Equivalence Checking of Array-Intensive Programs

C. Karfa, K. Banerjee, D. Sarkar, C. Mandal

Dept of Computer Sc & Engg
IIT Kharagpur

July 5, 2011

Table of Parts I

Part I

Motivations and Objectives

Loop and Arithmetic Transformations

- Loop transformations:
  - Loop splitting, distributions, unrolling, fusion, fission, tiling, etc.
- Arithmetic transformations:
  - Common sub-expression elimination, copy propagation, constant folding, etc.
  - Algebraic transformations like associative, distributive, commutative.

Loop and Arithmetic Transformations

- Loop transformations:
  - Loop splitting, distributions, unrolling, fusion, fission, tiling, etc.
- Arithmetic transformations:
  - Common sub-expression elimination, copy propagation, constant folding, etc.
  - Algebraic transformations like associative, distributive, commutative.

Ruled Based and ADDG Based Approaches

- *Rule based approaches*: (Pnueli et al. 2005)[5], (Menon et al. 2003)[1].
  - *Limitations:*
    - Need the order in which transformations have been applied from the synthesis tool.
    - completeness of the rule set.
- *ADDG based approach:* (Shashidhar et al. 2008)[2].
  - *Restrictions on input programs:*
    - static control-flow,
    - affine indices and bounds,
    - uniform recurrence and
    - single assignment form.
  - *Limitations:*
    - (i) non-uniform recurrence: (Verdoolaege et al. 2009)[3]
    - (ii) data-dependent assignments and accesses (Verdoolaege et al. 2010)[4]
    - (iii) arithmetic transformations.

Ruled Based and ADDG Based Approaches

- *Rule based approaches*: (Pnueli et al. 2005)[5], (Menon et al. 2003)[1].
    - *Limitations:*
        - Need the order in which transformations have been applied from the synthesis tool.
        - completeness of the rule set.
- *ADDG based approach:* (Shashidhar et al. 2008)[2].
    - *Restrictions on input programs:*
        - static control-flow,
        - affine indices and bounds,
        - uniform recurrence and
        - single assignment form.
    - *Limitations:*
        - (i) non-uniform recurrence: (Verdoolaege et al. 2009)[3]
        - (ii) data-dependent assignments and accesses (Verdoolaege et al. 2010)[4]
        - (iii) arithmetic transformations.

Contributions of the Paper

- An equivalence checking method which will be capable of verifying all kinds of loop transformations and also several arithmetic transformations.
- Considers the same class of programs considered by Shashidhar et al. [2] and their ADDG based modelling of programs. The contributions of the present work are:
  - defining the characteristic formula of a slice of ADDGs.
  - redefining the equivalence of ADDGs based on slice level characterization.
  - normalization of arithmetic expressions and some simplification rules

Part II

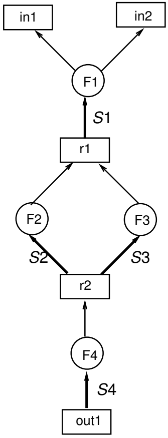
Contributions

ADDG

$for(i = 1; i \leq M; i = i + 1)$
$\quad for(j = 4; j \leq N; j = j + 1)$
$\qquad S1 : \; r1[i + 1][j - 3] = F1(in1[i][j], in2[i][j]);$

$for(l = 3; l \leq M; l = l + 1)\{$
$\quad for(m = 3; m \leq N - 1; m = m + 1)\{$
$\qquad if(l + m \leq 7)$
$\qquad\quad S2 : \; r2[l][m] = F2(r1[l - 1][m - 2]);$
$\qquad$ else
$\qquad\quad S3 : \; r2[l][m] = F3(r1[l][N - 3]);$
$\qquad S4 : \; out1[l][m] = F4(r2[l][m]);$
$\}\}$

(a)



(b)

ADDG (contd.)

$$
\begin{array}{l}
for(i_1 = L_1; i_1 \leq H_1; i_1 += r_1) \\
\quad for(i_2 = L_2; i_2 \leq H_2; i_2 += r_2) \\
\qquad \vdots \\
\qquad\quad for(i_x = L_x; i_x \leq H_x; i_x += r_x) \\
\qquad\qquad if(C_D)\ then \\
\qquad\qquad\quad S:\ d[e_1]\ldots[e_k] = f(u_1[e'_{11}]\ldots[e'_{1l_1}],\ \ldots,\ u_m[e'_{m1}]\ldots[e'_{ml_m}]);
\end{array}
$$

Definition (Iteration domain of the statement S (I_S))

$$I_S = \{[i_1, i_2, \ldots, i_x] \mid \bigwedge_{k=1}^{x} (L_k \leq i_k \leq H_k \wedge C_D \wedge \exists \alpha_k \in \mathbb{N}(i_k = \alpha_k r_k + L_k))\}$$

where $i_k, L_k, H_k, r_k, 1 \leq k \leq x$, are integers.

ADDG (contd.)

Definition (Definition domain (${}_SD_d$))

${}_SD_d \subseteq \mathbb{Z}^k = \{[e_1(\vec{v}), \ \ldots, \ e_k(\vec{v})] \mid \vec{v} \in I_S\}$.

Definition (Definition mapping (${}_SM_d^{(d)}$))

${}_SM_d^{(d)} = I_S \ \rightarrow \ {}_SD_d$ s.t. $\forall \vec{v} \in I_S, \ \vec{v} \mapsto [e_1(\vec{v}), \ \ldots, \ e_k(\vec{v})] \in \ {}_SD_d$.

Definition (Operand domain (${}_SU_{u_n}$))

${}_SU_{u_n} \subseteq \mathbb{Z}^{l_n} = \{[e_{n1}(\vec{v}), \ \ldots, \ e_{nl_n}(\vec{v})] \mid \vec{v} \in I_S\}$

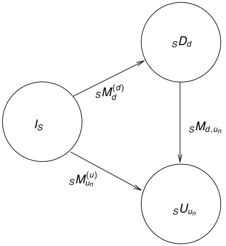
Definition (Operand mapping (${}_SM_{u_n}^{(u)}$))

${}_SM_{u_n}^{(u)} = I_S \rightarrow \ {}_SU_{u_n}$ s.t. $\forall \vec{v} \in I_S, \ \vec{v} \mapsto [e_{n1}(\vec{v}), \ \ldots, \ e_{nl_n}(\vec{v})] \in \ {}_SU_{u_n}$.

Dependence Mapping

Definition (Dependence mapping ($_SM_{d,u_n}$))

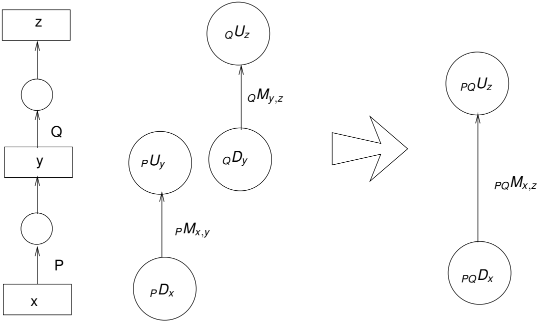
$$_SM_{d,u_n} = \{[i_1, \ldots, i_k] \rightarrow [j_1, \ldots, j_{l_n}] \mid ([i_1, \ldots, i_k] \in \ _SD_d \wedge [j_1, \ldots, j_{l_n}] \in \ _SU_{u_n} \wedge \exists \vec{v} \in I_S \mid ([i_1, \ldots, i_k] = \ _SM_d^{(d)}(\vec{v}) \ \wedge \ [j_1, \ldots, j_{l_n}] = \ _SM_{u_n}^{(u)}(\vec{v})))\}$$



$$_SM_{d,u_n} = (_SM_d^{(d)})^{-1} \diamond \ _SM_{u_n}^{(u)}$$

Figure: Computation of dependence mapping

Transitive Dependence



$$_{PQ}D_x = {}_PM_{x,y}^{-1}({}_QM_{y,z}^{-1}({}_QU_z) \cap {}_PM_{x,y}({}_PD_x))$$
$$_{PQ}U_z = {}_QM_{y,z}[{}_PM_{x,y}({}_PD_x) \cap_Q M_{y,z}^{-1}({}_QU_z)]$$

Figure: Transitive dependence

Example of dependence mappings

Example

$$
\begin{aligned}
&{}_{S1}D_{r1} = \{[i+1, j-3] \mid [i,j] \in I_{S1}\} \\
&{}_{S1}M^{(d)}_{r1} = \{[i,j] \rightarrow [i+1, j-3] \mid [i,j] \in I_{S1}\} \\
&{}_{S1}U_{in1} = \{[i,j] \mid [i,j] \in I_{S1}\} \\
&{}_{S1}M^{(u)}_{in1} = \{[i,j] \rightarrow [i,j] \mid [i,j] \in I_{S1}\} \\
&{}_{S1}M_{r1,in1} = ({}_{S1}M^{(d)}_{r1})^{-1} \diamond {}_{S1}M^{(u)}_{in1} \\
&= \{[i,j] \rightarrow [i-1, j+3] \diamond [i-1, j+3] \rightarrow [i-1, j+3]\} \mid [i,j] \in {}_{S1}D_{r1}\} \\
&= \{[i,j] \rightarrow [i-1, j+3] \mid [i,j] \in {}_{S1}D_{r1}\} \\
&{}_{S4S2}M_{out1,r1} = {}_{S4}M_{out1,r2} \diamond {}_{S2}M_{r2,r1} \\
&= \{[l,m] \rightarrow [l,m] \mid [l,m] \in {}_{S4}D_{out1}\} \\
&\qquad\qquad \diamond \{[l,m] \rightarrow [l-1, m-2] \mid [l,m] \in {}_{S2}D_{r2}\} \\
&= [l,m] \rightarrow [l-1, m-2] \mid [l,m] \in {}_{S4}D_{out1}\}
\end{aligned}
$$

Slices

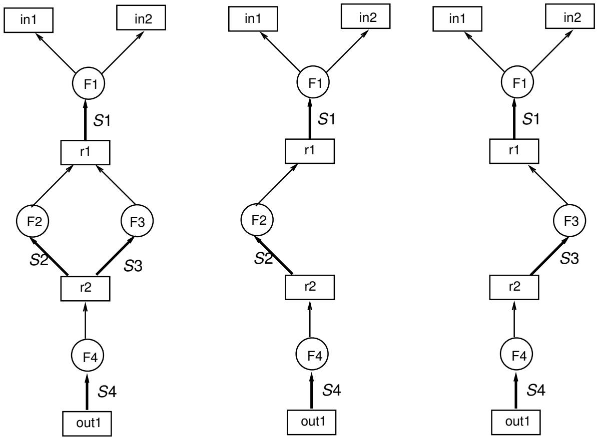
Definition (Slice)

A slice is a connected subgraph of an ADDG which has an array node as its start node (having no edge incident on it), only array nodes as its terminal nodes (having no edge emanating from them), all the outgoing edges (read edges) from each of its operator nodes and at most one outgoing edge (write edge) from each of its array nodes other than the terminal nodes.

Definition (IO-slice)

A component slice is said to be an IO-slice iff its start node is an output array node and all the terminal nodes are input array nodes.

An Example of slice



Characteristic formula of a slice

Definition (Characteristic formula of a slice)

The *characteristic formula* of a slice $g(a, \langle v_1, \ldots, v_n \rangle)$ is given as the tuple $\tau_g = \langle r_g, \langle {}_gM_{a \rightsquigarrow v_1}, \ldots, {}_gM_{a \rightsquigarrow v_n} \rangle \rangle$.

For the slice $g = \langle S4S2S1 \rangle$,

$r_g = out1 \Leftarrow F4(r2)$ [at the node $r2$],

$\Leftarrow F4(F2(r1))$ [at the node $r1$],

$\Leftarrow F4(F2(F1(in1, in2)))$ [at the nodes $in1$ and $in2$]

Definition (Matching IO-slices)

Two IO-slices g_1 and g_2 are said to be matching, denoted as $g_1 \approx g_2$, if the data transformations of both the slices are equivalent.

Equivalence of ADDGs

Definition (IO-slice class)

Maximum set of matching IO-slices of the ADDG.

Definition (IO-slice class equivalence:)

Two slice classes C_1 and C_2 are equivalent, denoted as $C_1 \simeq C_2$, iff (i) r_{C_1} and r_{C_2} are same. (ii) Corresponding dependence mappings in the two classes are same.

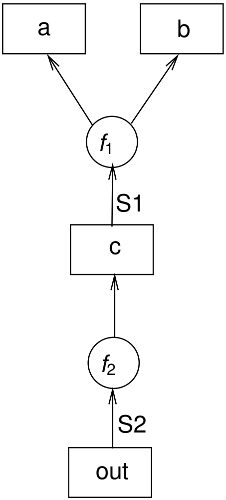
Definition (Equivalence of ADDGs:)

An ADDG G_S is said to be equivalent to an ADDG G_T iff for each IO-slice class C_S in G_S, there exists an IO-slice class C_T in G_T such that $C_S \simeq C_T$, and vice-versa.

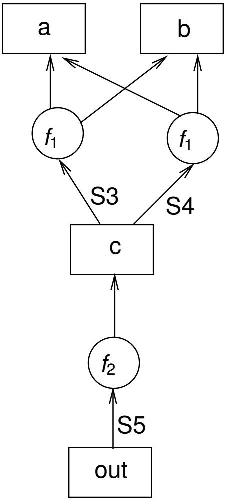
An example of slice classes

```
for(k = 0; k <= 100; k++)
  S1: c[k] = f1(a[2k], b[k+1]);
for(i=0; i<=50; i++)
  for(j=0; j<=50; j++)
    S2: out[i][j] = f2(c[i+j]);
    (a) original program
for(k = 0; k <= 100; k +=2){
  S3: c[k] = f1(a[2k], b[k+1]);
  S4: c[k+1] = f1(a[2k+2], b[k+2]);}
for(i=0; i<=50; i++)
  for(j=0; j<=50; j++)
    S5: out[i][j] = f2(c[i+j]);
    (b) transformed program
```

An example of slice classes (contd.)



(c) ADDG of (a)



(d) ADDG of (b)

Objectives of normalization

- Canonical form does not exist for integer arithmetic,
- we represent the data transformation and the dependence mappings of a slice in normalized form.
- The normalization process reduces many computationally equivalent formulas syntactically identical as it forces all the formulas to follow a uniform structure

Normalized Sum

Definition (Grammar of normalized sums)

1. $S \rightarrow S + T \mid c_s$, where c_s is any integer.
2. $T \rightarrow T * P \mid c_t$, where c_t is any integer.
3. $P \rightarrow S \uparrow C_e \mid abs\,(S) \mid (S)\ mod\ (S) \mid S \div C_d \mid c_m$, where c_m is a symbolic constant.
4. $C_e \rightarrow S \uparrow C_e \mid S$
5. $C_d \rightarrow S \div C_d \mid S$.

Example

The expression $c(b+a)(c+a)$ is represented as $1*a*b*c + 1*a*b*c + 1*a*c*c + 1*b*c*c + 0$, where the order of the variables is $a \prec b \prec c$.

Normalization of dependence mappings

Definition (Grammar of dependence mapping)

1. $M \rightarrow \langle D, U, Q \rangle$,
2. $D \rightarrow D, S \mid \varepsilon$,
3. $U \rightarrow U, S \mid \varepsilon$,
4. $Q \rightarrow \forall \exists Q \mid (A) \mid \varepsilon$,
5. $A \rightarrow A \wedge C \mid \varepsilon$,
6. $C \rightarrow SR0$,
7. $R \rightarrow \leq \mid \geq \mid = \mid != $.

Normalization: an example

Example

$M = \{[i][j][k] \rightarrow [10i + 50j + k][k] \mid 0 \leq i \leq 10 \wedge \exists\alpha_i \in \mathbb{N}(i = 2\alpha_i) \wedge$
$0 \leq j \leq 50 \wedge \exists\alpha_j \in \mathbb{N}(j = 3\alpha_j) \wedge 0 \leq k \leq 20 \wedge \exists\alpha_k \in \mathbb{N}(k = 2\alpha_i)\}$.
The normalized representation of this mapping is $M = \langle D, U, Q\rangle$,
where $D = 1 * i + 0, 1 * j + 0, 1 * k + 0$,
$U = 10 * i + 50 * j + 1 * k + 0, 1 * k + 0$ and $Q = \forall i \exists\alpha_i \forall j \exists\alpha_j \forall k \exists\alpha_k$
$(1 * i + 0 \geq 0 \wedge 1 * i + 0 \leq 10 \wedge 1 * i + (-2) * \alpha_i = 0 \wedge$
$1 * j + 0 \geq 0 \wedge 1 * j + 0 \leq 50 \wedge 1 * j + (-3) * \alpha_j = 0 \wedge$
$1 * k + 0 \geq 0 \wedge 1 * k + 0 \leq 20 \wedge 1 * k + (-2) * \alpha_k = 0)$.

Simplification rules

- The dependence mappings are ordered according to the occurrence of the array names.
- $1 * a * b^{(1)} + 2 * b^{(2)} + 1$. Then τ would be $\langle r_g, \langle {}_gM_{out,a}, {}_gM_{out,b^{(1)}}, {}_gM_{out,b^{(2)}} \rangle\rangle$.
- If an array name occurs more than once (as primaries) in a term of r_g, then their dependence mappings are ordered according to the lexicographic ordering of the dependence mappings.
- Let r_g of a slice g be $1 * a * b^{(1)} * b^{(2)} + 2 * b^{(3)} + 0$. Let ${}_gM_{out,b^{(1)}} = \{[i] \rightarrow [2i] \mid 1 \leq i \leq N\}$ and ${}_gM_{out,b^{(2)}} = \{[i] \rightarrow [i+5] \mid 1 \leq i \leq 2 * N\}$. Then τ would be $\langle r_g, \langle {}_gM_{out,a}, {}_gM_{out,b^{(2)}}, {}_gM_{out,b^{(1)}}, {}_gM_{out,b^{(3)}} \rangle\rangle$

Simplification rules

- The dependence mappings are ordered according to the occurrence of the array names.
- $1 * a * b^{(1)} + 2 * b^{(2)} + 1$. Then τ would be $\langle r_g, \langle {}_gM_{out,a}, {}_gM_{out,b^{(1)}}, {}_gM_{out,b^{(2)}} \rangle \rangle$.
- If an array name occurs more than once (as primaries) in a term of r_g, then their dependence mappings are ordered according to the lexicographic ordering of the dependence mappings.
- Let r_g of a slice g be $1 * a * b^{(1)} * b^{(2)} + 2 * b^{(3)} + 0$. Let ${}_gM_{out,b^{(1)}} = \{[i] \rightarrow [2i] \mid 1 \leq i \leq N\}$ and ${}_gM_{out,b^{(2)}} = \{[i] \rightarrow [i+5] \mid 1 \leq i \leq 2 * N\}$. Then τ would be $\langle r_g, \langle {}_gM_{out,a}, {}_gM_{out,b^{(2)}}, {}_gM_{out,b^{(1)}}, {}_gM_{out,b^{(3)}} \rangle \rangle$

Simplification rules (contd.)

- If the data transformation contains CSE with the same non-zero constant primary, then the tuple of dependence mappings corresponding to those terms are ordered according to the ordering of the corresponding dependence mappings.
- Let r_g be $1 * a^{(1)} + 1 * a^{(2)} + 0$. Let ${}_gM_{out,a^{(1)}} = \{[i] \rightarrow [i+1] \mid 1 \leq i \leq N\}$ and ${}_gM_{out,a^{(2)}} = \{[i] \rightarrow [2i] \mid 1 \leq i \leq N\}$. Then τ_g would be $\langle r_g, \langle {}_gM_{out,a^{(1)}}, {}_gM_{out,a^{(2)}} \rangle \rangle$.
- The occurrences of a CSE are collected together if the dependence mappings from the output array to each of the (input) arrays involved in the occurrences of the CSE are equal.

Simplification rules (contd.)

- If the data transformation contains CSE with the same non-zero constant primary, then the tuple of dependence mappings corresponding to those terms are ordered according to the ordering of the corresponding dependence mappings.
- Let r_g be $1 * a^{(1)} + 1 * a^{(2)} + 0$. Let ${}_gM_{out,a^{(1)}} = \{[i] \rightarrow [i+1] \mid 1 \leq i \leq N\}$ and ${}_gM_{out,a^{(2)}} = \{[i] \rightarrow [2i] \mid 1 \leq i \leq N\}$. Then τ_g would be $\langle r_g, \langle {}_gM_{out,a^{(1)}}, {}_gM_{out,a^{(2)}} \rangle \rangle$.
- The occurrences of a CSE are collected together if the dependence mappings from the output array to each of the (input) arrays involved in the occurrences of the CSE are equal.

An Example

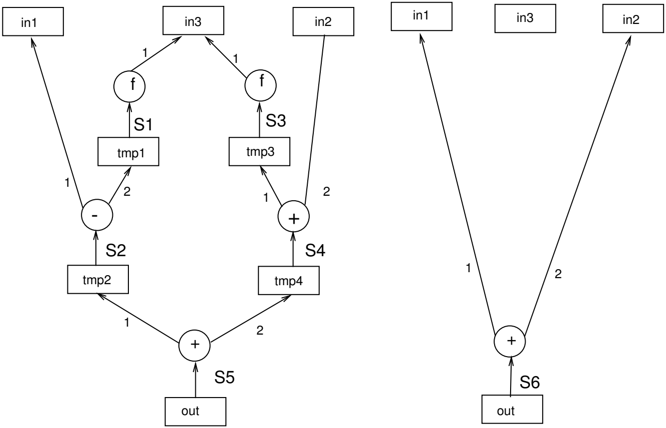
```
for(k=0; k<64; k++){
    tmp1[k] = f(in3[k+1]); //S1
    tmp2[k] = in1[2k] - tmp1[k];} //S2
for(k=5; k<69; k++){
    tmp3[k] = f(in3[k-4]); //S3
    tmp4[k-5] = tmp3[k] + in2[k-3];} //S4
for(k=0; k<64; k++){
    out[k] = tmp2[k] + tmp4[k];} //S5
```

(a) Original Program

```
for(k=0; k<64; k++) {
    out[k] = in1[2k] + in2[k+2]; } //S6
```

(b) Transformed program

An Example (contd.)



(a)

(b)

An example (contd.)

Example

The data transformation is $r_{g_1} = in1 + f(in3^{(1)}) - f(in3^{(2)}) + in2$. Here,
${}_{g_1}M_{out \rightsquigarrow in3^{(1)}} = \{[k] \rightarrow [k+1] \mid 0 \leq k \leq 64\}$ and
${}_{g_1}M_{out \rightsquigarrow in3^{(2)}} = \{[k] \rightarrow [k+1] \mid 0 \leq k \leq 64\}$.
The dependence mappings for both the occurrences of $in3$ in the data transformation are the same. Hence, the data transformation r_{g_1} is reduced to $in1 + in2$.
The data transformation of the slice, g_2 say, in the ADDG in figure (b) is $in1 + in2$.

Experimental Results

- Our method relies on the OMEGA calculator.
- SOB1: loop fusion, commutative and distributive,
- SOB2: loop reorder, commutative and distributive,
- WAVE: loop un-switching and commutative,
- LAP1: expression splitting and loop fission,
- LAP2: loop unrolling, commutative and distributive, and
- LAP3: loop spreading, commutative and remaining.

Experimental Results

Cases	*nests*	*loops*		*arrays*		*slices*		*Exec time (sec)*		
		src	*trans*	*src*	*trans*	*src*	*trans*	*equiv*	*not-eq1*	*not-eq2*
SOB1	2	3	1	4	4	1	1	01.53	0.62	0.75
SOB2	2	3	3	4	4	1	1	11.21	0.72	0.46
WAVE	1	1	2	2	2	4	4	07.05	0.73	0.59
LAP1	2	1	3	2	4	1	1	02.31	0.43	0.32
LAP2	2	1	1	2	2	1	2	07.58	0.26	0.24
LAP3	2	1	4	2	4	1	2	02.12	1.14	1.13

Table: Results for several benchmarks

Conclusions

- An ADDG based equivalence checking method is proposed for verification of loop and arithmetic transformations of array intensive behaviours.
- The method relies on normalization of arithmetic expressions and simplification rules to handle arithmetic transformations applied along with loop transformations.
- Unlike many other reported techniques, our method is strong enough to handle arithmetic transformations like associative, commutative, distributive, arithmetic expressions simplifications, common sub-expressions elimination, constant unfolding, etc.

References

Vijay Menon, Keshav Pingali, and Nikolay Mateev.
Fractal symbolic analysis.
ACM Trans. Program. Lang. Syst., 25(6):776–813, 2003.

K. C. Shashidhar.
Efficient Automatic Verification of Loop and Data-flow Transformations by Functional Equivalence Checking.
PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2008.

Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe.
Equivalence checking of static affine programs using widening to handle recurrences.
In *Proceedings of CAV '09*, pages 599–613, 2009.

Sven Verdoolaege, Martin Palkovič, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor.
Experience with widening based equivalence checking in realistic multimedia systems.
J. Electron. Test., 26(2):279–292, 2010.

Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu.
Translation and run-time validation of loop transformations.
Form. Methods Syst. Des., 27(3):335–360, 2005.

Thank You