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

Part II: Contributions
Part I

Motivations and Objectives

1. Behavioural Transformations
2. Literature Survey
3. Summery of Contributions
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

4. Array Data Dependence Graph

5. Slice

6. Equivalence of ADDGs

7. Normalization of arithmetic expressions

8. Experimentation

9. Conclusions
Array Data Dependence Graph

```
for(i = 1; i ≤ M; i = i + 1)
    for(j = 4; j ≤ N; j = j + 1)
        S1 : r1[i + 1][j - 3] = F1(in1[i][], in2[i][j]);
for(l = 3; l ≤ M; l = l + 1){
    for(m = 3; m ≤ N - 1; m = m + 1){
        if(l + m ≤ 7)
            S2 : r2[l][m] = F2(r1[l - 1][m - 2]);
        else
            S3 : r2[l][m] = F3(r1[l][N - 3]);
        S4 : out1[l][m] = F4(r2[l][m]);
    }
}
```
ADDG (contd.)

\[
\text{for}(i_1 = L_1; i_1 \leq H_1; i_1 + = r_1) \\
\text{for}(i_2 = L_2; i_2 \leq H_2; i_2 + = r_2) \\
\vdots \\
\text{for}(i_x = L_x; i_x \leq H_x; i_x + = r_x) \\
\text{if}(C_D) \text{ then} \\
S : d[e_1] \ldots [e_k] = f(u_1[e'_11] \ldots [e'_{1l1}], \ldots, u_m[e'_m1] \ldots [e'_{mlm}]);
\]

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

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

where $i_k, L_k, H_k, r_k, 1 \leq k \leq x$, are integers.
ADDG (contd.)

Definition (Definition domain ($S D_d$))

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

Definition (Definition mapping ($S M_d^{(d)}$))

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

Definition (Operand domain ($S U_{u_n}$))

$$S U_{u_n} \subseteq \mathbb{Z}^l = \left\{ [e_{n_1}(\vec{v}), \ldots, e_{n_l}(\vec{v})] \mid \vec{v} \in I_S \right\}$$ 

Definition (Operand mapping ($S M_{u_n}^{(u)}$))

$$S M_{u_n}^{(u)} = I_S \rightarrow S U_{u_n} \text{ s.t. } \forall \vec{v} \in I_S, \vec{v} \mapsto [e_{n_1}(\vec{v}), \ldots, e_{n_l}(\vec{v})] \in S U_{u_n}. $$
Definition (Dependence mapping \(sM_{d,u_n}\))

\[
sM_{d,u_n} = \{ [i_1, \ldots, i_k] \rightarrow [j_1, \ldots, j_{l_n}] \mid ([i_1, \ldots, i_k] \in sD_d \\
\land [j_1, \ldots, j_{l_n}] \in sU_{u_n} \land \exists \vec{v} \in I_S \mid ([i_1, \ldots, i_k] = sM^{(d)}_d(\vec{v}) \land [j_1, \ldots, j_{l_n}] = sM^{(u)}_{u_n}(\vec{v}))\}\}
\]

**Figure:** Computation of dependence mapping
Transitive Dependence

\[ PQD_x = P^{-1}_{x,y}(Q^{-1}_{y,z}(QU_z) \cap P_{x,y}(PD_x)) \]

\[ PQU_z = Q^{-1}_{y,z}(P^{-1}_{x,y}(PD_x) \cap Q^{-1}_{y,z}(QU_z)) \]

**Figure:** Transitive dependence
Example of dependence mappings

Example

\( S_1 D_{r1} = \{ [i + 1, j - 3] \mid [i, j] \in I_{S1} \} \)
\( S_1 M^{(d)}_{r1} = \{ [i, j] \rightarrow [i + 1, j - 3] \mid [i, j] \in I_{S1} \} \)
\( S_1 U_{in1} = \{ [i, j] \mid [i, j] \in I_{S1} \} \)
\( S_1 M^{(u)}_{in1} = \{ [i, j] \rightarrow [i, j] \mid [i, j] \in I_{S1} \} \)
\( S_1 M_{r1, in1} = (S_1 M^{(d)}_{r1})^{-1} \diamond S_1 M^{(u)}_{in1} \)
\( = \{ [i, j] \rightarrow [i - 1, j + 3] \diamond [i - 1, j + 3] \rightarrow [i - 1, j + 3] \} \mid [i, j] \in S_1 D_{r1} \} \)
\( = \{ [i, j] \rightarrow [i - 1, j + 3] \mid [i, j] \in S_1 D_{r1} \} \)

\( S_4 S_2 M_{out1, r1} = S_4 M_{out1, r2} \diamond S_2 M_{r2, r1} \)
\( = \{ [l, m] \rightarrow [l, m] \mid [l, m] \in S_4 D_{out1} \} \)
\( \diamond \{ [l, m] \rightarrow [l - 1, m - 2] \mid [l, m] \in S_2 D_{r2} \} \)
\( = [l, m] \rightarrow [l - 1, m - 2] \mid [l, m] \in S_4 D_{out1} \} \)
**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 $\tau_g = \langle r_g, \langle gM_{a \mapsto v_1}, \ldots, gM_{a \mapsto v_n} \rangle \rangle$.

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

$r_g = \text{out1} \iff F4(r2)$ [at the node $r2$],

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

$\iff F4(F2(F1(in1, in2)))$ [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] = f_1(a[2k], b[k+1]);
for(i=0; i<=50; i++)
    for(j=0; j<=50; j++)
        S2: out[i][j] = f_2(c[i+j]);
(a) original program
for(k = 0; k <= 100; k +=2){
    S3: c[k] = f_1(a[2k], b[k+1]);
    S4: c[k+1] = f_1(a[2k+2], b[k+2]);}
for(i=0; i<=50; i++)
    for(j=0; j<=50; j++)
        S5: out[i][j] = f_2(c[i+j]);
(b) transformed program
An example of slice classes (contd.)

Equivalence of ADDGs

Characteristic formula of a slice

Figure: An example for matching slices and slice class

C. Karfa, K. Banerjee, D. Sarkar, C. Mandal (IIT Kharagpur)
ISVLSI 2011
July 5, 2011 19 / 33
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. $S \rightarrow S + T | c_s$, where $c_s$ is any integer.
2. $T \rightarrow T * P | c_t$, where $c_t$ is any integer.
3. $P \rightarrow S \uparrow C_e | \text{abs}(S) | (S) \mod(S) | S \div C_d | c_m$, where $c_m$ is a symbolic constant.
4. $C_e \rightarrow S \uparrow C_e | S$
5. $C_d \rightarrow S \div C_d | S$

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. \( A \rightarrow A \land C \mid \varepsilon, \)
6. \( C \rightarrow SR0, \)
7. \( R \rightarrow \leq | \geq | = | !=. \)
Normalization: an example

Example

\[ M = \{ [i][j][k] \rightarrow [10i + 50j + k][k] \mid 0 \leq i \leq 10 \land \exists \alpha_i \in \mathbb{N}(i = 2\alpha_i) \land \\
0 \leq j \leq 50 \land \exists \alpha_j \in \mathbb{N}(j = 3\alpha_j) \land 0 \leq k \leq 20 \land \exists \alpha_k \in \mathbb{N}(k = 2\alpha_k) \}. \]

The normalized representation of this mapping is \( M = \langle D, U, Q \rangle \), where

\[ D = 1 \cdot i + 0, 1 \cdot j + 0, 1 \cdot k + 0, \]
\[ U = 10 \cdot i + 50 \cdot j + 1 \cdot k + 0, 1 \cdot k + 0 \quad \text{and} \quad Q = \forall i \exists \alpha_i \forall j \exists \alpha_j \forall k \exists \alpha_k \]
\[ (1 \cdot i + 0 \geq 0 \land 1 \cdot i + 0 \leq 10 \land 1 \cdot i + (-2) \cdot \alpha_i = 0 \land \\
1 \cdot j + 0 \geq 0 \land 1 \cdot j + 0 \leq 50 \land 1 \cdot j + (-3) \cdot \alpha_j = 0 \land \\
1 \cdot k + 0 \geq 0 \land 1 \cdot k + 0 \leq 20 \land 1 \cdot k + (-2) \cdot \alpha_k = 0). \]
The dependence mappings are ordered according to the occurrence of the array names.

$1 \ast a \ast b^{(1)} + 2 \ast b^{(2)} + 1$. Then $\tau$ would be $\langle r_g, \langle g_{M_{out}, a}, g_{M_{out}, b^{(1)}}, g_{M_{out}, b^{(2)}} \rangle \rangle$.

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 $1 \ast a \ast b^{(1)} \ast b^{(2)} + 2 \ast b^{(3)} + 0$. Let $g_{M_{out}, b^{(1)}} = \{ [i] \rightarrow [2i] \mid 1 \leq i \leq N \}$ and $g_{M_{out}, b^{(2)}} = \{ [i] \rightarrow [i + 5] \mid 1 \leq i \leq 2 \ast N \}$. Then $\tau$ would be $\langle r_g, \langle g_{M_{out}, a}, g_{M_{out}, b^{(2)}}, g_{M_{out}, b^{(1)}}, g_{M_{out}, b^{(3)}} \rangle \rangle$.
Simplification rules

- The dependence mappings are ordered according to the occurrence of the array names.

\[ 1 \times a \times b^{(1)} + 2 \times b^{(2)} + 1. \] 
Then \( \tau \) would be \( \langle r_g, \langle g^{M_{out,a}}, g^{M_{out,b^{(1)}}}, g^{M_{out,b^{(2)}}} \rangle \rangle \).

- 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 \( 1 \times a \times b^{(1)} \times b^{(2)} + 2 \times b^{(3)} + 0. \) Let 
\[ g^{M_{out,b^{(1)}}} = \{ [i] \rightarrow [2i] \mid 1 \leq i \leq N \} \] and 
\[ g^{M_{out,b^{(2)}}} = \{ [i] \rightarrow [i + 5] \mid 1 \leq i \leq 2 \times N \}. \] 
Then \( \tau \) would be \( \langle r_g, \langle g^{M_{out,a}}, g^{M_{out,b^{(2)}}}, g^{M_{out,b^{(1)}}}, g^{M_{out,b^{(3)}}} \rangle \rangle \).
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 \( 1 \times a^{(1)} + 1 \times a^{(2)} + 0 \). Let
  \[
  g^{M_{out,a^{(1)}}} = \{ [i] \rightarrow [i + 1] \mid 1 \leq i \leq N \} \quad \text{and} \quad g^{M_{out,a^{(2)}}} = \{ [i] \rightarrow [2i] \mid 1 \leq i \leq N \}.
  \]
  Then \( \tau_g \) would be \( \langle r_g, \langle g^{M_{out,a^{(1)}}, g^{M_{out,a^{(2)}}} \rangle \rangle \rangle \).

- 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.
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 $1 \times a^{(1)} + 1 \times a^{(2)} + 0$. Let

$g_{M_{out,a}^{(1)}} = \{ [i] \rightarrow [i+1] \mid 1 \leq i \leq N \}$ and
$g_{M_{out,a}^{(2)}} = \{ [i] \rightarrow [2i] \mid 1 \leq i \leq N \}$. Then $\tau_g$ would be $\langle r_g, \langle g_{M_{out,a}^{(1)}}, g_{M_{out,a}^{(2)}} \rangle \rangle$.

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

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

(a) Original Program

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

(b) Transformed program
An Example (contd.)
An example (contd.)

Example

The data transformation is $r_{g_1} = in1 + f(in3^{(1)}) - f(in3^{(2)}) + in2$. Here,

$$g_1 M_{out\rightarrow in3^{(1)}} = \{ [k] \rightarrow [k + 1] | 0 \leq k \leq 64 \} \text{ and }$$

$$g_1 M_{out\rightarrow in3^{(2)}} = \{ [k] \rightarrow [k + 1] | 0 \leq k \leq 64 \}.$$ 

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 $in1 + in2$. 
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

<table>
<thead>
<tr>
<th>Cases</th>
<th>nests</th>
<th>loops</th>
<th>arrays</th>
<th>slices</th>
<th>Exec time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>src</td>
<td>trans</td>
<td>src</td>
<td>trans</td>
</tr>
<tr>
<td>SOB1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>SOB2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>WAVE</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LAP1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>LAP2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LAP3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

**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.


Thank You