dimensional rankings  $\mathcal{W} = \mathbb{N}^p$ with lexicographic order.

Affine ranking  $\sigma(k, \vec{x}) = A_k \vec{x} + \vec{b}_k$  $\Rightarrow$  Farkas lemma.

Similar to multi-dimensional scheduling for loops, except:

- Higher dimension $n$ (number of relevant variables).
- Flow not always lexicographically positive  $\Rightarrow$  recurrence equations.
- Hidden “counters” (number $p$ of dimension of the ranking).
Phase 3: ranking function to prove termination

**Ranking function** Mapping $\sigma : K \times \mathbb{Z}^n \rightarrow (\mathcal{W}, \preceq)$, decreasing on each transition, where $(\mathcal{W}, \preceq)$ is a well-founded set.

**Multi-dimensional rankings** $\mathcal{W} = \mathbb{N}^p$ with lexicographic order.

**Affine ranking** $\sigma(k, \vec{x}) = A_k \vec{x} + b_k$ $\implies$ Farkas lemma.

- Similar to multi-dimensional scheduling for loops, except:
  - **Higher dimension** $n$ (number of relevant variables).
  - Flow not always lexico-positive $\implies$ recurrence equations.
  - Hidden “counters” (number $p$ of dimension of the ranking).
Phase 4: bound on the number of program steps

Worst-case computational complexity (WCCC): maximum number of transitions fired by the automaton:

\[ WCCC \leq \# \bigcup \sigma(k, P_k) \leq \sum_k \# \sigma(k, P_k) \]

Counting points in (images of) polyhedra: Ehrhart polynomials, projections, Smith form, union of polyhedra, etc.

\[ WCCC \leq \# \sigma(\text{init}, P_{\text{init}}) + \# \sigma(\text{loop}, P_{\text{loop}}) + \# \sigma(\text{end}, P_{\text{end}}) \]

\[ = 2 + \# \{(1, i) \mid 1 \leq i \leq 2r + 2\} = 2r + 4 \]
Greedy algorithm

1. \( i = 0 \); \( T = \mathcal{T} \), set of all transitions.
2. While \( T \) is not empty do
   1. Find a 1D affine function \((X, \rho_S)\), not increasing for any transitions, and decreasing for as many transitions as possible.
   2. Let \( \sigma_i = X \); \( i = i + 1 \);
   3. If no transition is decreasing, return \text{FALSE}.
   4. Remove from \( T \) all decreasing transitions.
3. \( d = i \), return \text{TRUE}.

Theorem 7 (Completeness of greedy algorithm w.r.t. invariants)

If an affine interpreted automaton, with associated invariants, has a multi-dimensional affine ranking function, then the greedy algorithm generates one such ranking. Moreover, the dimension of the generated ranking is minimal.
Yet another example

```c
y = 0;
x = m;
while(x>=0 && y>=0){
    if(indet()){  
        while(y <= m && indet())
            y++;
        x--;
    }
    y--;
}
```

<table>
<thead>
<tr>
<th>start</th>
<th>$m \geq 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_{bl4}$</td>
<td>$m \geq x &gt; 0$, $m \geq y &gt; 0$</td>
</tr>
<tr>
<td>$l_{bl5}$</td>
<td>$m \geq x \geq 0$, $m \geq y \geq 0$</td>
</tr>
<tr>
<td>$l_{bl6}$</td>
<td>$m \geq x \geq 0$, $m + 1 \geq y \geq 0$</td>
</tr>
<tr>
<td>$l_{bl10}$</td>
<td>$m \geq x \geq -1$, $m + 1 \geq y \geq 0$</td>
</tr>
</tbody>
</table>

\[
WCCC = 5 + 7m + 4m^2
\]
Link with Karp, Miller, Winograd’s decomposition


Gulwani (2009) very different but similar theoretical power.

- Iteration domains \(\Leftrightarrow\) Invariants.
- Loop counters \(\Leftrightarrow\) Integer variables involved in the control.
- Dependences: partial order \(\Leftrightarrow\) Evolution of variables.
- Scheduling functions \(\Leftrightarrow\) Ranking functions.
- Latency \(\Leftrightarrow\) Worst-case execution time (ideal).
- Parallelism \(\Leftrightarrow\) Non determinism.
- In both cases, algorithm depth = measure of sequentiality.
Outline

1. The polyhedral model
2. Scheduling, SURES, and approximated loops
3. Data mapping & communication optimizations
   - Lattice-based memory reduction
   - Communication optimizations for remote data
   - Conclusion
Example of intermediate buffer: DCT-like example

Two synchronized, pipelined (ASAP) processes, communicating through a shared buffer $A$.

$$\text{DO } b_r = 0, 63$$
$$\quad \text{DO } b_c = 0, 63$$
$$\quad \text{DO } r = 0, 7$$
$$\quad S: A(b_r, b_c, r, *) = \ldots$$
$$\quad \text{ENDDO}$$
$$\quad \text{ENDDO}$$
$$\quad \text{ENDDO}$$

$$\text{DO } b_r = 0, 63$$
$$\quad \text{DO } b_c = 0, 63$$
$$\quad \text{DO } c = 0, 7$$
$$\quad T: \ldots = A(b_r, b_c, *, c)$$
$$\quad \text{ENDDO}$$
$$\quad \text{ENDDO}$$
$$\quad \text{ENDDO}$$

Full array (no reuse) $64 \times 64 \times 8 \times 8 = 2^{18} = 256K$.

“Intuitive solution” write in $A(b_r \mod 2, b_c \mod 2, r, c)$ (4 blocks)
Example of intermediate buffer: DCT-like example

Two synchronized, pipelined (ASAP) processes, communicating through a shared buffer $A$.

\[
\begin{align*}
&\text{DO } b_r = 0, 63 \\
&\quad \text{DO } b_c = 0, 63 \\
&\quad \text{DO } r = 0, 7 \\
&\quad \quad \text{S: } A(b_r, b_c, r, \ast) = \ldots \\
&\quad \text{ENDDO} \\
&\text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
&\text{DO } b_r = 0, 63 \\
&\quad \text{DO } b_c = 0, 63 \\
&\quad \text{DO } c = 0, 7 \\
&\quad \quad \text{T: } \ldots = A(b_r, b_c, \ast, c) \\
&\quad \text{ENDDO} \\
&\text{ENDDO} \\
&\text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]

Full array (no reuse) $64 \times 64 \times 8 \times 8 = 2^{18} = 256K$.

“Intuitive solution” write in $A(b_r \ mod \ 2, b_c \ mod \ 2, r, c)$ (4 blocks)

Best linear allocation 112 with $\sigma = \begin{cases} r \ mod \ 4 \\ 16(b_r + b_c) + 2r + c \ mod \ 28 \end{cases}$
Memory reuse for scheduled programs

Given

- An array $A$ with multiple reads and writes.
- Scheduled program or communicating processes, thanks to $\theta$.

Goal

- reduction of the allocation size (size of buffer);
- simplicity of the addressing functions.

Solutions

- Optimal size with Ehrhart counting \(\approx\) approximations?
- Approximation of maximal number of live values \(\approx\) mapping?
- Modular mapping \(i \mapsto A\bar{i} \mod b\) \(\approx\) simple and quite efficient.
Modular mapping and admissible lattice

**Definition 1 (Modular mapping)**

A modular mapping \((M, \vec{b})\), with \(M \in M_{p,n}(\mathbb{Z})\) and \(\vec{b} \in \mathbb{N}^p\), maps index \(\vec{i}\) to \(\sigma(\vec{i}) = M\vec{i} \mod \vec{b}\) in \(p\)-dimensional array with shape \(\vec{b}\).

**Definition 2 (Lifetime analysis)**

Two indices \(\vec{i}\) and \(\vec{j}\) of \(\mathbb{Z}^n\) are conflicting \((\vec{i} \bowtie \vec{j})\) if they correspond to two simultaneously live values in the schedule \(\theta\).

Define \(DS = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}\). ✪ Can be over-approximated.
**Definition 1 (Modular mapping)**

A *modular mapping* \((M, \vec{b})\), with \(M \in \mathcal{M}_{p,n}(\mathbb{Z})\) and \(\vec{b} \in \mathbb{N}^p\), maps index \(\vec{i}\) to \(\sigma(\vec{i}) = M\vec{i} \mod \vec{b}\) in \(p\)-dimensional array with shape \(\vec{b}\).

**Definition 2 (Lifetime analysis)**

Two indices \(\vec{i}\) and \(\vec{j}\) of \(\mathbb{Z}^n\) are *conflicting* \((\vec{i} \bowtie \vec{j})\) if they correspond to two simultaneously live values in the schedule \(\theta\).

Define \(DS = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}\). ✪ Can be over-approximated.

**Lemma 6**

The modular mapping \(\sigma = (M, \vec{b})\) is *valid* iff \(DS \cap \ker \sigma = \{0\}\)

✪ \(\ker \sigma\) admissible lattice for \(DS\).
Critical and admissible lattices

<table>
<thead>
<tr>
<th>Integer points</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Critical and admissible lattices

0−Symmetric Polytope: vertices (8,1), (−8,−1), (−1,5), and (1,−5)
Critical and admissible lattices

Lattice: Basis (7,0), (0,5)

Determinant: 35

(i mod 7, j mod 5)

Second minimum = 9/41 > 1/5
First minimum = 6/41 > 1/7
Critical and admissible lattices

Lattice: Basis (9,0), (0,6)
Determinant: 54
(i mod 9, j mod 6)
Critical and admissible lattices

Lattice: Basis (9,0), (0,5)  Determinant: 45  (i mod 9, j mod 5)
Critical and admissible lattices

Lattice: \( (8,0), (6,6) \)  

Determinant: 48  

\( (i-j \mod 8, j \mod 6) \)
Critical and admissible lattices

Lattice: Basis (8,0), (4,4)  Determinant: 32  (i−j mod 8, j mod 4)
Critical and admissible lattices

Lattice: Basis (8,0), (3,4)  
Determinant: 32  
4i−3j mod 32
Critical and admissible lattices

Lattice: Basis (7,0), (4,4)  
Determinant: 28  
(i−j mod 7, j mod 4)
Critical and admissible lattices

Critical Lattice: Basis (4,3), (8,0)  
Determinant: 24  
3i–4j mod 24
Lattice-based memory allocation: process

1. **Lifetime analysis** of the array elements of $A$, w.r.t. $\theta$.

2. **Relation $\bowtie$**: Build the polytope of conflicting differences.

3. **Admissible lattice**: Build an admissible $\Lambda$ of small determinant.

4. **Modulo function**: Compute $\sigma = (M, \vec{b})$ such that $\ker \sigma = \Lambda$.

5. **Code generation**: Define new array $A'$ and replace each occurrence of $A(i)$ with $A'(Mi \mod \vec{b})$.

✍ Not a perfect scheme, does not reach minimal size, but: robust, expressed in terms of $\theta$, usable with approximations.
Outline

1. The polyhedral model
2. Scheduling, SURES, and approximated loops
3. Data mapping & communication optimizations
   - Lattice-based memory reduction
   - Communication optimizations for remote data
   - Conclusion
Optimize DDR accesses for bandwidth-bound accelerators.

- Use tiling for **data reuse** and to enable **burst communication**.
- Use fine-grain software pipelining to **pipeline DDR requests**.
- Use double buffering to **hide DDR latencies**.
- Use coarse-grain software pipelining to **hide computations**.
Source-to-source communication optimizations for HLS

Optimize DDR accesses for bandwidth-bound accelerators.

- Use tiling for **data reuse** and to enable **burst communication**.
- Use fine-grain software pipelining to **pipeline DDR requests**.
- Use double buffering to **hide DDR latencies**.
- Use coarse-grain software pipelining to **hide computations**.
Overview of the method (for C2H Altera HLS tool)

Derive automatically C2H-compliant C functions for the pipelined accelerators: **load, store, and compute**. Blocks are obtained by loop tiling, pipelined in a “double-buffering” scheme.

1. **Communication coalescing**: prefetches data out of tile, following rows, and exploits data reuse.
   - Array access analysis.
   - Tiling + software pipelining = schedule $\theta$.

2. **Local memory management**: defines memory elements, reduces size, and computes access functions.
   - Lifetime analysis w.r.t. $\theta$.
   - Lattice-based memory reduction of intermediate buffers.

3. **Code generation**: generates final C code in a linearized form while optimizing accesses to the DDR.
   - Placement of FIFO synchronizations.
   - Boulet-Feautrier’s method for polytope scanning.
Formalization of valid, exact, and approximated load

Valid load

(i) Load at least what is needed but not previously produced:
\[ \cup_{t \leq T} \{ \text{In}(t) \setminus \text{Out}(t' < t) \} \subseteq \text{Load}(t \leq T) \]

(ii) Do not overwrite locally-defined data:
\[ \text{Out}(t < T) \cap \text{Load}(T) = \emptyset \]
Formalization of valid, **exact**, and approximated load

**Exact load**

(i) Load *exactly* what is needed but not previously produced:

\[
\forall T, \bigcup_{t \leq T} \{ \text{In}(t) \setminus \text{Out}(t' < t) \} = \text{Load}(t \leq T)
\]

(ii) All loads should be disjoint (no redundant transfers):

\[
\text{Load}(T) \cap \text{Load}(T') = \emptyset, \forall T \neq T'
\]
Formalization of valid, exact, and approximated load

(i) Load at least the exact amount of data:
\[ \bigcup_{t \leq T} \{ \overline{\text{In}}(t) \setminus \text{Out}(t' < t) \} \subseteq \text{Load}(t \leq T) \]

(ii) Do not overwrite possibly locally-defined data:
\[ \overline{\text{Out}}(t < T) \cap \text{Load}(T) = \emptyset \]
Formalization of valid, exact, and approximated load

Valid approximated load

(i) Load at least the exact amount of data:
\[ \bigcup_{t \leq T} \{ \overline{\text{In}}(t) \setminus \text{Out}(t' < t) \} \subseteq \text{Load}(t \leq T) \]

(ii) Do not overwrite possibly locally-defined data:
\[ \overline{\text{Out}}(t < T) \cap \text{Load}(T) = \emptyset \]
Handling approximations of data accesses

Exact situation:
\[
\begin{align*}
\text{Load}(T) &= \text{In}(T) \setminus \{\text{In}(t < T) \cup \text{Out}(t < T)\} \\
&= \text{FirstReadBeforeWrite} \cap T \\
\text{Store}(T) &= \text{Out}(T) \setminus \text{Out}(t > T) = \text{LastWrite} \cap T
\end{align*}
\]

Approximated situation:
\[
\begin{align*}
\text{Load}(T) &= \overline{\text{In}}(T) \setminus \{\overline{\text{In}}(t < T) \cup \overline{\text{Out}}(t < T)\} \\
\text{Store}(T) &= \overline{\text{Out}}(T) \setminus \overline{\text{Out}}(t > T)
\end{align*}
\]

Theorem 8 (Valid approximated load and store operators)

The previous load and store operators are valid, for any tile \( T \):

(i) \( \overline{\text{Out}}(T) \subseteq \overline{\text{In}}(t \leq T) \cup \overline{\text{Out}}(t > T) \cup \text{Out}(t \leq T) \).

(ii) \( \overline{\text{In}}(T) \cap \{\overline{\text{Out}}(t < T) \setminus \overline{\text{Out}}(t < T)\} \subseteq \overline{\text{In}}(t < T) \).

Possible solution:
\[
\left\{ \begin{array}{l}
\overline{\text{Out}}(T) \setminus \overline{\text{Out}}(T) \subseteq \overline{\text{In}}(T) \\
\overline{\text{In}}(T) = \cup_{t > T} \{\overline{\text{In}}(t) \cap (\overline{\text{Out}}(t' \leq T) \setminus \overline{\text{Out}}(t' < t))\} \cup \overline{\text{In}}(T)
\end{array} \right\}
\]
Outline

1 The polyhedral model

2 Scheduling, SURES, and approximated loops

3 Data mapping & communication optimizations
   - Lattice-based memory reduction
   - Communication optimizations for remote data
   - Conclusion
The polytope model: more than an exact representation

Discuss correctness and optimality with respect to a description.

- Parallelism detection with respect to dependence abstraction.
- More accurate for uniform dependences and Allen & Kennedy.
- Optimality in a class of functions.

Try to not assume that some information is exactly described, i.e., take into account approximations. Think conservative!

- Dependence and lifetime analysis.
- Array references and sets of data.
- Memory mapping and conflicts.
- Iteration domains? If conversion? Non-determinism?

☞ Approximating the control remains a major difficulty.
☞ Incorporate more techniques such as abstract interpretation.