PolyGLoT: A Polyhedral Loop Transformation Framework for a Graphical Dataflow Language

Somashekaracharya G. Bhaskaracharya[1,2] and Uday Bondhugula[1]

[1] Department of Computer Science and Automation, Indian Institute of Science
[2] National Instruments, Bangalore, India

Abstract. Polyhedral techniques for program transformation are now used in several proprietary and open source compilers. However, most of the research on polyhedral compilation has focused on imperative languages such as C, where the computation is specified in terms of statements with zero or more nested loops and other control structures around them. Graphical dataflow languages, where there is no notion of statements or a schedule specifying their relative execution order, have so far not been studied using a powerful transformation or optimization approach. The execution semantics and referential transparency of dataflow languages impose a different set of challenges. In this paper, we attempt to bridge this gap by presenting techniques that can be used to extract polyhedral representation from dataflow programs and to synthesize them from their equivalent polyhedral representation.

We then describe PolyGLoT, a framework for automatic transformation of dataflow programs which we built using our techniques and other popular research tools such as Clan and Pluto. For the purpose of experimental evaluation, we used our tools to compile LabVIEW, one of the most widely used dataflow programming languages. Results show that dataflow programs transformed using our framework are able to outperform those compiled otherwise by up to a factor of seventeen, with a mean speed-up of $2.30\times$ while running on an 8-core Intel system.

1 Introduction and Motivation

Many computationally intensive scientific and engineering applications that employ stencil computations, linear algebra operations, image processing kernels, etc. lend themselves to polyhedral compilation techniques [2, 3]. Such computations exhibit certain properties that can be exploited at compile time to perform parallelization and data locality optimization.

Typically, the first stage of a polyhedral optimization framework consists of polyhedral extraction. Specific regions of the program that can be represented using the polyhedral model, typically affine loop-nests, are analyzed. Such regions have been termed Static Control Parts (SCoPs) in the literature. Results of the analysis include an abstract mathematical representation of each statement in the SCoP, in terms of its iteration domain, schedule, and array accesses. Once dependences are analyzed, an automatic parallelization and locality optimization tool such as Pluto [16] is used to perform high-level optimizations. Finally, the transformed loop-nests are synthesized using a loop generation tool such as CLooG [4].

Regardless of whether an input program is written in an imperative language, a dataflow language, or using another paradigm, if a programmer does care about performance, it is important for the compiler not to ignore transformations that yield significant performance gains on modern architectures. These transformations include, for example, ones that enhance locality by optimizing for cache hierarchies and exploiting register reuse or those that lead to effective coarse-grained parallelization on multiple cores. It is thus highly desirable to have techniques and abstractions that could bring the benefit of such transformations to all programming paradigms.

There are many compilers, both proprietary and open-source which now use the polyhedral compiler framework [12, 15, 6, 16]. Research in this area, however, has predominantly focused on imperative languages such as C, C++, and Fortran. These tools rely on the fact that the code can be viewed as a sequence of statements executed one after the other. In contrast, a graphical dataflow program consists of an interconnected set of nodes that represent specific computations with data flowing along edges that connect the nodes, from one to another. There is no notion of a statement or a mutable storage allocation in such programs. Conceptually, the computation nodes can be viewed as consuming data flowing in to produce output data. Nodes become ready to be 'fired' as soon as data is available at all their inputs. The programs are thus inherently parallel. Furthermore, the transparency with respect to memory referencing allows such a program to write every output data value produced to a new memory location. Typically, however, copy avoidance strategies are employed to ensure that the output data is *inplace* to input data wherever possible. Such *inplaceness* decisions can in turn affect the execution schedule of the nodes.

The polyhedral extraction and code synthesis for dataflow programs, therefore, involves a different set of challenges to those for programs in an imperative language such as C. In this paper, we propose techniques that address these issues. Furthermore, to demonstrate their practical relevance, we describe an automatic loop transformation framework that we built for the LabVIEW graphical dataflow programming language, which uses all of these techniques. To summarize, our contributions are as follows:

- We provide a specification of parts of a dataflow program that lends itself to the abstract mathematical representation of the polyhedral model.
- We describe a general approach for extracting the polyhedral representation for such a dataflow program part and also for the inverse process of code synthesis.
- We present an experimental evaluation of our techniques for LabVIEW and comparison with the LabVIEW production compiler.

The rest of the paper is organized as follows. Section 2 provides the necessary background on LabVIEW, dataflow languages in general, the polyhedral model, and introduces notation used in the rest of the paper. Section 3 deals with extracting a polyhedral representation from a dataflow program, and Section 4 addresses the inverse process of code synthesis. Section 5 describes our PolyGLoT framework. Section 6 presents an experimental evaluation of our techniques. Related work and conclusions are presented in Section 7 and Section 8 respectively.
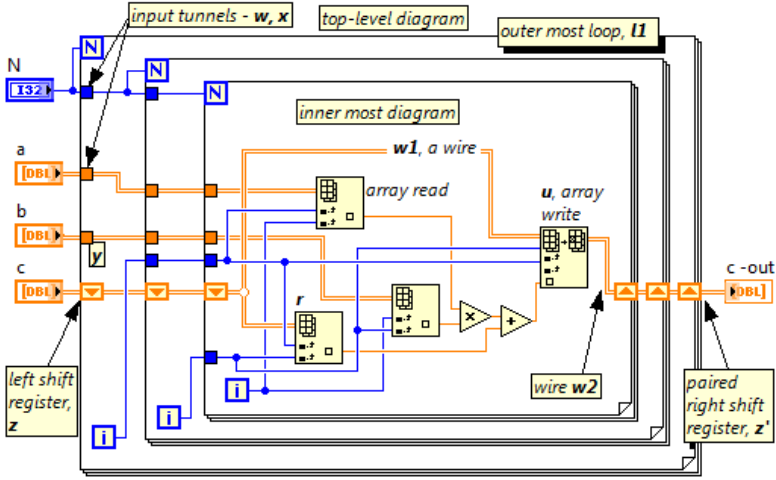
Fig. 1. *matmul* in LabVIEW. LabVIEW for-loops are unit-stride for-loops with zero-based indexing. A loop iterator node in the loop body (the [i] node) produces the index value in any iteration. A special node on the loop boundary (the N node) receives the upper loop bound value. The input arrays are provided by the nodes *a, b* and *c*. The output array is obtained at node *c-out*. The color of the wire indicates the type of data flowing along it e.g. blue for integers, orange for floats. Thicker lines are indicative of arrays.
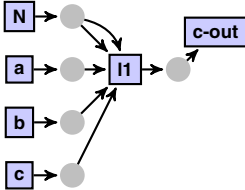
Fig. 2. DAG of the top-level diagram of *matmul*. In this abstract model, the gray nodes are wires. The 4 source nodes (N, a, b, c), the sink node (c-out) and the outermost loop are represented as the 6 blue nodes. Directed edges represent the connections from inputs/outputs of computation nodes to the wires, e.g. data from source node N flows over a wire into two inputs of the loop node. Hence the two directed edges from the corresponding wire node.

2   Background

2.1   LabVIEW – Language and Compiler

LabVIEW is a graphical, dataflow programming language from National Instruments Corporation (NI) that is used by scientists and engineers around the world. Typically, it is used for implementing control and measurement systems, and embedded applications. The language itself, due to its graphical nature, is referred to as the G language. A LabVIEW program called a Virtual Instrument (VI) consists of a front panel (the graphical user interface) and a block diagram, which is the graphical dataflow diagram. Instead of textual statements, the program consists of specific computation nodes. The flow of data is represented by a *wire* that links the specific output on a source node to the specific input on a sink node. The block diagram of a LabVIEW VI for matrix multiplication is shown in Figure 1.

Loop nodes act as special nodes that enclose the dataflow computation that is to be executed in a loop. Data that is only read inside the loop flows through a special node on the boundary of the loop structure called the *input tunnel*. A pair of boundary nodes called the *left* and *right shift registers* are used to represent loop-carried dependence. Data flowing into the right shift register in one iteration flows out of the left shift register in the subsequent iteration. The data produced as a result of the entire loop computation flows out of the right shift register. Additionally, some boundary nodes are also used for the loop control. In addition to being inherently parallel because of the dataflow programming paradigm, LabVIEW also has a parallel for loop construct that can be used to parallelize the iterative computation [7].

The LabVIEW compiler first translates the G program into a Data Flow Intermediate Representation (DFIR) [14]. It is a high-level, hierarchical and graph-based representation that closely corresponds to the G code. Likewise, we model the dataflow program as being conceptually organized in a hierarchy of diagrams. It is assumed that the diagrams are free of dead-code.

2.2   An Abstract Model of Dataflow Programs

Suppose N is the set of computation nodes and W is the set of wires in a particular diagram. Each diagram is associated with a directed acyclic graph (DAG), $G = (V, E)$, where $V = N \cup W$ and $E = E_N \cup E_W$. $E_N \subseteq N \times W$ and $E_W \subseteq W \times N$. Essentially, E_N is the set of edges that connect the output of the computation nodes to the wires that will carry the output data. Likewise, E_W is the set of edges that connect the input of computation nodes to the wires that propagate the input data. We follow the convention of using small letters v and w to denote computation nodes and wires respectively. Any edge *(v, w)* represents a particular output of node v and any edge *(w, v)* represents a particular input of node v. So, the edges correspond to memory locations. The wires serve as copy nodes, if necessary.

For every $n \in N$ that is a loop node, it is associated with a DAG, $G_n = (V_n, E_n)$ which corresponds to the dataflow graph describing the loop body. The loop inputs and outputs are represented as source and sink vertices. The former have no incoming edges, whereas the latter have no outgoing edges. Let I and O be the set of inputs and outputs. Furthermore, a loop output vertex may be paired with a loop input vertex to signify a loop-carried data dependence i.e., data produced at the loop output in one iteration flows out of the input for the next iteration (Fig 1).

Inplaceness.   In accordance with the referential transparency of a dataflow program, each edge could correspond to a new memory location. Typically, however, a copy-avoidance strategy may be used to re-use memory locations. For example, consider the array element write node u in Figure 1, and its input and output wires, w_1 and w_2. The output array data flowing along w_2 could be stored in the same memory location as the input array data flowing along w_1. The output data can be *inplace* to the input data. The *can-inplace* relation $(w_1, u) \rightsquigarrow (u, w_2)$ is said to hold.

In general, for any two edges *(x, y)* and *(y, z)*, *(x, y)* $\rightsquigarrow$ *(y, z)* holds iff the data inputs or outputs that these edges correspond to *can* share the same memory location (regardless

of whether a specific copy-avoidance strategy chooses to re-use the memory location or not). The can-inplace relation is an equivalence relation. A path $\{x_1, x_2, \ldots, x_n\}$ in a graph $G = (V, E)$, such that $(x_{i-1}, x_i) \rightsquigarrow (x_i, x_{i+1})$ for all $2 \leq i \leq n\text{-}1$, is said to be a *can-inplace* path. Note that by definition, the can-inplace relation $(w_1, v) \rightsquigarrow (v, w_2)$ implies that the node v can overwrite the data flowing over w_1. And in such a case, we say that the relation vXw_1 holds. However, the can-inplace relation $(v_1, w) \rightsquigarrow (w, v_2)$ does not necessarily imply such a destructive operation as the purpose of a wire is to propagate data, not to modify it.
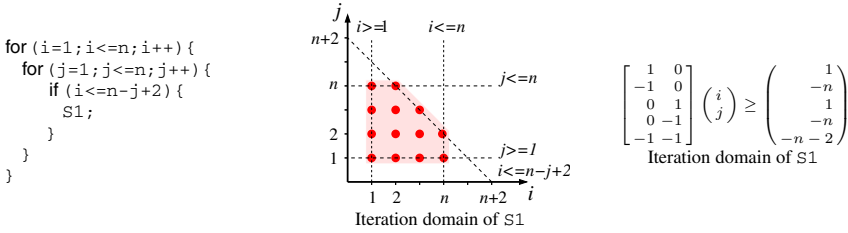
Suppose $<_s$ is a binary relation on V which specifies a total ordering of the computation nodes. The relation $<_s$ specifies a *valid* execution schedule iff $(v_1 <_s v_2)$ implies that there does not exist a directed path in graph G, from v_2 to v_1 for any $v_1, v_2 \in V$ i.e., the schedule respects all dataflow dependences. As we shall see later, the problem of scheduling the computation nodes is closely related to inplaceness. Memory re-use due to copy-avoidance can create additional dependences. A conjunction of scheduling relations $\bigwedge (v_1 <_s v_2)$ is said to be *consistent* with a conjunction of can-inplace relations $\bigwedge ((x, y) \rightsquigarrow (y, z))$, for $x, y, z \in N \cup V$, iff such a schedule does not violate the dependences imposed by such an inplaceness choice.

Loop Inputs and Outputs. Data flowing into and out of a loop is classified as either *loop-invariant input data* or *loop-carried data*. Loop-invariant input data is that which is only read in every iteration of the loop. Let *Inv* be the set of loop-invariant data inputs to the loop. The LabVIEW equivalent for such an input is an input tunnel. In Figure 1, for the outermost loop l_1, *Inv* = $\{w, x, y\}$. Loop-carried data is that which is part of a loop-carried dependence inducing dataflow. The paired loop inputs and outputs represent such a dependence. Let *ICar, OCar* be sets of these loop inputs and outputs. The loop-carried dependence is represented by the one-to-one mapping $lcd : OCar \rightarrow ICar$. The LabVIEW equivalent for such a pair are the left and right shift registers. In Figure 1, for loop l_1, *ICar* = $\{z\}$, *OCar* = $\{z'\}$, $(z, z') \in lcd$.
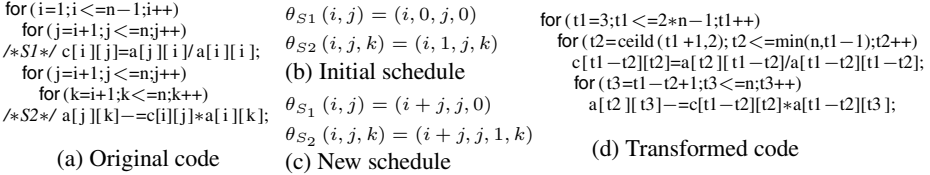
Array Accesses. In Figure 1, the array read access is a node that takes in an array and the access index values to produce the indexed array element value. The array write access, takes the same set of inputs and the value to be written to produce an array with the indexed element overwritten. We model the array read and write accesses similarly. Notice that the output array of an array write, v need not be inplace to the input array flowing through a wire w_1. If it is, then vXw_1.

2.3   Overview of the Polyhedral Model

The polyhedral model provides an abstract mathematical model to reason about program transformations. Consider a program part that is a sequence of statements with zero or more loops surrounding each statement. The loops may be imperfectly nested. The dynamic instances of a statement S, are represented by the integer points of a polyhedron whose dimensions correspond to the enclosing loops. The set of dynamic instances of a statement is called its *iteration domain*, D. It is represented by the polyhedron, defined by a conjunction of affine inequalities that involve the enclosing loop

```
for (i=1;i<=n;i++){
  for (j=1;j<=n;j++) {
    if (i<=n-j+2){
      S1;
    }
  }
}
```


$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{pmatrix} 1 \\ -n \\ 1 \\ -n \\ -n-2 \end{pmatrix}$$

Iteration domain of S1

Fig. 3. Polyhedral representation of a loop-nest in geometric and linear algebraic form

```
for (i=1;i<=n-1;i++)
  for (j=i+1;j<=n;j++)
/*S1*/ c[i][j]=a[j][i]/a[i][i];
  for (j=i+1;j<=n;j++)
    for (k=i+1;k<=n;k++)
/*S2*/ a[j][k]-=c[i][j]*a[i][k];
```

(a) Original code

$\theta_{S1}(i,j) = (i, 0, j, 0)$
$\theta_{S2}(i,j,k) = (i, 1, j, k)$

(b) Initial schedule

$\theta_{S_1}(i,j) = (i+j, j, 0)$
$\theta_{S_2}(i,j,k) = (i+j, j, 1, k)$

(c) New schedule

```
for (t1=3;t1<=2*n-1;t1++)
  for (t2=ceild(t1+1,2); t2<=min(n,t1-1);t2++)
    c[t1-t2][t2]=a[t2][t1-t2]/a[t1-t2][t1-t2];
    for (t3=t1-t2+1;t3<=n;t3++)
      a[t2][t3]-=c[t1-t2][t2]*a[t1-t2][t3];
```

(d) Transformed code

Fig. 4. An example showing the input code, the corresponding original schedule, a new schedule that fuses j loops of both statements while skewing the outermost loop with respect to the second outermost one, and code generated with the new schedule.

iterators and global parameters. Each dynamic instance is uniquely identified by its *iteration vector*, i.e., the vector i_S of enclosing loop iterator values. Figure 3 shows the polyhedral representation of a loop-nest in its geometric and linear algebraic form.

Schedules. Each statement, or more precisely its domain, has an associated schedule, which is a multi-dimensional affine function mapping each integer point in the statement's domain to a unique time point that determines when it is to be executed. Code generated from the polyhedral representation scans integer points corresponding to all statements globally in the lexicographic order of the time points they are mapped to. For example, $\theta_S(i,j,k) = (i+j, j, k)$ is a schedule for a 3-d loop nest with original loop indices i, j, k. Changing the schedule to $(i+j, k, j)$ would interchange the two inner loops. The reader is referred to [3] for more detail on the polyhedral representation.

The initial schedule which is extracted, corresponding to the original execution order, is referred to as an *identity schedule*, i.e., if it is not modified, code generation will lead to the same code as the one from which the representation was extracted. A dimension of the multi-dimensional affine scheduling function is called a *scalar dimension* if it is a constant. In Figure 4(b), the second dimension of both statements' schedules are scalar dimensions. In Figure 4(c) schedules the third dimension is a scalar one. Polyhedral optimizers have models to pick the right schedule among valid ones. A commonly used model that minimizes dependence distances in the transformed space [5], thereby optimizing locality and parallelism simultaneously is implemented in Pluto [16].

3   Extracting the Polyhedral Representation

The polyhedral representation of a SCoP typically consists of an abstract mathematical description of the iteration domain, schedule and array accesses for each statement.

The array accesses are also classified as either read or write accesses. Each of these are expressed as affine functions of the enclosing loop iterators and symbolic constants.

3.1 Challenges

As mentioned earlier, a graphical dataflow program has no notion of a statement. The program is a collection of nodes that represent specific computations, with data flowing along edges that connect one node to another. Referential transparency ensures that each edge could be associated with its own distinct memory location. Generally, copy-avoidance strategies are used to maximize inplaceness of output and input data. However, the exact memory allocation depends on the specific strategy used. Additionally, the problem of copy-avoidance is closely tied with the problem of scheduling the computation nodes. In the matmul program (Figure 1), consider the array write u and the array read r that share the same data source (say, v). If no array copy is to be created, the read must be scheduled ahead of the write, i.e., $u <_s r$ is not *consistent* with (v, w_1) $\rightsquigarrow (w_1, u) \wedge (w_1, u) \rightsquigarrow (u, w_2) \wedge (v, w_1) \rightsquigarrow (w_1, r)$. If the write is scheduled first, the read must work on a copy of the array as the write is likely to overwrite the array input. Abu-Mahmeed et al. [1] have looked into the problem of scheduling to maximize the inplaceness of aggregate data. To summarize, the main challenges in the extraction of the polyhedral representation for a graphical dataflow program are as follows:

1. A graphical dataflow program cannot be viewed as a sequence of statements executed one after the other.
2. While the access expressions could be analyzed just like parse trees, it is difficult to relate the access to a particular array definition as the exact memory allocation depends on the specific copy-avoidance strategy used.
3. The actual execution schedule of the computation nodes determined depends on the copy-avoidance decisions.

A trivial polyhedral representation can be extracted by treating each node of a graphical data flow program as a statement analogue while making the conservative assumption that data is copied over each edge. As most compilers make use of copy-avoidance strategies, such a polyhedral representation most certainly over-estimates the amount of data space required. This also results in an over-estimation of the computation e.g. an array copy. Therefore, the problem of polyhedral transformation in such a representation begins with a serious limitation in terms of dataspace and computation over-estimation. In essence, extraction of a polyhedral representation of a dataflow program part cannot negate the copy-avoidance optimizations. The inplaceness opportunities in the dataflow program must be factored into the analysis.

3.2 Static Control Dataflow Diagram (SCoD)

A SCoP is defined as a maximal set of consecutive statements without while loops, where loop bounds and conditionals may only depend on invariants within this set of statements. Analogous to this, we now characterize a canonical graphical dataflow program, a Static Control Dataflow Diagram (SCoD), which lends itself well to existing

polyhedral techniques for program transformation. The reasoning behind each individual characteristic is provided later.

1. It is a maximal dataflow diagram without constructs for loops that are not countable, where the countable loop bounds and conditionals, in any diagram, only depend on parameters that are invariant for that diagram. Nodes in the SCoD (and its nested diagrams) must be functional, without causing run-time side-effects or relying on any run-time state.

2. The only array primitives that feature as nodes in a SCoD and its nested diagrams are those which read an array element or write to an array element. More importantly, primitives that output array data that cannot be inplace to an input array data cannot be present in the diagrams.

3. For an array data source in any diagram, (v_1, w), there exists at most one node v_2 such that $(v_1, w) \rightsquigarrow (w, v_2) \wedge v_2 X w$.

4. Data flowing into a loop in any diagram is either loop-invariant data or loop-carried scalar data or loop-carried array data that has an associated can-inplace path through the loop body, which creates the loop-carried dependence, i.e., loop input $x \in I \Rightarrow x \in Inv \vee (x \in ICar \wedge (isScalarType(x, w_x) \vee (isArrayType(x, w_x) \wedge (x, w_x) \rightsquigarrow (w_y, y))))$ where $y = lcd^{-1}(x)$, w_x and w_y the input and output wires.

5. In any diagram, there is no can-inplace path from a loop-invariant data input to the loop-carried data input of a inner loop or to the array input of an array element write node, i.e., in any DAG, $G = (N, E)$ that corresponds to the body of a loop, if (v_1, w_1) is the loop invariant input, then there does not exist any edge (w_2, v_2) such that $(v_1, w_1) \rightsquigarrow (w_2, v_2) \wedge v_2 X w_2$.

The first characteristic is closely tied with the characterization of a SCoP. The rest of the characterization specifies a canonical form of dataflow diagram which has can-inplace relations that facilitate polyhedral extraction. As explained earlier, a naive implementation of a dataflow language could write each new output into a new memory location. The question of whether a particular wire vertex gets a new memory allocation or not depends on the actual copy-avoidance strategy employed by the compiler. The problem of extracting the polyhedral representation of an arbitrary dataflow diagram, therefore depends on the copy-avoidance strategy. In order to make the polyhedral extraction independent of it, we canonicalize the dataflow in a given diagram in accordance with the above characteristics.

An operation such as appending an element to an input array data is a perfectly valid dataflow operation. Clearly, the output array cannot be inplace to the input array. (2) ensures that such array operations are disallowed. Furthermore, it is possible in a dataflow program to overwrite multiple, distinct copies of the same array data. In such a case, a copy-avoidance strategy would inplace only one of the copies with the original data and the rest of them would be separate copies of data. (3) precludes such a scenario. It is important to note that it however, still allows multiple writes. (4) ensures that loop-carried dependence involving array data is tied to a single array data source. Assuming the absence of (5), data flowing from loop-invariant source vertex to a loop-carried input of an inner loop would necessitate a copy because the source data would have a pending read in subsequent iterations of the outer loop.

Theorem 1. *In any diagram of the SCoD, G = (V, E), there exists a schedule $<_s$ of the computation nodes V, which is consistent with the conjunction of all possible can-inplace relations, $\bigwedge_{x,\,y,\,z\,\in\,V} ((x, y) \rightsquigarrow (y, z))$, where isArrayType(x, y) $\wedge$ isArrayType(y, z) holds.*

Proof. Consider an array data source *(v, w)* in G with $v \in I$. If $v_1, v_2,\ldots,v_n$ are the nodes that consume the array data, then in accordance with characteristic (3), there is at most one node v_i such that *(v, w)* $\rightsquigarrow$ *(w, v_i)* $\wedge$ $v_i Xw$. Without any loss of generality, we can assume that $i = n$. This implies that any valid schedule $<_s$ where $v_1 <_s v_2 <_s v_3 \ldots <_s v_n$ holds is consistent with $\bigwedge_{i=1}^{n} ((v, w) \rightsquigarrow (w, v_i))$. Similar scheduling constraints can be inferred for the nodes that consume the array data produced by v_n and so on, thereby ensuring that all the can-inplace relations are satisfied for array dataflow. The new constraints inferred cannot contradict an existing constraint as the graph is acyclic. Therefore, any valid schedule $<_s$ where all the inferred scheduling constraints are satisfied is consistent with maximum array inplaceness in the diagram.   □

Essentially, in a SCoD, it is possible to schedule the computation nodes such that no new memory allocation need be performed for any array data inside the SCoD, i.e., all the array data consumed inside the SCoD will then have an inplace source that ultimately lies outside the SCoD.

Lemma 1. *In any diagram of the SCoD, G = (V, E), for any sink vertex $t \in O$, that has array data flowing into it, a can-inplace path exists from a source vertex $s \in I$ to t.*

Proof. There must exist a node v_1 which produces the array data flowing into *t* through wire *w*. So, *(v_1, w)* $\rightsquigarrow$ *(w, t)* holds. In accordance with the model, v_1 can either be an array write node or a loop. In either case, there must exist a node v_2 which produces the array data flowing into v_1 and so on until a source vertex *s* is encountered. The path traversed backwards from *t* to *s* clearly constitutes a can-inplace path.   □

3.3   A Multi-dimensional Schedule of Compute-Dags

A *compute-dag*, $T = (V_T, E_T)$ in a diagram $G = (V, E)$, is a sub-graph of G where there exists a node, $r \in V_T$ such that for every other $x \in V_T$ there exists a path from *x* to *r* in T (the node *r* will hereafter be referred to as the root node). As it is possible to pick inplace opportunities such that no array data need be copied on any edge in the SCoD, any diagram in the SCoD can be viewed as a sequence of computations that write on the incoming array data. Instead of statements, compute-dags, which are essentially dags of computation nodes can be identified. Consider an array write node or a loop node, both of which can overwrite an input array. Starting with a dag that is just this node as the root, the compute-dag can be built recursively by adding nodes which produce data that flows into any of the nodes in the dag. Such a recursive sweep of the graph stops on encountering another array write or loop node. However, while identifying compute-dags in a diagram, it is necessary to account for all the data produced by the nodes in the diagram.

Theorem 2. *In any diagram of the SCoD, G = (V, E), for every edge (x, y) $\in$ E where x is a computation node, there exists a compute-dag T_i = (V_i, E_i) in the set Σ' = {T_1, T_2,...,T_m} of compute-dags rooted at array write or loop nodes, such that x $\in$ V_i iff only array data flows out of every diagram.*

Proof. In accordance with Lemma 1, for any sink vertex $t \in O$, with array data flowing into it, there exists a can-inplace path $p_{s \to t}$ = {s,..., v, w, t} from a source vertex s to t. Also, every node x, which is not dead-code, must have a path $q_{x \to t_j}$ = {x, y, ..., v_j, w_j, t_j} to at least one sink vertex $t_j \in O$. If only array data flows into every sink vertex, consider the first vertex z at which the path $q_{x \to t_j}$ overlaps with $p_{s_j \to t_j}$. z must either be a loop-node or an array write node. In either case x must be part of a compute-dag rooted at z (in the former case, if x $\neq$ z, notice also that the data flowing along *(x, y)* must be the intermediate result of a computation that produces the loop-invariant data for the loop z). On the other hand, if scalar data can flow into a sink vertex t, clearly, the path from node x to t is not guaranteed to have either an array write node or a loop node. Consequently, the node x is not guaranteed to be part of any compute-dag. Likewise, if loops can have scalar loop-carried data since a scalar loop-carried output is represented as a sink vertex in the loop body DAG.   □
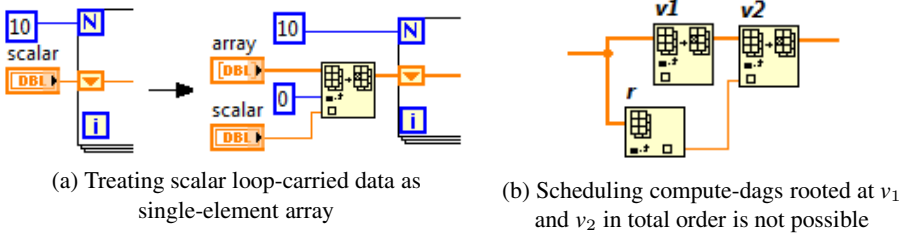
In order to address the consequences of Theorem 2, it is necessary to treat scalar data flowing out of the diagram as a single-element array. This results in a compute-dag that accounts for the scalar dataflow. Likewise, loop-carried scalar data must also be treated as single-element array. The dataflow into the loop-carried input is treated as a write to the array resulting in a corresponding compute-dag (refer Fig 5(a)). (Hereafter, we assume in the following discussion, that scalar data flowing out of a diagram and loop-carried scalar data are treated specially in this way as single-element arrays).

Each diagram in a SCoD is analyzed for compute-dags, starting from the top-level diagram. Suppose θ is the scheduling function. At each diagram level, d, the set of roots of the compute-dags in Σ' = {T_1, T_2,...,T_m} are ordered as follows:

– If data produced by a root node n_1 is consumed by root node n_2, then $\theta_{n_1}^d \prec \theta_{n_2}^d$.
– In accordance with Theorem 1, if there is an array write in a compute-dag rooted at n_1 and an array read in a compute-dag rooted at n_2, both of which are dependent on the same array data source, then $\theta_{n_1}^d \succ \theta_{n_2}^d$. Scheduling n_1 ahead of n_2 in the polyhedral representation would be unsafe. Such a schedule would only be possible if n_2 were to read a copy of the input array, allowing n_1 to overwrite the input array. The safe schedule ensures that an array copy is not required.
– If neither of the above hold for the two root nodes, either $\theta_{n_1}^d \succ \theta_{n_2}^d$ or $\theta_{n_1}^d \prec \theta_{n_2}^d$ should hold true.

Each diagram in the diagram hierarchy of the SCoD contributes to a dimension in the global schedule. Each loop encountered adds an additional dimension. The total order on the compute-dag roots in any diagram determines the time value at which each compute-dag can be scheduled in that dimension. The global schedule is obtained by appending its time value in the owning diagram to the schedule of the owning loop, if any, together with the loop dimension.

Apart from ensuring that all the data produced by the nodes in a diagram are accounted for, it must also be possible to schedule the compute-dag roots in a total order.

(a) Treating scalar loop-carried data as single-element array

(b) Scheduling compute-dags rooted at v_1 and v_2 in total order is not possible

Fig. 5. Single-element arrays and contradiction in schedule of compute-dags

Theorem 3. *In any diagram of the SCoD, G = (V, E), it is possible to schedule the set of roots of the compute-dags in $\Sigma' = \{T_1, T_2, \ldots, T_m\}$ in a total order if for every path $p_{v_1 \to v_2}$ between a pair of roots, v_1 and v_2, there does not exist an array read node, r in the compute-dag $T(v_2)$, such that r and v_1 share the same data source (x, w) with $v_1 X w$ and a path $q_{r \to v_2}$ exists that does not include v_1 on it.*

Proof. Consider a pair of root nodes v_1 and v_2 (refer Fig 5(b)). In accordance with the scheduling constraints specified above, $\theta_{v_1}^d \prec \theta_{v_2}^d$ if a path $p_{v_1 \to v_2}$ exists in G. This scheduling order is contradicted only if for some reason $\theta_{v_1}^d \succ \theta_{v_2}^d$ must hold, which can only happen if the compute-dag $T(v_2)$ contains a node that must be schedule ahead of v_1, i.e., an array read node that shares the same source as v_1. If such a node does not exist, then the contradiction never arises leading to a total ordering of the compute-dag roots. Similarly, there is no contradiction in schedule order if a path $p_{v_1 \to v_2}$ does not exist. □

In order to address the consequence of Theorem 3, while building compute-dag rooted at v_2, it is also necessary to stop on encountering an array read node r when there is a node v_1 with the same array source, overwriting the incoming array, such that there exists a path from r to v_2 which does not include v_1. A separate compute-dag rooted at such an array read must be identified, thereby breaking the compute-dag that would have been identified otherwise (rooted at v_2) into two different dags.

The set of actual statement analogues is $\Sigma = \{T_1, T_2, \ldots, T_n\}$ such that the *root(T_i)* for any $T_i \in \Sigma'$ is not a loop. Algorithm 1 provides a procedure for identifying the set of statement analogues in a given dataflow graph G = (V, E) of a particular diagram. It is possible for two statement analogues to have common sub-expressions. However, the nodes in a SCoD are functional, making the common sub-expressions also so.

Analysis of Iteration Domains. We assume that loop normalization has been done, i.e., all for-loops have a unit stride and a lower bound of zero. Analyzing the iteration domain of a for loop only involves the analysis of the dataflow computation tree that computes the upper bound of the for loop. This analysis is very similar to parsing an expression tree. Symbolic constants are identified as scalar data sources that lie outside the SCoD. Loop iterators and constant data sources are explicitly represented as nodes in our model.

Analysis of Array Accesses. The access expression trees for the array reads and array writes which are present in the compute-dags of the statement analogues are analyzed to

Algorithm 1. *identify-compute-dags(G = (V, E))*

```
Require: Treat scalar data flowing out of diagram as single element arrays
Require: Loop-invariant computations have not been code-motioned out into an
    enclosing diagram
 1: procedure IDENTIFY-COMPUTE-DAGS(G = (V, E))
 2:     Σ = ∅
 3:     for all n ∈ V | isArrayWriter(n) ∧ !isloop(n) do
 4:         Σ =Σ ∪ build-compute-dag(n, G)        ▷ compute-dag from G, with root n
 5:     for all n ∈ V | root-candidate[n] do
 6:         Σ = Σ ∪ build-compute-dag(n, G)
 7:     return Σ

 8: procedure BUILD-COMPUTE-DAG(n, G = (V, E))
 9:     V_T = {n}, E_T = ∅
10:     while (x, y) = get-new-node-for-dag(n, T = (V_T, E_T), G) do
11:         V_T = V_T ∪ x, E_T = E_T ∪ {(x, y)}
12:     return T = (V_T, E_T)

13: procedure GET-NEW-NODE-FOR-DAG(n, T = (V_T, E_T), G = (V, E))
14:     for each (x, y) ∈ E do
15:         if (x, y) ∉ E_T ∧ y ∈ V_T ∧ !isArrayWriter(x) ∧ !isloop(x) then
16:             if isArrayReader(x) then
17:                 z = get-array-write-off-same-source-if-any(x)
18:                 if there exists a path p_{z→n} then
19:                     root-candidate[x] = true
20:                     continue
21:             return (x, y)
22:     return ∅
```

obtain the access functions. The most important problem of tying the array access to a particular memory allocation is resolved easily. Due to a carefully determined scheduling order, which schedules array reads ahead of an array write having the same source, all the accesses can be uniquely associated with array data sources that lie outside the SCoD. This is regardless of the actual copy-avoidance strategy that may be used. Additionally, the scalar data produced by an array read that is the root of its own compute-dag and a node in another compute-dag is treated as a single-element array, thereby encoding the corresponding dependence in the array accesses of both the compute-dags. So, each statement analogue has exactly one write access.

4   Code Synthesis

A polyhedral optimizer can be used to perform the required program transformations on the polyhedral representation of the SCoD. We now consider the problem of synthesizing a SCoD given its equivalent polyhedral representation.

4.1   Input

The input polyhedral representation must capture the iteration domain, access and scheduling information of the statement analogues i.e., the set $\Sigma = \{T_1, T_2, \ldots, T_n\}$ of compute-dags, which are also available as input. Each compute-dag, derived perhaps from an earlier polyhedral extraction phase, has exactly one array write node, which is the root of the dag.

Algorithm 2. *Synthesize-SCoD()*

```
1:  Convention: If s represents a source vertex, the paired sink is s′
2:  procedure SYNTHESIZE-SCOD( )
3:      Let G₀ be the DAG of the top level diagram, G₀ = (∅, ∅)
4:      create-source-vertex-for-each-global-parameter(G₀)
5:      for each statement analogue, T in global schedule order do
6:          Read domain (D), identity schedule (θ) and access (A) matrices
7:          l = create-or-get-loop-nest(G₀, D, θ)              ▷ l, innermost loop
8:          add-compute-dag(l, T, G₀)
9:          for each (variable, read access) pair (v, a) in A do
10:             (s₀, s′₀) = create-source-and-sink-vertices-if-none(v, G₀)
11:             create-or-get-dataflow(s₀, s′₀, l, G₀, READ)
12:             array-read-node-access(a, T, l, G₀)   ▷ node reads data flowing into
    l through loop-invariant input or data flowing into the loop carried output if
    it exists. Create array index expression tree using a
13:             (v, a) = get-variable-write-access-pair(A)
14:             (s₀, s′₀) = create-source-and-sink-vertices-if-none(v, G₀)
15:             create-or-get-dataflow(s₀, s′₀, l, G₀, WRITE)
16:             insert-array-write-node(a, T, l, G₀)              ▷ node is added to flow
    path so that it overwrites the data flowing into the loop-carried output of l.
    Create array index expression tree
17:             create-dataflow-from-parameters-and-iterators(c, G₀)
18:     return G₀
```

The polyhedral representation must have identity schedules. Any polyhedral representation with non-identity schedules can be converted to one with identity schedules by performing code generation and extracting the generated code again into the polyhedral representation. In this manner, scheduling information gets into statement domains and the schedule extracted from the generated code is an identity one. Once an equivalent polyhedral representation in this form has been obtained, the approach described in the rest of this section is used to synthesize a SCoD.

4.2   Synthesizing a Dataflow Diagram

The pseudocode for synthesizing a dataflow diagram is presented in Algorithms 2 and 3. The statement analogues are processed in their global schedule order (line 2.5). The iteration domain and scheduling information of a statement analogue are together used to create the surrounding loop-nest (line 2.7). Lower and upper bounds are inferred for each loop iterator. In case the for-loop is a normalized for-loop as in our abstract model, the actual upper bound will be a difference of the minimum and maximum of the inferred upper and lower bounds plus one. Built-in primitives for various operations such as max, min, floor, ceil etc. may be used to set up the loop-control. Note that if the required loop-nest has been created already for a statement analogue scheduled earlier, it need not be created again. The compute-dag is then added to the dataflow graph of the enclosing loop (line 2.8).

Inherent Parallelism – the Factor to Consider. Dataflow programs are inherently parallel. A computation node is ready to be fired for execution as soon as all its inputs are available. It is essential to exploit this inherent parallelism during code synthesis. In order to infer such parallelism and exploit it, we reason in terms of *coalesced dependences*. A coalesced dependence is the same as a regular data dependence except that two accesses are considered to be in conflict if they even access the same variable (potentially an aggregate data type), as opposed to the same location in the aggregate data.

For example, an array access that writes to odd locations does not conflict with another that reads from even locations. However, a coalesced dependence exists between the two. Analogous to regular data dependences, we now also use the terms flow, anti, and output coalesced dependences.

A unique source-sink vertex pair (s_0, s_0') is created in the top-level DAG, G_0, of the top-level diagram for each variable v access is described in the access matrices (lines 2.10, 2.14). A dataflow path is also created from s_0 to s_0'. The problem of synthesizing a dataflow diagram is essentially a problem of synthesizing the dependences between the given set $\Sigma = \{T_1, T_2, \ldots, T_n\}$ of compute-dags in terms of edges that will connect them together. Specifically, as all the dependences involve array variables (may be single-element), these interconnecting edges represent the dataflow between array read or write nodes in the compute-dags, through intervening loops. Consider set of array write nodes $U = \{u_1, u_2, \ldots, u_n\}$, which correspond to write accesses on the same variables in a particular time dimension such that u_i is scheduled ahead of u_j for all $i < j$ (i.e., the corresponding compute-dags).

Theorem 4. *All coalesced output dependences on a variable in the polyhedral representation are satisfied by a synthesized dataflow diagram if in any diagram, all array write nodes $u_1, u_2, \ldots, u_n$ corresponding to write accesses to that variable lie on the same can-inplace path $p_{u_1 \to u_n}$.*

Proof. Suppose all the nodes in $U = \{u_1, u_2, \ldots, u_n\}$ are scheduled in the outermost diagram. A coalesced output dependence exists between any pair of write nodes scheduled in this diagram, thereby defining a total ordering on the set U. Therefore, all the corresponding array write nodes must be inserted along the can-inplace path $p_{s_0 \to s_0'}$. Now consider a write node u scheduled in an inner loop. A coalesced output dependence exists between u and any array write node $u_i \in U$. This is ensured by inserting the inner loop along the path $p_{s_0 \to s_0'}$, in accordance with its schedule order relative to the other writes nodes on the path. The incoming and outgoing edges of the loop node on the can-inplace path must correspond to the loop-carried input and its paired output, which in turn serve as the source and sink vertices in the DAG of the loop body.   □

Theorem 5. *A coalesced flow dependence in the polyhedral representation is satisfied by a synthesized dataflow diagram if the array write node and read node associated with the dependence lie on the same can-inplace path.*

Proof. Each of the array write nodes $u_1, u_2, \ldots, u_m$ lies on the can-inplace path $p_{u_1 \to u_m}$ due to Theorem 4. A coalesced flow dependence exists between the the write access u_m and read access r. Therefore, there must be a path $p_{u_m \to r}$, which means that all of these nodes must lie on the same can-inplace path $p_{u_1 \to r}$. If a read access r is the only access to a variable inside an inner loop l, the coalesced flow dependence between r and any u_i scheduled earlier is satisfied by a can-inplace path $p_{u_1 \to l}$. The incoming edge to l on this path need only correspond to a loop-invariant input. It acts as a data source for r in the loop body.   □

Together, from Theorem 4 and Theorem 5, it can be seen that the path $p_{u_1 \to r}$ diverges from the path $p_{u_1 \to u_n}$ at u_m i.e., the last write scheduled ahead of r. This enables the concurrent execution of the array write node u_{m+1} and r, thereby exposing the inherent

parallelism in a dataflow diagram discussed earlier. There is no coalesced output or coalesced flow dependence between u_{m+1} and r. Also, just as the output array of an array write node can be inplace to the input array, loop-carried array outputs of a loop node can be inplace to the corresponding input. Similarly, a loop-invariant array input corresponds to the array input of a read node, as they do not have a corresponding output that can be inplace. Due to this symmetric relationship, based on coalesced dependences, we can infer inherent parallelism in the following scenarios:

- Consider two compute-dags, T_1 and T_2, scheduled in the same time dimension, d, such that $\theta^d_{T_1} \prec \theta^d_{T_2}$ with no coalesced output or coalesced flow dependence between them e.g. the two compute-dags have array accesses on disjoint sets of arrays. T_1 and T_2 then constitute two tasks that can executed in parallel in a dataflow program.
- Consider two loops, l_x and l_y, scheduled in the same time dimension such that there is no coalesced output or coalesced flow dependence between compute-dags in one loop and those of the other e.g. compute-dags in l_x only read a particular array variable, where those in l_y only write to it. The two loops can be executed as parallel tasks. This can be particularly crucial in obtaining good performance.
- Similarly, a loop and a compute-dag scheduled in the same time dimension with no coalesced output or coalesced flow dependence between the compute-dag and those in the loop.

Note that coalesced anti-dependences do not inhibit parallelism. The read and write access on the same variable may share the same data source. The read access can be performed on a copy of the data, while the write access is performed on the source data.

A dataflow diagram synthesized as described in the proofs for Theorem 4 and 5 is indeed a SCoD. The characteristics (1) and (2) are trivially satisfied. The dataflow diagram also meets characteristic (3) as all the array write nodes are serialized in accordance with Theorem 4. Furthermore, the construction described in the proof for Theorem 4 also ensures that whenever a loop-carried input-output pair is created, the corresponding source and sink vertices have an associated can-inplace path, thereby ensuring characteristic (4). Finally, the proof for Theorem 5 also implies a loop-carried input for a particular variable access is created on a loop only when all the accesses to a variable inside the loop-nest are read accesses. Therefore, a flow path from a loop-invariant source vertex to a loop-carried input never exists, ensuring characteristic (6).

Algorithm 2 processes the read accesses of a statement analogue first and then the write access. Algorithm 3, briefly explained below, describes the creation of the array dataflow paths for the corresponding read and write nodes in the compute-dag.

Read Accesses: Suppose the array read node is scheduled to execute in loop l_m. The closest enclosing loop l_c that has an array write node (for a write access on the same variable) in its body, and therefore, an associated loop-carried input s_c is found (line 3.4). A dataflow through loop-invariant inputs is then created to propagate the data flowing into the loop-carried output s'_c (line 3.12) to the inner loop l_m (line 3.14). This is the data produced by the write node associated with the last write access on the variable. However, if part of such a flow through loop-invariant inputs already exists for an intervening loop, it is extended to reach l_m (line 3.13).

Algorithm 3. *Creation of loop-carried and loop-invariant dataflow*

```
 1: procedure CREATE-OR-GET-DATAFLOW(s₀, s₀', lₘ, G₀, access − type)
 2:     {l₁,...,lₘ} = get-enclosing-loops(lₘ)              ▷ {G₁,...,Gₘ} be their DAGs
 3:     sources = get-inflow-if-any(s₀, lₘ, G₀)
 4:     c = max i | i ∈ {0, 1,...,|sources|} and sᵢ ∈ ICar[lᵢ] for i > 0
 5:     if access-type == WRITE then
 6:         if c < m then
 7:             find v | (v, w), (w, s_c') ⊂ E_{|c|}
 8:             create flow path from v to s_c' through loop lₘ via loop-carried
    inputs/outputs (transforming s_{c+1},...,s_{|sources|} into loop-carried inputs)
 9:             replace flow path (v, w, s_c') with this new flow path
10:     else if |sources|< m then                         ▷ must be a read access
11:         if c == |sources| then                        ▷ use data overwritten in outer loop
12:             find v | (v, w), (w, s_c') ⊂ E_{|c|}
13:         else v = s_{|sources|}  ▷ extend loop-invariant flow
14:         create a flow path through loop-invariant inputs from v to lₘ
15:     return

16: procedure GET-INFLOW-IF-ANY(s₀, lₘ, G₀)
17:     {l₁,...,lₘ} = get-enclosing-loops(lₘ)                        ▷ l₁ outermost
18:     s = s₀, H = G, U = V, F = E, sources = ∅
19:     for i ← 1, m do
20:         wᵢ = wire carrying data from s
21:         if ∃ w ∈ U | (w, lᵢ) ∈ F ∧ (s, wᵢ) ⤳ (w, lᵢ) then
22:             H = DAG that describes body of loop lᵢ
23:             s = source vertex in H that corresponds to loop input (w, lᵢ)
24:             sources = append s to the sources list
25:         else break
26:     return sources
```

Write Accesses: Suppose the array write node is scheduled in a loop l_m. As in the case of a read access, the loop l_c is found (line 3.4). Any flow of data through loop-invariant inputs of intervening loops, from the source v of the loop-carried output s_c' is transformed to a flow of data through loop-carried inputs to the inner loop l_m. The newly created data flow through loop-carried inputs and outputs replaces the existing flow path *(v, w, s_c')* (line 3.7-line 3.9).

Once the dataflow from the variable source vertex is created to the loop enclosing the access node, it can read or write the data flowing in. The required access computation trees are created using the access information (usually represented by a matrix). The data outputs from these trees serve as the index inputs to access node.

Loop Iterators and Global Parameters: Besides the variable accesses, considered so far, there might still be other nodes whose input dataflow is yet to be created. The sources of these node inputs are either the loop iterators or global parameters for the SCoD e.g. consider the compute-dag that corresponds to `(b[i] = a[i] + i)`, the `i` input to the add node in the compute-dag still needs an input dataflow. Two mappings, `paramSource` and `iteratorSource`, from the set of node inputs to the sets of global parameters and loop iterators, can be used to create the input dataflow from the corresponding source vertices. In an actual implementation, these mappings have to be derived from the earlier phases of polyhedral extraction and optimization.

5   The PolyGLoT Auto-transformation Framework

We employed the techniques described so far to build PolyGLoT, a polyhedral automatic transformation framework for LabVIEW. The LabVIEW compiler translates the source dataflow diagram into a hierarchical, graph-based dataflow intermediate representation (DFIR). Several standard compiler optimizations are performed on this intermediate representation. We implemented a separate pass that uses PolyGLoT to perform polyhedral extraction, auto-transformation and dataflow diagram synthesis in that order. The optimized DFIR graph is then translated to the LLVM IR, which is then further optimized by the LLVM backend compiler that finally emits the machine code.
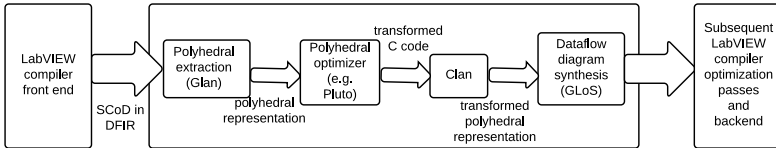
Fig. 6. A high-level overview of PolyGLoT

PolyGLoT consists of four stages. The first stage extracts the polyhedral representation from a user-specified SCoD using the techniques described in Section 3. The translation is performed on DFIR. Glan (named after its C counterpart), a G loop analysis tool, was implemented to serve this purpose. The polyhedral representation extracted is used as an input to Pluto, an automatic parallelizer and locality optimizer. Pluto then applies a sequence of program transformations that include loop interchange, skewing, tiling, fusion, and distribution. Pluto internally calls into CLooG to output the transformed program as C code. Glan was used to produce a representative text (encoding a compute-dag id and also text describing the array accesses) for each compute-dag. Thus, we ensured that the transformed C code produced by Pluto included statements that could be matched with the computed-dags identified during extraction.

Clan was used to extract the polyhedral representation of the transformed C code, which was finally used as the input for GLoS (G loop synthesis). GLoS is a tool that synthesizes DFIR from the input polyhedral representation as per techniques developed in Section 4. Pluto was also used to produce scheduling information of loops that it would parallelize using OpenMP. This information was used by GLoS to parallelize the corresponding loops in the synthesized DFIR using the LabVIEW parallel for loop feature.

6   Experimental Evaluation

For the purpose of experimental evaluation, we implemented many of the benchmarks in the publicly available Polybench/C 3.2 [17] suite in LabVIEW. The matmul and ssymm benchmarks from the example test suite in Pluto were also used. Each of these benchmarks were then compiled using five different configurations.

– *lv-noparallel* is the configuration that simply uses the LabVIEW production compiler. This is the baseline for configurations that do not parallelize loops.

Table 1. Summary of performance (sequential and parallel execution on an 8-core machine)

Benchmark	Problem size	Execution time (seq)		Speedup	Execution time (8 cores)			Speedup over *lv-par*	
		lv-nopar	*pg-loc*	(local)	*lv-par*	*pg-par*	*pg-loc-par*	*pg-par*	*pg-loc-par*
atax	NX=4096, NY=4096	0.456s	0.567s	0.80	0.707s	0.642s	0.167s	1.10	4.23
bicg	NX=4096, NY=4096	0.409s	0.689s	0.59	0.409s	0.220s	0.093s	1.86	4.40
doitgen	NQ=NR=NP=128	7.476s	7.344s	1.02	0.976s	0.999s	0.934s	0.98	1.04
floyd-warshall	N=1024	86.06s	91.89s	0.94	82.76s	13.64s	4.909s	6.07	16.9
gemm	NI=NJ=NK=1024	60.40s	24.20s	2.50	7.026s	5.473s	3.628s	1.28	1.94
gesummv	N=4096	0.488s	0.536s	0.91	0.078s	0.069s	0.074s	1.13	1.05
matmul	N=2048	688.5s	196.3s	3.51	89.49s	94.70s	27.44s	0.94	3.26
mvt	N=4096	1.248s	0.828s	1.51	0.195s	0.334s	0.105s	0.58	1.86
seidel	N=1024, T= 1024	44.82s	44.79s	1.00	45.03s	9.797s	8.364s	4.60	5.38
ssymm	N=2048	122.8s	177.4s	0.69	15.03s	55.45s	23.85s	0.27	0.63
syr2k	NI=1024, NJ=1024	34.03s	30.86s	1.10	4.190s	4.423s	4.223s	0.95	0.99
syrk	NI=1024, NJ=1024	24.44s	22.01s	1.11	2.974s	3.118s	2.793s	0.95	1.06
trmm	N=2048	231.7s	64.62s	3.59	41.29s	39.94s	11.42s	1.03	3.62

- *pg-loc* uses the LabVIEW compiler but with our transformation pass enabled to perform locality optimizations.
- *lv-parallel* again uses the LabVIEW production compiler, but with loop parallelization. The parallel for loop feature in LabVIEW [19, 7] is used to parallelize loops when possible in the G code.
- *pg-par* is with our transformation pass enabled to perform auto-parallelization but without any locality optimizing transformations. In order to realize a parallel loop identified as parallelizable, Bordelon et al [7]'s solution is used. The parallel loops in the transformed code are identified using Pluto [16].
- *pg-loc-par* is with our transformation pass enabled to perform both locality optimizations and auto-parallelization.

The comparisons of the runtime performance with various configurations can be found in Table 1. The performance numbers were obtained on on a dual-socket Intel Xeon CPU E5606 (2.13 GHz, 8 MB L3 cache) machine with 8 cores in all, and 24 GB of RAM. LLVM 2.8 was the final backend compiler used by LabVIEW.

Table 1 shows that the benchmarks gemm, matmul, mvt, syr2k, syrk and trmm benefit from locality-enhancing optimizations, in particular, loop tiling, and in addition, loop fusion and other unimodular transformations [2, 18]. Table 1 also shows the effect of locality optimizations in conjunction with loop parallelization. It can be seen that for floyd-warshall and seidel, loop skewing exposes loop parallelism that could not have been exploited without it. The benchmarks, atax, bicg, floyd-warshall, gemm, matmul, mvt, seidel, syrk, trmm benefit from more coarse-grained parallelism, i.e., a reduced frequency of shared-memory synchronization between cores as a result of loop tiling. In some cases, we see a slow down with PolyGLoT, often by about 10%. We believe that this is primarily due to transformed code generated by PolyGLoT not being optimized by subsequent passes within LabVIEW and the backend compiler (LLVM) as well as the baseline (*lv-noparallel* and *lv-parallel*). This is also partly supported by the fact that *pg-loc* itself produces this slow down, for example, for ssymm. Better downstream optimization within LabVIEW and in LLVM after PolyGLoT has been run can address this. In addition, the loop fusion heuristic used by Pluto can be tailored for LabVIEW code to obtain better performance. Overall, we see a mean speedup of 2.30× with PolyGLoT (*pg-loc-par*) over the state-of-the-art (*lv-parallel*).

7   Related Work

Much work has been done on using polyhedral techniques in the compilation of imperative languages [9, 11, 5]. Clan is a widely used research tool for extracting a polyhedral representation from C static control parts [3]. Production compilers with polyhedral framework implementations include IBM XL [6], RSTREAM [15], and LLVM [12].

Ellmenreich et al. [8] have considered the problem of adapting the polyhedral model to parallelize a functional program in Haskell. The source program is analyzed to obtain a set of parallelizable array definitions. Dependence analysis on each array set is then performed to parallelize all the computations within the set. Johnston et al. [13] review the advances in dataflow programming over the decades. Ample work has been done on parallelizing dataflow programs. It includes the work on loop parallelization analysis by Yi et al. [19]. Dependences between array accesses are analyzed using standard techniques to determine if a given user-specified loop in a graphical dataflow program can be parallelized. In contrast to these works, the focus of our work is not really on parallelization but on leveraging existing polyhedral compilation techniques to perform dataflow program transformations. Parallelism detection is but a small component of a loop-nest optimization framework. The complete polyhedral representation that we extract from a given dataflow program part can be used to drive automatic transformations, many of which can actually aid parallelization. Furthermore, to the best of our knowledge, no prior art exists that tackles this problem and the problem of dataflow program part synthesis from an equivalent polyhedral representation by exploiting the inplaceness opportunities that can be inferred from the dataflow program. The work of Yi et al. [19] is commercially available as the parallel for loop feature in LabVIEW, and we compared with it through experiments in Section 6. Given an iterative construct in a dataflow program that is marked parallel, Bordelon et al. [7] studied the problem of parallelizing and scheduling it on multiple processing elements. Our system uses it to eventually realize parallel code from the transformed DFIR.

The interplay between scheduling and maximizing the inplaceness of aggregate data has been studied by Abu-Mahmeed et al. [1]. Recently, Gerard et al. [10] have built on this work to provide a solution for inter-procedural inplaceness using language annotations that express inplace modifications. The soundness of such an annotation scheme is guaranteed by a semi-linear type system, where a value of a semi-linear type can be read multiple times and then updated once. For any array data source in any diagram of the SCoD, there is at most one node that can overwrite it. During the polyhedral extraction, by scheduling a write node after all the read nodes which share the same data source, we in effect choose semi-linear type semantics on the array data in the dataflow diagram. It also allows us to infer an inplace path of array updates. The inplace path is used for associating the accesses to an array definition in the polyhedral representation, which can have multiple write accesses to the same definition.

8   Conclusions

We have addressed the problem of extracting polyhedral representations from graphical dataflow programs that can be used to perform high-level program transformations

automatically. Additionally, we also studied the problem of synthesizing dataflow diagrams from their equivalent polyhedral representation. To the best of our knowledge, this is the first work which tackles these problems, and does this while exploiting in-placeness opportunities inherent in a dataflow program. We also demonstrated that our techniques are of practical relevance by building an automatic transformation framework for the LabVIEW compiler that uses them. In several cases, programs compiled through our framework outperformed those compiled otherwise by significant margins, sometimes by a factor as much as seventeen. A mean speed-up of $2.30\times$ was observed over state-of-the-art.

References

[1] Abu-Mahmeed, S., McCosh, C., Budimlić, Z., Kennedy, K., Ravindran, K., Hogan, K., Austin, P., Rogers, S., Kornerup, J.: Scheduling Tasks to Maximize Usage of Aggregate Variables in Place. In: de Moor, O., Schwartzbach, M.I. (eds.) CC 2009. LNCS, vol. 5501, pp. 204–219. Springer, Heidelberg (2009)

[2] Aho, A.V., Sethi, R., Ullman, J.D., Lam, M.S.: Compilers: Principles, Techniques, and Tools, 2nd edn. Prentice-Hall (2006)

[3] Bastoul, C.: Clan: The Chunky Loop Analyzer. The Clan User Guide

[4] Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT, pp. 7–16 (September 2004)

[5] Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 132–146. Springer, Heidelberg (2008)

[6] Bondhugula, U., Gunluk, O., Dash, S., Renganarayanan, L.: A model for fusion and code motion in an automatic parallelizing compiler. In: PACT. ACM (2010)

[7] Bordelon, A., Dye, R., Yi, H., Fletcher, M.: Automatically creating parallel iterative program code in a data flow program (20100306733) (December 2010),
http://www.freepatentsonline.com/y2010/0306733.html

[8] Ellmenreich, N., Lengauer, C., Griebl, M.: Application of the Polytope Model to Functional Programs. In: Carter, L., Ferrante, J. (eds.) LCPC 1999. LNCS, vol. 1863, pp. 219–235. Springer, Heidelberg (2000)

[9] Feautrier, P.: Some efficient solutions to the affine scheduling problem: Part I, one-dimensional time. International Journal of Parallel Programming 21(5), 313–348 (1992)

[10] Gérard, L., Guatto, A., Pasteur, C., Pouzet, M.: A modular memory optimization for synchronous data-flow languages: application to arrays in a lustre compiler. In: LCTES, pp. 51–60 (2012)

[11] Griebl, M.: Automatic Parallelization of Loop Programs for Distributed Memory Architectures. University of Passau, Habilitation thesis (2004)

[12] Grosser, T., Zheng, H., Aloor, R., Simbrger, A., Grolinger, A., Pouchet, L.-N.: Polly: Polyhedral optimization in LLVM. In: IMPACT (2011)

[13] Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. 36(1) (March 2004)

[14] NI LabVIEW Compiler: Under the Hood,
     `http://www.ni.com/white-paper/11472/en`
[15] Meister, B., Vasilache, N., Wohlford, D., Baskaran, M.M., Leung, A., Lethin, R.: R-stream
     compiler. In: Encyclopedia of Parallel Computing, pp. 1756–1765. Springer (2011)
[16] PLUTO: An automatic polyhedral parallelizer and locality optimizer for multicores,
     `http://pluto-compiler.sourceforge.net`
[17] Polybench, `http://polybench.sourceforge.net`
[18] Wolfe, M.: High Performance Compilers for Parallel Computing. Addison-Wesley Long-
     man Publishing Co., Inc., Boston (1995)
[19] Yi, H., Fletcher, M., Dye, R., Bordelon, A.: Loop parallelization analyzer for data flow
     programs (20100306753) (December 2010),
     `http://www.freepatentsonline.com/y2010/0306753.html`