Verification of Source Code Transformations by Program Equivalence Checking

K.C. Shashidhar[1,2], Maurice Bruynooghe[2],
Francky Catthoor[1,3], and Gerda Janssens[2]

[1] Interuniversitair Micro-Elektronica Centrum (IMEC) vzw, Leuven, Belgium
[2] Departement Computerwetenschappen, Katholieke Universiteit Leuven, Belgium
[3] Departement Elektrotechniek (ESAT), Katholieke Universiteit Leuven, Belgium
{kodambal, catthoor}@imec.be, {maurice, gerda}@cs.kuleuven.ac.be

Abstract. Typically, a combination of manual and automated transformations is applied when algorithms for digital signal processing are adapted for energy and performance-efficient embedded systems. This poses severe verification problems. Verification becomes easier after converting the code into dynamic single-assignment form (DSA). This paper describes a method to prove equivalence between two programs in DSA where subscripts to array variables and loop bounds are (piecewise) affine expressions. For such programs, geometric modeling can be used and it can be shown, for groups of elements at once, that the outputs in both programs are the same function of the inputs.

1 Introduction

In the recent years, embedded processor systems have emerged as pervasive platforms for multimedia and telecom systems. They are highly resource-constrained and there is an increasing stress on rigorous optimization of the software that runs on them. Current compiler optimizations, though powerful, are insufficient to meet the resource constraints. Designers apply domain specific optimizations to obtain programs with a better performance/energy consumption trade-off.

Accesses to the data memory hierarchy are the most time and energy consuming operations in data-intensive applications. Globally applied loop transformations, expression propagations and algebraic transformations can reduce their cost. Guided by elaborate cost models, experienced designers apply them manually or use ad-hoc tools in a transformation phase prior to compilation. The process is error prone and testing hampers designer's productivity. We present a formal and automated method for the verification of such transformations.

Fig. 1 shows an artificial example where program (b) has been derived from (a) through expression propagations, loop and algebraic transformations. The functions, when executed, take inputs `A[]` and `B[]`, and assign the computed values to the elements of the output array `C[]`. Ignoring possible overflow, integer addition is both associative and commutative. Hence, both programs compute

R. Bodik (Ed.): CC 2005, LNCS 3443, pp. 221–236, 2005.

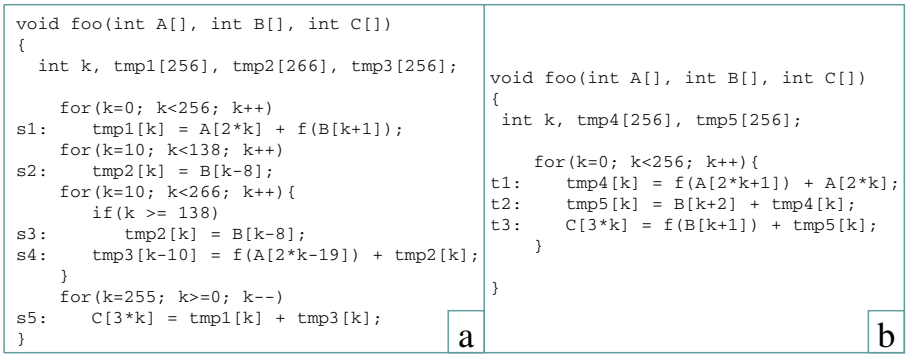

Fig. 1. Example of an original (a) and transformed (b) program function pair

the same outputs for the same inputs, i.e., they are *input-output equivalent*. Our method automates the checking of their input-output equivalence.

The method handles a decidable subset of structured, imperative programs that are in dynamic single-assignment form, have only piecewise-affine expressions as subscripts to array variables and bounds of `for`-loops, and have static control-flow free from side-effects. It relies on code pre-processing methods to convert programs commonly seen in practice into the subset. For programs in this subset, we introduce a representation that captures both computation and the *true* data dependencies (Sec. 2). This representation exposes the invariant properties for the transformations and can deal with algebraic transformations (Sec. 3). Equivalence is shown by checking that a one-to-one correspondence exists between the two programs in their computation and in the data dependencies between the individual elements of their observable array variables (Sec. 4). It neither relies on any information about the particular instances of the transformations that were applied nor on the order of their application. It scales well for larger problem sizes (Sec. 5). Prior work outlines our method and discusses its application in embedded systems design [13]. This paper formally presents the method and explains how recurrences in data dependencies are handled. In Sec. 6, we situate our work with respect to other approaches.

2 Program Representation

We assume an imperative programming language that has array data structures and has a form of `for`-loops to control iteration. Our current tools are focused on C. The analysis is intra-procedural and the equivalence is checked between two procedures (functions). They can call other functions (common to both) to the extent that those functions can be considered as side effect free operators.

2.1 Class of Allowed Programs

Programs we can handle have the following properties:

1. *Dynamic single-assignment:* Every memory location is written only once. Optimizing compilers use the *static single-assignment* (SSA) form [6] to facilitate optimizations which still can write the same array element several times. This is not the case with *dynamic single-assignment* (DSA) form; it eliminates all false dependencies. Methods for conversion to DSA are described in [7, 16]. We also require that functions are free from side-effects.
2. *Piecewise-affine expressions:* Subscripts in the arrays and expressions in the bounds of the `for`-loops are all piece-wise affine in the iterator variables of the enclosing `for`-loops. Additionally, the expressions can also include operators like `mod`, `div`, `max`, `min`, `floor` and `ceil`. This allows representing the addressing relationships between elements of arrays as affine inequalities in integers and makes it possible to use well-understood dependence tests (for example, the Omega test [12]) to solve those systems.
3. *Static control-flow:* There are no data-dependent `while`-loops in the programs. We assume that data-dependent `while`-loops have been converted to `for`-loops with worst-case bounds and a global if-condition on its body; and the data-dependent if-conditions in the program have been converted into data dependencies by using if-conversion [1].
4. *No pointer references:* Programs are free from pointer references. Pointer-to-array conversion methods (for example, [15]) can be used here.

The class is not unduly restrictive for the application domain. In fact, it is advantageous to bring programs into such a form before applying global transformations as this form creates more freedom for the transformations and the tools used for guiding the transformations can do a better job [5].

2.2 Array Data Dependence Graphs

Scalars can be considered as one element arrays. Hence, which element is assigned by a (assignment) statement depends on the instantiation of the subscripts of the assigned array. The subscripts can depend on the values of the surrounding iterators when the statement appears inside a nest of `for`-loops. Which values the subscripts take during execution can be described in closed form as an integer domain in a multi-dimensional geometrical space. Such descriptions which record a variety of information related to the statements and dependencies among them are together referred to as the geometrical or polyhedral representation. This representation is commonly used for dependence analysis by optimizing compilers [2, 3, 17]. Here we briefly review the main elements.

Let us consider a statement `s` of the form

$$\texttt{s:}\quad \texttt{v[}f_{i_1}(\vec{k}_d)\texttt{]}\ldots\texttt{[}f_{i_n}(\vec{k}_d)\texttt{] = exp(}\cdots\texttt{, u[}f_{j_1}(\vec{k}_d)\texttt{]}\ldots\texttt{[}f_{j_m}(\vec{k}_d)\texttt{]},\cdots\texttt{);}$$

where $\vec{k}_d = (k_1, \ldots, k_r, \ldots, k_d)$ is the vector of iterator variables of the surrounding `for`-loops. Let $l_r(\vec{k}_{r-1})$, $u_r(\vec{k}_{r-1})$ and $s_r(\vec{k}_{r-1})$ be affine functions defining

respectively the lower and upper bounds, and the stride of iterator k_r. Finally, assume execution of the `for`-loops is controlled by affine expressions $c_r(\vec{k}_{r-1})$ and execution of the statement `s` by $c_{d+1}(\vec{k}_d)$. Then we can define the following:

Definition 1 (Iteration Domain, D). *Integer domain in which each point* $[k_1, \ldots, k_d]$ *represents exactly one execution of the statement* `s`*:*

$$\mathrm{D} := \{[k_1, \ldots, k_d] \mid (\bigwedge_{r=1}^{d} k_r \in \mathbb{Z} \wedge (l_r(\vec{k}_{r-1}) \leq k_r \leq u_r(\vec{k}_{r-1})) \wedge c_r(\vec{k}_{r-1}) \wedge (\exists \alpha_r \in \mathbb{Z} \mid k_r = \alpha_r s_r(\vec{k}_{r-1}) + l_r(\vec{k}_{r-1}))) \wedge c_{d+1}(\vec{k}_d)\}.$$

Definition 2 (Definition Domain, $\mathrm{W_v}$). *Integer domain in which each point* $[i_1, \ldots, i_n]$ *represents exactly one* write *to* `v[`i_1`]`$\ldots$`[`i_n`]`*, an element of the array* `v` *defined by the statement* `s` *with iteration domain* D*:*

$$\mathrm{W_v} := \{[i_1, \ldots, i_n] \mid (\bigwedge_{r=1}^{n} i_r = f_{i_r}(\vec{k})) \wedge \vec{k} \in \mathrm{D}\}.$$

Definition 3 (Operand Domain, $\mathrm{R_u}$). *Integer domain in which each point* $[j_1, \ldots, j_m]$ *represents exactly one* read *from an element* `u[`j_1`]`$\ldots$`[`j_m`]`*, of an operand array* `u` *in statement* `s` *with iteration domain* D*:*

$$\mathrm{R_u} := \{[j_1, \ldots, j_m] \mid (\bigwedge_{r=1}^{m} j_r = f_{j_r}(\vec{k})) \wedge \vec{k} \in \mathrm{D}\}.$$

Definition 4 (Dependency Mapping, $\mathrm{M_{v,u}}$). *A mapping associated with a statement, between a defined array* `v` *and an operand array* `u`*. Each instance* $[i_1, \ldots, i_n] \rightarrow [j_1, \ldots, j_m]$ *in the mapping indicates that element* `u[`j_1`]`$\ldots$`[`j_m`]` *is read when the element* `v[`i_1`]`$\ldots$`[`i_n`]` *is written by the statement* `s` *with iteration domain* D*:*

$$\mathrm{M_{v,u}} := \{[i_1, \ldots, i_n] \rightarrow [j_1, \ldots, j_m] \mid (\bigwedge_{r=1}^{n} i_r = f_{i_r}(\vec{k})) \wedge (\bigwedge_{r=1}^{m} j_r = f_{j_r}(\vec{k})) \wedge \vec{k} \in \mathrm{D}\}.$$

For example, the definitions given above for statement `s4` in the original function in Fig. 1 are:

$\mathrm{D} := \{[k] \mid 10 \leq k < 266 \wedge k \in \mathbb{Z}\}$	
$\mathrm{W_{tmp3}} := \{[d] \mid d = k - 10 \wedge k \in \mathrm{D}\}$	$\mathrm{R_A} := \{[d] \mid d = 2 * k - 19 \wedge k \in \mathrm{D}\}$ $\mathrm{R_{tmp2}} := \{[d] \mid d = k \wedge k \in \mathrm{D}\}$
$\mathrm{M_{tmp3,A}} := \{[d_1] \rightarrow [d_2] \mid d_1 = k - 10 \wedge d_2 = 2 * k - 19 \wedge k \in \mathrm{D}\}$ $\mathrm{M_{tmp3,tmp2}} := \{[d_1] \rightarrow [d_2] \mid d_1 = k - 10 \wedge d_2 = k \wedge k \in \mathrm{D}\}$	

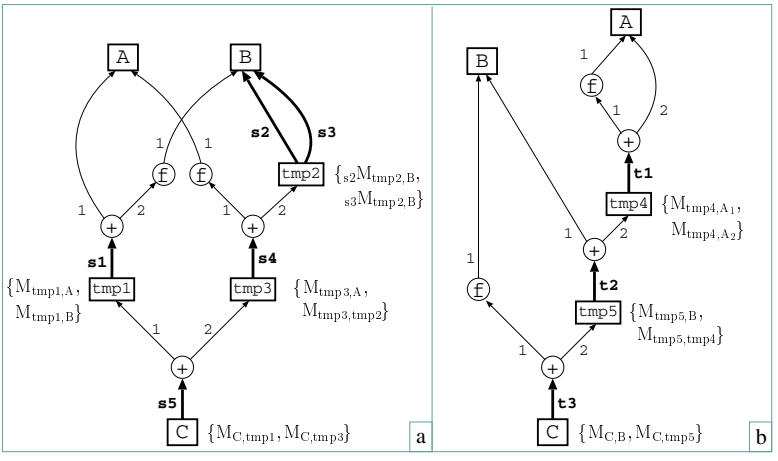

Fig. 2. The `ADDGs` of program functions in Fig. 1. Array A_1 and A_2 in the dependency mapping of `tmp4` in (b) refer to different occurrences of `A` in statement `t1`

A data dependence exists between two statements s and t when s produces values and t consumes them, i.e., s has definition domain W_v, t has operand domain R_v and $W_v \cap R_v \neq \emptyset$. Dependencies are represented at a fine grained level. The assigned array depends either on the consumed array or on the main operator of the rhs. In the latter case, the operator in turn depends on its arguments which are either other operators or arrays. The set of all dependencies can be represented as an array data dependence graph (`ADDG`).

Definition 5 (Array Data Dependence Graph, `ADDG`). *The* `ADDG` *of a program is a directed graph $G = (V, E)$, where the node set V is the union of arrays used in the program (array nodes) and the operator occurrences (operator nodes) of the statements and the edge set E represents the dependencies. An edge with operator node as source is labeled by the operand position of its destination; an edge with an array as source is labeled with the statement identifier of the assignment. Array nodes of defined arrays are annotated with the dependency mappings of the statement.*

Whereas standard data dependence graphs used in high-performance compilers represent dependencies at the statement level, we use more detailed dependencies. Also, a data dependence (*reverse flow*), denoted by a directed edge, refers not just to a single value, but to a *set of values*. A dependency mapping (Def. 4) corresponds to a path with the defined array as source and the operand array as destination (paths that pass through zero or more operators). Fig. 2 shows the `ADDG` representations of the programs of Fig. 1.

An array `v` is an *internal array* if $\bigcup W_v = \bigcup R_v$, i.e. each produced element is consumed; it is an *input array* if $\bigcup W_v = \emptyset$, i.e., no element is produced; and an *output array* if $\bigcup R_v \subset \bigcup W_v$, i.e. some of its elements are not consumed.

In the example, the original function has $\{\texttt{A}, \texttt{B}\}$ as input, $\{\texttt{tmp1}, \texttt{tmp2}, \texttt{tmp3}\}$ as internal and $\{\texttt{C}\}$ as output arrays. A path $u, o_1, \ldots, o_n, v$ with u and v array nodes and $o_1, \ldots, o_n$ operator nodes represents a data-flow between v and u which is described by the dependency mapping $\mathrm{M}_{\texttt{u},\texttt{v}}$. We can also associate a dependency mapping with a path across several array nodes.

Definition 6 (Transitive Dependency Mapping, $\mathrm{M}^*_{\texttt{v}_0,\texttt{v}_n}$). *Let p be a path in an* ADDG *starting in array node v_o, ending in array node v_n and passing through array nodes $v_1, \ldots, v_{n-1}$ $(n \geq 0)$. Using $\bowtie$ for the natural join*[4] *of two relations:*

$$\mathrm{M}^*_{\texttt{v}_0,\texttt{v}_n} := \begin{cases} I \text{ (the identity)} & n = 0 \\ \mathrm{M}_{\texttt{v}_0,\texttt{v}_1} & n = 1 \\ \mathrm{M}_{\texttt{v}_0,\texttt{v}_1} \bowtie \mathrm{M}_{\texttt{v}_1,\texttt{v}_2} \bowtie \ldots \bowtie \mathrm{M}_{\texttt{v}_{n-1},\texttt{v}_n} & \text{otherwise} \end{cases}$$

Definition 7 (Data Dependence Path). *A path between two array nodes is a data dependence path iff its transitive dependency mapping is non-empty.*

The transitive dependency mapping from an output to an input node is called the *output-to-input mapping*. The set of output-to-input mappings characterizes the data-flow of the computation.

For example, in the ADDG of the original function in Fig. 2 the output-to-input mapping from C to B on the rightmost path is given by

$$\begin{aligned} \mathrm{M}^*_{\texttt{C},\texttt{B}} &:= \mathrm{M}_{\texttt{C},\texttt{tmp3}} \bowtie \mathrm{M}_{\texttt{tmp3},\texttt{tmp2}} \bowtie \mathrm{M}_{\texttt{tmp2},\texttt{B}} \\ &:= \{[d_1] \rightarrow [d_2] \mid d_1 = 3 * k \wedge d_2 = k + 2 \wedge 128 \leq k < 256 \wedge k \in \mathbb{Z}\}. \end{aligned}$$

The data dependence paths from a node v can be used to identify the program slices contributing to the computation of the elements of v. The outgoing edges of v partition the elements of the array and different paths correspond to different slices of the computation. Also an operator node has different outgoing edges. They correspond to different operands of the operator; they all contribute to the computation by the operator and hence belong to the same slice.

An ADDG can have cycles, in which case it has cyclic paths. A cyclic data dependence path indicates the presence of a *recurrence* in the computation: arrays in a cyclic path have elements whose value depend on other elements of the same array. While an ADDG with a cycle has infinite paths, all data dependence paths are finite as the program is composed of terminating `for`-loops. We return to recurrences in Sec. 4.2.

An internal array node acts as a buffer and can be eliminated from a given path (because the program is in DSA).

Operation 1 (Internal Array Node Elimination). *Let the outgoing edges of an internal array node w be $(w, x_1), \ldots, (w, x_k)$ with labels $s_1, \ldots, s_k$ and let $\mathrm{M}_{\texttt{w},\texttt{t}_1}, \ldots, \mathrm{M}_{\texttt{w},\texttt{t}_k}$ be the corresponding dependency mappings. Let p be a path*

[4] $x \rightarrow z \in F \bowtie G \Leftrightarrow \exists y$ s.t. $x \rightarrow y \in F \wedge y \rightarrow z \in G$.

(possibly including operators) from an array node u to w and s be the label of the outgoing edge of u on p and $\mathrm{M}_{u,w}$ the associated dependency mapping. Let the incoming edge on w on p be (v, w) with the label l. The node w is eliminated on the path p from u as follows:

- *$\forall i, 1 \leq i \leq k$: add the edge (v, x_i) and if $u = v$, label it as $s.s_i$, else label it as l*
- *Replace $\mathrm{M}_{u,w}$ by the transitive dependency mappings $\mathrm{M}^*_{u,t_i}, \ldots, \mathrm{M}^*_{u,t_k}$*
- *Remove the edge (v, w).*

In the above operation, when v is an operator node and $k > 1$, v has multiple operands with the same position label. Such operands correspond to disjunct slices of the computation.

3 Transformations and Their Effect

In this section, we discuss three categories of transformations that we allow and their effect on the ADDG representation of the program function.

Global Loop Transformations. Loop transformations are usually classified into *structure preserving* and *structure modifying* categories. The former category includes such transformations as loop permutation, interchange, skewing, reversal and bumping, and those that can be derived from combining them. The latter includes loop distribution, fission, splitting, merging, folding, fusion, strip-mining, tiling and unrolling. Structure preserving transformations only affect the iteration domains of statements. While the graph structure of the ADDG remains, the associated dependency mappings are affected. A transformation preserves correctness when the output-to-input mappings for the paths of the same computation on the transformed ADDG is identical to the output-to-input mappings in the original ADDG. Structure modifying transformations can result in a re-distribution of definition domains of the involved arrays. For example, in the original function, the rightmost path splits at array node tmp2 and partitions the output-to-input mappings from the output array C to input array B for the same computation. Therefore, the invariant for the correctness of these transformations is that, the union of output-to-input mappings for the paths of the same computation on the transformed ADDG must be identical to a similar union of mappings in the original ADDG.

Expression Propagations. Expression propagation involves both introduction and elimination of intermediate arrays for partial computations in the program function. For example, a statement with a summation of three terms on the right-hand side can be converted into two statements with summation of two terms each, by the introduction of an intermediate array. Another possibility is that a set of values are recomputed, instead of reused. The effect of expression propagation on the ADDG of the program function is insertion/elimination of array nodes on the paths of the ADDG and/or duplication of sub-ADDGs. The invariant for the correctness of the propagation transformations is the same as for

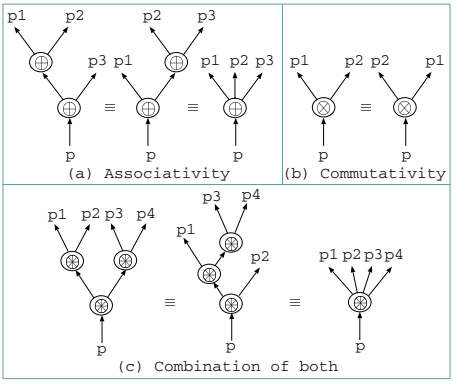

Fig. 3. AC transformations

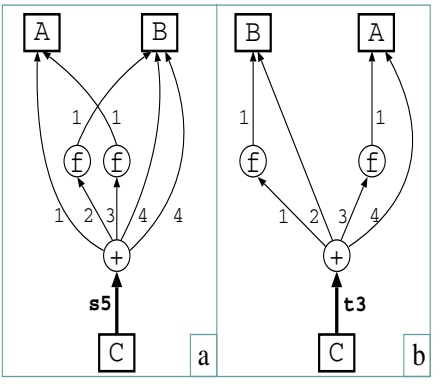

Fig. 4. `ADDGs` after flattening

loop transformations. That is, the output-to-input mappings for the paths of the same computation on the transformed `ADDG` is identical to the output-to-input mappings in the original `ADDG`.

Global Algebraic Transformations. Algebraic transformations exploit properties of operators and user-defined functions and modify the data-flow in order to improve efficiency or to enable the other transformations. Several statements can be involved as can be seen in Fig. 1, where these transformations have been applied across expressions of multiple statements. The `ADDGs` of the two functions, as shown in Fig. 2, also reflect this. Most of these transformations just rely on the associativity and/or commutativity properties of the operators like addition and multiplication on a data-type such as integer. We distinguish:

Associativity. Let $\oplus$ be an associative operator. Fig. 3(a) shows two computations that are equivalent due to associativity. To integrate associativity in our method, we replace the graph fragment by its normal form: A single $\oplus$ operator with a variable number of arguments as shown on the right of Fig. 3(a). This does not affect the output-to-input mappings of the `ADDG`. In addition, internal array nodes receiving input from another $\oplus$ operator can be eliminated. This results in the following operator:

Operation 2 (Flattening). *Process all successor nodes of an associative $\oplus$-node p as follows: if it is an internal array node, apply internal array node elimination. If it is another $\oplus$-node o, eliminate it: let l be the label of the edge (p, o) and let $(o, s_0), \ldots, (o, s_n)$ be the outgoing edges. For all the outgoing edges of p with label $(k > l)$, replace the label k by $k + n$ and add edges (p, s_i) with labels $l + i$. Remove the edge (p, o). Repeat flattening on p until all its successor nodes are either input nodes or operator nodes other than $\oplus$.*

Note that elimination of a node adds new children to the root node, which are in turn processed and that the order of the nodes is preserved. Fig. 4 shows

the effect on the ADDGs of Fig. 2. On the left, note the two outgoing edges with the same label, they correspond to disjunct slices of the computation.

Commutativity. A commutative operator allows to permute the arguments as shown in Fig. 3(b). As a consequence, one cannot use the labels on the edges to find corresponding arguments for operators that should perform the same computation. E.g., the +-nodes of Fig. 4 are commutative. To find the correspondence between their arguments, a matching operation is needed.

Operation 3 (Matching). *Given a pair of commutative operators in two different* ADDGs *matching selects pairs of corresponding edges. To do so, it has to look-ahead in the subtrees of the edges, using information about operator labels and transitive dependency mappings to eliminate candidates. This boils down to a recursive application of the method described in Sec. 4.*

Consider the two addition operators in the two ADDGs of Fig. 4. Edge 1 in the left ADDG pairs with edge 4 in the right ADDG, as they are the only ones leading to the input array A. Both +-nodes haves two edges leading to an operator labeled f, so further look-ahead is needed. In both cases, one of the operator nodes leads to input array A and the other to B, hence the correct pairing is $(2, 1)$ and $(3, 3)$. Finally, the left ADDG has two edges labeled 4, leading to input array B, also edge 2 of the right ADDG leads to B, resulting in two pairs $(4, 2)$.

Combination of associativity and commutativity. Operators can be both associative and commutative, increasing the number of equivalent forms, as illustrated in Fig. 3(c) for the $\circledast$-operator. As already explained on our example, the flattening operation has to be followed by a matching operation.

Other Properties. Operations for handling other properties (distributivity, inverse of an operator, identity element of an operator, evaluation of constant values) can be developed in a similar way by a combination of reduction to a suitable normal form and matching.

4 Equivalence Checking Method

We start by introducing a sufficient condition for equivalence between programs. Next, in Sec. 4.1, we explain a traversal based method to check the condition. Finally, in Sec. 4.2 we discuss how recurrences are tackled.

Two programs are equivalent when they have identical outputs for identical inputs. Assuming they have the same input and output arrays, we distinguish the following two conditions. For each output element in both programs:

COND-A: The set of output-to-input mappings is the same; and
COND-B: The computation is the same.

Together, they ensure that each output element is obtained by applying the same function on the same input elements, i.e., that both programs are

equivalent. The ADDG is an abstraction of the computation that allows one to do the verification for groups of elements at once. The verification is based on a synchronous traversal of the ADDGs from output to input. Using the structure of the ADDGs, the dependency mappings and the operators, it is verified whether both programs perform the same computation.

4.1 Synchronized Traversal of Two ADDGs

Starting with a proof obligation about the equality of the outputs we try to reduce it to proof obligations about equality of inputs that are trivially satisfied.

Definition 8 (Proof Obligation). *Given two* ADDGs*, G_1 and G_2. A* primitive proof obligation *is of the form $(v_1, v_2, \mathrm{M}^*_{\mathtt{O},\mathtt{v}_1}, \mathrm{M}^*_{\mathtt{O},\mathtt{v}_2})$, where v_1 and v_2 are arrays from G_1 and G_2, respectively, and $\mathrm{M}^*_{\mathtt{O},\mathtt{v}_1}$ and $\mathrm{M}^*_{\mathtt{O},\mathtt{v}_2}$ are transitive dependency mappings with identical domains, i.e., $dom(\mathrm{M}^*_{\mathtt{O},\mathtt{v}_1}) = dom(\mathrm{M}^*_{\mathtt{O},\mathtt{v}_2})$. A* proof obligation *is a conjunction of primitive proof obligations.*

Definition 9 (Truth of Proof Obligation). *A proof obligation is true if each of its primitive proof obligations is true. A primitive proof obligation $(v_1, v_2, \mathrm{M}^*_{\mathtt{O},\mathtt{v}_1}, \mathrm{M}^*_{\mathtt{O},\mathtt{v}_2})$ is true if $\mathtt{v}_1[\mathrm{M}^*_{\mathtt{O},\mathtt{v}_1}(i)] = \mathtt{v}_2[\mathrm{M}^*_{\mathtt{O},\mathtt{v}_2}(i)]$ for all i in $dom(\mathrm{M}^*_{\mathtt{O},\mathtt{v}_1})$ for any execution of the program.*

Operation 4 (Proof Initialization). *A first requirement is that the data-flow is correct, i.e., each read element is either input or has been written before. A second requirement is that both programs output the same set of elements. These requirements need to be checked before the actual verification by inspecting definition and operand domains of statements.*

For each output array O_i in both G_1 and G_2, let W_i be the total definition domain of O_i (the union of the definition domains of the defining statements). Let p_i be the primitive proof obligation $(O_i, O_i, \mathrm{M}^*_{\mathtt{O}_i,\mathtt{O}_i}, \mathrm{M}^*_{\mathtt{O}_i,\mathtt{O}_i})$ with $dom(\mathrm{M}^*_{\mathtt{O}_i,\mathtt{O}_i}) = W_i$. The initial proof obligation is the conjunction of all p_i.

Obviously, the initial proof obligation implies equivalence of both programs.

Definition 10 (Terminal Proof Obligation). *A primitive proof obligation $p = (v_1, v_2, \mathrm{M}^*_{\mathtt{O},\mathtt{v}_1}, \mathrm{M}^*_{\mathtt{O},\mathtt{v}_2})$ is terminal iff v_1 and v_2 are input arrays.*

A terminal proof obligation is true according to Def. 9 iff $v_1 = v_2$ and $\mathrm{M}^*_{\mathtt{O},\mathtt{v}_1} = \mathrm{M}^*_{\mathtt{O},\mathtt{v}_2}$, i.e., the output-to-input mappings select the same elements in the same input arrays.

The following reduction introduces primitive proof obligations where the nodes are not arrays; such obligations are auxiliary obligations, which have not been given a formal meaning. They are further reduced in subsequent reductions.

Operation 5 (Reduction of Primitive Proof Obligation). *Let the primitive proof obligation to be reduced be $p = (v_1, v_2, \mathrm{M}^*_{\mathtt{O},\mathtt{v}_1}, \mathrm{M}^*_{\mathtt{O},\mathtt{v}_2})$. The reduction generates a set (conjunction) of new primitive proof obligations that replaces p.*

Case 1. *v_1 is an array node. For each successor node of v_1 that is an array node an* array-array reduction *is applied and for each successor node of v_1 that is an operator node an* array-operator reduction *is applied.*

- Array–array reduction. *Suppose that the successor node is the array node a. For every dependency mapping $\mathrm{M}_{\mathtt{v_1,a}}$, $\mathrm{M}^*_{\mathtt{0,a}} := \mathrm{M}^*_{\mathtt{0,v_1}} \bowtie \mathrm{M}_{\mathtt{v_1,a}}$ is computed, and the proof obligation $(a, v_2, \mathrm{M}^*_{\mathtt{0,a}}, \mathrm{restrict}(\mathrm{M}^*_{\mathtt{0,v_2}}))$ is added, where $\mathrm{restrict}(\mathrm{M}^*_{\mathtt{0,v_2}})$ is the projection of $\mathrm{M}^*_{\mathtt{0,v_2}}$ on $dom(\mathrm{M}^*_{\mathtt{0,a}})$.*
- Array–operator reduction. *Suppose that the successor node is the operator node f. The proof obligation $(f, v_2, \mathrm{M}^*_{\mathtt{0,v_1}}, \mathrm{M}^*_{\mathtt{0,v_2}})$ is added.*

Case 2. *v_2 is an array node: this case is similar to* **Case 1**.

Case 3. *v_1 and v_2 are both operator nodes $v_1 = v_2 = \odot$. If $\odot$ is associative, apply flattening on $\odot$-node on both sides. Let $x_1, \ldots, x_{k'}$ and $y_1, \ldots, y_{l'}$ be the successor nodes of v_1 and v_2, with labels $\{1, \ldots, k\}$ and $\{1, \ldots, l\}$ respectively, for edges between them (where $k \leq k'$ and $l \leq l'$). If $\odot$ is commutative, apply matching. Let x_i be matched with $y_{m(w_i)}$, where $w_i = label(v, x_i)$. If $\odot$ is neither associative nor commutative, then $m(w_i) = w_i$. For each pair $(x_i, y_{m(w_i)})$, $\forall i, 1 \leq i \leq k'$, $(x_i, y_{m(w_i)}, M_1, M_2)$ is added, such that, if x_i (resp. $y_{m(w_i)}$) is an operator node, then $M_1 = \mathrm{M}^*_{\mathtt{0,v_1}}$ (resp. $M_2 = \mathrm{M}^*_{\mathtt{0,v_2}}$), else $M_1 := \mathrm{M}^*_{\mathtt{0,v_1}} \bowtie \mathrm{M}_{\mathtt{v_1,x_i}}$ (resp. $M_2 := \mathrm{M}^*_{\mathtt{0,v_2}} \bowtie \mathrm{M}_{\mathtt{v_2,y_{m(w_i)}}}$).*

The method is summarized in Algorithm 1. The actual implementation uses the proof obligations and reasons over the program representation without manipulating its initial structure.

Algorithm 1: Outline of the equivalence checker.

```
Input: ADDGs G1 and G2 of the two functions.
Output: If they are equivalent, return True, else return False, with diagnostics.
P ⟵ ProofInitialization()
while P ≠ ∅ do
    p ⟵ SelectObligation()
    if TerminalObligation(p) then
        if not TrueObligation(p) then
            return (False, errorDiagnostics)
    else
        newObligations ⟵ ReduceObligation(p)
        if newObligations = ∅ then
            return (False, errorDiagnostics)
        else
            P ⟵ (P \ {p}) ∪ newObligations
return True
```

4.2 Handling Recurrences in the ADDG

Recurrences are detected when reduction leads to an array node that has already been visited. Clearly, it is inefficient to step through each instance of a recurrence.

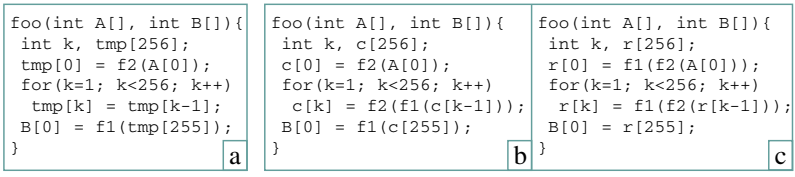

Fig. 5. Example program functions with recurrences

In most practical cases it can be avoided by computing the relation with the set of values at the end of the coil of recurrence, called the *across-recurrence mapping*. The key operation that enables such a computation is the positive *transitive closure* of an integer tuple relation.

Definition 11 (Across-recurrence Mapping). *Suppose we have a recurrence with* $\mathtt{v}, \mathtt{w}_1, \ldots, \mathtt{w}_k, \mathtt{v}$ *as the internal array nodes in the cycle that is entered on a path from array* u. *Then the transitive dependency mapping for the cycle from* v *back to* v *is given by,* $\mathrm{M}^*_{\mathtt{v},\mathtt{v}} := \mathrm{M}_{\mathtt{v},\mathtt{w}_1} \bowtie \mathrm{M}_{\mathtt{w}_1,\mathtt{w}_2} \bowtie \cdots \bowtie \mathrm{M}_{\mathtt{w}_k,\mathtt{v}}$. *The across-recurrence mapping between* u *and* v *is the transitive dependency mapping between* u *and* v *that is across the recurrence on* v *and it relates the elements of* u *to the elements of* v *that are assigned outside the cycle on the same path. It is defined as,* $\mathrm{M}^R_{\mathtt{u},\mathtt{v}} = \mathrm{M}_{\mathtt{u},\mathtt{v}} \bowtie \mathrm{M}'_{\mathtt{v},\mathtt{v}}$, *where* $\mathrm{M}'_{\mathtt{v},\mathtt{v}}$ *is calculated as follows:*

- *Compute positive transitive closure of the recurrent mapping:* $m := (\mathrm{M}^*_{\mathtt{v},\mathtt{v}})^+$
- *Get domain and range of the computed closure:* $d := \mathrm{domain}(m)$; $r := \mathrm{range}(m)$
- *Get domain and range of the end-to-end mapping:* $d' := (d - r)$; $r' := (r - d)$
- *Restrict the closure to the tuples in the end-to-end mapping:*
  $\mathrm{M}'_{\mathtt{v},\mathtt{v}} := \{x \to y \,|\, x \to y \in m \wedge x \in d' \wedge y \in r'\}$.

For a tuple relation F, its positive transitive closure F^+, is a tuple relation defined as $x \to z \in F^+ \Leftrightarrow x \to z \in F \vee \exists y$ s.t. $x \to y \in F \wedge y \to z \in F^+$. A remark here is that exact transitive closure of a relation in closed form is not computable in the general case. A sufficient condition [9] for its computation is that, if the tuple of the relation is $[\vec{k_1}] \to [\vec{k_2}]$, then $\vec{k_2} = \vec{k_1} + \vec{c}$, where $\vec{c}$ is a vector of integer constants.

Depending on the nodes that appear in the cycle of recurrence, we distinguish two possible cases of recurrences in an ADDG.

Recurrence without computation. In this case, no operator nodes are present in the recurrence cycle. Fig. 5(a) shows an example program having such a recurrence without computation. During traversal (or during array node elimination), if such a recurrence is encountered on a given path, the across-recurrence mapping is computed and this essentially eliminates the cycle on the path. This is illustrated in the Fig. 6(a), where v is the array at the entry to the cycle and no operator nodes exist on the path p.

Recurrence with computation. In this case, operator nodes are present in the recurrence cycle. Fig. 5(b) and (c) show an example of equivalent program pair

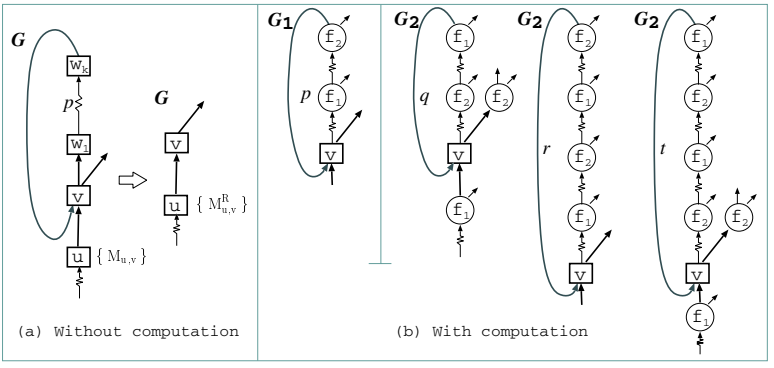

Fig. 6. Two cases of recurrence

that have such a recurrence with computation. When confronted with this recurrence, it is required that the across-recurrence mapping be computed on the two corresponding `ADDGs` in a synchronized way. That is, we need to ensure that the new dependency mappings computed account for the same computation. In order to be able to do that we first have to get identical sequence of operators on the recurrence cycles on both the `ADDGs`. This is achieved by *unfolding*.

Operation 6 (Unfolding). *Suppose G_1 and G_2 are the* `ADDGs` *being traversed in synchronization and we detect a recurrence on one of them, say, G_1, with $(f_1, \ldots, f_k, f_1)$ as operator nodes on the cycle. The traversal ensures that the corresponding nodes traversed on G_2 are also $(f_1, \ldots, f_k, f_1)$. If a recurrence is also detected at this point on G_2, we are done. Otherwise, we unfold G_1, by stepping through the recurrence along with G_2 as many times as it takes to reveal a cycle with identical sequence of operators on G_2.*

Fig. 6(b) shows G_1 with cycle p and G_2 with the basic possibilities for a cycle, viz,, operators shifted by one (q), unfolded once completely (r) and both unfolded once and shifted by one (t). In the example pair in Fig. 5(b) and (c), the operator is shifted by one in the transformed program.

Once we have established matching cycles on the two sides by unfolding, we have transitive dependency mappings for the two corresponding cycles, $M_1 := \{[\vec{a_1}] \rightarrow [\vec{a_2}] \,|\, C_1\}$ and $M_2 := \{[\vec{c_1}] \rightarrow [\vec{c_2}] \,|\, C_2\}$, where C_1 and C_2 are affine constraint expressions. Now, in order to compute the across-recurrence mapping that ensures same computation on both sides we combine the two transitive dependency mappings and use the *combined mapping* M as the dependency mapping for the cycle, given by, $M := \{[\vec{a_1}, \vec{c_1}] \rightarrow [\vec{a_2}, \vec{c_2}] \,|\, C_1 \wedge C_2\}$, where the vector variables in the formulae describing M_1 and M_2 are made distinct by renaming. This mapping is used for the computation of the mapping M' as described in the Def. 11. M' is then split into M'_1 and M'_2 along the same dimensions that were combined earlier. These mappings are used in calculating the across-recurrence mappings on the respective `ADDGs`.

5 Discussion

As we described, the method is a synchronized traversal on the two ADDGs. Our method traverses corresponding paths only once and tables all established equivalences. Therefore, if we assume that the number of maximal slices of computation in the ADDGs is very small compared to their sizes, the complexity of the traversal is linear in the size of the larger of the two ADDGs, i.e., $O(\max(|V_1| + |E_1|, |V_2| + |E_2|))$. The operations on the integer domains and relations, that our method calls, are based on checking the validity of Presburger formulae, whose best known upper bound has triple-exponential complexity in the length of the constraint expressions. However, the expressions are usually small enough in practice and the operations are feasibly computed with a dependence test like Omega Test [12]. Therefore, we can assume the time for these operations to be bounded by a constant. Hence, the overall complexity is still in the order of the traversal.

With a prototype implementation of the method, we have been able to check equivalences of real-life program pairs efficiently. For programs with 1000 lines of uncommented C code, with control and data-flow complexity comparable to real-life signal processing algorithm kernels, the tool took less than 100 seconds on a standard desktop [14].

Typically, as can be expected, the original and the transformed program pairs seen in practice do not fall in the class that we have assumed for our method, at least not in all respects. But as discussed in Sec. 2.1, some restrictions can be relaxed by using code-preprocessing tools. They are used to pre-process the initial and the transformed programs separately, before passing them to our equivalence checker. For instance, using tools that are available to us in-house, we are able to handle programs that are not in DSA and also not having static control-flow (because of data-dependent if-conditions). Additionally, since ours is an intra-procedural method, by inlining functions in both programs using a function-inlining tool, we are able to verify correctness of inter-procedural code transformations from the categories that we handle.

6 Related Work

Undecidability of the program equivalence problem implies that any effort start with the definition of a decidable class of programs that is of interest. Hence, the problem has been addressed by various researchers for different program classes with different applications in mind. The problem we address is distinct by its central requirement to represent and maintain the relationships among elements of the arrays in the programs in closed form. Unrolling deeply nested loops with large bounds is clearly infeasible for real-life signal processing programs. To add to this, algebraic transformations will require an infeasible search for normalization on a combination of the unrolled statements. Hence, we restrict our discussion of related work to methods that do not propose loop unrolling.

Translation validation [8, 11] and fractal symbolic analysis [10], both present methods which show semantic equivalence of two versions of programs. In the case of the former, the comparison is between the source and the target code. These methods are distinct from ours in that they essentially try to heuristically *infer* a sequence of legal transformations that can relate the two programs. Instead, we are able to directly check for equivalence of programs that are in a suitable language class. Also, their methods do not handle algebraic transformations. The work most related to ours, because we address the same class of programs, is the algorithm recognition method presented in [4]. Again, algebraic transformations are not handled by them. Another distinction is that, all these methods do not pay attention to debugging support which is very important in the context of source code transformations.

7 Conclusions

We have presented a program equivalence checking method that enables verification of global source code transformations. The transformations considered are the ones that are widely reported in current practice relating to development of data-intensive software for high-performance and low-power systems. The program class handled is also the one that is often referred to in the literature relevant to the application domain of the transformations. The method is fully automatic and efficient. Hence, we believe that it provides a practical addition to the toolbox used by programmers applying source code transformations.

References

1. J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren. Conversion of control dependence to data dependence. In *POPL*, pp. 177–189. ACM, 1983.
2. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, 2001.
3. U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Publishers, 1988.
4. D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. In *8th Euro-Par*, pp. 309–313. Springer, 2002.
5. F. Catthoor, S. Wuytack, E. de Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design.* Kluwer Publishers, 1998.
6. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
7. P. Feautrier. Array expansion. In *ICS*, pp 429–441. ACM, 1988.
8. B. Goldberg, L. Zuck, and C. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. In *International Workshop on Compiler Optimization Meets Compiler Verification*, ENTCS. Elsevier, 2004.
9. W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. *Intl. Journ. of Parallel Prog.*, 24(6):579–598, 1996.

10. N. Mateev, V. Menon, and K. Pingali. Fractal symbolic analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):776–813, 2003.
11. G. C. Necula. Translation validation for an optimizing compiler. In *SIGPLAN Programming Language Design and Implementation*, pp. 83–95. ACM, 2000.
12. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.
13. K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *Design, Automation and Test in Europe*. IEEE, 2005.
14. K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Automatic Verification of Source Code Transformations on Array-Intensive Programs: Demonstration with Real-life Examples. Tech. Rep. CW 401, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, 2005.
15. R. A. van Engelen and K. A. Gallivan. An efficient algorithm for pointer-to-array access conversion for compiling and optimizing DSP applications. In *International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems*, pp. 80–89. IEEE, 2001.
16. P. Vanbroekhoven, G. Janssens, M. Bruynooghe, H. Corporaal, and F. Catthoor. A step towards a scalable dynamic single assignment conversion. Tech. Rep. CW 360, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, 2003.
17. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.