Experience with Widening based Equivalence Checking in Realistic Multimedia Systems

Sven Verdoolaege *Martin Palkovic* *Maurice Bruynooghe*
Gerda Janssens *Francky Catthoor*

Report CW 572, December 2009

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Experience with Widening based Equivalence Checking in Realistic Multimedia Systems

Sven Verdoolaege *Martin Palkovic*[*] *Maurice Bruynooghe*
Gerda Janssens *Francky Catthoor*[†]

Report CW 572, December 2009

Department of Computer Science, K.U.Leuven

Abstract

The application of loop and data transformations to array and loop intensive programs is crucial to obtain a good performance. Designers often apply these transformations manually or semi-automatically. For the class of static affine programs, automatic methods exist for proving the correctness of these transformations. Realistic multimedia systems, however, often contain constructs that fall outside of this class. We present an extension of a widening based approach to handle the most relevant of these constructs, viz. accesses to array slices, data dependent accesses and data dependent assignments, and report on some experiments with non-trivial applications.

[*]IMEC, Leuven, Belgium
[†]IMEC, Leuven, Belgium and Katholieke Universiteit Leuven, Belgium

1 Introduction

Especially in embedded systems, program transformations are unavoidable when going from the initial specification of an algorithm to its final implementation targeting the best performance, energy and/or area on a given platform [5, 21]. These transformations can be automatic, semi-automatic or manual. In all cases it is crucial to know that the transformed program preserves the behavior of the original. This knowledge can be obtained by formally verifying the equivalence of the two programs, where two programs are considered to be equivalent if they produce the same outputs when given the same inputs.

Programs for our target domain of multimedia and communication systems typically take one or more arrays as input, apply some operations in multiple nested loops, and then produce one or more arrays as outputs. Two such programs are then considered equivalent if the values of the output arrays in both programs are the same whenever the values of the input arrays are.

For pairs of programs that strictly belong to the class of static affine programs, i.e., programs with static control flow and piecewise affine expressions for all loop bounds, conditions and array accesses, this equivalence problem has been addressed by three approaches [3, 16, 18]. These approaches can handle any per statement or per array piecewise quasi-affine loop or data transformation, including combinations of loop interchange, loop reversal, loop skewing, loop distribution, loop tiling, loop unrolling, loop splitting, loop peeling and data-reuse transformations, without any a priori knowledge about which transformations have been performed. The approaches also detect and handle recurrences in both programs *fully automatically*, with some restrictions, depending on the particular approach.

However, realistic multimedia applications typically contain constructs that do not fit entirely into the class of static affine programs. As proposed by [15], some of these constructs can be hidden using preprocessing, but only if the undesirable constructs are left untouched by the transformations. Furthermore, some preprocessing steps may replace one undesirable construct by another or they may abstract away so much information so as to render equivalence checking impossible. Instead, we propose to handle the most crucial such constructs in our target domain directly. In particular, in this paper we address reads and writes to entire rows (or higher-dimensional slices) of arrays, data-dependent read and write accesses and data-dependent assignments.

Illustrative Example The pair of programs shown in Figure 1 illustrates some constructs that appear in realistic multimedia programs. Some of these can be handled using our basic approach of [18, 19], while some others require the extensions discussed in this paper. In order not to overload the example, it does not illustrate all the loop and data transformations listed above that both our basic and our extended approaches can handle.

Both programs take two arrays as input (`in1[M][N]` and `in2[M]`) and produce an array as output (`out[M]`). Furthermore, we assume that the elements of `in2` have values between -1 and 1. Both programs also have two temporary arrays, one zero-dimensional array (`m` in Program 1 and `t` in Program 2), i.e., a scalar, and one two-dimensional array (`A` in Program 1 and `B` in Program 2). The following transformations have been applied to Program 1 to arrive at Program 2: the call of `h` has been moved up; the two outer loops have been merged, after shifting the iterator of the second loop by one; the outer iterations of the merged loop have been peeled off and the rows of the `B` array have been reversed with respect to those of the `A` array. It should be noted that the equivalence checking procedure is not informed about the transformations that have been applied. The only input is the pair of programs. The two programs are not completely equivalent because Program 1 contains an extra assignment to `A` in Line 7, which is missing from Program 2. However, despite this difference, many elements in the output arrays will still be equivalent and our procedure determines the set of indices in the output arrays where this condition holds.

Line 4 of Program 1 contains a recurrence on `m`, while Line 10 of Program 2 contains a similar recurrence on `t`. Our basic approach automatically discovers that `t` is equal to `h(m)` for all iterations of the enclosing loops by applying a widening step during the equivalence checking. This widening step is summarized in Section 3.

Our basic approach assumes that the index expressions are complete, i.e., that individual array elements are accessed, and completely known (as an expression in terms of the loop iterators and

```
1  for (i = 0; i < M; ++i) {
2    m = i+1;
3    for (j = 0; j < N; ++j)
4      m = g(h(m), in1[i][j]);
5    compute_row(h(m), A[i]);
6  }
7  A[5][6] = 0;
8  for (i = 0; i < M; ++i)
9    if (i+in2[i]>=0 && i+in2[i]<M)
10     out[i] = f(A[i + in2[i]]);
11   else
12     out[i] = 0;
```

(a) Program 1

```
1  if (M > 0) {
2    t = h(1);
3    for (j = 0; j < N; ++j)
4      t = h(g(t, in1[0][j]));
5    compute_row(t, B[M - 1]);
6  }
7  for (i = 1; i < M; ++i) {
8    t = h(i+1);
9    for (j = 0; j < N; ++j)
10     t = h(g(t, in1[i][j]));
11   compute_row(t, B[M - 1 - i]);
12   if (M-i-in2[i-1]>=0&&M-i-in2[i-1]<M)
13     out[i-1] = f(B[M - i - in2[i-1]]);
14   else
15     out[i-1] = 0;
16 }
17 if (M > 0) {
18   if (-in2[M-1] >= 0 && -in2[M-1] < M)
19     out[M-1] = f(B[-in2[M-1]]);
20   else
21     out[M-1] = 0;
22 }
```

(b) Program 2

Figure 1: A pair of almost equivalent programs.

the parameters) at analysis time. The programs in Figure 1 contain several constructs where these conditions do not hold. In particular, Line 5 of Program 1 contains a call to `compute_row` that writes a complete row of `A` rather than an individual element. Similarly, Line 10 contains a call to a function `f` *reading* an entire row of `A`. Furthermore, the index of the row being read is partly determined by runtime information (the value of `in2[i]`). The same runtime information also determines whether the function is called at all, as it appears in the condition in Line 9.

In this paper, we propose extensions to our basic approach for handling such constructs. With these extensions, we can correctly identify that all elements except those in the range $[4, 6]$ of the `out` array are equivalent. The update of `A[5][6]` in Line 7 means that row 5 of `A` as a whole is different from the corresponding row $M - 6$ of `B` in the call of `f`. Since `in2[i]` ranges between -1 and 1, this row may be used in the computation of elements 4 to 6 of the output array, explaining how our extended approach detects that these elements are not guaranteed to have the same values in the two programs.

After an overview of related work in Section 2, we summarize our basic approach in Section 3. Extensions for handling reads and write to array slices and data-dependent reads, writes and assignments are worked out in Section 4 and form the main contribution of this paper. Section 5 describes how to locate the source of an error. In Section 6, we discuss our experiments and we conclude in Section 7. This paper is a revision and extension of our earlier work [20]. In particular, we now also describe how to handle data dependent writes (Section 4.6) and data dependent assignments (Section 4.7) and we describe how to locate the source of an error (Section 5).

2 Related Work

Many approaches have been proposed for equivalence checking, but few include some form of induction, which is needed for handling recurrences. Even general theorem provers such as ACL2 [10] require manual intervention by the user to specify induction hypotheses. Regression verification [9] performs induction and can handle recurrences, but only if they have been rewritten as recursive function calls. Their technique for matching functions in the two input programs prohibits them from verifying any non-trivial loop transformation.

Three approaches that target the fully automatic verification of static affine programs with recurrences are our own widening based approach [18, 19], which serves as a basis for the extensions proposed in this paper, that of Barthou et al. [3] and that of Shashidhar et al. [16]. The latter two both rely on a transitive closure operation [11], which basically restricts their approaches to programs containing only uniform recurrences. Our widening based approach handles both uniform and non-uniform recurrences. Furthermore, the approach of [16] cannot even accurately handle the recurrences in Figure 1 because it cannot express that the value of `m` in Line 5 of Program 1 for a given iteration of the `i` loop depends on the *entire* `i`th row of the input array `in1`. Since the extensions of this paper involve similar relations, they cannot be applied to the approach of [16]. The approach of [3], on the other hand, could serve as a basis for our extensions, but unlike the approach of [18], it does not handle transformations that exploit the commutativity of operations.

For dealing with constructs beyond static affine programs, some authors have proposed to perform some preprocessing steps on both programs before the actual equivalence checking. For example, Shashidhar [15, Chapter 9] proposes the application of pointer conversion, if-conversion, function encapsulation, function inlining and dynamic single assignment (DSA) conversion. We do not use the latter preprocessing step as we effectively construct a DSA representation in the first step of our approach, while the other preprocessing steps are mostly complementary to our extensions. Function inlining could help to remove accesses to array slices, but it may introduce other undesirable constructs, while none of these steps can help to remove data dependent accesses. Replacing the standard exact dataflow analysis in our approach by fuzzy dataflow analysis [2] would allow us to handle more general data dependent constructs, but it would require further extensions beyond those proposed in this paper and it would not eliminate the need for our proposed extensions. We consider these further extensions as part of future work.

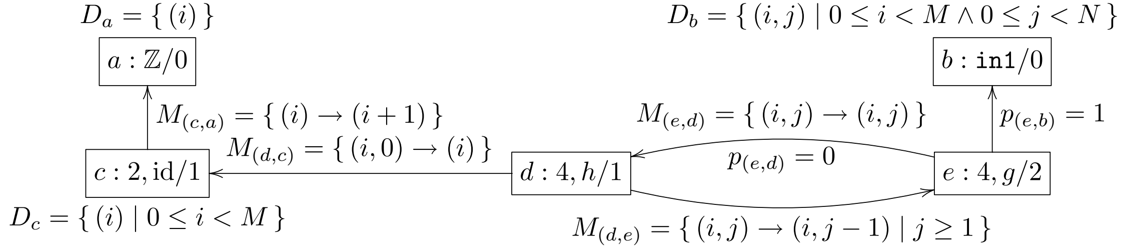

Figure 2: Partial dependence graph for Program 1 in Figure 1(a). Each node is labeled "node name: location,operation name/arity". the location is undefined for input computations and is therefore omitted on computations a and b.

3 Basic Methodology

Our equivalence checking procedure takes two programs as input and determines whether they are equivalent. The procedure consists of two main steps. In the first step we set up a *dependence graph* that specifies for each value used in the program where and how it was computed. The second step performs the actual equivalence checking on this dependence graph abstraction.

3.1 Dependence Graphs

The construction of a dependence graph is itself performed in three substeps. First, the *iteration domains* and *access relations* are extracted from the program. Then, these iteration domains and access relations are used to perform *dependence analysis*. Finally, the dependence graph is constructed based on information from the input program and the results of the dependence analysis. In this section, we describe how to construct the part of the dependence graph that corresponds to the statements in Line 4 of Program 1 in Figure 1(a) and above. This dependence graph fragment is shown in Figure 2. The construction of the remainder of the dependence graph requires some of the extensions described in Section 4.

The programs we consider as inputs to our equivalence checking procedure consist of several loop nests with integer loop iterators. Each execution of a statement within a program can be represented by the values of the loop iterators of the enclosing loops. The list of these values, ordered from outermost to innermost loop, is called an *iteration vector*. For example, the first iteration of the statement in Line 4 of Program 1 is represented by the iteration vector $(0, 0)$, while the last iteration is $(M - 1, N - 1)$, assuming M and N are strictly positive. The set of all such iteration vectors associated to a given statement is called the *iteration domain* of the statement. For example, the iteration domain of the statement in Line 4 is

$$D_4 = \{ (i,j) \in \mathbb{Z}^2 \mid 0 \le i < M \wedge 0 \le j < N \}.$$

The constraints that define these iteration domains are obtained from the lower and upper bounds of all enclosing loops and from the conditions in all enclosing `if`-statements. The symbolic constants M and N in these constraints are called *parameters*.

Each read or write access in a statement is represented by an *access relation*. This access relation maps the iteration vector to an array index. For example, the read access `in1[i][j]` in Line 4 is represented by the access relation $\{ (i,j) \to (i,j) \}$, while the accesses to `m` are represented by access relations $\{ (i,j) \to () \}$. Since scalars such as `m` are treated as zero-dimensional arrays, the only possible index is the zero-dimensional $()$.

Standard exact *dependence analysis* [7] is then used to figure out for each read from an element of an array (or a scalar), where the element was written for the last time prior to the read. Each dataflow analysis invocation takes a list of write access relations and a single read access relation as input, together with information about the relative positions in the program text of the statements

containing these accesses. Dataflow analysis determines for each read, which of the writes was the last to write to the same array element before the given read. The result is a *dependence relation* that maps iterations of the statement containing the read to iterations of the statement containing the write. By default, a read from an uninitialized array element, i.e., a read without a corresponding write, is treated as a read from an input array. For example, the source of the value of `m` read in Line 4 of Program 1 depends on the value of the j iterator. For $j = 0$, the source is the copy operation in Line 2, resulting in a dependence relation that maps the first iteration of the inner loop to the same iteration of the outer loop, i.e., $M_{(d,c)} = \{ (i,j) \to (i) \mid j = 0 \}$. For $j \geq 1$, the source is the call to `g` in the previous iteration of the j loop, resulting in the dependence relation $M_{(d,e)} = \{ (i,j) \to (i, j-1) \mid j \geq 1 \}$. The subscripts on these dependence relations correspond to the nodes in Figure 2, as explained below.

In order to be able to apply exact dataflow analysis, the input programs need to satisfy the usual restrictions of static affine programs, i.e., static control flow, quasi-affine loop bounds and quasi-affine index expressions. Recall that quasi-affine expressions consist of additions, constant multiplication and integer division by a constant. We also assume that all functions called in the program are pure. In Section 4, we will somewhat relax the quasi-affine index expression requirement.

The dependence graph $G = (V, E)$ is constructed from the input program and the results of the dependence analysis. We first describe the nodes of the graph and then continue with the edges. It is customary to take the statements of the program as nodes in a dependence graph. However, a statement may contain several nested operations (or function calls), which need to be represented separately in order to be able to match them in the two programs. We therefore use *computations* as nodes in our dependence graph, where a computation is the set of all executions of a given operation, i.e., an operation together with the iteration domain of the statement in which it appears. Each computation has the following characteristics,

- operation name
- arity r (number of arguments)
- iteration domain D
- location, e.g., a line number.

The location describes the location in the program text where the operation is performed and is only used to help the user to distinguish between computations that perform the same operation. For example, the statement in Line 4 of Program 1 yields two computations, one (e) for the call to `g` and one (d) for the call to `h`. Both of these computations have iteration domain $D_d = D_e = D_4$. The names of the computations are those from Figure 2. The first computation calls "g/2", with g the name of the operation and 2 its arity, while the second calls "h/1".

Besides computations derived from operations, we also introduce some additional computations. First, we introduce a computation for each input array. The "iteration domain" of such a computation is the set of array elements, its name is the name of the input array and its arity is zero. For example, the iteration domain of the input array `in1` is $D_b = \{ (i,j) \in \mathbb{Z}^2 \mid 0 \leq i < M \wedge 0 \leq j < N \}$, while that of input array `in2` is $D_j = \{ (i) \in \mathbb{Z} \mid 0 \leq i < M \}$. Second, we introduce a special "computation" that represents the integers, with iteration domain $\mathbb{Z}$, name "$\mathbb{Z}$" and arity zero. Any time an affine combination of the enclosing loop iterators and the parameters is used outside of an index expression, it is implicitly replaced by a read from this "$\mathbb{Z}$" array. For example, the affine expression `i+1` in Line 2 of Program 1 is considered as a read from the element of array $\mathbb{Z}$ with index $i + 1$. Third, for each statement that does not perform any operation, we introduce a "copy computation" with name "id" and arity 1. For example, since the affine expression in Line 2 is treated as a read from an array, the statement contains no operations and therefore has a corresponding copy computation (c) that copies a value from the "$\mathbb{Z}$" array to the (zero-dimensional) `m` array. Finally, we introduce a special output computation, with as iteration domain the set of elements of the output array, name "Out" and arity 1. For simplicity we assume here that the program contains a single output array, in this case `out`. Our

implementation allows any number of output arrays. Of these additional computations, only the copy computations have a location. The others each perform a unique operation and therefore do not require an identifying location.

The edges of the dependence graph are derived from the results of the dependence analysis. In particular, there is an edge between any pair of computations a and b such that some iteration of computation a requires a value computed by some iteration of computation b as one of its arguments. Each edge $e = (a, b)$ has the following characteristics,

- a *dependence relation* $M_e \subseteq D_a \to D_b$, with D_a and D_b the iteration domains of a and b.
- an *argument position* p_e, with $1 \leq p_e \leq r_a$, indicating which argument of the operation of a depends on a value from b.

Unsurprisingly, if computation a has an array access as one of its arguments, then there is an edge from a to the computation that last wrote to the accessed array element. The dependence relation on such an edge is exactly the dependence relation produced by dependence analysis. Examples of such edges are those between nodes d and e and between nodes d and c. The dependence relations on these edges have been derived above. Instead of an array access, a given argument of computation a may also be another computation b. In this case, both computations involved have the same iteration domain and the dependence relation is simply the identity mapping. An example of such a dependence is that between the calls of functions `g` and `h` in Line 4 of Program 1. The resulting edge in the dependence graph is that from node e to node d in Figure 2.

3.2 Equivalence Checking

The equivalence checking is performed by propagating desired equivalences from the output computations of the dependence graphs to the leaves of these graphs (i.e., input computations or constant functions). Once the leaves have been reached, the equivalences that we are actually able to prove are then propagated back to the output computations. This two way propagation is useful not only to see which of the elements of the output arrays are equivalent in case we cannot prove that the full arrays are equivalent, but also in our treatment of recurrences and commutative operations. We will briefly discuss recurrences below and refer to [18, 19] for commutative operations, where we also explain our treatment of associative operations.

During the equivalence checking, an *equivalence tree* is constructed. Each node in this tree expresses a desired equivalence between iterations of a pair of computations and, after the second phase, the actual equivalence that we are able to prove. In particular, to each node in the equivalence tree, we associate a pair of computations $(v_1, v_2) \in V_1 \times V_2$, a relation $R^{\text{want}} \subseteq D_{v_1} \times D_{v_2}$ containing pairs of computation iterations for which we want to prove equivalence, and a relation $R^{\text{lost}} \subseteq R^{\text{want}}$ containing those pairs of iterations for which we have *not* been able to prove equivalence. The pairs of iterations that have been proven equivalent are then given by $R^{\text{want}} - R^{\text{lost}}$. The children of a node describe equivalences that are needed to prove the equivalence of the parent. The root node of the equivalence tree expresses the fact that we want the output computations of both programs to compute the same values. That is, the nodes v_1 and v_2 of the root are the output computations and the R^{want} relation is an identity mapping between the iteration domains of these two computations, where we assume that these iteration domains are identical. The R^{lost} relation is initially undefined and will be filled in by the second pass of the algorithm. In our example, the output computations have iteration domain $\{\, (i) \in \mathbb{Z} \mid 0 \leq i < M \,\}$. The R^{want} relation of the root node of the equivalence tree is therefore

$$\{\, (i) \leftrightarrow (i) \mid 0 \leq i < M \,\}.$$

Our procedure treats all functions as black boxes and so it will only consider a pair of iterations of two computations to be equivalent if it can prove that all the corresponding arguments are equivalent. More specifically, an equivalence node n between v_1 and v_2 such that v_1 and v_2 perform different operations and such that neither of these operations is the special copy operation id, has

its R_n^{lost} relation set to its R_n^{want} relation. Otherwise, child nodes c_{e_1,e_2} are created for each pair of edges $e_1 = (v_1, u_1)$ and $e_2 = (v_2, u_2)$ with the same argument position $p_{e_1} = p_{e_2}$ emanating from both computations in the corresponding dependence graphs. The set of all such pairs is denoted T_n. In case one or both operations are copy operations, we only propagate along one of the copy operations.

The dependence analysis ensures that for any computation v and for any argument position $p \leq r_v$, the iteration domain of v is partitioned by the domains of the dependence relations of the edges (v, u) with $p_{(v,u)} = p$. For each pair of edges with the same argument position from the two dependence graphs, we therefore need to prove that the same values arrive at this argument position for all elements in the intersection of the domains of the dependence relations. This subset of the original R_n^{want} relation then needs to be reformulated in terms of the iterations of the computations at the other ends of the edges. That is, we need to prove equivalence according to the following R^{want} relation for each child node c_{e_1,e_2}, created by following the edges $e_1 = (v_1, u_1)$ and $e_2 = (v_2, u_2)$:

$$R_{c_{e_1,e_2}}^{\text{want}} = (M_{e_1} \oplus_{\rightarrow}^{\leftrightarrow} M_{e_2})\, R_n^{\text{want}}. \tag{1}$$

The $\oplus_{\rightarrow}^{\leftrightarrow}$ operator combines mappings of type $D_{v_1} \rightarrow D_{u_1}$ and $D_{v_2} \rightarrow D_{u_2}$ into one of type $(D_{v_1} \leftrightarrow D_{v_2}) \rightarrow (D_{u_1} \leftrightarrow D_{u_2})$, i.e., $M_{e_1} \oplus_{\rightarrow}^{\leftrightarrow} M_{e_2}$ is equal to

$$\{\, (\mathbf{i}_1 \leftrightarrow \mathbf{i}_2) \rightarrow (\mathbf{i}'_1 \leftrightarrow \mathbf{i}'_2) (\mathbf{i}_1) \rightarrow (\mathbf{i}'_1) \in M_{e_1} \wedge (\mathbf{i}_2) \rightarrow (\mathbf{i}'_2) \in M_{e_2} \,\}.$$

In our example, assume that there is a node in the equivalence tree relating computation d (Figure 2) in the dependence graph of Program 1 to the computation that corresponds to the call of `h` in Line 2 of Program 2 (node n_{23} in Figure 11) with $R_n^{\text{want}} = \{(0, 0) \leftrightarrow ()\}$. Since both computations perform operation `h`, propagation proceeds with the creation of children. In this case, the computation in the first dependence graph has two outgoing edges, while the computation in the second dependence graph has one outgoing edge, an edge that points to the $\mathbb{Z}$ computation with dependence relation $\{() \rightarrow (1)\}$. Two child nodes s_1 and s_2 are therefore created. The first child node s_1 has

$$\begin{aligned} R_{s_1}^{\text{want}} &= (\{(i, 0) \rightarrow (i)\} \oplus_{\rightarrow}^{\leftrightarrow} \{() \rightarrow (1)\})\, \{(0, 0) \leftrightarrow ()\} \\ &= \{((i, 0) \leftrightarrow ()) \rightarrow ((i) \leftrightarrow (1))\}\{(0, 0) \leftrightarrow ()\} \\ &= \{(0) \leftrightarrow (1)\}. \end{aligned}$$

The second child node has $R_{s_2}^{\text{want}} = \emptyset$, because the constraint $j \geq 1$ on the dependence relation between computations d and e conflicts with the fact that the single element in R_n^{want} has $j = 0$. Since computation c (Figure 2) performs a copy operation, a further propagation on node s_1 yields a node t relating the two $\mathbb{Z}$ computations with

$$\begin{aligned} R_t^{\text{want}} &= (\{(0) \rightarrow (1)\} \oplus_{\rightarrow}^{\leftrightarrow} 1_{\mathbb{Z}\rightarrow\mathbb{Z}})\, \{(0) \leftrightarrow (1)\} \\ &= \{(1) \leftrightarrow (1)\}. \end{aligned}$$

At some point, forward propagation reaches leaves in the dependence graphs and R^{lost} relations can be propagated back. Once the $R_{c_{e_1,e_2}}^{\text{lost}}$ relations of all $(e_1, e_2) \in T_n$ have been computed, they can be combined to compute the R_n^{lost} relation of the parent equivalence node,

$$R_n^{\text{lost}} = \bigcup_{(e_1,e_2)\in T_n} \left((M_{e_1}^{-1} \oplus_{\rightarrow}^{\leftrightarrow} M_{e_2}^{-1})\, R_{c_{e_1,e_2}}^{\text{lost}} \right) \cap R_n^{\text{want}}. \tag{2}$$

Note that the exposition in [18] performs this backpropagation on R^{got} instead of R^{lost}. We prefer R^{lost} here because it is required for backpropagation over expansion edges as explained in Section 4.4 and because in the ideal case, most R^{lost} relations are empty and therefore easier to manipulate. In the special case where the computations of n have arity zero, $T_n = \emptyset$ and so (2) simplifies to $R_n^{\text{lost}} = \emptyset$. For input computations, however, we are only allowed to assume that

```
for (j = 0; j <= 31; j++)
  MPG_IMDCT_Win(buffer[j], rawout);
```

Figure 3: Reading of a row from the `buffer` array and writing of the entire `rawout` array in a IMDCT call in the MP3 application.

elements with the same index are equivalent. In this case we therefore use $R_n^{\text{lost}} = R_n^{\text{want}} \setminus \{(\mathbf{i}) \leftrightarrow (\mathbf{i})\}$. Continuing the example above, we find $R_t^{\text{lost}} = \emptyset$ and then also

$$\begin{aligned} R_{s_1}^{\text{lost}} &= \left(\{(0) \to (1)\}^{-1} \oplus_{\to}^{\leftrightarrows} 1_{\mathbb{Z}\to\mathbb{Z}}\right) \emptyset \\ &= \emptyset. \end{aligned}$$

If there are any recurrences in the input programs, then we cannot simply apply the above propagation step, because the number of iterations in the loop may be a parameter and therefore unknown at analysis time. Basic propagation would then result in an infinite sequence of equivalence nodes with the same pair of computations. Our solution is to apply a "widening" operation [6] as soon as we detect a recurrence, replacing the infinite sequence by a finite sequence. In particular, if we create a new node n with the same pair of computations as an ancestor a in the equivalence tree, such that $R_n^{\text{want}} \not\subseteq R_a^{\text{want}}$, then we replace the R_a^{want} relation of the original equivalence node by the integer affine hull of $R_n^{\text{want}} \cup R_a^{\text{want}}$, intersected with the iteration domains of the corresponding computations, and start over from a. We continue taking affine hulls until $R_n^{\text{want}} \subseteq R_a^{\text{want}}$, at which point we apply an induction step by assuming that R_a^{lost} will be the empty set. The number of widenings is finite, because the dimension of R^{want} is increased in each step. The induction hypothesis is checked once we have handled the base case that escapes from the recurrence. It is therefore crucial that we propagate the R^{lost} relations back to a.

Consider for example the recurrences on `m` and `t` in Line 4 of Program 1 and Line 10 of Program 2. Let v_1 and v_2 be the computations that call the function `g` in these statements. Let us assume that the first equivalence node a relating these computations that we encounter has $R_a^{\text{want}} = \{\, (i, N-1) \leftrightarrow (i, N-1) \mid 1 \le i < M \,\}$. Propagation through `h` (of the same iteration in Program 1 and of the previous iteration in Program 2) yields a new equivalence node n with $R_n^{\text{want}} = \{\, (i, N-2) \leftrightarrow (i, N-2) \mid 0 \le i < M \,\}$, assuming $N \ge 3$. The affine hull of $R_a^{\text{want}} \cup R_n^{\text{want}}$ is $\{\, (i,j) \leftrightarrow (i,j) \,\}$ and intersection with the iteration domains yields $\{\, (i,j) \leftrightarrow (i,j) \mid 1 \le i < M \wedge 0 \le j \le N-1 \,\}$. Replacing R_a^{want} by this intersection, we again arrive at a node n, now with $R_n^{\text{want}} = \{\, (i,j) \leftrightarrow (i,j) \mid 1 \le i < M \wedge 0 \le j \le N-2 \,\}$ and so $R_n^{\text{want}} \subseteq R_a^{\text{want}}$. One of the other children of a follows the edge (d, c) in Figure 2 and establishes the base case. For more information on this widening operation and the corresponding induction, we refer to [18], which also explains the "narrowing" operation that is applied when induction fails.

4 Extensions

4.1 Motivation

The technique presented in Section 3 forms a solid basis for equivalence checking. This subsection motivates our extensions, the major contribution of this paper, based on real-life applications.

While performing program optimizations such as loop transformations, we often consider blocks such as FFT or IDCT as monolithic black boxes that have already been well optimized for the target platform. In those situations, we see from the source code only a function call to this block and we know only the signature of the function. Usually, entire (rows of) arrays are read and/or written in those function calls. Such a situation occurs, e.g., when calling IMDCT in the MP3 audio decoder [12]. This situation is depicted in Figure 3, where the row `buffer[j]` is read and the entire array `rawout` is written. This behavior can also be observed in other real-life applications.

```
dist = f(...,prev_sub2_frame[...
          +176*v4x[11*j+l]+2*v4y[11*j+l]]);
```

Figure 4: Data dependent addressing with known bounds (elements of `v4x` and `v4y` in interval $[-4, 4]$) in the motion estimation part of QSDPCM.

Support for accesses to array slices is therefore an important extension of our equivalence checking technique.

Another important extension is dealing with data dependent array indexing, which often occurs in real-life applications. For example, Figure 4 shows a code fragment from the QSDPCM video encoder [17] where this situation occurs. Typically, the designer knows the bounds for the data dependent part of the indexing. Combining this knowledge with our extension for data dependent addressing allows us to formally verify also this type of code.

In the sequel we will describe the technical details of these two important extensions of our base formal verification technique. Later, in Section 6 we will use the extensions to verify transformations performed on both the MP3 audio decoder and the QSDPCM video encoder. In the case of MP3, the correctness of the applied loop transformation itself was checked by the in-house transformation tool used to perform the transformation, but it is still interesting to verify equivalence of original and transformed program independently. For QSDPCM, the program pair we consider in the experiments was obtained using manual transformations.

4.2 Expansion Edges

In principle, the extensions we describe in this paper do not require any changes to the equivalence checking procedure. Instead, they can be handled by creating the appropriate input for this procedure, i.e., by adapting the iteration domains and access relations used in dependence analysis and by introducing extra computations and edges in the dependence graphs. We will, however, also introduce a new kind of edges in the dependence graphs, called *expansions*, and these edges need to be treated differently during the equivalence checking procedure. These edges will be used to encode reads from array slices as well as data-dependent accesses. They will also be useful for handling reductions, i.e., associative operations applied to the elements of an array slice, in future work. We describe the common characteristics of expansions and the required modifications to the equivalence checking procedure here.

In regular dependence relations, any element in the domain of the relation is mapped to a single element in the range. Expansion edges, by contrast, are used to express the dependence of an iteration on *multiple* iterations of another computation. A single element in the domain of the dependence relation on an expansion edge may therefore be mapped to multiple elements in the range. Two iterations of computations are then only considered equivalent if all the iterations these iterations map to in their respective dependence relations are *pairwise* equivalent. For example, in Section 4.4, we will use expansion edges to express the fact that, basically, a row of an array depends on all its elements. Two rows are then only considered equivalent if all the elements in the rows are pairwise equivalent. In particular, an expansion edge has a dependence relation of the form $M = \{ (\mathbf{i}) \to (\mathbf{i}, \mathbf{b}) \mid \mathbf{b} \in B \}$, where $B \subseteq \mathbb{Z}^d$ is called the *expansion domain*. In the example of reading an array row, B would contain the indices of the elements in the row.

Since an expansion may map a single element to multiple elements, we need to make some adjustments to the equivalence checking procedure. First of all, when performing a propagation, if one of the dependence graph edges used is an expansion, then the other is required to be an expansion too. Furthermore the expansion domains B_1 and B_2 need to have the same dimension d. If these conditions do not hold, then we cannot prove equivalence and we simply set $R^{\text{lost}} = R^{\text{want}}$. We also cannot simply apply (1) to propagate the R^{want} relation. Since both M_{e_1} and M_{e_2} map a single iteration to many elements, a direct application would result in an R^{want} relation that

```
1 compute_row(h(m), &slice);
2 for (k = 0; k < N; ++k)
3   A[i][k] = at(slice, k);
```

Figure 5: Encoding of the call `compute_row(h(m), A[i])` writing to an `A` row of size `N`.

expresses that each of these elements in one dependence graph needs to be equal to each of these elements in the other dependence graph, while we should only require that corresponding elements are equal. In the case of reading a row from an array, this would mean that all elements in the row in one program should be equal to all elements in the row in the other program, while we only need the elements at the same positions to be equal.

The solution is to add the necessary equalities to the resulting R^{want} relation. In particular, (1) is replaced by

$$R^{\text{want}}_{c_{e_1,e_2}} = \left((M_{e_1} \oplus^{\leftrightarrows}_{\rightarrow} M_{e_2})\, R^{\text{want}}_n\right) \cap \left((D_{v_1} \leftrightarrow D_{v_2}) \oplus^{\times}_{\leftrightarrow} 1_{\mathbb{Z}^d \leftrightarrow \mathbb{Z}^d}\right),$$

with $e_1 = (v_1, u_1)$ and $e_2 = (v_2, u_2)$. That is, we intersect the transformed R^{want} with a relation that forces the final d dimensions of each argument to be equal to each other. We have to be careful, though, that these equalities only match pairs of elements without removing any of them. In particular, adding these equalities should not remove any elements from the projection of $R' = (M_{e_1} \oplus^{\leftrightarrows}_{\rightarrow} M_{e_2})\, R^{\text{want}}_n \subseteq D_{u_1} \times D_{u_2}$ onto either D_{u_1} or D_{u_2}. If the expansion domains are independent of the iteration domains, as is the case here, then we only need to check for identical expansion domains. In the general case, e.g., for handling reductions, we effectively check that projecting $R^{\text{want}}_{c_{e_1,e_2}}$ onto D_{u_1} and D_{u_2} yields the same results as projecting R' onto the same domains. If these tests fail, then we again set $R^{\text{lost}} = R^{\text{want}}$.

Applying (2) on expansion edges during backpropagation does not require any adjustments. Since we propagate the correspondences that we have *not* been able to prove, a pair of iterations will be considered not proven equivalent as soon as any single pair of elements that the original pair was mapped to cannot be proven to be equivalent. In the row reading example, this means that array rows are only considered equivalent if all of their elements have been proven equivalent.

4.3 Writing Array Slices

In the basic methodology of Section 3, we have tacitly assumed that both programs only access individual array elements. However, as we have seen above, realistic programs often call functions that read or write entire array rows or even larger array slices. Writes to such slices are fairly easy to handle. We simply treat the extended write as a write to a temporary `slice` scalar representing the array slice, followed by a loop that fills in the individual array elements of the slice based on the `slice` scalar and the index into the slice.

Consider, for example, the write to a row of the two-dimensional array `A` in Line 5 of Program 1. This statement is treated as if it had been replaced by the code fragment in Figure 5. Any subsequent read from `A` that is determined by the dataflow analysis to read a value written in the given statement then results in the fragment of the dependence graph shown in the top part of Figure 6 (box α). The upward branch in this figure expresses the dependence on the rows. During the equivalence checking this branch will enforce that rows with the same values are read in both programs. The leftward branch expresses the dependence on the indices of the row elements. During equivalence checking this branch will then enforce that elements with the same index are being read. Note that the *index of the row* need not be the same in both programs as a data transformation may have reordered the rows in `A`. However, such a data transformation cannot have also transformed the columns as that would require a transformation of the function `compute_row`. The remainder of Figure 6 will be explained in the following sections.

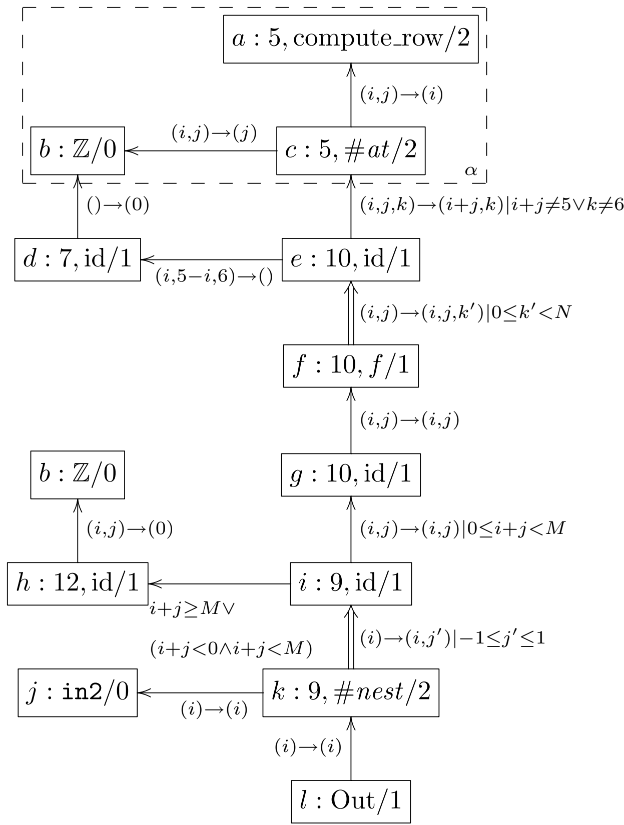

Figure 6: Final part of dependence graph for Program 1 in Figure 1(a). Two-line arrows represent expansions. To avoid clutter, the b node is shown twice.

```
1 slice = init;
2 for (k = 0; k < N; ++k)
3   slice = append(slice, B[i][k]);
4 A[i] = f(slice);
```

Figure 7: Encoding of a call `A[i] = f(B[i])` reading a `B` row of size `N`.

In general, the iteration domain D_t of the extra "#at" computation t is the Cartesian product of the iteration domain D_c of the computation c computing the slices and the set of indices of elements D_a in the slice. If n is the dimension of D_a, then there are $1 + n$ edges leaving t, one to c with a dependence relation that projects $D_t = D_c \times D_a$ onto D_c and n edges to $\mathbb{Z}$, each with a dependence relation that projects D_t onto the corresponding dimension. It is important to note that the input programs are not changed in any way. Figure 5 only serves as a motivation for modeling a dependence on a write to an array slice as the fragment shown in the top part of Figure 6. In principle, the encoding of Figure 5 could also be used during the dataflow analysis, but it is simpler to treat the write to the array slice as writing to each individual element and then apply the standard dataflow analysis. That is, the range of the access relation is extended with D_a before it is used in the dependence analysis. For example, the write to the row of `A` in Line 5 is represented by the access relation $\{ (i) \to (i, k) \mid 0 \le k < N \}$. The $\oplus^{\times}_{\leftrightarrow}$ and $\oplus^{\times}_{\to}$ operators used below are defined similarly.

4.4 Reading Array Slices

Just like writes of array slices, reads of array slices could also be handled using the primitives from our basic methodology. We would again (implicitly) introduce a `slice` scalar, but in this case, we would have to construct the value of this imaginary scalar from the individual elements. That is, a statement `A[i] = f(B[i])` inside some `i`-loop, where `B` is a two-dimensional array, would be encoded as shown in Figure 7. Note that this encoding introduces a recurrence on the `slice` scalar. Our procedure would have no problem handling this recurrence, but the relation between the computation of `f` and the computation that writes `B`, recovered in the resulting widening step, is already explicitly available in the input program. We therefore prefer a more direct representation using the expansions of Section 4.2.

In particular, if a program contains a read access to an array slice of dimension d from a computation c with iteration domain D_c, then we introduce a new id computation t in the dependence graph with iteration domain $D_t = D_c \times B$ and add an expansion edge from c to t with dependence relation $M_{(c,t)} = \{ (\mathbf{i}) \to (\mathbf{i}, \mathbf{b}) \mid \mathbf{b} \in B \}$, where the expansion domain $B \subseteq \mathbb{Z}^d$ contains all indices of the array slice. Figure 6 contains such an expansion edge between nodes f and e. The edges leaving the extra computation t (e in the example) are determined by standard dataflow analysis on the complete array. To be able to perform this analysis the original access relation $A_c \subseteq D_c \to D_a$, with D_a the domain of the array slices, is first lifted to $A_t = A_c \oplus^{\times}_{\to} 1_{B \to B}$, with $1_{B \to B}$ the identity mapping on B. For example, our handling of data dependent accesses discussed below will first express the data dependent access `A[i + in2[i]]` in Line 10 of Program 1 as $\{ (i, j) \to (i + j) \}$, with j representing the value of `in2[i]`. To handle the fact that a whole row of `A` is being read, this access relation is further extended to $\{ (i, j, k) \to (i + j, k) \}$. Dependence analysis then determines that the source of this read is the assignment in Line 7 of Program 1 in case $i + j = 5$ and $k = 6$, and the row write in Line 5 otherwise.

4.5 Data Dependent Read Accesses

A data dependent access is an access of which the index expression contains nested integer array accesses or calls to integer valued functions. We first discuss read accesses, delaying data dependent write accesses to Section 4.6. Throughout, we will also assume that the data dependent constructs

```
type pick(type old, type cur, int i1, int i2)
{
  return i1 == i2 ? cur : old;
}
```

```
value = init;
for (j = -1; j <= 1; ++j)
  value = pick(value, A[i+j], j, in2[i]);
out[i] = f(value);
```

Figure 8: Encoding of the call `out[i] = f(A[i+in2[i]])` with values of `in2` between -1 to 1.

in the index expressions appear in the same order in both programs. This assumption can be removed by combining the technique of this section with the technique for handling commutative operations from [18]. Moreover, we will assume that bounds are known on each of the nested constructs (array accesses or function calls). In our implementation, we expect the user to provide these bounds explicitly through a `#pragma`, but static analysis techniques have been developed for deriving such bounds automatically.

Our treatment of data dependent read accesses is based on the simple idea that two such accesses are certainly equivalent if the following two conditions are satisfied:

- all pairs of nested constructs are equivalent
- the outer access is equivalent for all possible values of the nested constructs.

Consider, for example, the statement in Line 10 of Program 1, with a nested access to the `in2` array with values between -1 and 1. This statement could be encoded using the code fragment at the bottom of Figure 8, where the function `pick` shown at the top is responsible for picking the value where the actual index is equal to a particular value among all possible values. Like all the other functions called within the two programs, this `pick` function could simply be treated by the equivalence checking as a black box. However, as in the case of reading array slices, we prefer a more direct representation in the dependence graph using expansions.

The solution is again to add extra dimensions to the iteration domains. In this case, the extra dimensions correspond to the values of the nested constructs. However, unlike the case of reading array slices, we not only need to make sure that both programs compute the same values for each value of these nested constructs, we also need to make sure that in both programs the nested constructs themselves have the same value. To impose this restriction, we introduce a computation s with an $(n+1)$-ary operator "*#nest*", that represents the data-dependent read. The computation that contains the data-dependent read is connected to s through an edge with identity dependence relation. The first n arguments of s correspond to the nested constructs and are treated as usual. If any of these constructs is itself an array access with nested constructs, then the technique is applied recursively. These first n arguments ensure that the nested constructs have the same value in both programs. The final argument of s is connected through an expansion edge to an additional id computation t. The domain of t is $D_t = D_s \times B$, with D_s the iteration domain of s and $B \subseteq \mathbb{Z}^n$ the known bounds on the values of the nested constructs. The dependence relation on the expansion edge is $M_{(s,t)} = \{ (\mathbf{i}) \to (\mathbf{i}, \mathbf{b}) \mid \mathbf{b} \in B \}$. Finally, t is connected to one or more computations writing to the array based on standard dependence analysis. During this dependence analysis, the iteration domain is taken to be that of t, i.e., D_t, while the access relation is given by replacing each of the nested constructs by the corresponding dimension of B.

For example, consider the data dependent access in Line 10 of Program 1. As we will see in Section 4.7, our treatment of the data dependent conditions in Line 9 also introduces extra dimensions for the data dependent constructs. Since these constructs are the same as those used in Line 10, there is no need to add these extra dimensions again. However, if the access would

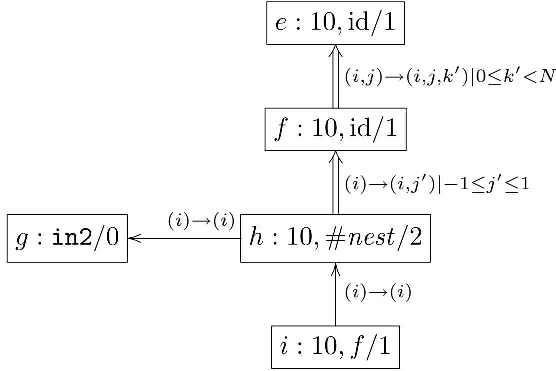

Figure 9: Dependence graph fragment for isolated data dependent read in Line 10 of Program 1 in Figure 1(a).

```
1 for (k = l; k <= u; ++k)
2   A[i + k] = k == f(i) ? g(B[i]) : A[i + k];
```

Figure 10: Encoding of a call `A[i + f(i)] = g(B[i])`, with values of `f(i)` in the interval $[l, u]$.

have appeared in isolation, i.e., in absence of the data dependent conditions, then it would have resulted in the dependence graph fragment shown in Figure 9. Node h in this graph corresponds to node s above, while node f corresponds to node t. Node e is the same as that of Figure 6. The iteration domain of the statement in Line 10 is $D_{10} = \{ (i) \mid 0 \le i < M \}$. Since the bounds on the values of `in2` are given by $B = \{ (j) \mid -1 \le j \le 1 \}$, the iteration domain of f is $D_{10} \times B = \{ (i, j) \mid 0 \le i < M \wedge -1 \le j \le 1 \}$, while the access relation used during dependence analysis is $\{ (i, j) \to (i + j) \}$, where the value of `in2[i]` is represented by j.

4.6 Data Dependent Write Accesses

Assume the input program contains a statement of the form `A[i + f(i)] = g(B[i])`, where as before we assume that the values of `f(i)` are bounded to lie in some interval, say $[l, u]$. Since at analysis time, we do not know what the value of `f(i)` will be at run time, we can only assume that all elements $i + k$ with $k \in [l, u]$ may be written by the statement. The statement is therefore treated as writing a whole segment of the `A` array, as shown in Figure 10. The value assigned to each element in this segment depends on whether k is equal to `f(i)`. If it is, then the value is that prescribed by the statement. Otherwise, it is the original value of the array element.

This encoding hides the data-dependent access and allows us to apply standard exact dependence analysis. During equivalence checking, the right-hand side is treated as a function with three arguments, `f(i)`, `B[i]` and `A[i + k]`, each of which needs to be equivalent to its counterpart for the data-dependent writes to be equivalent. It should be noted that fuzzy dependence analysis would not require this hiding and may in fact exploit the extra data-dependent access information to derive more accurate dependence relations.

4.7 Data Dependent Assignments

In our treatment of data dependent reads, we have assumed that the range of values of nested constructs has known bounds. However, in typical multimedia applications, and also in our illustrative example, these bounds are not sufficient to preclude out-of-bounds accesses. Instead, statements containing such data dependent reads are often guarded by conditionals that ensure the accesses stay within the bounds of the array. If the conditions are not satisfied, then usually

some default value is used instead. As an example, consider the data dependent read in Line 10 of Program 1. The conditions in Line 9 ensure that the value of the index expression remains in the interval $[0, M)$. In the else branch (Line 12), some default value is written to the same array element that is written in the then branch. In this section, we describe how we handle such conditional assignments, where the condition is data dependent.

The first step is to move the condition inside the statement, i.e., to replace the compound statement `if (c) a = e; else a = d;` by `a = c ? e : d;`. Similarly, `if (c) a = e;` is replaced by `a = c ? e : a;`. In its latter form, the statement can be analyzed using standard exact dependence analysis. However, a straightforward application will ignore the conditions in `c` during the analysis of the accesses in `e` and `d`. This means that if the conditions were used to guard against out-of-bounds accesses, the dependence analysis step will not be aware of these guards and will therefore detect out-of-bounds accesses. By default, such out-of-bounds accesses are treated as reads from input arrays. During equivalence checking these spurious accesses will also be required to be equivalent, which means that the index expressions need to have the same value in both programs. In other words, by ignoring the data dependent conditions, we exclude any data transformations on the array with data dependent accesses. Consider, for example, the case where the value of `in2[M-1]` is equal to 1 in our running example. This means that the else branch is taken in Line 9 of Program 1 and Line 18 of Program 2, and that `out[M-1]` is assigned the value 0 in both programs. However, if the data dependent conditions are ignored, then the equivalence checker would also check that the value of `A[M]` and that of `B[-1]` are equal. These values are never written and because the indices are distinct, the equivalence checker can only assume that the values stored in these elements may be different.

The solution is to pass along the information of the data dependent conditions to the dependence analysis. In particular, the iteration domains are extended with dimensions that correspond to the values of the data dependent constructs. In this way, the data dependent conditions can be expressed in terms of the dimensions of the iteration domain and can therefore be added to the iteration domain of "`e`". Similarly, the iteration domain of "`d`" is intersected with the complement of the data dependent conditions. For example, the iteration domain of the statement in Line 10 of Program 1 is taken to be

$$\{ (i, j) \mid 0 \leq i < M \wedge -1 \leq j \leq 1 \wedge 0 \leq i + j < M \}$$

where j represents the value of `in2[i]`. Using this modified iteration domain and the access relation from Section 4.4, i.e., $\{ (i, j, k) \to (i + j, k) \}$, dependence analysis no longer detects any spurious out-of-bounds accesses.

In the dependence graph, the extra dimensions are again introduced through a *#nest* computation with an outgoing expansion edge. The iteration domain of the id computation at the other end of this expansion edge is then split according to the data dependent conditions. In our example, the data dependent assignment guarded by the conditions in Line 9 of Program 1 is represented by node k in the dependence graph in Figure 6. The extra dimension is introduced in the expansion edge to i, after which the domain is split in two parts, with the iteration domain of node g representing the cases where the data dependent conditions hold and that of node h representing the cases where the conditions do not hold. No extra computations have to be introduced to handle the data dependent read in f, because the extra dimensions have already been added at this point.

4.8 The Illustrative Example Revisited

Consider once more the pair of programs in Figure 1. The first part of the dependence graph of Program 1 is shown in Figure 2, while the second part is shown in Figure 6. As mentioned before, there are two expansion edges in this graph, one from node k to node i, corresponding to the data dependent assignment, and one from node f to node e, corresponding to the read access to a row of array `A`. The dependence graph of Program 2 is shown in Figure 11 and is very similar, except that there is no node that corresponds to d as Program 2 does not have any assignment that

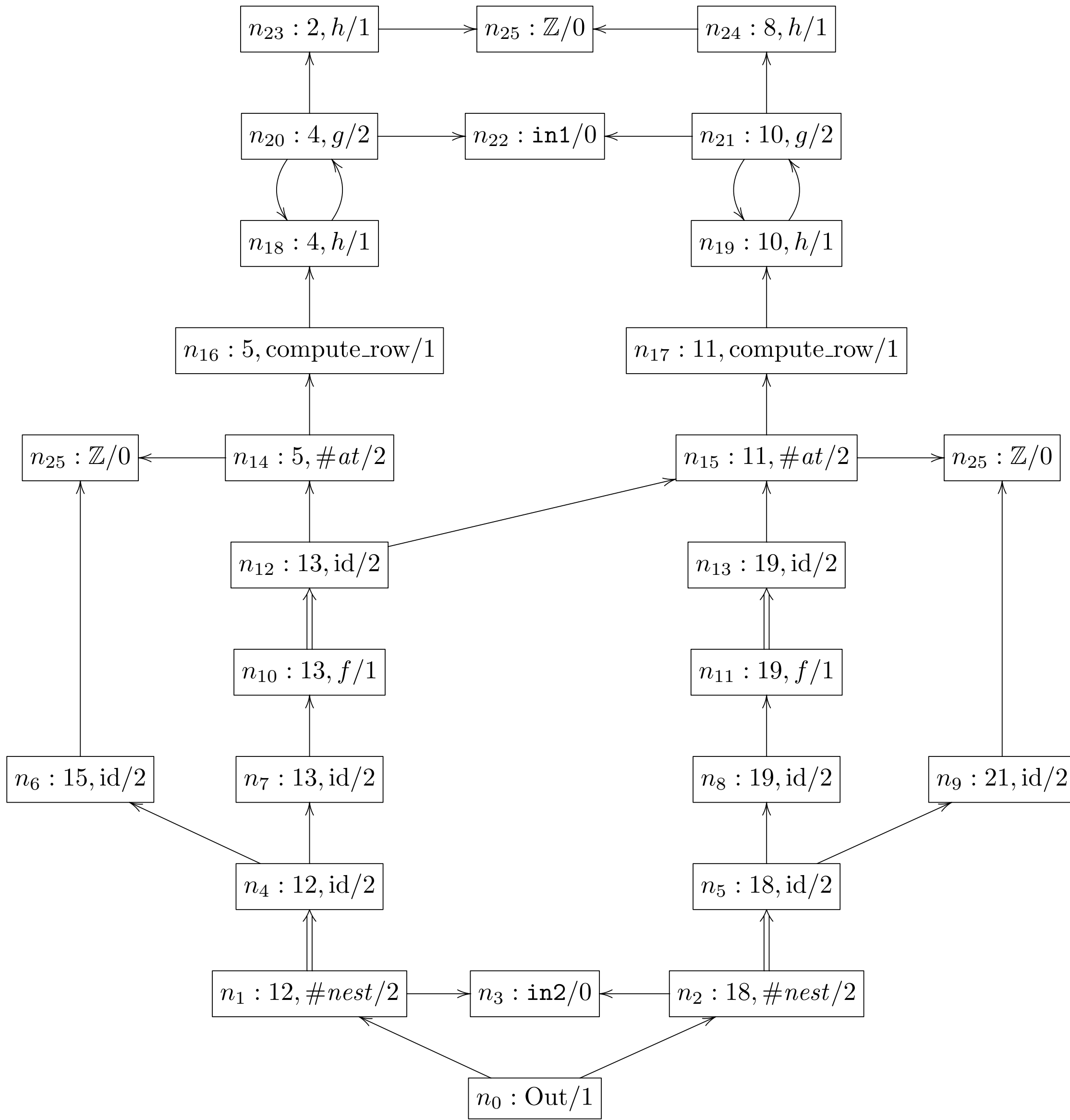

Figure 11: Dependence graph for Program 2 in Figure 1(b). Node n_{25} is shown three times.

corresponds to the assignment in Line 7 of Program 1. Furthermore, some iteration domains are shifted with respect to those of Program 1 and some nodes are duplicated (with different iteration domains).

Let us consider what happens during the forward propagation. Since the output array is one-dimensional, the initial correspondence to be proven between the output computations of both programs is $\{ (i) \leftrightarrow (i') \mid 0 \le i = i' < M \}$. In the dependence graph of Program 1 there is only one edge leaving the output computation l, but in that of Program 2 there are two, because `out` is written in both the data dependent assignment starting in Line 12 and the data dependent assignment starting in Line 18. Let us follow the first edge. The desired correspondence between node k and node n_1 is now $\{ (i) \leftrightarrow (i') \mid 1 \le i+1 = i' < M \}$. Following the first expansion edge in both graphs yields $\{ (i,j) \leftrightarrow (i',j') \mid 1 \le i+1 = i' < M \wedge -1 \le j = j' \le 1 \}$. Continuing upward in Figure 6, we add the conditions $0 \le i + j < M$ (the equivalent conditions $M \ge i' + j' > 0$ are added in the other graph) and perform another expansion. The resulting R^{want} relation is $\{ (i,j,k) \leftrightarrow (i',j',k') \mid 1 \le i+1 = i' < M \wedge -1 \le j = j' \le 1 \wedge 0 \le i+j < M \wedge 0 \le k = k' < N \}$.

At this point, we have reached node e in Figure 6. Propagation through c continues until the input computations and returns with an empty R^{lost}. However, since there is no node in the second graph that corresponds to d, the R^{lost} relation between e and n_{15} will be $\{ (i,j,k) \leftrightarrow (i',j',k') \mid 1 \le i+1 = i' < M \wedge -1 \le j = j' \le 1 \wedge i+j = 5 \wedge k = k' = 6 < N \}$. This single problematic array element taints the whole row and applying (2) we obtain $R^{\text{lost}} = \{ (i,j) \leftrightarrow (i',j') \mid 1 \le i+1 = i' < M \wedge -1 \le j = j' \le 1 \wedge i+j = 5 \}$ at the level of f. Performing further backpropagation steps to the level of k yields $R^{\text{lost}} = \{ (i) \leftrightarrow (i') \mid 5 \le i' = i+1 \le 7 \}$. After the final backpropagation step, we obtain the results mentioned in the illustrative example of Section 1.

5 Locating Errors

In the illustrative example, we have seen that our approach is able to detect that elements 4 to 6 of the output array may not have the same values in the two programs. Usually, we expect all elements to be equal and we may want to find the source of this difference. These sources are easy to find in our approach. Whenever we set the R^{lost} relation of some node in the equivalence tree to a non-empty set, we have found a potential source of an error. In principle, this R^{lost} relation could simply be reported to the user, but the description of this relation may be fairly complicated. Instead, we report a single element of the relation, i.e., a specific pair of iteration vectors for a specific value of the parameters. Selecting such a single element from a relation is a built-in operation of the `isl` library we use in our prototype tool (see Section 6).

During backtracking, we can build up a trace from the source of the error to the pair of output computations, in each step describing the pair of computations involved together with a pair of iteration vectors. This pair of iteration vectors is updated in the same way as the R^{lost} relation, i.e., by applying (2). If one of dependence relations in (2) is not injective, then the result of this application may contain more than a single pair of iteration vectors. We therefore select a single element from the result after each such application.

For example, the first pair of computations where an error is detected in the illustrative example, is computation b in Figure 6 and computation n_{15} in Figure 11. More specifically, the error occurs at iterations (0) and $(5,6)$ respectively. That is, the equivalence checker expects that the value stored in `B[M - 1 - 5][6]` should be equal to 0, but it has no way of proving this equivalence. During backtracking, the equivalence checker moves back to the row read in Line 13 of Program 2 represented by node n_{12}. The dependence relation on the traversed edge is $\{ (i,j,k) \to (i+j-1,k) \}$. (This relation differs from the relation between e and c because the iterations of Line 13 have been shifted by one in Program 2.) Our sample point $\{ (0) \leftrightarrow (5,6) \}$ is mapped to the relation $\{ (0) \leftrightarrow (i,j,6) \mid i+j-1 = 5 \wedge -1 \le j \le 1 \}$, from which we pick an arbitrary point, say, $\{ (0) \leftrightarrow (5,1,6) \}$. Continued backtracking now moves in the first dependence graph, first to node d, with pair of iterations $\{ () \leftrightarrow (5,1,6) \}$ and then to node e, with pair of iterations $\{ (4,1,6) \leftrightarrow (5,1,6) \}$. Note that in this case we do not need to pick a point because we have already selected a value for j in the second graph and the R^{want} relation forces the j

values in both graphs to be the same. In the following step, the expansion edge is traversed backward in both programs, leading to the pair of iterations $\{ (4,1) \leftrightarrow (5,1) \}$. After a couple of uneventful backpropagation steps, we arrive at the second expansion edge and move back to the pair of *#nest* computations with pair of iterations $\{ (4) \leftrightarrow (5) \}$. The final backpropagation to the output computations then gives a pair of iterations $\{ (4) \leftrightarrow (4) \}$. In other words, the fact that `B[M - 1 - 5][6]` is not assigned the value zero explains why the elements at position 4 of the output arrays are not guaranteed to be equal.

In each step of building up the trace above, we propagate back a pair of iterations to the parent node. There are two cases where we cannot perform this backpropagation, when passing through a commutative operator and when passing through a widening step. In the case of commutative operators (not discussed in detail in this paper), our basic approach essentially considers all possible permutations of the arguments. We therefore initially expect many more pairs of computations to be equivalent than strictly necessary and even if it turns out no permutation works for a given pair of iterations of the commutative operations, there is no way of knowing which permutation the user expected to result in an equivalence. We therefore turn off error tracing as soon as we propagate beyond a commutative operator and take the pair of commutative operators to be source of the error, if any. By adding extra output arrays, the user can direct the equivalence checker to the permutation of arguments that is expected to be valid for the given pair of iterations.

When propagating back over a widening step, we can continue building the error trace, but we cannot propagate back the pair of iterations in a systematic way. We therefore select an arbitrary pair of iterations from the R^{lost} relation on the pair of computations that caused the widening, independent of the pair of iterations in the widened relation. This is the (small) price we have to pay for using a widening based approach instead of an approach based on transitive closures.

6 Experiments

Both the basic methodology and the extensions proposed in this paper have been implemented as part of our C++ `isa` (`http://www.kotnet.org/~skimo/loop/isa-0.10.tar.bz2`) prototype tool set. The only exception is the data dependent write accesses of Section 4.6, which do not occur in any of our experiments. The `isa` tool set contains a polyhedral extractor from C based on SUIF [1] and an exact dependence analysis tool. We use our own C `isl` library to manipulate sets of integers defined by linear inequalities and integer divisions. All experiments in this section were performed on an Intel Xeon W3520 @ 2.66GHz.

The experimental results are shown in Table 1. The first five rows reproduce an experiment we performed in [19] using our basic approach. This experiment consists of 105 pairs of equivalent programs that were generated in different ways. The first four rows are individual pairs, while the fifth row is a summary of all 105 pairs. Program 1 and 2 are those from Figure 1.

The MP3 applications are based on [12]. The original version was first preprocessed by applying the scenario methodology [8] to avoid data dependent conditions. The data dependent conditions trigger different program paths based on the input audio frame type. The most promising program paths were grouped together and the remaining data dependent conditions were moved down and encapsulated as explained in [13]. The result of this preprocessing is MP3 1. MP3 2 is the result of applying loop transformations on MP3 1 and MP3 3 is a manually corrupted version of MP3 2.

The QSDPCM application [17] is a video encoder for slow motion pictures (e.g., teleconferencing). The original version is QSDPCM 1, while QSDPCM 2 is the result of manual transformations, including loop fusion, loop shifting and array contraction, with the aim of improving data locality. The difference is explained in more detail in [14], where this pair of programs was used in a feasibility study of the trade-off between inter-array and intra-array in-place memory compaction. QSDPCM 2 was also used as a test driver for a data memory optimization tool [4]. QSDPCM 3 is a manually corrupted version of QSDPCM 2. Each of these versions contains data dependent assignments (Section 4.7) and could therefore not be handled in our previous work [20]. In the experiments of [20] we instead used a version that we call p-QSDPCM 1 here, which is the result of aggressive semi-automatic preprocessing, hiding all data dependent conditions, including

Program 1	program 2	equiv.	cases	stats	comps	d	row access	dd index	ec time	∇	Δ
CLooG-isl 1	CLooG-PolyLib 1	✓	1	133	135	3			0.158s	51	0
CLooG-isl 2	CLooG-PolyLib 2	✓	1	1090	1092	3			3.790s	116	0
CLooG-isl 3	CLooG-PolyLib 3	✓	1	27	29	5			0.248s	29	0
CLooG-isl 4	CLooG-PolyLib 4	✓	1	48	52	2			6.424s	201	0
CLooG-isl	CLooG-PolyLib	✓	105	4290	4570	5			20.235s	1880	0
Program 1	Program 1	✓	1	10	32	3	✓	✓	0.020s	1	0
Program 2	Program 2	✓	1	16	50	3	✓	✓	0.019s	1	0
Program 1	Program 2		1	13	41	3	✓	✓	0.036s	2	0
MP3 1	MP3 1	✓	1	34	58	4	✓		0.032s	2	0
MP3 2	MP3 2	✓	1	34	58	4	✓		0.032s	2	0
MP3 1	MP3 2	✓	1	34	58	4	✓		0.032s	2	0
MP3 1	MP3 3		1	34	59	4	✓		0.050s	2	14
p-QSDPCM 1	p-QSDPCM 2	✓	1	202	252	8		✓	20.090s	270	0
QSDPCM 1	QSDPCM 1	✓	1	228	544	10	✓	✓	17.113s	380	0
QSDPCM 2	QSDPCM 2	✓	1	244	560	12	✓	✓	22.868s	402	0
QSDPCM 1	QSDPCM 2	✓	1	236	552	12	✓	✓	18.990s	392	0
QSDPCM 2	QSDPCM 3		1	244	561	12	✓	✓	102.407s	432	3389
QSDPCM 2	QSDPCM 3		1	244	561	12	✓	✓	24.347s	422	N/A
QSDPCM 1	QSDPCM 3		1	236	553	12	✓	✓	27.227s	392	N/A

Table 1: Experimental results of equivalence checking. Meaning of the columns: program 1 and 2: input programs; equiv.: input programs are equivalent; cases: number of pairs of programs; stats: number of assignment statements; comps: number of computations; d: maximal dimension of iterations domains; ec time: equivalence checking time; ∇: number of widenings; Δ: number of narrowings or N/A if narrowing is turned off.

those used to guard against out-of-bounds data dependent array accesses. This version has been used as the basis of an automatic transformation. However, p-QSDPCM 2 is not the result of this automatic transformation. Instead it is the result of a manual application of transformations that are similar to those that have been applied to QSDPCM 1, but only to that part of the code that does not have any influence on any of the data dependent assignments. Because crucial information about these data dependent assignments was hidden in the p-QSDPCM program, we would not have been able to prove equivalence otherwise. In particular, we are unable to compare p-QSDPCM 1 to the result of the automatic transformation.

While the p-QSDPCM program had seen too much preprocessing, the QSDPCM programs had not seen enough. We have therefore manually applied some preprocessing to the QSDPCM program pair. In particular, we applied if-conversion on all data dependent conditions not involved in data dependent assignments, we inlined a function in QSDPCM 1 that had been inlined in QSDPCM 2 and we added some extra initializations. These extra initialization were needed to prevent the dependence analysis tool from detecting loop-carried dependences that do not occur in practice, but that require analysis of data dependent conditions to disprove. Finally, we added annotations to express the bounds on all data dependent constructs.

The columns of Table 1 show whether the program pair(s) are equivalent, the number of programs pairs, the total number of statements, the total number of computations, the maximal loop nest depth, the presence of row accesses or data dependent accesses, the total computation time of the equivalence checking, the number of widenings performed and the number of narrowings. The results indicate that the extensions of this paper as such do not make the equivalence checking more expensive. The QSDPCM application, however, is a particularly difficult program to analyze because of the many nested data dependent assignments. The table also shows that in this kind of applications, it is more difficult to check the equivalence of a pair of programs that is *not* equivalent (unless, say, different functions are called close to the output computations, in which case the equivalence checker can abort immediately). The main cause of this slowdown appears to be the explosion in narrowing steps. Although narrowing steps can sometimes be useful to salvage some information after an over-optimistic widening step, such over-optimistic widening steps rarely occur in practice. Turning off narrowing then has no effect on the ability of the equivalence checker to detect and locate errors. The final two rows of the table show results with narrowing turned off. Performing an equivalence check of QSDPCM 1 and QSDPCM 3 with narrowing turned on did not terminate in a reasonable amount of time because of an inefficiency in our handling of tabling, which is only noticeable in the presence of a massive amount of narrowings.

7 Conclusion

We have extended our widening based method for automatically proving the equivalence of static affine programs to a wider class of programs containing write and read accesses to array slices, data dependent read and write accesses and data dependent assignments. For an efficient treatment of some of these constructs, we have introduced special expansion edges in our dependence graph representation of the input programs and we have extended the equivalence checking procedure to perform propagation over such edges. The extended approach has been fully implemented, with the exception of data dependent write accesses, and is publicly available. Furthermore, we have explained and implemented a mechanism for locating the source of an inequivalence. The experiments show that the extensions themselves are not necessarily more difficult to handle. It is, however, important that sufficient information is available. In particular, data dependent assignments should not be hidden away in lower level functions. A large number of narrowings can still lead to excessive running times. Possible future work is to improve the efficiency of the procedure, (possibly using a simplified form of dataflow analysis), extend the class of programs to include more data dependent constructs through the use of fuzzy dataflow analysis and extend the equivalence checker to handle general reductions.

References

[1] Amarasinghe, S., Anderson, J., Lam, M.S., Tseng, C.W.: An overview of the SUIF compiler for scalable parallel machines. In: Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing. San Francisco, CA (1995)

[2] Barthou, D., Collard, J.F., Feautrier, P.: Fuzzy array dataflow analysis. J. Parallel Distrib. Comput. **40**(2), 210–226 (1997)

[3] Barthou, D., Feautrier, P., Redon, X.: On the equivalence of two systems of affine recurrence equations. In: Euro-Par Conf., *LNCS*, vol. 2400, pp. 309–313. Paderborn (2002)

[4] Brockmeyer, E., Miranda, M., Corporaal, H., Catthoor, F.: Layer assignment echniques for low energy in multi-layered memory organisations. In: 2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), pp. 11,070–11,075 (2003)

[5] Catthoor, F., Danckaert, K., Kulkarni, C., Brockmeyer, E., Kjeldsberg, P., Van Achteren, T., Omnés, T.: Data access and storage management for embedded programmable processors. Kluwer Academic Publishers, Boston, USA (2002)

[6] Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: PLILP'92, *LNCS*, vol. 631, pp. 269–295. Leuven, Belgium (1992)

[7] Feautrier, P.: Dataflow analysis of array and scalar references. Int. J. of Parallel Programming **20**(1), 23–53 (1991)

[8] Gheorghita, S.V., Palkovic, M., Hamers, J., Vandecappelle, A., Mamagkakis, S., Basten, T., Eeckhout, L., Corporaal, H., Catthoor, F., Vandeputte, F., De Bosschere, K.: System-scenario-based design of dynamic embedded systems. ACM Transactions on Design Automation of Electronic Systems **14**(1) (2009)

[9] Godlin, B., Strichman, O.: Regression verification. In: 46th Design Automation Conference (DAC'09), pp. 466–471 (2009)

[10] Kaufmann, M., Moore, J.S., Manolios, P.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Norwell, MA, USA (2000)

[11] Kelly, W., Pugh, W., Rosser, E., Shpeisman, T.: Transitive closure of infinite graphs and its applications. Int. J. Parallel Program. **24**(6), 579–598 (1996)

[12] Lagerström, K.: Design and implementation of an mp3 decoder (2001)

[13] Palkovic, M., Corporaal, H., Catthoor, F.: Dealing with data dependent conditions to enable general global source code transformations. International Journal of Embedded Systems **4**(1), 27–39 (2009)

[14] Palkovic, M., Corporaal, H., Catthoor, F.: Trade-offs in loop transformations. ACM Transactions on Design Automation of Electronic Systems **22** (2009)

[15] Shashidhar, K.C.: Efficient automatic verification of loop and data-flow transformations by functional equivalence checking. Ph.D. thesis, Katholieke Universiteit Leuven, Leuven, Belgium (2008)

[16] Shashidhar, K.C., Bruynooghe, M., Catthoor, F., Janssens, G.: Verification of source code transformations by program equivalence checking. In: CC 2005, Proceedings, *Lecture Notes in Computer Science*, vol. 3443, pp. 221–236. Springer-Verlag, Berlin (2005)

[17] Strobach, P.: Qsdpcm – a new technique in scene adaptive coding. In: Proc. 4th Eur. Signal Processing Conf. (EUSIPCO-88), pp. 1141–1144. Elsevier Publ., Grenoble, France (1988)

[18] Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. In: Computer Aided Verification 21, pp. 599–613. Springer (2009)

[19] Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. Report CW 565, Department of Computer Science, K.U.Leuven, Leuven, Belgium (2009)

[20] Verdoolaege, S., Palkovic, M., Bruynooghe, M., Janssens, G., Catthoor, F.: Experience with widening based equivalence checking in realistic multimedia systems. In: Proceedings of IEEE International High Level Design Validation and Test Workshop, pp. 122–129. IEEE Press, San Francisco, CA, USA (2009)

[21] Verma, M., Marwedel, P.: Advanced Memory Optimization Techniques for Low-Power Embedded Processors. Springer Publishing Company, Incorporated (2007)