Translation Validation of Loop and Arithmetic Transformations in the Presence of Recurrences

Kunal Banerjee *, Chittaranjan Mandal, Dipankar Sarkar

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur, India
{kunalb, chitta, ds}@cse.iitkgp.ernet.in

Abstract

Compiler optimization of array-intensive programs involves extensive application of loop transformations and arithmetic transformations. Hence, translation validation of array-intensive programs requires manipulation of intervals of integers (representing domains of array indices) and relations over such intervals to account for loop transformations and simplification of arithmetic expressions to handle arithmetic transformations. A major obstacle for verification of such programs is posed by the presence of recurrences, whereby an element of an array gets defined in a statement S inside a loop in terms of some other element(s) of the same array which have been previously defined through the same statement S. Recurrences lead to cycles in the data-dependence graph of a program which make dependence analyses and simplifications (through closed-form representations) of the data transformations difficult. Another technique which works better for recurrences does not handle arithmetic transformations. In this work, array data-dependence graphs (ADDGs) are used to represent both the original and the optimized versions of the program and a validation scheme is proposed where the cycles due to recurrences in the ADDGs are suitably abstracted as acyclic subgraphs. Thus, this work provides a unified equivalence checking framework to handle loop and arithmetic transformations along with most of the recurrences – this combination of features had not been achieved by a single verification technique earlier.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Translation validation, equivalence checking, loop transformations, arithmetic transformations, recurrence, array data-dependence graph (ADDG)

1. Introduction

Loop transformations together with arithmetic transformations are applied extensively in the domain of multimedia and signal processing applications to obtain better performance in terms of energy, area and/or execution time. For example, the work reported in (Bouchebaba et al. 2007) applies loop fusion and loop tiling to several nested loops and parallelizes the resulting code across different processors for multimedia applications. Minimization of the total energy while satisfying the performance requirements for applications with multi-dimensional nested loops is another example (Kadayif et al. 2005). Application of arithmetic transformations can improve the performance of computationally intensive applications (Potkonjak et al. 1993; Zory and Coelho 1998). These transformations have also been successfully employed in minimizing critical path lengths (Landwehr and Marwedel 1997). Often loop transformation and arithmetic transformation techniques are applied incrementally since application of one may create scope for application of several other techniques. In all these cases, it is crucial to ensure that the intended behaviour of the program has not been altered faultily during transformation.

Verification of loop transformations in array-intensive programs has been extensively investigated. Reported work on translation validation include an approach based on transformation specific rules for verification of loop interchange, skewing, tiling and reversal transformations (Zuck et al. 2005), and a method called fractal symbolic analysis (Menon et al. 2003). The main drawback of these approaches is that they require information such as the list of transformations applied and their order of application; however, such information need not be readily available from the compilers. Another approach is a symbolic simulation based method for verification of loop transformations for programs with no recurrences and with affine indices and bounds (Matsumoto et al. 2007). This method, however, does not maintain explicit edges for multiple occurrences of the same array in a computation in its internal representation and hence this method fails in the presence of many arithmetic transformations. Another approach for verifying array-intensive programs can be to use off-the-shelf SMT solvers or theorem provers since the equivalence between two programs can be modeled with a formula such that the validity of the formula implies the equivalence (Karfa et al. 2013a). Although SMT solvers and theorem provers can efficiently handle linear arithmetic, they are not equally suitable for non-linear arithmetic. Since array-intensive programs often contain non-linear arithmetic in the data transformation, these tools are found to be inadequate for establishing equivalence of such programs (Karfa et al. 2013a). Some methods (Barthou et al. 2002; Shashidhar et al. 2005; Shashidhar 2008) consider a restricted class of programs which must have static control-flows, valid schedules, affine indices and bounds and single assignment forms. The original and the transformed programs may be modeled as two ADDGs and the correctness of the loop transformations is established by showing equivalence between the two ADDGs (Shashidhar et al. 2005; Shashidhar 2008). This technique is capable of handling a wide variety of loop transformation

* K. Banerjee is presently a Research Scientist at Intel Parallel Computing Lab, Bangalore, India.

LCTES'16, June 13–14, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4316-9/16/06...$15.00
http://dx.doi.org/10.1145/2907950.2907954

techniques without taking any information from the compilers. Another method (Verdoolaege et al. 2009, 2012) extends the ADDG model to a dependence graph model to handle recurrences along with associative and commutative operations. All the above methods, however, fail if the transformed program is obtained from the original program by application of arithmetic transformations such as, distributive transformations, arithmetic expression simplification, common sub-expression elimination, constant unfolding, etc., along with loop transformations. An ADDG based method is presented in (Karfa et al. 2013b) which compares ADDGs at slice-level rather than at the path-level (Shashidhar 2008) and employs a normalization technique (Sarkar and De Sarkar 1989) for the arithmetic expressions to verify a wide variety of loop transformations and a wide range of arithmetic transformations applied together in array-intensive programs. However, it cannot verify programs involving recurrences because recurrences lead to cycles in the ADDGs rendering the existing data-dependence analysis and simplification (through closed-form representations) of the data transformations in ADDGs inapplicable.

In this work, the slice based equivalence checking framework (Karfa et al. 2013b) based on ADDGs to handle loop and arithmetic transformations is extended so that most recurrences are also handled – this combination of features has not been achieved by a single verification technique earlier. The validation scheme proposed here identifies suitable subgraphs (arising from recurrences) in the ADDGs and treats them separately; each such cyclic subgraph in the original ADDG is compared with its corresponding subgraph in the transformed ADDG in isolation and if all such pairs of subgraphs are found equivalent, then the entire ADDGs, with the subgraphs replaced by some uninterpreted functions of proper arities, are compared in the same way as in the method described in (Karfa et al. 2013b). The experimental results demonstrate the efficacy of the method.

The rest of the paper is organized as follows. Section 2 explains the ADDG based equivalence checking scheme as described in (Karfa et al. 2013b) with the help of an illustrative example. Section 3 provides an extension of the equivalence checking scheme to handle recurrences. The correctness and the complexity of the prescribed method are formally treated in Section 4. The experimental results are given in Section 5. Section 6 underlines some limitations of the present method. The paper is concluded in Section 7.

2. Overview of ADDG Based Equivalence Checking Method

The source and the transformed programs are represented as ADDGs in this work. Details about the model and the associated verification technique can be found in (Karfa et al. 2013b); here we concisely present only the essential concepts and then explain the equivalence checking method with the aid of an example. The vertex set in an ADDG comprises each individual array and each operation (corresponding to a statement) in a program, while the edge set captures the define-use dependencies between the arrays through the operations. *A program with static control flow, affine indices and bounds, valid schedule and single assignment form can be represented as an ADDG (Shashidhar 2008).* The elements of an output array are defined in terms of the elements of the input arrays by resolving the dependencies on the intermediate arrays, if any. The goal is to show that each element of an output array is defined identically in both the source and the transformed programs with respect to the input arrays. A left hand side (lhs) array may be defined using multiple right hand side (rhs) arrays in a statement enclosed in a loop and the whole array may be defined over multiple statements; therefore, we need to determine the definition domain of the lhs array (i.e., the elements of the lhs array defined by a statement) and the operand domain of each rhs array (i.e., the elements of the rhs array used in a statement) for a statement. Note that edges emanating from array nodes and entering operator nodes are called as *write edges* whereas, edges emanating from operator nodes and entering array nodes are called as *read edges*. Also, there may be transitive define-use dependence between the arrays. Consequently, transitive dependence over a sequence of statements needs to be computed. In this way, eventually the relations between the input array elements and the output array elements are computed. This mapping is termed as dependence mapping and the graph that represents the dependence is termed as IO-slice. One IO-slice may only define a subset of an output array. Therefore, there may be multiple IO-slices for an output array. Furthermore, it has to be identified how each output array is functionally dependent on the input arrays in an IO-slice as an algebraic expression e over the input arrays, such that e represents the symbolic value of the output array after execution of the IO-slice; this is denoted as data transformation of the IO-slice.

For computing and comparing dependence mappings of the IO-slices, the method of (Karfa et al. 2013b) relies on the Integer Set Library (ISL) (Verdoolaege 2010). ISL is a library for manipulating sets and relations of integer points bounded by linear constraints. On the other hand, the data transformations may be modified by the arithmetic transformations and in some cases by the loop transformations. Therefore, there must be a uniform representation for the arithmetic expressions that involve arrays and their indices. For this purpose, the normalization technique of (Sarkar and De Sarkar 1989), which is later improved upon in (Karfa et al. 2013b; Banerjee et al. 2014) by addition of further simplification rules, is used. Now, we explain the equivalence checking method of (Karfa et al. 2013b) with an example.

$for(k = 0; k < 94; k++)\{$
$\quad T1[k] = B[2*k+1] + C[2*k];$
$\quad T2[k] = A[k] * T1[k];\}$
$for(k = 5; k < 99; k++)\{$
$\quad T3[k] = A[k-5] - C[2*k-10];$
$\quad T4[k] = T3[k] * B[2*k-9];\}$
$for(k = 0; k < 94; k++)\{$
$\quad T5[k] = A[k] * C[2*k];$
$\quad Out[k] = T2[k] - T4[k+5] + T5[k];\}$

(a)

$for(k = 0; k < 94; k++)\{$
$\quad T[k] = 2*A[k] + B[2*k+1];$
$\quad Out[k] = C[2*k] * T[k];\}$

(b)

Figure 1. (a) Original program. (b) Transformed program.

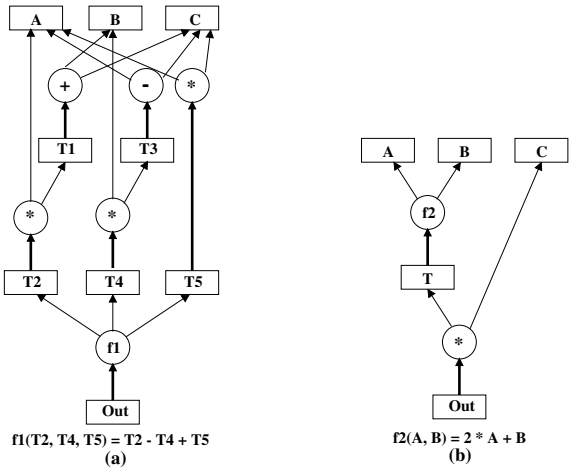


Figure 2. (a) ADDG of the original program. (b) ADDG of the transformed program.

EXAMPLE 1. Let us consider the programs of Figure 1 and their corresponding ADDGs in Figure 2. Both the programs actually compute $Out[k] = C[2*k]*(2*A[k] + B[2*k+1])$, $0 \leq k \leq 93$. It may be noted that both the ADDGs have only one IO-slice. Let the IO-slices of the source program and the transformed program be denoted as s and t, respectively. The method of (Karfa et al. 2013b) first extracts the data transformation of each IO-slice and the dependence mapping of each path of the IO-slices s and t. The data transformation r_s of the IO-slice s is $A*(B+C) - B*(A-C) + A*C$ and that of the IO-slice t, namely r_t, is $C*(2*A+B)$. The normalized representation of r_s is $1*A*B + (-1)*A*B + 1*A*C + 1*A*C + 1*B*C + 0$. In this normalized expression, the terms 1 and 2 can be eliminated because the dependence mappings from the output array Out to the arrays A and B are identical. In particular, the dependence mapping from Out to A is $M_{Out,A} = \{[k] \rightarrow [k] \mid 0 \leq k \leq 93\}$ and the dependence mapping from Out to B is $M_{Out,B} = \{[k] \rightarrow [2*k+1] \mid 0 \leq k \leq 93\}$ in both term 1 and term 2. Similarly, term 3 and term 4 of the normalized expression have the same dependence mappings from Out to A and the dependence mappings from Out to C. In particular, the dependence mappings from Out to A and from Out to C are $M_{Out,A} = \{[k] \rightarrow [k] \mid 0 \leq k \leq 93\}$ and $M_{Out,C} = \{[k] \rightarrow [2*k] \mid 0 \leq k \leq 93\}$, respectively. So, term 3 and term 4 can be combined. Therefore, after application of simplification rules, r_s becomes $2*A*C + 1*B*C + 0$. The normalized representation of r_t is $2*A*C + 1*B*C + 0$. Therefore, $r_s = r_t$. After simplification, the data transformations of the IO-slices consist of three input arrays including two occurrences of C. So, we need to check four dependence mappings, each one from Out to the input arrays A, B, $C^{(1)}$ and $C^{(2)}$. The respective dependence mappings are $M_{Out,A} = \{[k] \rightarrow [k] \mid 0 \leq k \leq 93\}$, $M_{Out,B} = \{[k] \rightarrow [2*k+1] \mid 0 \leq k \leq 93\}$, $M_{Out,C^{(1)}} = \{[k] \rightarrow [2*k] \mid 0 \leq k \leq 93\}$ and $M_{Out,C^{(2)}} = \{[k] \rightarrow [2*k] \mid 0 \leq k \leq 93\}$, respectively in the IO-slice s. It can be shown that $M_{Out,A}$, $M_{Out,B}$, $M_{Out,C^{(1)}}$ and $M_{Out,C^{(2)}}$ in IO-slice t are the same as those in the IO-slice s. So, the IO-slices s and t are equivalent. Hence, the ADDGs are equivalent and, in turn, the programs they represent are equivalent. ■

It is worth noting that there may be multiple IO-slices having identical data transformation due to transformations such as, loop fission. All the IO-slices having identical data transformation are clubbed together to form an IO-slice class; an IO-slice class has data transformation equivalent to that of any of its constituent IO-slice and the definition domain of the output array and the dependence mapping(s) for the input array(s) in an IO-slice class are obtained by taking a union of the individual definition domains and the respective dependence mappings for each input array. Two IO-slice classes, one from the source ADDG and the other from the transformed ADDG, are said to be equivalent if they have identical data transformations, definition domains and dependence mappings (these attributes are together termed as *characteristic formula*). Algorithm 1 presents the broad steps of the equivalence checking method reported in (Karfa et al. 2013b).

3. Extension of the Equivalence Checking Scheme to Handle Recurrences

The discussion given in the previous section clearly reveals that an ADDG without recurrence is basically a directed acyclic graph (DAG) which captures the data-dependence and the functional computation of an array-intensive program. Presence of recurrences introduces cycles into the ADDG and consequently, the equivalence checking strategy outlined so far fails. Consider, for example, the following code snippet:

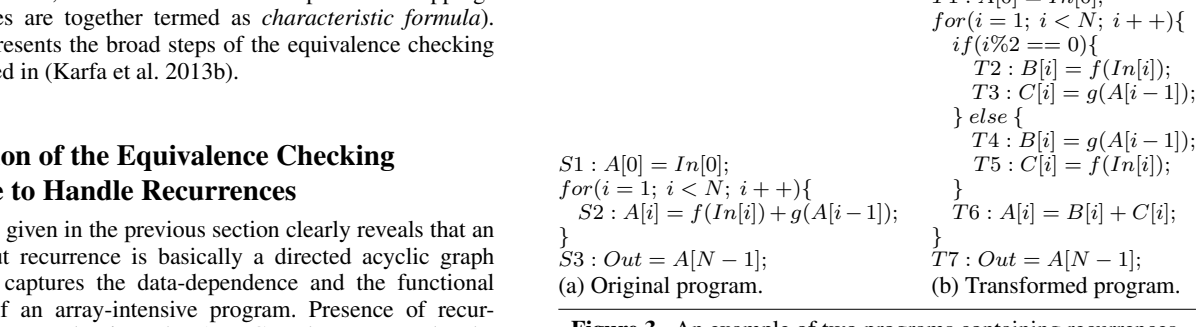
$A[0] = B[0];$
$for(i = 1; i < N; i++)$
$\quad A[i] = A[i-1] + B[i];$

The array element $A[i], i \geq 0$, has the value $\sum_{j=0}^{i} B[j]$, where the array B must have been defined previous to this code segment (otherwise, the program would not fulfill the requirement of having a valid schedule). As is evident from this example, it is not possible to compute the characteristic formula of a slice involving recurrence(s) using the present method. It is also not possible to obtain closed form representations of recurrences in general. Even for the restricted class of programs for which ADDGs can be constructed, it may not be always viable to obtain the corresponding closed form representations mechanically from the recurrences. As a solution, we consider separation of suitable subgraphs with cycles from the acyclic subgraphs of the ADDG. For the subgraphs with cycles, the back edges are removed to make them acyclic. Once equivalence of the resulting acyclic subgraphs from the two ADDGs is established, their corresponding original cyclic subgraphs may be replaced by identical uninterpreted functions. This process results in transforming the original ADDGs to DAGs. Now, the existing equivalence checking procedure of (Karfa et al. 2013b), as described in the previous section, can be employed to check equivalence of the transformed ADDGs. This technique to establish equivalence of subgraphs with cycles in ADDGs is now illustrated through the following example borrowed from existing literature (Verdoolaege et al. 2012).

Algorithm 1 ADDG_EQX (ADDG G_1, ADDG G_2)

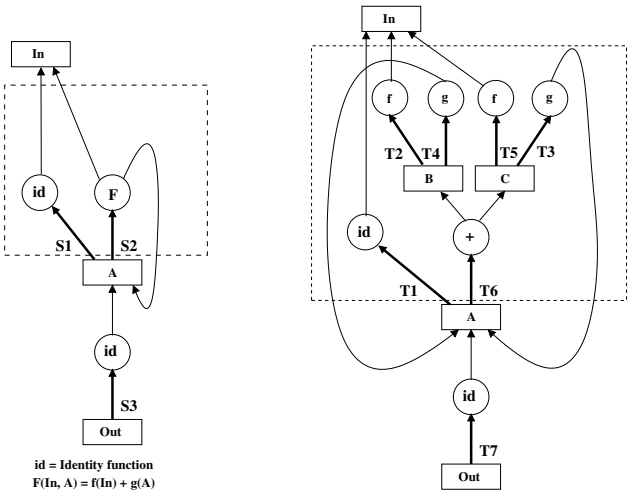
Inputs: Two ADDGs G_1 and G_2.
Outputs: Boolean value *true* if G_1 and G_2 are equivalent, *false* otherwise; in case of failure, it reports the possible source of non-equivalence.

1: Find the set of IO-slices in each ADDG; find the characteristic formulae of the IO-slices.
2: Use arithmetic simplification rule to the data transformation of the IO-slices of G_1 and G_2.
3: Obtain the IO-slice classes and their characteristic formulae in each ADDG; let $\mathcal{H}_{G_1}$ and $\mathcal{H}_{G_2}$ be the respective sets of IO-slice classes in both the ADDGs.
4: **for** each IO-slice class g_1 in $\mathcal{H}_{G_1}$ **do**
5: $\quad g_k = findEquivalentSliceClass(g_1, \mathcal{H}_{G_2})$.
$\quad$ /* This function returns the equivalent IO-slice class of g_1 in $\mathcal{H}_{G_2}$ if found; otherwise it returns NULL. */
6: $\quad$ **if** $g_k = NULL$ **then**
7: $\quad\quad$ **return** *false* and report g_1 as a possible source of non-equivalence.
8: $\quad$ **end if**
9: **end for**
10: Repeat the above loop by interchanging G_1 and G_2.
11: **return** *true*.



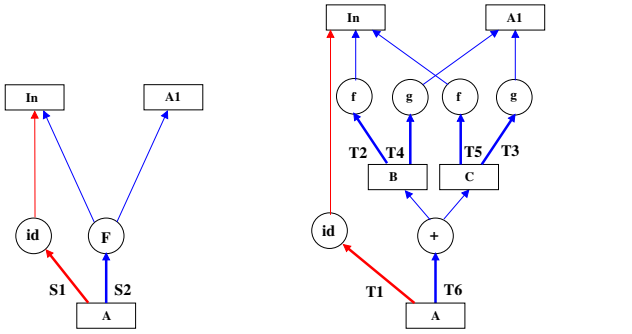
(a) Original program. (b) Transformed program.

Figure 3. An example of two programs containing recurrences.



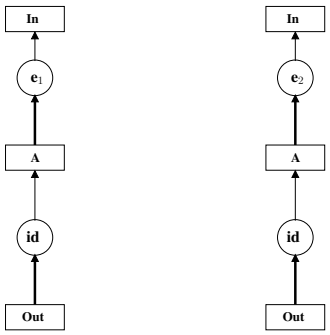
(a) Original ADDG. (b) Transformed ADDG.

Figure 4. ADDGs for the programs given in Figure 3.



(a) Original subADDG. (b) Transformed subADDG.

Figure 5. Modified subADDGs corresponding to subgraphs marked by the dotted lines in Figure 4.



(a) Original ADDG (b) Transformed ADDG

Figure 6. Modified ADDGs with new uninterpreted function for the programs given in Figure 3.

EXAMPLE 2. Let us consider the pair of equivalent programs involving recurrences shown in Figure 3. The corresponding ADDGs shown in Figure 4 have cycles since the array A has dependence upon itself.

Consider the *subgraphs containing the cycles* marked by the dotted lines in Figure 4(a) and Figure 4(b). Let the respective functional transformations computed by the subgraphs be $A[i] \Leftarrow e_1(In[i])$ and $A[i] \Leftarrow e_2(In[i]), 0 \leq i < N$. For equivalence, the equality $e_1(In[i]) = e_2(In[i])$ should hold for all $i, 0 \leq i < N$.

Suppose we proceed to prove the above equality by induction on i. For the basis case, therefore, $e_1(In[0]) = e_2(In[0])$ should hold. It may be noted that the slices indicated by the red edges in Figure 5(a) and Figure 5(b) depict respectively the functional transformations $A[0] \Leftarrow e_1(In[0])$ and $A[0] \Leftarrow e_2(In[0])$ and hence proving the basis case reduces to proving the equivalence of these two slice classes; we refer to such slices as *basis slices*; in fact, since, in general, there may be several cases under the proof of the basis step, we have *basis slice classes* together constituting a basis subgraph.

For the induction step, let the hypothesis be $e_1(In[i]) = e_2(In[i]), 0 \leq i < m$. We have to show that $e_1(In[m]) = e_2(In[m])$. Now, let the transformed array $A[0], \ldots, A[m-1]$ be designated as $A1[0], \ldots, A1[m-1]$. Specifically, the induction hypothesis permits us to assume that the parts of the array A over the index range $[0, m-1]$ is identically transformed and the induction step necessitates us to show that $A1[m]$ is equivalent based on this assumption. The slices indicated by the blue edges in Figure 5(a) and Figure 5(b) capture the respective transformations of the m-th element (for any m) and proving the inductive step reduces to proving the equivalence of these two slices; accordingly, these respective slices in the two ADDGs are referred to as *induction slice classes*. The general possibility of proof of the induction step by case analysis is manifested by having several induction slice classes constituting an induction subgraph.

Hence the method consists of breaking the cycles by removing the backward edges identified in a depth-first traversal from the output array vertex, by incorporating a new array of the same name in both the ADDGs and then proceeding as follows.

Removing the cycles will make the resulting subADDGs given in Figure 5(a) and Figure 5(b) DAGs thereby permitting application of the equivalence checking scheme explained in Section 2. In the sequel, we refer to this scheme as "ADDG_EQX" to distinguish it from its enhanced version for handling recurrences. The equivalence of these two subADDGs is established by showing the equivalence of the basis slice classes (i.e., the red edged slice classes of the subADDGs in Figure 5) first and then the equivalence of the induction slice classes (i.e., the blue edged slices). The first task is simple because they have identical data transformation and index domains in both the ADDGs.

The equivalence of the induction slice classes is shown as follows. For the ADDG shown in Figure 5(a), $M_{A,In} = \{[m] \rightarrow [m] \mid 1 \leq m < N\}$, $M_{A,A1} = \{[m] \rightarrow [m-1] \mid 1 \leq m < N\}$ and $r_{A,\{In,A1\}} = f(In) + g(A1)$. For the ADDG shown in Figure 5(b), $M_{B,In} = \{[m] \rightarrow [m] \mid \exists k \in \mathbb{Z}(m = 2 \times k), 1 \leq m < N\}$, $M_{C,A1} = \{[m] \rightarrow [m-1] \mid \exists k \in \mathbb{Z}(m = 2 \times k), 1 \leq m < N\}$, $M_{B,A1} = \{[m] \rightarrow [m-1] \mid \exists k \in \mathbb{Z}(m = 2 \times k + 1), 1 \leq m < N\}$, $M_{C,In} = \{[m] \rightarrow [m] \mid \exists k \in \mathbb{Z}(m = 2 \times k + 1), 1 \leq m < N\}$, $M_{A,B} = \{[m] \rightarrow [m] \mid 1 \leq m < N\}$, $M_{A,C} = \{[m] \rightarrow [m] \mid 1 \leq m < N\}$. Now we find that $r^{(1)}_{A,\{In,A1\}} = f(In) + g(A1)$ for the domain $\{[m] \mid \exists k \in \mathbb{Z}(m = 2 \times k), 1 \leq m < N\}$ and $r^{(2)}_{A,\{In,A1\}} = f(In) + g(A1)$ for the domain $\{[m] \mid \exists k \in \mathbb{Z}(m = 2 \times k + 1), 1 \leq m < N\}$; (the superfixes refer to the data transformations of the array elements referred through the terms in the rhs expressions;) note that the normalization technique accommodates the commutativity of the '+' operation. Since the data transformation is identical in both the domains, they constitute a slice class with domain $\{[m] \mid 1 \leq m < N\}$. Hence, we arrive at the same mapping and data transformation as that of Figure 5(a). Note that the method automatically extracts two pairs of slice classes from the subADDGs in Figure 5 – one cor-

$for(i = 0; i <= 50; i++)$
$S1 : Z[2 * i] = In[i];$
$for(i = 1; i <= 99; i+ = 2)$
$S2 : Z[i] = Z[i - 1] + Z[i + 1];$

Figure 7. An example where back edges exist in the absence of recurrence.

responding to the basis cases and the other corresponding to the inductions.

Having established that the two subgraphs with cycles in the respective ADDGs are equivalent, we construct another pair of modified ADDGs as shown in Figure 6. The ADDGs in Figure 6(a) and Figure 6(b) contain the new uninterpreted functions e_1 and e_2, respectively, where $e_1 = e_2$. It is to be noted that scalar variables, such as Out, are treated as array variables of unit dimension, e.g., Out is considered as $Out[0]$. Now showing equivalence of the two entire ADDGs of Figure 6 (which are basically DAGs) is straightforward by the ADDG equivalence checking method (ADDG_EQX) given in Section 2. ■

Before formalizing the above mechanism, we underline the fact that while recurrences imply back edges in an ADDG, the converse is not true; we may have back edges even when there is no recurrence. This is illustrated through the following example.

EXAMPLE 3. Consider the program segment given in Figure 7. While constructing the ADDG corresponding to this program, there will be two back edges in the ADDG for statement $S2$ corresponding to the two rhs terms $Z[i-1]$ and $Z[i+1]$; however, note that statement $S2$ does not have any data dependency on itself since the whole of its operand domain has already been defined in statement $S1$. These cases can be handled within the framework of the equivalence checking method described in Section 2 by permitting computation of transitive dependence through the cycles only once and then continuing through the other edges emanating from the array vertex Z. ■

Example 3 leads us to the following definition.

DEFINITION 1 (Recurrence Array Vertex). *An array vertex Z which is identified as the destination of a back edge (of a cycle c) during a depth-first traversal of an ADDG starting from an output array such that ${}_cD_Z \bigcap {}_cU_Z \neq \emptyset$ is called a* recurrence array vertex.

Let us first elaborate what the above definition entails. Obviously, recurrences lead to cycles and a back edge basically identifies the source of the recurrence namely, the array Z which has been defined in terms of itself. Let the statement leading to a recurrence in some program be
$S : Z[l] = f(Y[r_0], Z[r_1], Z[r_2], Z[r_3]);$
Let the symbol $Z^{(i)}$ represent the i-th occurrence of Z in the rhs of S. For Z to qualify as a recurrence array vertex, $({}_SM^{(d)}_{Z,Z^{(1)}}(I_S) = {}_SD_Z) \bigcap ({}_SM^{(u)}_{Z,Z^{(1)}}(I_S) = {}_SU_{Z^{(1)}}) \neq \emptyset$ or ${}_SD_Z \bigcap {}_SU_{Z^{(2)}} \neq \emptyset$ or ${}_SD_Z \bigcap {}_SU_{Z^{(3)}} \neq \emptyset$ should hold (other arrays such as, Y, do not participate in this procedure). In case the recurrence occurs through a cycle c involving multiple statements, we shall have to consider the transitive dependences over c. This check for overlapping definition and operand domains helps in segregating cases of *true* recurrences from those as shown in Example 3.

As indicated in Example 2, for each recurrence array vertex, we need to find a minimum subgraph with cycle(s) so that the computed values of the recurrence array elements can be captured by an uninterpreted function with proper arguments. Finding such minimum subgraphs with cycles in an ADDG involves prior identification of strongly connected components (SCCs) in the ADDG.

DEFINITION 2. *(Basis Subgraph $\mathcal{B}(Z, \{Y^i_1, \ldots, Y^i_{k_i}, k_i \geq 1, 1 \leq i \leq m\})$ for the recurrence array vertex Z). Let $\mathcal{C}(Z)$ be an SCC in the ADDG having Z as the recurrence array vertex. Let $e_i = \langle Z, f_i \rangle, 1 \leq i \leq m$, be the m write edges which are not contained in $\mathcal{C}(Z)$; also, let $\langle f_i, Y^i_1 \rangle, \ldots, \langle f_i, Y^i_{k_i} \rangle, k_i \geq 1$, be all the existing read edges emanating from the operator vertex f_i in the ADDG. The subgraph $\mathcal{B}(Z, \{Y^i_1, \ldots, Y^i_{k_i}, k_i \geq 1, 1 \leq i \leq m\})$ containing the array vertex Z, all such operator vertices $f_i, 1 \leq i \leq m$, and the corresponding array vertices $Y^i_1, \ldots, Y^i_{k_i}$ along with the connecting edges $\langle Z, f_i \rangle, \langle f_i, Y^i_1 \rangle, \ldots, \langle f_i, Y^i_{k_i} \rangle$ is called a* basis subgraph. *For brevity, we represent such a basis subgraph as $\mathcal{B}(Z, \overline{Y})$, where $\overline{Y}$ represents the set $\{Y^i_1, \ldots, Y^i_{k_i}, k_i \geq 1, 1 \leq i \leq m\}$ ordered using a consistent ordering of all the array names used in the program.*

Note that the subgraph $\mathcal{B}(Z, \overline{Y})$ is the minimum subgraph that encompasses the basis step(s) of a recurrence involving the recurrence array Z in the program. The arrays $Y^i_j, 1 \leq j \leq k_i, 1 \leq i \leq m$, in $\overline{Y}$ need not be all distinct. If there are multiple basis steps with different data transformations, then there will be multiple basis slice classes; it is to be noted that all such basis slice classes are collectively covered in $\mathcal{B}(Z, \overline{Y})$.

DEFINITION 3. *(Induction Subgraph $\mathcal{D}(Z, \{T^i_1, \ldots, T^i_{k_i}, k_i \geq 1, 1 \leq i \leq m\}, \{W^i_1, \ldots, W^i_{k_i}, k_i \geq 1, 1 \leq i \leq n\})$ for the recurrence array vertex Z). Let $\mathcal{C}(Z)$ be an SCC in the ADDG having Z as the recurrence vertex.*
1) Let $e_i = \langle X_i, h_i \rangle, 1 \leq i \leq m$, be the m write edges where $X_i (\neq Z)$ is an array vertex in $\mathcal{C}(Z)$ such that e_i is not contained in $\mathcal{C}(Z)$; also, let $\langle h_i, T^i_1 \rangle, \ldots, \langle h_i, T^i_{k_i} \rangle, k_i \geq 1, 1 \leq i \leq m$, be all the existing read edges emanating from the operator vertex h_i in the ADDG. Let the connected subgraph containing $\mathcal{C}(Z)$ along with the vertices $h_i, T^i_1, \ldots, T^i_{k_i}, k_i \geq 1, 1 \leq i \leq m$, be designated as $\mathcal{D}'(Z, T^i_1, \ldots, T^i_{k_i}, k_i \geq 1, 1 \leq i \leq m)$.
2) Let $g_i, 1 \leq i \leq n$, be n operator vertices in $\mathcal{C}(Z)$ such that there is a read edge e emanating from g_i which is not contained in $\mathcal{C}(Z)$; also, let $\langle g_i, W^i_1 \rangle, \ldots, \langle g_i, W^i_{k'_i} \rangle, k'_i \geq 1, 1 \leq i \leq n$, be all the existing read edges emanating from the operator vertex g_i in the ADDG which are not already covered in $\mathcal{C}(Z)$; the connected subgraph containing $\mathcal{D}'(Z, T^i_1, \ldots, T^i_{k_i}, k_i \geq 1, 1 \leq i \leq m)$ along with all such array vertices $W^i_1, \ldots, W^i_{k'_i}, k'_i \geq 1, 1 \leq i \leq n$, is called an induction subgraph *and is represented as $\mathcal{D}(Z, \overline{V})$, where $\overline{V}$ is the set $\{T^i_1, \ldots, T^i_{k_i}, k_i \geq 1, 1 \leq i \leq m\} \bigcup \{W^i_1, \ldots, W^i_{k'_i}, k'_i \geq 1, 1 \leq i \leq n\})$ ordered using a uniform ordering of all the array names used in the program.*

Note that the subgraph $\mathcal{D}(Z, \overline{V})$ is the minimum subgraph that encompasses the induction step(s) of a recurrence involving the array Z in the program. Once the induction subgraph is found, we do not need to distinguish between the arrays marked as T's and W's; hence they are jointly denoted as $\overline{V}$. The arrays $T^i_j, 1 \leq j \leq k_i, 1 \leq i \leq m$, and $W^i_j, 1 \leq j \leq k_i, 1 \leq i \leq n$, in $\overline{V}$ need not be all distinct. If there are multiple induction steps with different data transformations, then there will be multiple induction slice classes; it is to be noted that all such induction slice classes are collectively covered in $\mathcal{D}(Z, \overline{V})$.

DEFINITION 4. *(Recurrence Subgraph $\mathcal{E}(Z, \overline{Y}, \overline{V})$ for the basis subgraph $\mathcal{B}(Z, \overline{Y})$ and the induction subgraph $\mathcal{D}(Z, \overline{V})$). The subgraph $\mathcal{E}(Z, \overline{Y}, \overline{V})$ which is obtained by combining a basis subgraph $\mathcal{B}(Z, \overline{Y})$ and an induction subgraph $\mathcal{D}(Z, \overline{V})$ having the same recurrence array vertex Z in an ADDG is called a* recurrence subgraph.

$Z[1] = f_1(Y_1^1[j_1^1], \ldots, Y_{k_1}^1[j_{k_1}^1]);$

$\vdots$

$Z[m] = f_m(Y_1^m[j_1^m], \ldots, Y_{k_m}^m[j_{k_m}^m]);$

$for(i = m+1, i \leq N; i++)\ \{$

$\quad if(\ Cond(i)\ )\ \{$

$\quad\quad X_1[i] = g_1(Z[0], \ldots, Z[m], W_1^1[i], \ldots, W_{l_1}^1[i]);$

$\quad\quad \vdots$

$\quad\quad X_p[i] = g_p(Z[0], \ldots, Z[m], W_1^p[i], \ldots, W_{l_p}^p[i]);$

$\quad\}\ else\ \{$

$\quad\quad X_1[i] = h_1(T_1^1[i], \ldots, T_{d_1}^1[i]);$

$\quad\quad \vdots$

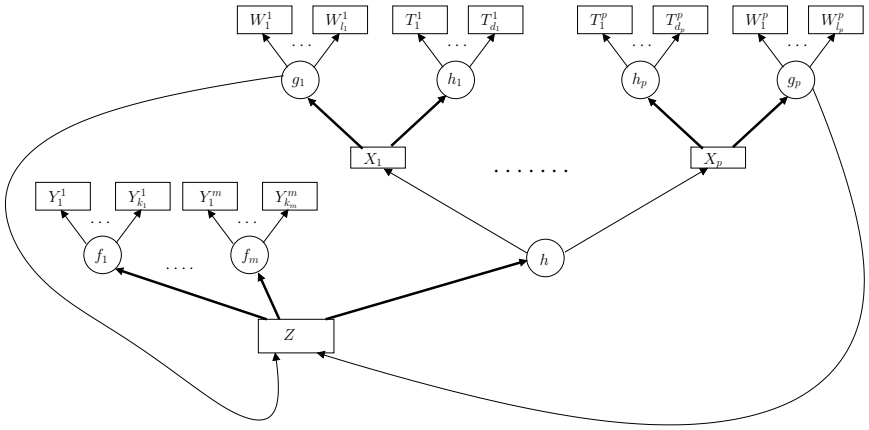
$\quad\quad X_p[i] = h_p(T_1^p[i], \ldots, T_{d_p}^p[i]);$

$\quad\}$

$\quad Z[i] = h(X_1[i], \ldots, X_p[i]);$

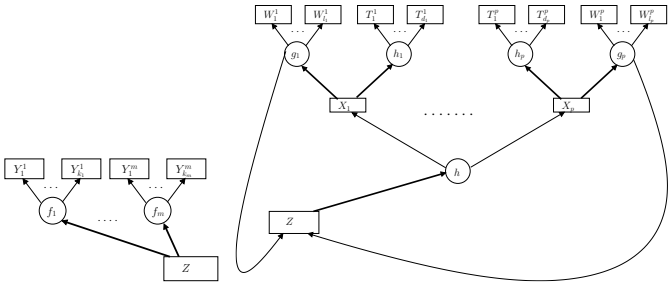
$\}$

(a)



(b)

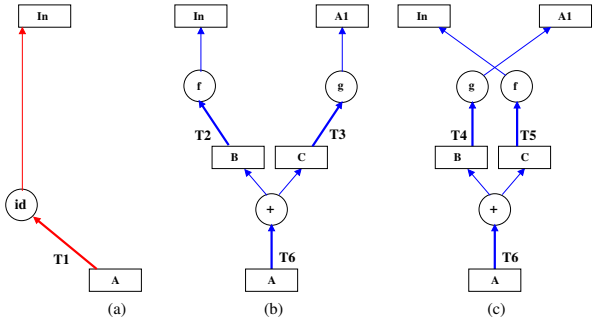
Figure 8. (a) Original program. (b) Corresponding ADDG.



(a) (b)

Figure 9. (a) Basis subgraph. (b) Induction subgraph corresponding to Figure 8.

Note that the subgraph $\mathcal{E}(Z, \overline{Y}, \overline{V})$ is the minimum subgraph that encompasses the recurrence involving the array Z in the program.



(a) (b) (c)

Figure 10. (a) Basis slice. (b) Valid induction slice 1. (c) Valid induction slice 2, corresponding to Figure 5(b).

Figure 8(a) shows a schema of a program involving a recurrence and Figure 8(b) shows its ADDG representation; note that we have shown a single back edge from the operator vertices g_1 and g_p instead of m such back edges for clarity; the corresponding basis subgraph and the induction subgraph have been given in Figure 9(a) and Figure 9(b), respectively.

In order to compare two recurrence subgraphs, $\mathcal{E}_1(Z, \overline{Y_1}, \overline{V_1})$ and $\mathcal{E}_2(Z, \overline{Y_2}, \overline{V_2})$, say, from two different ADDGs, and represent them as uninterpreted functions subsequently, the arguments $\overline{Y}$ and $\overline{V}$ must be identical, i.e., $\overline{Y_1} = \overline{Y_2}$ and $\overline{V_1} = \overline{V_2}$ have to hold. In case they do not hold, we identify the *uncommon arrays*, i.e., arrays that appear in only one of the ADDGs but not both, occurring in $\overline{Y_i}$ and $\overline{V_i}, i \in \{1, 2\}$; for each such array, T say, we identify the minimum subgraph $G_T(T, \overline{X_T})$, say (with T as the start array vertex and $\overline{X_T}$ as the terminal array vertices), where $\overline{X_T}$ comprises only *common arrays*, i.e., arrays that appear in both the ADDGs. If $T \in \overline{Y_i}$ ($\overline{V_i}$), then the basis subgraph $\mathcal{B}_i(Z, \overline{Y_i})$ (induction subgraph $\mathcal{D}_i(Z, \overline{V_i})$) is extended by including G_T and treat the resulting subgraph as the basis subgraph (induction subgraph) for establishing equivalence of the two ADDGs. Algorithm 2 captures this process, where the module "extendSubgraph" basically extends the passed recurrence subgraph along each uncommon array vertex A by including all edges of the form $\langle A, f_i \rangle$ and $\langle f_i, X_i \rangle, 1 \leq i \leq p$,; this process of extension is repeated for each uncommon array vertex $X_i, 1 \leq i \leq p$, until each of the terminal nodes of the extended subgraph is a common array vertex.

Algorithm 2 extendRecurrenceSubgraph (ADDG G, Subgraph $\mathcal{E}(Z, \overline{Y}, \overline{V})$, List L)

Inputs: An ADDG G, a recurrence subgraph $\mathcal{E}(Z, \overline{Y}, \overline{V})$ and a list L of common arrays (including the input and the output arrays).

Outputs: An extended recurrence subgraph $\mathcal{E}'(Z, \overline{Y}', \overline{V}')$ whose each terminal array vertex belonging to $\overline{Y}'$ or $\overline{V}'$ is a common array belonging to L.

1: Set $\mathcal{A} \leftarrow \{\overline{Y}\} \bigcup \{\overline{V}\}$.
2: Let $\overline{Y}' \leftarrow \overline{Y}$; $\overline{V}' \leftarrow \overline{V}$; $\mathcal{B}'(Z, \overline{Y}') \leftarrow \mathcal{B}(Z, \overline{Y})$ and $\mathcal{D}'(Z, \overline{V}') \leftarrow \mathcal{D}(Z, \overline{V})$.
3: **for** all $T \in (\mathcal{A} - L)$ **do**
4: $\quad G_T(T, \overline{X_T}) \leftarrow$ extendSubgraph $(G, \mathcal{E}, T)$.
5: $\quad$ **if** $T \in \overline{Y}$ **then**
6: $\quad\quad$ Let $\overline{Y}' \leftarrow \overline{Y}'\{T \leftarrow \overline{X_T}\}$;
$\quad\quad$ /* In the ordered tuple $\overline{Y}'$, T is substituted by $\overline{X_T}$ and then the whole tuple is ordered using the consistent ordering of all the array names of L. */
$\quad\quad \mathcal{B}'(Z, \overline{Y}') \leftarrow$ compose $(\mathcal{B}'(Z, \overline{Y}'), G_T(T, \overline{X_T}))$.
7: $\quad$ **else**
8: $\quad\quad$ Let $\overline{V}' \leftarrow \overline{V}'\{U \leftarrow \overline{X_U}\}$;
$\quad\quad \mathcal{D}'(Z, \overline{V}') \leftarrow$ compose $(\mathcal{D}'(Z, \overline{V}'), G_T(T, \overline{X_T}))$.
9: $\quad$ **end if**
10: **end for**
11: Let $\overline{Y} \leftarrow \overline{Y}'$; $\overline{V} \leftarrow \overline{V}'$;
$\mathcal{E}'(Z, \overline{Y}', \overline{V}') \leftarrow$ compose $(\mathcal{B}'(Z, \overline{Y}'), \mathcal{D}'(Z, \overline{V}'))$.
12: **return** $\mathcal{E}'(Z, \overline{Y}', \overline{V}')$.

For example, from Figure 4(b), we shall get an SCC $\mathcal{C}(A)$, say, comprising the vertices (A, $+$, B, C, g_{T4}, g_{T3}) with A designated as the recurrence array vertex; the corresponding basis subgraph $\mathcal{B}(A, In)$ comprises the vertices $\{A, id_{T1}, In\}$ and the connecting edges; the corresponding induction subgraph $\mathcal{D}(A, In)$ comprises $\mathcal{C}(A)$, the vertices $\{f_{T_2}, f_{T_5}, In\}$ and the connecting edges. The recurrence subgraph $\mathcal{E}(A, In)$ is obtained by taking a union of the vertices and edges in $\mathcal{B}(A, In)$ and $\mathcal{D}(A, In)$. Although B and C are uncommon arrays, extraction of the minimum induction subgraph itself will not identify them as terminal arrays due to clause (1) of Definition 3; no extra extension step will be needed.

The method initially finds five slices corresponding to the sub-ADDG given in Figure 5(b) as given below (in terms of the involved statements): $\mathscr{G}_1 = \langle T1\rangle$, $\mathscr{G}_2 = \langle T6, T2, T5\rangle$, $\mathscr{G}_3 = \langle T6, T2, T3\rangle$, $\mathscr{G}_4 = \langle T6, T4, T5\rangle$, $\mathscr{G}_5 = \langle T6, T4, T3\rangle$. However, out of these five slices, slices $\mathscr{G}_2$ and $\mathscr{G}_5$ are deemed invalid because they contain conflicting conditions, namely $i\%2 == 0$ and $i\%2 != 0$, in the dependence mapping ${}_{\mathscr{G}_2}M_{A,In}$ and ${}_{\mathscr{G}_5}M_{A,A1}$, respectively. Thus, the validity of an induction slice is to be checked before it is added to the respective set of induction slice classes. Note that the function module "compose" prunes such invalid slices. The basis slice and the valid induction slices have been shown in Figure 10.

It is to be noted that the method of (Verdoolaege et al. 2012) resolves the equivalence of the data transformations of the two programs shown in Figure 3 by comparing all possible permutations of the involved commutative operator ($+$ in this case). This method, however, would not have been successful in establishing equivalence if the statement $S2$ in Figure 3(a) had been replaced by $S2 : A[i] = f(In[i]) + g(A[i-1]) + 4$; and 2 had been added to each of the rhs expressions in the statements $T2, T3, T4$ and $T5$ in Figure 3(b) (e.g., statement $T2$ in Figure 3(b) was replaced by $T2 : B[i] = f(In[i]) + 2$;). Our method, unlike that of (Verdoolaege et al. 2012) which only checks syntactic equivalence of the operands (except for commutative and associative operations), can show the equivalence even under such a scenario since it employs the normalization technique of (Karfa et al. 2013b) to check equivalence of arithmetic transformations.

Our overall equivalence checking method is now presented in Algorithm 3, where the function module "getRecurrenceSubgraph" is responsible for identifying true recurrences and excluding those SCCs which may have been generated by cases such as in Example 3; the module "obtainDAG" takes a (cyclic) recurrence subgraph $\mathcal{E}(Z, \overline{Y}, \overline{V})$ as input, produces a subgraph $\mathcal{E}'(Z, \overline{Y}, \overline{V}, Z1)$ by replacing all back edges of the form $\langle f, Z\rangle$ by $\langle f, Z1\rangle$ in $\mathcal{E}'$ and copies the dependence mappings ${}_{\varepsilon_1}M_{Z,Z}$ as ${}_{\varepsilon'_1}M_{Z,Z1}$, where $Z1$ is a new array vertex (not already present in the ADDG); the function module "replaceRecurrenceSubgraphByUF" takes an ADDG G, a recurrence subgraph $\mathcal{E}(Z, \overline{Y}, \overline{V})$ and an uninterpreted function f as input and replaces the recurrence subgraph $\mathcal{E}(Z, \overline{Y}, \overline{V})$ in G by a directed acyclic subgraph $\mathcal{H}(Z, \overline{Y}, \overline{V})$ having a start vertex Z, terminal vertices $\overline{Y}$ and $\overline{V}$, a single operator f, write edge $\langle Z, f\rangle$ and a set $\{\langle f, Y_i\rangle, Y_i \in \overline{Y}\} \bigcup \{\langle f, V_i\rangle, V_i \in \overline{V}\}$ of read edges; the mappings ${}_{\mathcal{H}}M_{Z,\overline{Y}}$ and ${}_{\mathcal{H}}M_{Z,\overline{V}}$ are kept the same as ${}_{\varepsilon}M_{Z,\overline{Y}}$ and ${}_{\varepsilon}M_{Z,\overline{V}}$, respectively.

Algorithm 3 equivalenceChecker (ADDG G_1, ADDG G_2)

Inputs: Two ADDGs G_1 and G_2.
Outputs: Boolean value *true* if G_1 and G_2 are equivalent, *false* otherwise; in case of failure, it reports the possible source of non-equivalence.

1: Set $L \leftarrow$ findCommonArrays (G_1, G_2); $G'_i \leftarrow G_i, i \in \{1, 2\}$.
2: Set $\mathcal{C}_i$ of SCCs $\leftarrow$ findStronglyConnectedComponents (G_i), $i \in \{1, 2\}$.
3: **for** each SCC $c_1(Z_1) \in \mathcal{C}_1$ **do**
4: $e_1(Z_1, \overline{Y_1}, \overline{V_1}) \leftarrow$ getRecurrenceSubgraph ($G_1, c_1(Z_1)$); $e'_1(Z_1, \overline{Y_1}', \overline{V_1}') \leftarrow$ extendRecurrenceSubgraph (G_1, $e_1(Z_1, \overline{Y_1}, \overline{V_1}), L$); $e''_1(Z_1, \overline{Y_1}', \overline{V_1}', Z1_1) \leftarrow$ obtainDAG ($e'_1(Z_1, \overline{Y_1}', \overline{V_1}')$).
5: **end for**
6: **for** each $c_2(Z_2) \in \mathcal{C}_2$ **do**
7: $e_2(Z_2, \overline{Y_2}, \overline{V_2}) \leftarrow$ getRecurrenceSubgraph ($G_2, c_2(Z_2)$); $e'_2(Z_2, \overline{Y_2}', \overline{V_2}') \leftarrow$ extendRecurrenceSubgraph (G_2, $e_2(Z_2, \overline{Y_2}, \overline{V_2}), L$); $e''_2(Z_2, \overline{Y_2}', \overline{V_2}', Z1_2) \leftarrow$ obtainDAG ($e'_2(Z_2, \overline{Y_2}', \overline{V_2}')$).
8: **end for**
9: **for** each $e''_1(Z_1, \overline{Y_1}', \overline{V_1}', Z1_1)$ **do**
10: **for** each $e''_2(Z_2, \overline{Y_2}', \overline{V_2}', Z1_2)$ **do**
11: **if** $Z_1 = Z_2 \wedge \overline{Y_1}' = \overline{Y_2}' \wedge \overline{V_1}' = \overline{V_2}' \wedge$ ADDG_EQX ($e''_1(Z_1, \overline{Y_1}', \overline{V_1}', Z1_1)$, $e''_2(Z_2, \overline{Y_2}', \overline{V_2}', Z1_2)$) $\equiv$ *true* **then**
12: $G'_1 \leftarrow$ replaceRecurrenceSubgraphByUF (G'_1, $e'_1(Z_1, \overline{Y_1}', \overline{V_1}')$, f). /* where f is some new uninterpreted function */
13: $G'_2 \leftarrow$ replaceRecurrenceSubgraphByUF (G'_2, $e'_2(Z_2, \overline{Y_2}', \overline{V_2}')$, f). /* here f is the same uninterpreted function as in the previous step */
14: **end if**
15: **end for**
16: **if** no match for $e''_1(Z_1, \overline{Y_1}', \overline{V_1}', Z1_1)$ is found **then**
17: **return** *false* and report $e''_1(Z_1, \overline{Y_1}', \overline{V_1}', Z1_1)$ as a possible source of non-equivalence.
18: **end if**
19: **end for**
20: **if** some $e''_2(Z_2, \overline{Y_2}', \overline{V_2}', Z1_2)$ exists which is not found to have equivalence with any $e''_1(Z_1, \overline{Y_1}', \overline{V_1}', Z1_1)$ **then**
21: **return** *false* and report $e''_2(Z_2, \overline{Y_2}', \overline{V_2}', Z1_2)$ as a possible source of non-equivalence.
22: **end if**
23: **if** ADDG_EQX (G'_1, G'_2) $\equiv$ *true* **then**
24: **return** *true*.
25: **else**
26: **return** *false* and report the slice class in an ADDG which has no equivalent slice class in the other ADDG. /* NB: Our method is not complete and hence the output value *false* implies possible non-equivalence. */
27: **end if**

4. Correctness and Complexity

4.1 Correctness

THEOREM 1 (Soundness). *Let $\mathcal{E}_1(Z, \overline{Y}, \overline{V})$ be a recurrence subgraph of the ADDG G_1 of the source program P_1 and $\mathcal{E}_2(Z, \overline{Y}, \overline{V})$ be a recurrence subgraph of the ADDG G_2 of the transformed program P_2 with identical parameters $Z, \overline{Y}$ and $\overline{V}$. Let $\mathcal{E}_1(Z, \overline{Y}, \overline{V})$ have the basis subgraph $\mathcal{B}_1(Z, \overline{Y})$ and the induction subgraph $\mathcal{D}_1(Z, \overline{V})$; similarly, let $\mathcal{B}_2(Z, \overline{Y})$ and $\mathcal{D}_2(Z, \overline{V})$ respectively be the basis and the induction subgraphs of $\mathcal{E}_2(Z, \overline{Y}, \overline{V})$. Let $\Sigma_1 = \{\overline{Y}[d_{\overline{Y}}], d_{\overline{Y}} \in {}_{\mathcal{B}_1}U_{\overline{Y}}$, the used domain of $\overline{Y}$ in $\mathcal{B}_1\} \biguplus \{\overline{V}[d_{\overline{V}}], d_{\overline{V}} \in {}_{\mathcal{D}_1}U_{\overline{V}}$, the used domain of $\overline{V}$ in $\mathcal{D}_1\}$. Similarly, let $\Sigma_2 = \{\overline{Y}[d_{\overline{Y}}], d_{\overline{Y}} \in {}_{\mathcal{B}_2}U_{\overline{Y}}$, the used domain of $\overline{Y}$ in $\mathcal{B}_2\} \biguplus \{\overline{V}[d_{\overline{V}}], d_{\overline{V}} \in {}_{\mathcal{D}_2}U_{\overline{Y}}$, the used domain of $\overline{V}$ in $\mathcal{D}_2\}$. Let the function defined over the elements of Σ_1 yielding values for $Z[d], d \in {}_{\varepsilon_1}D_Z$, be represented as e_1 and the function defined over the elements of Σ_2 yielding values for $Z[d], d \in {}_{\varepsilon_2}D_Z$, be represented as e_2. Let the directed acyclic ADDGs corresponding to the subgraphs $\mathcal{D}_1(Z, \overline{V})$ and $\mathcal{D}_2(Z, \overline{V})$ be $\mathcal{D}'_1(Z, \overline{V}, Z1)$ and $\mathcal{D}'_2(Z, \overline{V}, Z1)$, respectively. If the equivalence checker ADDG_EQX ascertains that $\mathcal{B}_1(Z, \overline{Y}) \simeq \mathcal{B}_2(Z, \overline{Y})$ and $\mathcal{D}'_1(Z, \overline{V}, Z1) \simeq \mathcal{D}'_2(Z, \overline{V}, Z1)$, then $e_1 = e_2$.*

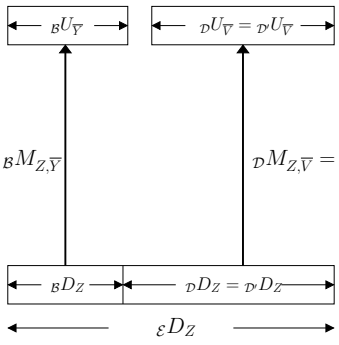


Figure 11. Relationship between different domains.

Proof: Figure 11 shows the relationship between different domains in a recurrence subgraph. Consider any element $d \in {}_{\mathcal{E}_1}D_Z$. By the single assignment (SA) property of the program P_1, ${}_{\mathcal{B}_1}D_Z$ and ${}_{\mathcal{D}_1}D_Z$ constitute a partition of ${}_{\mathcal{E}_1}D_Z$; hence, we have the following two *mutually exclusive* cases: (1) $d \in {}_{\mathcal{B}_1}D_Z$ and (2) $d \in {}_{\mathcal{D}_1}D_Z$. We carry out proof by cases.

Case 1 [$d \in {}_{\mathcal{B}_1}D_Z$]: Here,

$$\begin{aligned} Z[d] &= e_1(\overline{Y}[d_{\overline{Y}}]), \text{ for some } d_{\overline{Y}} = {}_{\mathcal{B}_1}M_{Z,\overline{Y}}(d) \in {}_{\mathcal{B}_1}U_{\overline{Y}} \\ &= r_{\mathcal{B}_1}(\overline{Y}[d_{\overline{Y}}]), \text{ where } r_{\mathcal{B}_1} \text{ is the data transformation of the subgraph } \mathcal{B}_1 \\ &= r_{\mathcal{B}_2}(\overline{Y}[d'_{\overline{Y}}]), \text{ for some } d'_{\overline{Y}} = {}_{\mathcal{B}_2}M_{Z,\overline{Y}}(d') \in {}_{\mathcal{B}_2}U_{\overline{Y}} \\ &= e_2(\overline{Y}[d'_{\overline{Y}}]), \text{ since the equivalence checker ADDG\_EQX ascertains } \mathcal{B}_1(Z,\overline{Y}) \simeq \mathcal{B}_2(Z,\overline{Y}), \text{ it must have found } r_{\mathcal{B}_1} = r_{\mathcal{B}_2} \text{ over the domain } \{\overline{Y}[d_{\overline{Y}}], d_{\overline{Y}} \in {}_{\mathcal{B}_1}U_{\overline{Y}} = {}_{\mathcal{B}_2}U_{\overline{Y}}\} \text{ and } {}_{\mathcal{B}_1}M_{Z,\overline{Y}} = {}_{\mathcal{B}_2}M_{Z,\overline{Y}} \text{ over the domains } {}_{\mathcal{B}_1}D_Z = {}_{\mathcal{B}_1}D_Z \end{aligned}$$

Thus, $d_{\overline{Y}} = {}_{\mathcal{B}_1}M_{Z,\overline{Y}}(d) = {}_{\mathcal{B}_2}M_{Z,\overline{Y}}(d') = d'_{\overline{Y}}$. Hence, from the soundness of ADDG_EQX equivalence checker, $e_1(\overline{Y}[d_{\overline{Y}}]) = e_2(\overline{Y}[d_{\overline{Y}}])$, $\forall d_{\overline{Y}} \in {}_{\mathcal{B}_1}U_{\overline{Y}} = {}_{\mathcal{B}_2}U_{\overline{Y}}$.

Case 2 [$d \in {}_{\mathcal{D}_1}D_Z$]: Here,

$$\begin{aligned} Z[d] &= e_1(\overline{V}[d_{\overline{V}}]), \text{ where } d_{\overline{V}} = {}_{\mathcal{D}_1}M_{Z,\overline{V}}(d) = {}_{\mathcal{D}'_1}M_{Z,\overline{V}}(d) \in {}_{\mathcal{D}_1}U_{\overline{V}}, \text{ since by construction of the directed acyclic version } \mathcal{D}'_1 \text{ of } \mathcal{D}_1, \ {}_{\mathcal{D}_1}M_{Z,\overline{V}} = {}_{\mathcal{D}'_1}M_{Z,\overline{V}} \text{ over the domain } {}_{\mathcal{D}_1}D_Z = {}_{\mathcal{D}'_1}D_Z \\ &= r_{\mathcal{D}'_1}(\overline{V}[d_{\overline{V}}], Z1[d_{Z1}]), \text{ for some } d_{Z1} \in {}_{\mathcal{D}'_1}U_{Z1} \quad \ldots(i) \end{aligned}$$

However, ${}_{\mathcal{D}'_1}U_{Z1} (= {}_{\mathcal{D}_1}U_{Z1}) \subseteq {}_{\mathcal{B}_1}D_Z \bigcup {}_{\mathcal{D}'_1}D_Z$ (owing to recurrence). Because of SA property of the program(s), ${}_{\mathcal{B}_1}D_Z \bigcap {}_{\mathcal{D}'_1}D_Z = \emptyset$. Hence we have the following two (mutually exclusive) subcases: (2.1) $d_{Z1} \in {}_{\mathcal{B}_1}D_Z$ and (2.2) $d_{Z1} \in {}_{\mathcal{D}'_1}D_Z$.

Subcase 2.1 [$d_{Z1} \in {}_{\mathcal{B}_1}D_Z$]: So, continuing from (i) we have,

$$\begin{aligned} Z[d] &= e_1(\overline{V}[d_{\overline{V}}]) \\ &= r_{\mathcal{D}'_1}(\overline{V}[d_{\overline{V}}], Z1[d_{Z1}]) \\ &= r_{\mathcal{D}'_1}(\overline{V}[d_{\overline{V}}], r_{\mathcal{B}_1}(\overline{Y}[d_{\overline{Y}}])) \\ &= r_{\mathcal{D}'_2}(\overline{V}[d_{\overline{V}}], r_{\mathcal{B}_2}(\overline{Y}[d_{\overline{Y}}])) \\ &= e_2(\overline{V}[d_{\overline{V}}]), \text{ for some } d_{\overline{V}} = {}_{\mathcal{B}_1}M_{Z,\overline{Y}}(d) = {}_{\mathcal{B}_2}M_{Z,\overline{Y}}(d) \text{ since the equivalence checker ADDG\_EQX finds } {}_{\mathcal{B}_1}M_{Z,\overline{Y}}(d) = {}_{\mathcal{B}_2}M_{Z,\overline{Y}}(d), r_{\mathcal{B}_1} = r_{\mathcal{B}_2} \text{ over } \{\overline{Y}[d_{\overline{Y}}], d_{\overline{Y}} \in {}_{\mathcal{B}_1}M_{Z,\overline{Y}}({}_{\mathcal{B}_1}D_Z) = {}_{\mathcal{B}_2}M_{Z,\overline{Y}}({}_{\mathcal{B}_2}D_Z)\} \text{ and } r_{\mathcal{D}'_1} = r_{\mathcal{D}'_2} \text{ over } \{\overline{V}[d_{\overline{V}}], d_{\overline{V}} \in {}_{\mathcal{D}_1}U_{\overline{V}} = {}_{\mathcal{D}'_1}U_{\overline{V}}\} \times \{Z1[d_{z1}], d_{z1} \in {}_{\mathcal{D}'_1}U_{Z1}\} \end{aligned}$$

Therefore, $e_1 = e_2$ for this subcase.

Subcase 2.2 [$d_{Z1} \in {}_{\mathcal{D}'_1}D_Z$]: It may be noted that from Definition 1 (recurrence array vertex) ${}_{\mathcal{D}'_1}D_Z \bigcap {}_{\mathcal{D}'_1}U_{Z1} \neq \emptyset$, i.e., some elements of Z defined through $\mathcal{D}'_1$ are used in defining further elements of Z in $\mathcal{D}'_1$ itself. Specifically, equation (i) $Z[d] = e_1(\overline{V}[d_{\overline{V}}]) = r_{\mathcal{D}'_1}(\overline{V}[d_{\overline{V}}], Z1[d_{Z1}])$ depicts that in order to prove that $Z[d]$ evaluated through e_1 is the same as that evaluated through e_2, we need to *assume* that in $\mathcal{D}'_1$, the evaluation of $Z[d_{Z1}]$ $(= Z1[d_{Z1}])$ should precede the evaluation of $Z[d]$, and likewise, in $\mathcal{D}'_2$. Such an assumption is nothing but the induction hypothesis. (This aspect justifies the nomenclature of the subgraphs $\mathcal{D}_1$ and $\mathcal{D}_2$ as the induction subgraphs because it supports the inductive step in the analysis of a recurrence.) In other words, an ordering over the elements of ${}_{\mathcal{D}'_1}D_Z$ $(= {}_{\mathcal{D}'_2}D_Z)$ is needed for validation of $e_1 = e_2$. Towards this, let us consider the following relation: $\forall d_1, d_2 \in {}_{\mathcal{D}'_1}D_Z$ $(= {}_{\mathcal{D}'_2}D_Z)$, $d_1 \prec d_2$, if the computation of the value of $Z[d_2]$ depends upon the computation of that of $Z[d_1]$[1].

Assuming that $\forall d' \prec d$, $Z[d']$ $(= Z1[d'])$ are evaluated identically by $\mathcal{D}'_1$ and $\mathcal{D}'_2$, i.e., $Z[d'] = e_1(\overline{V}[d'_{\overline{V}}]) = e_2(\overline{V}[d'_{\overline{V}}])$, we have

$$\begin{aligned} Z[d] &= e_1(\overline{V}[d_{\overline{V}}]) \\ &= r_{\mathcal{D}'_1}(\overline{V}[d_{\overline{V}}], Z1[d_{Z1}]) \\ &= r_{\mathcal{D}'_1}(\overline{V}[d_{\overline{V}}], e_1(\overline{V}[d'_{\overline{V}}])), \text{ where } d'_{\overline{V}} = {}_{\mathcal{D}_1}M_{Z,\overline{V}}(d) = {}_{\mathcal{D}_2}M_{Z,\overline{V}}(d) \\ &= r_{\mathcal{D}'_2}(\overline{V}[d'_{\overline{V}}], e_2(\overline{V}[d'_{\overline{V}}])) \\ &= e_2(\overline{V}[d'_{\overline{V}}]), \text{ by induction hypothesis and by the equivalence checker ADDG\_EQX applied on } \mathcal{D}'_1 \text{ and } \mathcal{D}'_2 \end{aligned}$$

Hence, $e_1 = e_2$ over the defined domain ${}_{\mathcal{D}'_1}D_{Z1} = {}_{\mathcal{D}'_2}D_{Z1}$. ■

4.2 Complexity

Let us determine the worst case time complexity of all the steps involved in Algorithm 3. Note that the analysis has been done assuming the number of arrays in each ADDG is a, the number of statements (write edges) is s, the maximum arity of a function is t and the number of recurrence subgraphs is γ.

Step 1: Finding the set of common arrays takes a^2 time and copying each ADDG takes $O(V + E)$, i.e., $O((a + s) + (a + s \times t))$ time.

Step 2: This step basically involves identification of SCCs in a directed graph using Tarjan's algorithm (Tarjan 1972), which can be done in a depth-first traversal of the graph. Therefore, this step also

[1] Such a definition essentially implies that there is a *valid schedule*, which is supported by the restrictions given in Section 2.

requires $O(V+E) = O((a+s)+(a+s\times t))$ time.
Steps 3—5: Each of the three functions mentioned in step 4 requires a depth-first traversal of the ADDG; therefore, the loop takes $O(\gamma \times ((a+s)+(a+s\times t)))$ time.
Steps 6—8: Similar to the steps 3—5.
Steps 9—19: In step 11, checking whether the arrays involved in two recurrence subgraphs are identical takes $O(a^2)$ time. Note that the complexity of comparing two normalized expressions is $O(2^{\|F\|})$, where $\|F\|$ is the length of a normalized formula in terms of its constituent variables and operators (Karfa et al. 2013b). However, representing the data transformation of the elements of the output array of a subgraph in terms of the elements of the input arrays of that subgraph may require substitution of the intermediate temporary arrays in a transitive fashion, which takes $O(a^2 \times 2^{\|F\|})$ time (Karfa et al. 2013b). For finding the transitive dependence and the union of the mappings, ISL (Verdoolaege 2010) has been used, whose worst case time complexity is the same as the deterministic upper bound on the time required to verify Presburger formulae, i.e., $O(2^{2^{2^n}})$, where $n = O(t \times a)$ is the length of the formula. Since we replace the recurrence subgraphs (except for the recurrence vertex and the input array vertices) with an operator vertex representing an uninterpreted function along with the edges connecting this operator vertex with the recurrence vertex and the input array vertices, step 12 takes $O(V) = O(a+s)$ time for each ADDG. We have to compare the data transformations of pairs of recurrence subgraphs, one from each ADDG; thus, the loop encompassing the steps 9–19 takes $O(g \times (2^{\|F\|} + a^2 \times 2^{\|F\|}))$ time.
Steps 20—22: This step simply reports a failure case; however, in the worst case, the faulty recurrence subgraph may cover almost the entire ADDG; thus, this step may also require $O(V+E) = O((a+s)\times(a+s\times t))$ time.
Steps 23—27: In step 23, the verifier of (Karfa et al. 2013b) is invoked whose worst case time complexity is reported to be $O(k^{b\times x} \times (2^{\|F\|} + a^2 \times 2^{\|F\|}) + k^{2\times b\times x} \times 2^{\|F\|})$, where b is the number of branching blocks in the control flow graph of the program, k is the maximum branches in a branching block and x is the maximum number of arrays defined in a branch. However, the authors of (Karfa et al. 2013b) had neglected the worst case time complexity for verifying Presburger formulae because this worst case behaviour was never exhibited in their experiments (which is the case for our set of experiments as well). Since this is the costliest step in the entire algorithm, the worst case time complexity of Algorithm 3 is $O(2^{2^{2^n}})$ which is identical to that of (Karfa et al. 2013b).

5. Experimental Results

Our method has been implemented in the C language and run on a 2.0 GHz Intel® Core™2 Duo machine. Our tool is available at `http://cse.iitkgp.ac.in/~chitta/pubs/EquivalenceChecker_ADDG.tar.gz` as an open source free software covered under GNU General Public License along with the benchmarks, installation and usage guidelines. It first constructs the ADDGs from the original and the transformed programs written in C and then applies our method to establish the equivalence between them. The tool has been tested on some benchmarks. The characteristics of the benchmarks and the time taken to establish equivalence by our tool and by those of [Ver] (Verdoolaege et al. 2012) and [Kar] (Karfa et al. 2013b) are given in Table 1; note that the symbol × has been used whenever a tool failed to establish the required equivalence. The benchmarks have been obtained by manually applying different transformations to the programs of Sobel edge detection (SOB), Debaucles 4-coefficient wavelet filter (WAVE), Laplace algorithm to edge enhancement of northerly directional edges (LAP), linear recurrence solver (LIN), successive over-relaxation (SOR), computation across (ACR), low-pass filter (LOWP), discrete cosine transform (DCT), matrix multiplication (MMUL), finite difference time domain (FDTD), modified versions of the example given in Figure 3 (EXM), summation of the elements of different input arrays (SUM) and the example shown in Figure 12. Note that all the tools succeeded in showing equivalence for benchmarks 1–6 which involved only loop transformations; the tool of (Verdoolaege et al. 2012) failed to establish equivalence for benchmarks 7–12 because they contained arithmetic transformations as well; the tool of (Karfa et al. 2013b) failed for benchmarks 13–16 because they involved loop transformations along with recurrences and both the tools of (Verdoolaege et al. 2012) and (Karfa et al. 2013b) failed for benchmarks 17–19 because they involved both arithmetic transformations and recurrence; our tool succeeded in showing equivalence in all the cases except for the last one – the reason for failure in this case is explained in Section 6. Although a comparative analysis with the method of (Shashidhar 2008) would have also been relevant, we could not furnish it since their tool is not available to us. To find out the set of loop transformations and arithmetic transformations supported by our tool, the readers are referred to (Karfa et al. 2013b). A pertinent point to note is that although our tool outperforms that of (Karfa et al. 2013b) with respect to execution time whenever both the tools are able to establish equivalence, the tool of (Verdoolaege et al. 2012) takes about 2.4 times less execution time on average than that of ours whenever it is successful – this is probably because our tool invokes ISL (Verdoolaege 2010) through system call and communicates with it via reading and writing to files whereas, ISL comes as an integrated package within (Verdoolaege et al. 2012) itself and hence it is faster.

Table 1. Results on some benchmarks

Sl No	Benchmark	C lines		loops		arrays		slices		Exec time (sec) [Ver]	Exec time (sec) [Kar]	Exec time (sec) [Our]
		src	trn	src	trn	src	trn	src	trn			
1	ACR1	14	20	1	3	6	6	1	1	0.18	0.76	0.28
2	LAP1	12	28	1	4	2	4	1	2	0.28	9.25	0.55
3	LIN1	13	13	3	3	4	4	2	2	0.12	0.62	0.24
4	LIN2	13	16	3	4	4	4	2	3	0.13	0.74	0.30
5	SOR	26	22	8	6	11	11	1	1	0.18	1.08	0.68
6	WAVE	17	17	1	2	2	2	4	4	0.31	6.83	0.53
7	ACR2	24	14	4	1	6	6	2	1	×	0.98	0.36
8	LAP2	12	21	1	3	2	4	1	1	×	2.79	0.35
9	LAP3	12	14	1	1	2	2	1	2	×	4.82	0.56
10	LOWP	13	28	2	8	2	4	1	2	×	9.17	0.63
11	SOB1	27	19	3	1	4	4	1	1	×	1.79	0.61
12	SOB2	27	27	3	3	4	4	1	1	×	1.85	0.57
13	DCT	20	48	6	18	5	7	1	1	0.57	×	1.61
14	EXM1	8	14	1	1	2	4	1	3	0.14	×	0.36
15	EXM2	8	15	1	1	3	5	1	3	0.19	×	0.48
16	MMUL	11	22	3	8	3	3	1	1	0.26	×	0.76
17	FDTD	26	27	9	8	6	8	1	1	×	×	1.17
18	SUM1	8	14	1	1	2	4	1	3	×	×	0.40
19	SUM2	16	19	4	4	4	6	2	4	×	×	0.72
20	Figure 12	6	6	1	1	1	1	1	1	0.26	×	×

In another set of experiments, we had borrowed the programs from (Karfa et al. 2013b) where the authors had manually injected data computation errors and/or wrongly written the loop iterators to enforce erroneous loop boundary calculations in order to check the efficacy of the equivalence checker in detecting incorrect code transformations. Our equivalence checker also reported *failure* in all these cases and correctly identified the subgraphs of the ADDGs where the errors had been injected as revealed by subsequent manual inspection.

6. Current Limitations

The technique outlined in Algorithm 3, however, has the following restrictions.
1) The arrays that participate in recurrences must be named identically in both the original program and the transformed program.

$A[0] = in;$	$A[0] = g(in);$
$for(i = 1; i <= N; ++i)\{$	$for(i = 1; i <= N; ++i)\{$
$\quad A[i] = f(g(A[i/2]));$	$\quad A[i] = g(f(A[i/2]));$
$\}$	$\}$
$out = g(A[N]);$	$out = A[N];$
(a) Original program.	(b) Transformed program.

Figure 12. A pair of equivalent programs with non-equivalent recurrence graphs.

We believe that this restriction can be eliminated by establishing a *name-correspondence* for the intermediary array names starting with identical output array names and ending with identical input array names (similar to state correspondence in bisimulation based methods (Kundu et al. 2010)).

2) The equivalence checking method outlined above does not presently support co-induction (mutual recurrence) and nested recurrence. While the former can be tackled by tracking SCCs with multiple recurrence array vertices, the latter can be alleviated by incorporating a recursive call in our equivalence checking algorithm (at steps 4 and 7) such that the innermost recurrence is first checked for equivalence before moving on to outer recurrences – these schemes are yet to be implemented in our tool.

3) Our method cannot verify programs if the recurrence in the original program is replaced by some iterative code without recurrence in the transformed program or vice versa since each recurrence subgraph in the original program is paired with a recurrence subgraph from the transformed program by our equivalence checker. However, it may be worth noting that we have not found any automated code optimizer to apply such transformation.

4) Figure 12 shows a pair of equivalent programs with non-equivalent recurrence subgraphs. Since the basis subgraphs and the induction subgraphs of the programs are different, our ADDG based equivalence checker will declare them to be possibly non-equivalent. However, an equivalence checking mechanism may be devised *that propagates the mismatched symbolic values (of* $A[0]$*)* in and $g(in)$ from the basis subgraphs to the induction subgraphs and even out of the recurrence subgraph, if required (as in the present scenario). Note that the method of (Verdoolaege et al. 2012) is able to establish equivalence of the two programs given in Figure 12 by employing a similar technique of symbolic value propagation.

5) Recurrences that employ reductions (Iooss et al. 2014) are not allowed because reductions involve accumulations of a parametric number of sub-expressions using an associative and commutative operator – which is currently not supported by our method.

7. Conclusion

Loop and arithmetic transformations are applied extensively in embedded system design to minimize execution time, power, area, etc. An ADDG based equivalence checking method has been proposed in (Karfa et al. 2011, 2013b) for translation validation of programs undergoing such transformations. This method, however, cannot be applied to verify programs that involve recurrences because recurrences lead to cycles in the ADDGs which render currently available dependence analyses and arithmetic simplification rules inadmissible. Another verification technique (Verdoolaege et al. 2009, 2012) which can verify programs with recurrences does not handle arithmetic transformations. This work, for the first time, provides a unified equivalence checking framework for handling loop transformations and arithmetic transformations along with most of the recurrences. The experimental results attest the effectiveness of the method. In our future work, we intend to alleviate the current limitation of the method to handle a more general class of programs.

Acknowledgments

The work of K. Banerjee was supported by TCS Research Fellowship. This work was also funded by DST Project No: SB/EMEQ-281/2013.

References

K. Banerjee, D. Sarkar, and C. Mandal. Extending the FSMD framework for validating code motions of array-handling programs. *IEEE Trans. on CAD of ICS*, 33(12):2015–2019, 2014.

D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations (research note). In *Euro-Par Conference on Parallel Processing*, pages 309–313, 2002.

Y. Bouchebaba, B. Girodias, G. Nicolescu, E. M. Aboulhamid, B. Lavigueur, and P. G. Paulin. MPSoC memory optimization using program transformation. *ACM Trans. Design Autom. Electr. Syst.*, 12(4), 2007.

G. Iooss, C. Alias, and S. V. Rajopadhye. On program equivalence with reductions. In *Static Analysis*, pages 168–183, 2014.

I. Kadayif, M. T. Kandemir, G. Chen, O. Ozturk, M. Karaköy, and U. Sezer. Optimizing array-intensive applications for on-chip multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 16(5):396–411, 2005.

C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Equivalence checking of array-intensive programs. In *ISVLSI*, pages 156–161, 2011.

C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Experimentation with SMT solvers and theorem provers for verification of loop and arithmetic transformations. In *I-CARE*, pages 3:1–3:4, 2013a.

C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Verification of loop and arithmetic transformations of array-intensive behaviours. *IEEE Trans. on CAD of ICS*, 32(11):1787–1800, 2013b.

S. Kundu, S. Lerner, and R. Gupta. Translation validation of high-level synthesis. *IEEE Trans on CAD of ICS*, 29(4):566–579, 2010.

B. Landwehr and P. Marwedel. A new optimization technique for improving resource exploitation and critical path minimization. In *ISSS*, pages 65–72, 1997.

T. Matsumoto, K. Seto, and M. Fujita. Formal equivalence checking for loop optimization in C programs without unrolling. In *ACST*, pages 43–48, 2007.

V. Menon, K. Pingali, and N. Mateev. Fractal symbolic analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):776–813, 2003.

M. Potkonjak, S. Dey, Z. Iqbal, and A. C. Parker. High performance embedded system optimization using algebraic and generalized retiming techniques. In *ICCD*, pages 498–504, 1993.

D. Sarkar and S. De Sarkar. A theorem prover for verifying iterative programs over integers. *IEEE Trans Software. Engg.*, 15(12):1550–1566, 1989.

K. C. Shashidhar. *Efficient Automatic Verification of Loop and Dataflow Transformations by Functional Equivalence Checking*. PhD thesis, Katholieke Universiteit Leuven, 2008.

K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *DATE*, pages 1310–1315, 2005.

R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

S. Verdoolaege. ISL: An Integer Set Library for the Polyhedral Model. In *ICMS*, pages 299–302, 2010.

S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In *CAV*, pages 599–613, 2009.

S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. Program. Lang. Syst.*, 34(3), 2012.

J. Zory and F. Coelho. Using algebraic transformations to optimize expression evaluation in scientific codes. In *PACT*, pages 376–384, 1998.

L. D. Zuck, A. Pnueli, B. Goldberg, C. W. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. *Formal Methods in System Design*, 27(3):335–360, 2005.