2011 IEEE Computer Society Annual Symposium on VLSI

Equivalence Checking of Array-Intensive Programs

C. Karfa, K. Banerjee, D. Sarkar, C. Mandal
Department of Computer Science and Engineering,
Indian Institute of Technology, Kharagpur 721302, India.
Email: {ckarfa, kunalb, ds, chitta}@cse.iitkgp.ernet.in

***Abstract*—An equivalence checking method for ensuring correctness of loop and arithmetic transformations in array intensive programs is presented here. The array data dependence graphs (ADDGs) are used to represent both the input and the transformed behaviours and the correctness of the transformations is ensured by proving equivalence of two ADDGs. In contrast to the existing path based one, we formalize a slice based equivalence of ADDGs. Moreover, normalization of arithmetic expressions and some simplification rules are incorporated to handle arithmetic transformations. Experimental results on several test cases demonstrate the effectiveness of our method.**

***Keywords*-Array Data Dependence Graph; Equivalence Checking; Slice; Embedded Systems.**

I. Introduction

Application of loop transformations along with arithmetic transformations targeting the best performance in terms of energy and/or area on a given platform is a common practice in the domain of multimedia and signal processing applications. These transformations can be automatic, semi-automatic or manual. The work reported in [1], for example, applied loop fusion and tiling to several nested loops and to parallelize the resulting code across different processors for some multimedia applications. A method to minimize the total energy while satisfying the performance requirements for application with multi-dimensional nested loops was proposed in [2]. Several loop transformation techniques and their effects in embedded system design may be found in [3], [4]. Applications of several arithmetic transformations can be found in [5], [6]. Importantly, loop transformation techniques and the arithmetic transformation techniques are applied dynamically since application of one may create scope of application of the other. In all cases, it is crucial to know that the transformed program preserves the behaviour of the original.

A translation validation approach capable of verifying structure preserving and reordering loop transformations has been introduced by Pnueli et al. [7]. A method called fractal symbolic analysis has been proposed in [8]. The power of these methods depends on the availability of the transformation dependant rule set and the information of the order in which the loop transformations are applied. An ADDG based equivalence checking method was proposed by Shashidhar et al. [9] [10] for a restricted class of programs which must have static control-flow, uniform recurrence, affine indices and bounds and single assignment form. This work is promising because they are capable of handling most of the loop transformation techniques without taking any information from the synthesis tools. The main limitations of the ADDG based method are its inability to handle the cases of (i) non-uniform recurrence, (ii) data-dependent assignments and accesses and (iii) arithmetic transformations. The method proposed in [11] extends the ADDG models to dependence graphs to handle non-uniform recurrences. This method is further extended in [12] to verify data-dependent assignments and accesses also. It has been discussed in [10] how the basic ADDG based method can be extended to handle associative and commutative transformations. All the above methods, however, fail if the transformed behaviour is obtained from the original behaviour by application of arithmetic transformations such as, distributive transformations, arithmetic expression simplification, common sub-expression elimination, constant unfolding, substitution of multiplications with constants by addition, etc, along with loop transformations. The definition of equivalence of ADDGs proposed by Shashidhar et al. [9] [10] cannot be extended (unlike the cases of commutative and associative transformations) to handle these arithmetic transformations. It is because of the fact that the basic assumption in their equivalence checking method is that the number of paths must be the same in two equivalent slices. However, this assumption may not hold in the cases of several arithmetic transformations.

The objective of this work is to develop an equivalence checking method which will be capable of verifying a wide variety of loop transformations and also several arithmetic transformations mentioned above applied together on array and loop intensive applications. We consider the same class of programs considered by Shashidhar et al. [9] and their ADDG based modelling of programs. The contributions of the present work are: (i) defining the characteristic formula of a slice, (ii) redefining the equivalence of ADDGs based on slice-level characterization rather than path based one, alleviating in the process, a shortcoming of the latter in handling equivalence of slices where the number of paths is unequal, (iii) incorporating normalization of arithmetic

This work was supported by Microsoft Corporation and Microsoft Research India under Microsoft Research India Ph.D. Fellowship Award.

978-0-7695-4447-2/11 $26.00 © 2011 IEEE
DOI 10.1109/ISVLSI.2011.61

expressions [13] and some additional simplification rules for normalized expressions for handling several arithmetic transformations applied along with loop transformations, (iv) providing an method for checking equivalence of ADDGs.

The rest of the paper is organized as follows. The ADDG model is briefly introduced in section II. The notions of a slice and its characteristic formula are defined in section III. The formulation of equivalence checking and the verification method are given in section IV. Some experimental results are given in section V. Finally, the paper is concluded in section VI.

II. Representation of behaviours as ADDGs

Definition 1 (Array Data Dependence Graph:) The ADDG of a sequential behaviour is a directed graph $G=(V,E)$, where the vertex set V is the union of the set A of array nodes and the set F of operator nodes and the edge set $E=\{\langle a,f\rangle \mid a\in A\wedge f\in F\}\cup\{\langle f,a\rangle \mid f\in F\wedge a\in A\}$. Edges of the form $\langle a,f\rangle$ are *write* edges and the edges of the form $\langle f,a\rangle$ are *read* edges. An assignment statement S of the form $l[\vec{l_k}]=f(r_1[\vec{i_1}],\ \ldots,\ r_k[\vec{i_k}])$ appears as a subgraph G_S of $G=(A\cup F,E)$, where $G_S=\langle V_S,\ E_S\rangle$, $V_S=A_S\cup F_S$, $A_S=\{l,r_1,\ \ldots,\ r_k\}\subseteq A$, $F_S=\{f\}\subseteq F$ and $E_S=\{\langle l,f\rangle\}\cup\{\langle f,r_i\rangle,\ 1\le i\le k\rangle\}\subseteq E$. The write edge $\langle l,f\rangle$ is labelled with the statement designation S.

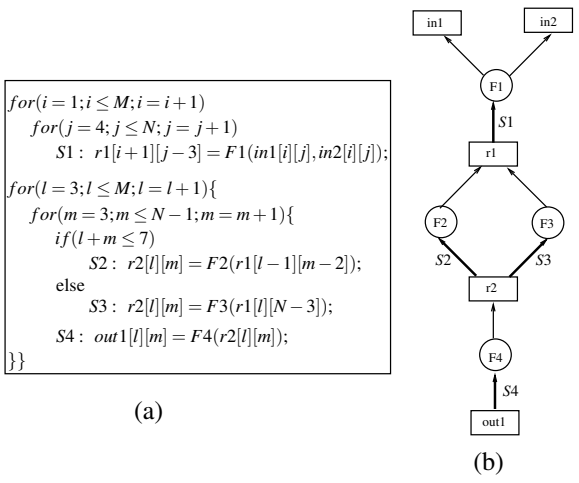

Figure 1. (a) A nested-loop behaviour and (b) its corresponding ADDG

Figure 1 shows a sequential behaviour and its corresponding ADDG. Certain information will be extracted from each statement S of the behaviour and associated with the write edge labelled with S in the ADDG. Let us first consider, for this purpose, the generalized n-nested loop structure in figure 2 in which S occurs. Each index i_k has a lower limit L_k and a higher limit H_k and a step constant (increment/decrement) r_k. The parameters L_k, H_k and r_k are all integers. The statement S executes under the condition C_D over the loop indices $i_k,\ 1\le k\le n$, within the loop body. All the index expressions $e_1,\ldots,e_k$ of the array d and the expressions $e'_{11},\ldots,e'_{1l_1},\ldots,e'_{m1},\ldots,e'_{ml_m}$ of the corresponding arrays $u_1,\ldots,u_m$ in the statement S are affine arithmetic expressions over the loop indices.

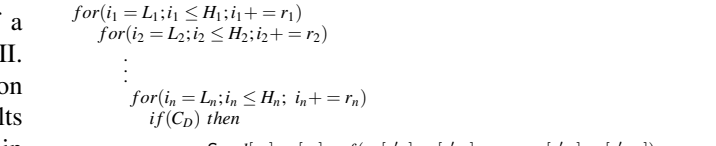

Figure 2. A generalized nested loop structure

The iteration domain of the statement S is given as

$$I_{\mathsf{S}}=\{[i_1,i_2,\ldots,i_n]\mid \bigwedge_{k=1}^{n}(L_k\le i_k\le H_k\wedge C_D\wedge\exists\alpha_k\in\mathbb{Z}(i_k=\alpha_k r_k+L_k))\}$$

where $i_k,L_k,H_k,r_k,1\le k\le n$, are integers.

The *definition domain* ${}_{\mathsf{S}}D_d$ of the left hand side (lhs) array d is given by ${}_{\mathsf{S}}D_d\subseteq\mathbb{Z}^k=\{[e_1(\vec{v}),\ \ldots,\ e_k(\vec{v})]\mid\vec{v}\in I_{\mathsf{S}}\}$. The *definition mapping* (${}_{\mathsf{S}}M_d^{(d)}$) is given by ${}_{\mathsf{S}}M_d^{(d)}=\{I_{\mathsf{S}}\rightarrow{}_{\mathsf{S}}D_d\mid\forall\vec{v}\in I_{\mathsf{S}},\ \vec{v}\mapsto[e_1(\vec{v}),\ \ldots,\ e_k(\vec{v})]\in{}_{\mathsf{S}}D_d\ \}$. The *operand domain* of the operand array u_n is given by ${}_{\mathsf{S}}U_{u_n}\subseteq\mathbb{Z}^{l_n}=\{[e_{n1}(\vec{v}),\ \ldots,\ e_{nl_n}(\vec{v})]\mid\vec{v}\in I_S\}$. The *operand mapping* (${}_SM_{u_n}^{(u)}$) is given by ${}_{\mathsf{S}}M_{u_n}^{(u)}=\{I_{\mathsf{S}}\rightarrow{}_{\mathsf{S}}U_{u_n}\mid\forall\vec{v}\in I_{\mathsf{S}},\ \vec{v}\mapsto[e_{n1}(\vec{v}),\ \ldots,\ e_{nl_n}(\vec{v})]\in{}_{\mathsf{S}}U_{u_n}\}$. There are m operand domains and m operand mappings, one for each operand array $u_1,\ \ldots,\ u_m$.

Definition 2 (Dependence mapping (${}_{\mathsf{S}}M_{d,u_n}$)): ${}_{\mathsf{S}}M_{d,u_n}=\{[i_1,\ldots,i_k]\rightarrow[j_1,\ldots,j_{l_n}]\mid([i_1,\ldots,i_k]\in{}_{\mathsf{S}}D_d\ \wedge[j_1,\ldots,j_{l_n}]\in{}_{\mathsf{S}}U_{u_n}\wedge\exists\vec{v}\in I_{\mathsf{S}}\mid([i_1,\ldots,i_k]={}_{\mathsf{S}}M_d^{(d)}(\vec{v})\ \wedge[j_1,\ldots,j_{l_n}]={}_{\mathsf{S}}M_{u_n}^{(u)}(\vec{v})))\}$

The dependence mapping ${}_{\mathsf{S}}M_{d,u_n}$ can be obtained as ${}_{\mathsf{S}}M_{d,u_n}=({}_{\mathsf{S}}M_d^{(d)})^{-1}\diamond{}_{\mathsf{S}}M_{u_n}^{(u)}$. The dependence mapping between the array x and the array z, i.e., ${}_{\mathsf{PQ}}M_{x,\,z}$, can be obtained from the mappings ${}_{\mathsf{P}}M_{x,\,y}$ and ${}_{\mathsf{Q}}M_{y,\,z}$ by right composition ($\diamond$) of ${}_{\mathsf{P}}M_{x,\,y}$ and ${}_{\mathsf{Q}}M_{y,\,z}$. The following definition captures this computation.

Definition 3 (Transitive Dependence Mapping): For two consecutive statements Q and P in the behaviour, ${}_{\mathsf{PQ}}M_{x,z}={}_{\mathsf{P}}M_{x,y}\diamond{}_{\mathsf{Q}}M_{y,z}=\{[i_1,\ldots,i_{l_1}]\rightarrow[k_1,\ldots,k_{l_3}]\mid\exists[j_1,\ \ldots,j_{l_2}]\ \mathtt{s.t.}\ [i_1,\ldots,i_{l_1}]\rightarrow[j_1,\ldots,j_{l_2}]\in{}_{\mathsf{P}}M_{x,y}\wedge[j_1,\ldots,j_{l_2}]\rightarrow[k_1,\ldots,k_{l_3}]\in{}_{\mathsf{Q}}M_{y,z}\}$.

The transitive dependence can be extended to a sequence of statements in a natural way.

Example 1: Let us consider the behaviour and its corresponding ADDG of figure 1. Let us now consider the statements $S4$ and $S2$ of the behaviour. We have

$I_{S4}=\{[l,\ m]\mid 3\le l\le M\wedge 3\le m\le N-1\wedge(\exists\alpha_1,\alpha_2\in\mathbb{Z}\mid l=\alpha_1+3\wedge m=\alpha_2+3)\}$,

${}_{S4}D_{out1}=I_{S4}$, ${}_{S4}U_{r2}=I_{S4}$

${}_{S4}M_{out1,r2}=\{[l,m]\rightarrow[l,m]\mid[l,m]\in{}_{S4}D_{out1}\}$,

$I_{S2}=\{[l,\ m]\mid 3\le l\le M\wedge 3\le m\le N-1\wedge l+m\le 7\wedge(\exists\alpha_1,\alpha_2\in\mathbb{Z}\mid l=\alpha_1+3\wedge m=\alpha_2+3)\}$,

$_{S2}D_{r2} = I_{S2}$,
$_{S2}U_{r1} = \{[l-1,\ m-2] \mid [l,m] \in I_{S2}\}$ and
$_{S2}M_{r2,r1} = \{[l,m] \rightarrow [l-1,m-2] \mid [l,m] \in\ _{S2}D_{r2}\}$.

The transitive dependence mapping $_{S4S2}M_{out1,r1}$ can be obtained from $_{S4}M_{out1,r2}$ and $_{S2}M_{r2,r1}$ by the composition operator $\diamond$ as follows:

$$\begin{aligned} {}_{S4S2}M_{out1,r1} &= {}_{S4}M_{out1,r2} \diamond {}_{S2}M_{r2,r1} \\ &= \{[l,m] \rightarrow [l,m] \mid [l,m] \in {}_{S4}D_{out1}\} \\ &\quad \diamond \{[l,m] \rightarrow [l-1,m-2] \mid [l,m] \in {}_{S2}D_{r2}\} \\ &= [l,m] \rightarrow [l-1,m-2] \mid [l,m] \in {}_{S4}D_{out1}\} \end{aligned}$$

□

III. Slices

Definition 4 (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) of its operator nodes and at most one outgoing edge (write edge) of its array nodes except the start node and terminal nodes.

Each statement in the ADDG in figure 1 represents a slice. Also, each of the statement sequences $\langle S2S1\rangle$, $\langle S3S1\rangle$, $\langle S4S2\rangle$, $\langle S4S3\rangle$, $\langle S4S2S1\rangle$ and $\langle S4S3S1\rangle$ represents a slice in this ADDG. The start array node of a slice depends on each of the terminal array nodes through a sequence of statements. Therefore, the dependence mapping between the start array node and each of the terminal array nodes of a slice can be computed as the transitive dependence mapping over the sequence of statements from the start node to the terminal node, in question, using definition 3. The dependence mappings capture the index mappings between the start array node and the terminal array nodes in a slice. In addition, it is required to store how the output array is dependent functionally on the input arrays in a slice. We denote this notion as the *data transformation* of a slice.

Definition 5 (Data transformation of a slice g (r_g)): It is an arithmetic expression e over the terminal arrays of the slice such that e represents the value of the output array elements of the slice after execution of the slice.

The data transformation r_g in a slice g can be obtained by using a backward substitution method [14] on the slice from its output array node up to the input array nodes. The backward substitution method of finding r_g is based on symbolic simulation. The steps of the backward substitution method, for example, for computation of data transformation for the slice represented by statement sequence $\langle S4S2S1\rangle$ in the ADDG in figure 1(b) are as follows:

$out1 \Leftarrow F4(r2)$ [at the node $r2$],
$\Leftarrow F4(F2(r1))$ [at the node $r1$],
$\Leftarrow F4(F2(F1(in1,in2)))$ [at the nodes $in1$ and $in2$]

The slice g is characterized by its data transformation and the list of dependence mappings between the source array and the terminal arrays.

Definition 6 (Characteristic formula of a slice): The *characteristic formula* of a slice g is given as the tuple $\tau_g = \langle r_g, \langle {}_gM_{a\rightsquigarrow v_1},\ \ldots,\ {}_gM_{a\rightsquigarrow v_n}\rangle\rangle$, where a is the start node of the slice, $v_i,\ 1 \le i \le n$, are the terminal nodes, r_g is an arithmetic expression over $v_1,\ \ldots,\ v_n$ representing the data transformation of g and ${}_gM_{a\rightsquigarrow v_i},\ 1 \le i \le n$, denotes the dependence mapping between a and v_i.

Definition 7 (IO-slice): A slice is said to be an IO-slice iff its start node is an output array node and the terminal nodes are input array nodes.

It is required to capture the dependence of each output array on the input arrays. Therefore, IO-slices are of our interest. It may be noted that the slices represented by the statement sequences $\langle S4S2S1\rangle$ and $\langle S4S3S1\rangle$ are the only two IO-slices in the ADDG in figure 1(b).

IV. Equivalence of ADDGs

As discussed in the introduction, Shashidhar et al. [9] [10] have proposed an equivalence checking method for ADDG based verification of loop transformations and some data-flow transformations. In characterizing the transformation over a slice, the method proposed in [9] relies on the signature of the individual paths [1] of the slice. The basic assumption in their equivalence checking method is that the number of paths from the output array to the input arrays must be the same in two equivalent IO-slices. Since, paths may be removed or the path signatures may be transformed significantly due to application of arithmetic transformations, computationally equivalent slices may have paths whose signatures are non-identical or have no correlation among them. Following example illustrates this fact.

Example 2: Let us consider two program fragments given in figure 3. The program in figure 3(b) is obtained from figure 3(a) by loop merging and simplification of arithmetic expressions. These two programs are actually equivalent. Let us now consider their corresponding ADDGs in figure 3(c) and figure 3(d), respectively. It may be noted that both the ADDGs contain a single IO-slice. These two IO-slices have different number of paths from the output array to the input arrays. Specifically, the IO-slice in figure 3(c) has two paths from the output node *out* to the input node *in*3; however, the slice in the ADDG in figure 3(d) has no such paths. □

The above example underlines the fact that while obtaining the equivalence of a slice, *one should compare the slice as a whole rather than the individual path signatures within the slice*. In this work, we redefine the equivalence of

[1] A *path* p from an array node a_1 to an array node a_n in an ADDG is of the form $a_1 \rightarrow f_1 \xrightarrow{l_1} a_2 \rightarrow f_2 \xrightarrow{l_2} a_3 \rightarrow \ \ldots\ \xrightarrow{l_{n-2}} a_{n-1} \rightarrow f_{n-1} \xrightarrow{l_{n-1}} a_n$, where the array nodes (a_i's) and the operator nodes (f_i's) alternate, $\langle a_k, f_k\rangle,\ 1 \le k \le n-1$, are the write edges, $\langle f_k,\ a_{k+1}\rangle,\ 1 \le k \le n-1$, are the read edges in the ADDG and $l_i, 1, \le i \le n-1$, are the labels of the read edges of the path. A label l of the read edge from an operator node f to an array node a denotes that a is the l^{th} argument of the operator f. The signature of that path is a tuple $\langle a_1, f_1, l_1, f_2, l_2,\ \ldots,\ l_{n-2}, f_{n-1}, l_{n-1}, a_n\rangle$.

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

(a) Original program

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

(b) Transformed program

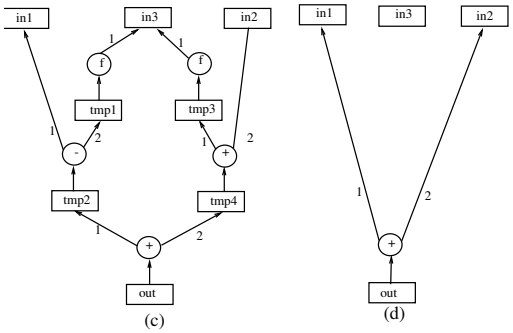

Figure 3. (a) Original programme (b) transformed programme (c) ADDG of the original program (d) ADDG of the transformed program

ADDGs based on our definition of characteristic formulas of slices. In addition, a normalization technique is incorporated in our method to represent the data transformations of the slices. Two simplification rules for normalized expressions are proposed in this work. Slice level characterization, inclusion of the normalization technique and the simplification rules enable us to handle several arithmetic transformations applied along with loop transformations. Let us first introduce the normalization procedure of data transformations and the simplification rules. We then formulate the equivalence problem of ADDGs.

A. Normalization of slice data transformations

The normalization process reduces many computationally equivalent formulas syntactically identical as it forces all the formulas to follow a uniform structure [13]. We use the following normal form adapted from [13].

The data transformation of a slice can be represented in the *normalized sum form*. A normalized sum is a sum of terms with at least one constant term; each term is a product of primaries with a non-zero constant primary; each primary is a storage variable, an input variable or of the form $abs(s)$, $mod(s_1,s_2)$, $exp(s_1,s_2)$ or $div(s_1,s_2)$, where s, s_1, and s_2 are normalized sums. The dependence mapping can be represented as three tuple – index expressions of the lhs array, index expressions of the rhs array and the quantified formula defining the domain of the mapping. Each of them are also represented in normalized form. In addition to the above structure, any normalized sum is arranged by a *lexicographic ordering of its constituent subexpressions from the bottom-most level, i.e., from the level of simple primaries*. This will help us handle the algebraic transformations efficiently.

Example 3: The expression $3yz - 4x^2 + 2$ will have the normal form $-4*x*x + 3*y*z + 2$, where the lexicographic order of the variables are $x \prec y \prec z$. 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 lexicographic order of the variables is $a \prec b \prec c$. □

The dependence mapping can be represented as three tuple – index expressions of the lhs array, index expressions of the rhs array and the quantified formula defining the domain of the mapping. The index expressions of lhs/rhs array is an ordered tuple of normalized sums where the i^{th} normalized sum represents the index expression of the i^{th} dimension of the array. The quantified formula is a conjunction of atomic formulas defined over the universally quantified variables. The increment/decrement of universally quantified variables is defined by existentially quantified variables.

B. Simplification rules for data transformation

We have proposed two simplification rules over a normalized expression. Normalization along with these simplification rules enables us handle arithmetic transformations efficiently. The simplification rules are as follows:

(1). (a) For a slice g, the dependence mappings in its characteristic formula τ_g are ordered according to the occurrence of the array names in r_g. (b) 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. (c) If the data transformation r_g in τ_g contains common sub-expressions (terms) 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 in the tuples of the terms.

(2). In the data transformation of a slice, the occurrences of a common sub-expression are collected together if the dependence mappings from the output array to each of the (input) arrays involved in the occurrences of the sub-expression are equal. If the collection of the sub-expressions cancels out through symbolic computation, then remove all the dependence mappings corresponding to those sub-expressions.

Example 4: Let us consider, for example, that the data transformation of a slice $g(a, \langle x,z \rangle)$ is $3x + 5z - 3x$, where x and z be two input arrays. Let the dependence mapping from the output array a of the slice to x corresponding to the first sub-expression $3x$ be ${}_gM^{(1)}_{a \rightsquigarrow x}$ and the same for x corresponding to the second sub-expression $3x$ be ${}_gM^{(2)}_{a \rightsquigarrow x}$.

This formula is reduced to $5z$ if the dependence mapping ${}_gM^{(1)}_{a\rightsquigarrow x} = {}_gM^{(2)}_{a\rightsquigarrow x}$. Similarly, the formula $3xy + 4z + 7xy$ is reduced to $10xy + 4z$ if the dependence mappings from output array to x are the same for both x (in $3xy$ and $7xy$) and the dependence mappings from output array to y are the same for both y (in $3xy$ and $7xy$). □

C. Equivalence Problem Formulation

Let G_S be the ADDG corresponding to the sequential behaviour and G_T be the ADDG corresponding to the transformed behaviour.

Definition 8 (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.

Definition 9 (IO-slice class): A slice class in an ADDG is the maximum set of matching IO-slices of the ADDG.

Let a slice class be $C_g(a,\ \langle v_1,\ \ldots, v_l\rangle) = \{g_1,\ldots,g_k\}$ where each slice involves l input arrays $v_1,\ \ldots,\ v_l$ and the output array a. The data transformation of C_g is the same as any of the member slices. Due to single assignment form of the behaviour, the domains of the dependence mappings between the output array a and the input array v_j in the slices of C_g must be non-overlapping. The domain of the dependence mapping ${}_{C_g}M_{a\rightsquigarrow v_m}$ from a to v_m over the entire class C_g is the union of the domains of ${}_{g_i}M_{a\rightsquigarrow v_m},\ 1 \le i \le k$. The data transformation of C_g is the data transformation of any of the slices in C_g.

Definition 10 (IO-slice class equivalence:): A slice class C_1 of an ADDG G_S is said to be equivalent to a slice class C_2 of G_T, denoted as $C_1 \simeq C_2$, iff

(i) The data transformation of C_1 and C_2 are same.

(ii) Both C_1 and C_2 consist of the same number of dependence mappings and the corresponding dependence mappings in the two classes are same.

Definition 11 (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.

The equivalence checking method is given as algorithm 1. Given the undecidability of this equivalence problem [14], completeness of the method is unattainable. Our method may produce false-negative results for some cases. Therefore, the method reports "ADDGs may not be equivalent" when it fails to show the equivalence of two ADDGs. Our method, however, can be proved to be sound.

Algorithm 1 Equivalence Checking Between two ADDGs

```
1: /* Input: Two ADDGs G_S and G_T;
   Output: Whether G_S and G_T are equivalent or not; */
2: Find the set of IO-slices in each ADDG. Find the characteristic
   formulae of the slices;
3: Use arithmetic simplification rule to the data transformation of
   the slices of G_S and G_T;
4: Obtain the slice classes and their characteristic formulas in
   each ADDG; Let C_{G_S} and C_{G_T} be the respective sets of slice
   classes in both the ADDGs;
5: for each slice class g_1 in C_{G_S} do
6:     g_k = findEquivalentSlices(g_1, C_{G_T});   /* this function
       returns the equivalent slice of g_1 in C_{G_T} if found; otherwise
       returns NULL; */
7:     if g_k = NULL then
8:         Report "ADDGs may not be equivalent;" exit(failure);
9:     end if
10: end for
11: Repeat the above loop by interchanging G_S and G_T;
12: Report "ADDGs are equivalent;" exit(success);
```

V. Experimental Results

The method has been implemented in C language and run on a 2.0 GHz Intel® Core™2 Duo machine. For the dependence mappings of the slices, our method relies on the OMEGA calculator [15]. The method has been tested on several instances of equivalence checking problems obtained manually from the sobel edge detection (SOB), Debaucles 4-coefficient wavelet filter (WAVE) and Laplace algorithm to edge enhancement of northerly directional edges (LAP). To create the test cases, we have considered variety of loop transformations and arithmetic transformations as shown in column two of table I. The maximum nesting of the loops, number of loop bodies, arrays and slices in source and transformed behaviours are shown in columns three to nine in table I. The execution time of our tool for these test cases are tabulated in column 10 of the table. In the last two cases, the number of slices differs from the source ADDG to the transformed ADDG, the method successfully formed the slice classes and established the equivalence. In all of the cases, our method was able to establish the equivalence in less than twelve seconds. It may be noted that the method reported in [9] fails in the cases of SOB1, SOB2, LAP2 because of application of distributive transformations. It is difficult to compare running times with this method [9] since the examples used there are not publicly available and also their tool is not available to us.

In our second experiment, we take the source behaviour and the transformed behaviours of the previous experiment. We, however, intentionally change the index expressions of some of the arrays or limits of some of the loops in the transformed behaviours. As a result, some dependence mappings (involving those arrays) do not match with the original behaviour. Similarly, we change the rhs expressions of some statements of the transformed behaviours to create another set of erroneous test cases. As a result, the data transformations of some of the slices do not match with the corresponding slices of the original behaviour. The execution time of our tool for these two cases are tabulated in columns eleven and twelve, respectively. Our tool is able to find the non-equivalence and locate the erroneous statement(s) in all the cases in less than two seconds as shown in table I.

Cases (1)	*transformations* (2)	*nests* (3)	*loops* *src* (4)	*loops* *trans* (5)	*arrays* *src* (6)	*arrays* *trans* (7)	*slices* *src* (8)	*slices* *trans* (9)	*Exec time (sec)* *equiv* (10)	*not-equiv1* (11)	*not-equiv2* (12)
SOB1	loop fusion, commutative and distributive	2	3	1	4	4	1	1	01.53	0.62	0.75
SOB2	loop reorder, commutative and distributive	2	3	3	4	4	1	1	11.21	0.72	0.46
WAVE	loop un-switching and commutative	1	1	2	2	2	4	4	07.05	0.73	0.59
LAP1	expression splitting and loop fission	2	1	3	2	4	1	1	02.31	0.43	0.32
LAP2	loop unrolling, commutative and distributive	2	1	1	2	2	1	2	07.58	0.26	0.24
LAP3	loop spreading, commutative and renaming	2	1	4	2	4	1	2	02.12	1.14	1.13

Table I
RESULTS FOR SEVERAL BENCHMARKS

VI. CONCLUSIONS

The present work is concerned with verification of loop transformations and arithmetic transformation techniques applied on loop and array intensive applications (common in the multimedia and signal processing domains). An ADDG based equivalence checking method is proposed for this purpose. 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 based on transformations employing arithmetic transformations associative, commutative, distributive, arithmetic expression simplifications, common subexpression elimination, constant unfolding, substitution of multiplication with constant by addition, etc. Experimental results have shown the efficiency of the method.

Future scope of the work includes identification of simplification rules to handle more sophisticated arithmetic transformations, such as, operator strength reduction, etc. Our current implementation supports only uniform recurrence permitting the use of OMEGA calculator to compute the dependence mappings. Our next objective is to find the scope of application of our normalization techniques to widening based approach [11] which can handle non-uniform recurrence.

REFERENCES

[1] Y. Bouchebaba, B. Girodias, G. Nicolescu, E. M. Aboulhamid, B. Lavigueur, and P. Paulin, "Mpsoc memory optimization using program transformation," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 4, pp. 43:1–43:39, 2007.

[2] M. Karakoy, "Optimizing array-intensive applications for on-chip multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 5, pp. 396–411, 2005.

[3] M. Qiu, E. H. M. Sha, M. Liu, M. Lin, S. Hua, and L. T. Yang, "Energy minimization with loop fusion and multi-functional-unit scheduling for multidimensional dsp," *J. Parallel Distrib. Comput.*, vol. 68, no. 4, pp. 443–455, 2008.

[4] M. Palkovic, F. Catthoor, and H. Corporaal, "Trade-offs in loop transformations," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, pp. 22:1–22:30, April 2009.

[5] B. Landwehr and P. Marwedel, "A new optimization technique for improving resource exploitation and critical path minimization," in *ISSS*, pp. 65–72, 1997.

[6] M. Potkonjak, S. Dey, Z. Iqbal, and A. Parker, "High performance embedded system optimization using algebraic and generalized retiming techniques," in *Proc. of ICCD*, pp. 498 –504, Oct. 1993.

[7] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu, "Translation and run-time validation of loop transformations," *Form. Methods Syst. Des.*, vol. 27, no. 3, pp. 335–360, 2005.

[8] V. Menon, K. Pingali, and N. Mateev, "Fractal symbolic analysis," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 6, pp. 776–813, 2003.

[9] 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.

[10] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens, "Functional equivalence checking for verification of algebraic transformations on array-intensive source code," in *Proc. of DATE'05*, pp. 1310–1315, 2005.

[11] S. Verdoolaege, G. Janssens, and M. Bruynooghe, "Equivalence checking of static affine programs using widening to handle recurrences," in *Proceedings of CAV '09*, pp. 599–613, 2009.

[12] S. Verdoolaege, M. Palkovič, M. Bruynooghe, G. Janssens, and F. Catthoor, "Experience with widening based equivalence checking in realistic multimedia systems," *J. Electron. Test.*, vol. 26, no. 2, pp. 279–292, 2010.

[13] D. Sarkar and S. De Sarkar, "A theorem prover for verifying iterative programs over integers," *IEEE Trans Software. Engg.*, vol. 15, no. 12, pp. 1550–1566, 1989.

[14] Z. Manna, *Mathematical Theory of Computation*. Tokyo: McGraw-Hill Kogakusha, 1974.

[15] W. Kelly, E. Rosser, B. Pugh, D. Wonnacott, T. Shpeisman, and V. Maslov, "The omega calculator and library." available at `http://www.cs.umd.edu/projects/omega/` .