Outline

1. The polyhedral model
2. Systems of uniform recurrence equations
3. Multi-dimensional scheduling and applications
   - Catalog of loop transformations
   - Detection of parallel loops
   - Multi-dimensional ranking and worst-case execution time

Loop distribution and loop fusion

```
DO i=1, N
  a(i) = b(i)
  d(i) = a(i-1)
ENDDO
```

Loop distribution

$\longrightarrow$

$\longleftarrow$

Loop fusion

```
DO i=1, N
  a(i) = b(i)
ENDDO
DO i=1, N
  d(i) = a(i-1)
ENDDO
```

Main consequences

- Loop distribution used to parallelize/vectorize loops.
- Loop fusion increases the granularity of computations.
- Loop fusion reduces loop overhead.
- Loop fusion usually improves spatial & temporal data locality.
- Loop fusion may enable array scalarization.

Loop shifting

```
DO i=1, N
  a(i) = b(i)
  d(i) = a(i-1)
ENDDO
```

Loop shifting

$\longleftrightarrow$

```
DO i=0, N
  IF (i > 0) THEN
    a(i) = b(i)
  IF (i < N) THEN
    d(i+1) = a(i)
ENDDO
```

Main consequences

- Similar to software pipelining.
- Creates prelude/postlude or introduces `if` statements.
- Can be used to align accesses and enable loop fusion.
- Particularly suitable to handle constant dependence distances.

Loop peeling

```
DO i=0, N
  IF (i > 0) THEN
    a(i) = b(i)
  IF (i < N) THEN
    d(i+1) = a(i)
ENDDO
```

Loop peeling

$\longrightarrow$

Loop sinking

$\longleftarrow$

```
d(1) = a(0)
DO i=1, N-1
  a(i) = b(i)
  d(i+1) = a(i)
ENDDO
a(N) = b(N)
```

Mais consequences

- Peeling removes a few iterations to make code simpler.
- Peeling extracts iterations with a specific behavior to enable more transformations.
- Peeling reduces the iteration domain (range of loop counter).
- Sinking is used to make loops perfectly nested.

Partial or total loop unrolling

```
DO i=1, 10
  a(i) = b(i)
  d(i) = a(i-1)
ENDDO
```

Unrolling by 2

$\longrightarrow$

```
DO i=1, 10, 2
  a(i) = b(i)
  d(i) = a(i-1)
  a(i+1) = b(i+1)
  d(i+1) = a(i)
ENDDO
```

Main consequences

- Replicates instructions to improve schedule & resource usage.
- Can be used for array scalarization.
- Increase code size.
- Total loop unrolling flattens the loops and changes structure.

Strip mining, loop coalescing

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

Strip mining $\longrightarrow$

$\longleftarrow$ Loop linearization

```
DO Is=1, N, s
  DO i=Is, min(N, Is+s-1)
    a(i) = b(i) + c(i)
  ENDDO
ENDDO
```

Main consequences

- Strip-mining performs parametric loop unrolling.
- It changes the structure and creates blocks of computations.
- It can be used as a preliminary step for tiling.
- Loop linearization can reduce the control of loops.
- It also reduces the problem dimension.

Loop interchange

Loop interchange: $(i, j) \mapsto (j, i)$.

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

Loop interchange $\longleftrightarrow$

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

Main consequences

- Can enable loop parallelism.
- Basis of loop tiling.
- Changes order of memory accesses and thus data locality.
- Needs bounds computations as in $\sum_{i=1}^{n} \sum_{j=1}^{i} S_{i,j} = \sum_{j=1}^{n} \sum_{i=j}^{n} S_{i,j}$.

Loop skewing, loop reversal, unimodular transformation

Loop skewing: $(i, j) \mapsto (i, j + i)$, loop iterations in the same order.

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

Loop reversal: $i \mapsto -i$, loop executed in opposite order.

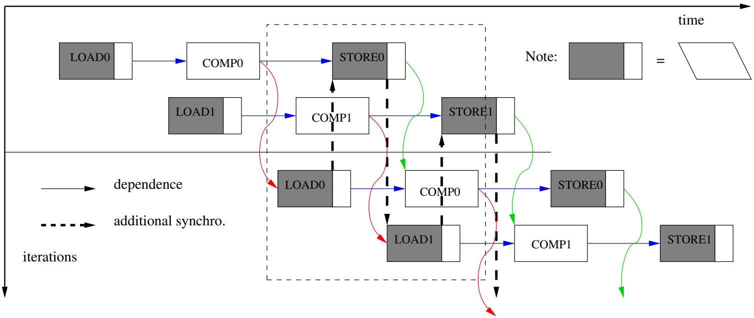
Unimodular = combination of reversal, skewing, interchange.

```
DO i=1, N                  DO t=2, 2N
  DO j=1, N                  DO p=max(1,t-N), min(N,t-1)
    a(i,j) = ...    <-->       a(p,t-p) = ...
  ENDDO                      ENDDO
ENDDO                      ENDDO
```

In practice, need to combine all. Ex: HLS with C2H Altera

Optimize DDR accesses for bandwidth-bound accelerators.

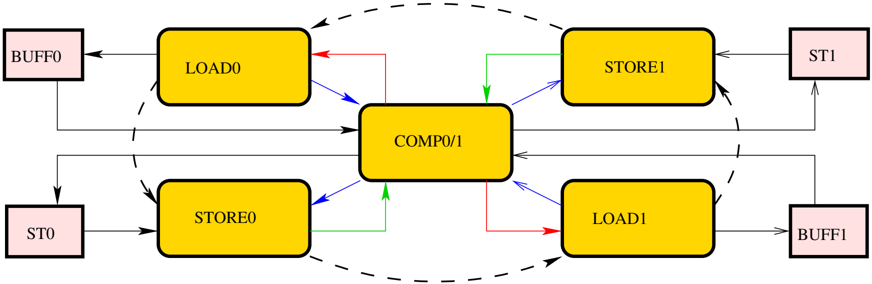
- Use tiling for data reuse and to enable burst communication.
- Use fine-grain software pipelining to pipeline DDR requests.
- Use double buffering to hide DDR latencies.
- Use coarse-grain software pipelining to hide computations.



In practice, need to combine all. Ex: HLS with C2H Altera

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.



Outline

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.

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

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

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 $E_{S,T} \subseteq E'_{S,T}$.

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

Distance set:

$$E = \left\{ \begin{pmatrix} i-j \\ j-i \end{pmatrix} \;\middle|\; i-j \geq 1,\ 1 \leq i,\ j \leq N \right\}$$

Polyhedral approximation:

$$E' = \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ -1 \end{pmatrix} \;\middle|\; \lambda \geq 0 \right\}$$

Direction vectors:

$$E' = \begin{pmatrix} + \\ - \end{pmatrix} = \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \mu \begin{pmatrix} 0 \\ -1 \end{pmatrix} \;\middle|\; \lambda,\ \mu \geq 0 \right\}$$

Level:

$$E' = ① = \begin{pmatrix} + \\ * \end{pmatrix} = \left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \lambda \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \mu \begin{pmatrix} 0 \\ 1 \end{pmatrix} \;\middle|\; \lambda \geq 0 \right\}$$

Uniformization of dependences: example

```
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
```

a(i,j) $\Rightarrow$ a(i-1,N)

Dep. distance $(1, j - N)$.

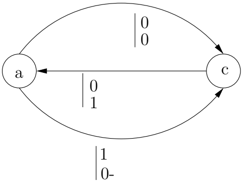
Uniformization of dependences: example

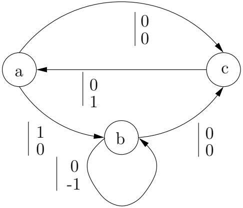
```
DO i=1, N
  DO j=1, N
    a(i,j) = c(i,j-1)
    c(i,j) = a(i,j) + a(i-1,N)
  ENDDO
ENDDO
```

a(i,j) ⇒ a(i-1,N)
Dep. distance $(1, j - N)$.

Direction vector $(1, 0-) = (1, 0) + k(0, -1)$, $k \geq 0$.
Also $X.(1, 0-) \geq 1 \Rightarrow X.(1, 0) \geq 1$ and $X.(0, -1) \geq 0$.

☛ SURE!





No parallelism ($d = 2$). Code appears (here it is) purely sequential.

Emulation of dependence polyhedra

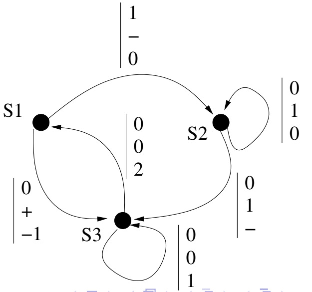
For a (self) dependence polyhedron $\mathcal{P}$, with vertex v and ray r:

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

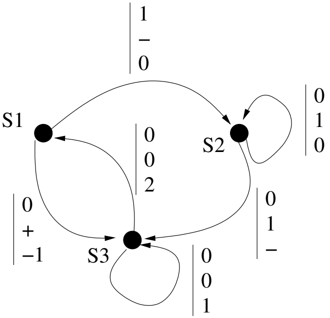
☛ Emulate vertices, rays, and lines.

Example with direction vectors:

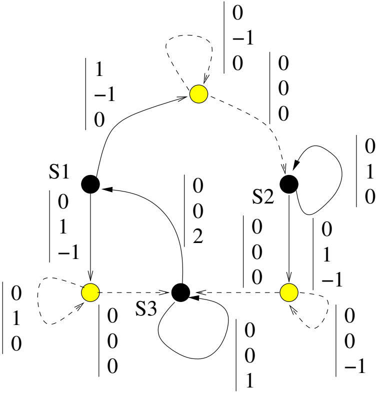
```
DO i= 1, N
  DO j = 1, N
    DO k = 1, j
      a(i,j,k) = c(i,j,k-1) + 1
      b(i,j,k) = a(i-1,j+i,k) + b(i,j-1,k)
      c(i,j,k+1) = c(i,j,k) + b(i,j-1,k+i)
                         + a(i,j-k,k+1)
    ENDDO
  ENDDO
ENDDO
```



Second example: dependence graphs

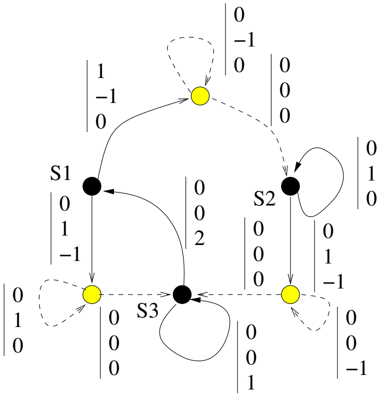


Initial RDG.

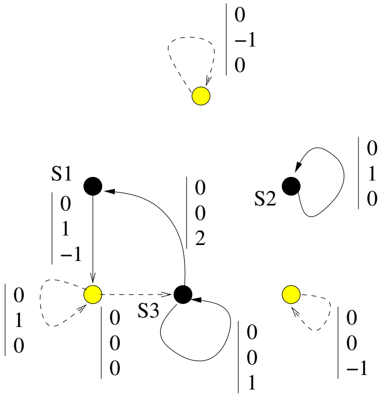


Uniformized RDG.

Second example: G and G'



Uniformized RDG.



G': zero-weight multi-cycles.

$\boxed{(2i, j) \text{ for } S_2,\ (2i+1, 2k) \text{ for } S_1, \text{ and } (2i+1, 2k+3) \text{ for } S_3.}$

Second exemple: 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
```

Allen-(Callahan)-Kennedy (1987): loop distribution

AK(G, k):

- Remove from G all edges of level $< k$.
- Compute $G_1, \dots, G_s$ the s SCCs of G in topological order.
  - If G_i has a single statement S, with no edge, generate DOPAR loops in all remaining dimensions, and generate code for S.
  - Otherwise:
    - Generate DOPAR loops from level k to level $l-1$, and a DOSEQ loop for level l, where l is the minimal level in G_i.
    - call AK(G_i, $l+1$). /* d_S sequential loops for statement S */

➽ Variant of (dual of) KMW with DOPAR as high as possible.

Allen-(Callahan)-Kennedy (1987): loop distribution

AK(G, k):

- Remove from G all edges of level $< k$.
- Compute $G_1, \ldots, G_s$ the s SCCs of G in topological order.
  - If G_i has a single statement S, with no edge, generate DOPAR loops in all remaining dimensions, and generate code for S.
  - Otherwise:
    - Generate DOPAR loops from level k to level $l-1$, and a DOSEQ loop for level l, where l is the minimal level in G_i.
    - call AK(G_i, $l+1$). /* d_S sequential loops for statement S */

➽ Variant of (dual of) KMW with DOPAR as high as possible.

Theorem 1 (Optimality of AK for dependence levels)

Nested loops $\mathcal{L}$, RDG G with levels. One can build nested loops $\mathcal{L}'$, with same structure and same RDG, with bounds parameterized by N such that, for each SCC G_i of G, there is a path in the EDG of $\mathcal{L}'$ that visits each statement S of G_i $\Omega(N^{d_s})$ times.

Darte-Vivien (1997): unimodular + shift + distribution

Boolean DV(G, k) /* G uniformized graph, with virtual and actual nodes */

- Build G' generated by the zero-weight multi-cycles of G.
- Modify slightly G' (technical detail not explained here).
- Choose X (vector) and, for each S in G', ρ_S (scalar) s.t.:

$$\begin{cases} \text{if } e = (u, v) \in G' \text{ or } u \text{ is virtual, } Xw(e) + \rho_v - \rho_u \geq 0 \\ \text{if } e \notin G' \text{ and } u \text{ is actual, } Xw(e) + \rho_v - \rho_u \geq 1 \end{cases}$$

  For each actual node S of G let $\rho_S^k = \rho_S$ and $X_S^k = X$.
- Compute $G'_1, \ldots, G'_s$ the SCC of G' with ≥ 1 actual node:
  - If G' is empty or has only virtual nodes, return TRUE.
  - If G' is strongly connected with ≥ 1 actual node, return FALSE.
  - Otherwise, return $\bigwedge_{i=1}^{s} \text{DV}(G'_i, k+1)$ ($\bigwedge$ = logical AND).

General affine multi-dimensional schedules

Affine dependences (or even relations): (S, I) depends on (T, J) if $(I, J) \in \mathcal{D}_e$ where $e = (T, S)$ and $\mathcal{D}_e$ is a polyhedron.

- Look for schedule σ such that $\sigma(T, J) <_{lex} \sigma(S, I)$ for all $(I, J) \in \mathcal{D}_e$. If σ is affine, use affine form of Farkas lemma.
- Write $\sigma(T, J) + \epsilon_e \leq \sigma(S, I)$ with $\epsilon \geq 0$ and maximize the number of dependence edges e such that $\epsilon_e \geq 1$.
- Remove edges e such that $\epsilon_e \geq 1$ and continue to get remaining dimensions ☛ multi-dimensional affine schedule.

To perform tiling, look for several dimensions (permutable loops) such that $\sigma(S, I) - \sigma(T, J) \geq 0$ instead of $\sigma(S, I) - \sigma(T, J) \geq 1$.

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.
- Bondhugula-Ramanujam-Sadayappan (2008): improved extension to permutable loops, with locality optimization.

Outline

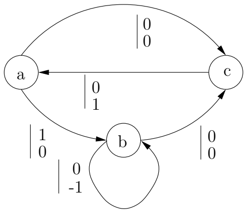
Yet another application of SUREs: understand "iterations"

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
```

C `for` and `while` loops:

```
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:

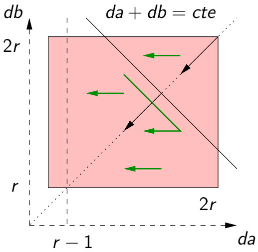
$$\forall p \in \{p = (i,j) \mid 1 \le i,j \le N\}$$

$$\begin{cases} a(i,j) = c(i,j-1) \\ b(i,j) = a(i-1,j) + b(i,j+1) \\ c(i,j) = a(i,j) + b(i,j) \end{cases}$$

Context: transforming WHILE loops into DO loops

Example of GCD of 2 polynomials

```
// 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.

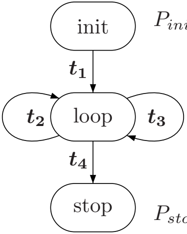
Sequential automaton similar to affine recurrence equations, with a different semantics: different relations express non-determinism.

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.

```
// 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;
}
```



$$P_{init} = 1 \leqslant r$$

$$P_{loop} = \begin{cases} r-1 \leqslant da \leqslant 2r \\ r \leqslant db \leqslant 2r \\ 1 \leqslant r \end{cases}$$

$$P_{stop} = \begin{cases} 1 \leqslant r \\ r \leqslant db \leqslant 2r \\ r-1 \leqslant da < r \end{cases}$$

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

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

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

☛ Similar to multi-dimensional scheduling for loops, except:

- Higher dimension n (number of relevant variables).
- Flow not always lexico-positive ➠ recurrence equations.
- Hidden "counters" (number p of dimension of the ranking).

Phase 3: ranking function to prove termination

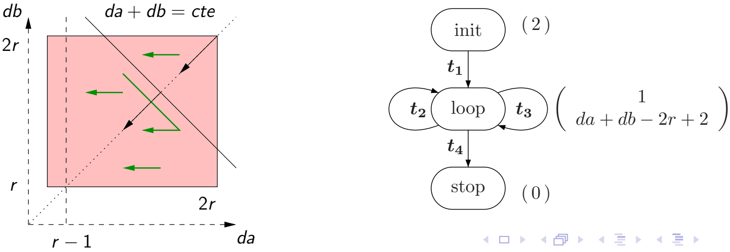
Ranking function Mapping $\sigma : \mathcal{K} \times \mathbb{Z}^n \to (\mathcal{W}, \preceq)$, decreasing on each transition, where $(\mathcal{W}, \preceq)$ is a well-founded set.

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

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

☛ Similar to multi-dimensional scheduling for loops, except:

- Higher dimension n (number of relevant variables).
- Flow not always lexico-positive ➠ recurrence equations.
- Hidden "counters" (number p of dimension of the ranking).



Phase 4: bound on the number of program steps

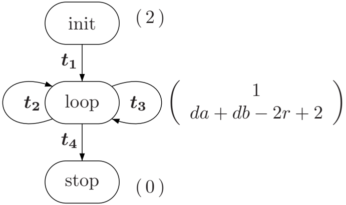
Worst-case computational complexity (WCCC): maximum number of transitions fired by the automaton:

$$WCCC \leq \# \bigcup \sigma(k, \mathcal{P}_k) \leq \sum_k \# \sigma(k, \mathcal{P}_k)$$

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

$$\begin{aligned} WCCC \leq\ & \#\sigma(\text{init}, \mathcal{P}_{\text{init}}) \\ & + \#\sigma(\text{loop}, \mathcal{P}_{\text{loop}}) \\ & + \#\sigma(\text{end}, \mathcal{P}_{\text{end}}) \end{aligned}$$

$$= 2 + \#\{(1, i) \mid 1 \leq i \leq 2r + 2\}$$
$$= 2r + 4$$



Alias-Darte-Feautrier-Gonnord (2010)

Greedy algorithm

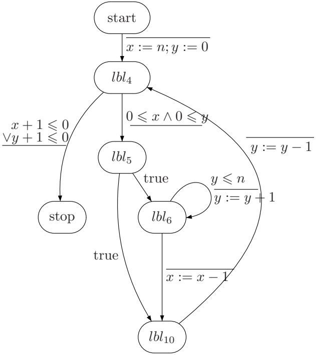
- $i = 0$; $T = \mathcal{T}$, set of all transitions.
- While T is not empty do
  - Find a 1D affine function (X, ρ_S), not increasing for any transitions, and decreasing for as many transitions as possible.
  - Let $\sigma_i = X$; $i = i + 1$;
  - If no transition is decreasing, return FALSE.
  - Remove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

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



start	$m \geq 0$	$2m+4$
lbl_4	$m \geq x > 0, m \geq y > 0$	$(2x+3, 3y+3)$
lbl_5	$m \geq x \geq 0, m \geq y \geq 0$	$(2x+3, 3y+2)$
lbl_6	$m \geq x \geq 0, m+1 \geq y \geq 0$	$(2x+2, m-y+1)$
lbl_{10}	$\begin{cases} m \geq x \geq -1, m+1 \geq y \geq 0 \\ 2m \geq x+y \end{cases}$	$(2x+3, 3y+1)$

$\text{WCCC} = 5 + 7m + 4m^2$

Link with Karp, Miller, Winograd's decomposition

Podelski-Rybalchenko (2004) $\sim$ URE $\sim$ Lamport (1974).
Bradley-Manna-Sipma (2005) $\sim$ Wolf-Lam (1991).
Colón-Sipma (2002) between Wolf-Lam & Darte-Vivien (1997).
Alias-Darte-Feautrier-Gonnord (2010) $\sim$ Feautrier (1992).

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.

Theorem 2 (Farkas' lemma)

Let A be a matrix and b a vector. There exists a vector $x \geq 0$ with $Ax = b$ if and only if $yb \geq 0$ for each row vector y with $yA \geq 0$.

Theorem 3 (Duality)

Provided that both sets are nonempty:

$$\max\{cx \mid Ax \leq b\} = \min\{yb \mid y \geq 0,\ yA = c\}$$

Theorem 4 (Complementary slackness)

If both optima are finite, x_0 and y_0 are optimum solutions if and only if they are feasible and $y_0(b - Ax_0) = 0$.

Theorem 5 (Affine form of Farkas' lemma)

If $Ax \leq b$ is nonempty then $cx \leq \delta$ for all x such that $Ax \leq b$ if and only if there exists $y \geq 0$ such that $c = yA$ and $yb \leq \delta$.