On Demand Parametric Array Dataflow Analysis

Sven Verdoolaege
Consultant for LIACS, Leiden
INRIA/ENS, Paris
Sven.Verdoolaege@ens.fr

Hristo Nikolov
LIACS, Leiden
nikolov@liacs.nl

Todor Stefanov
LIACS, Leiden
stefanov@liacs.nl

ABSTRACT

We present a novel approach for exact array dataflow analysis in the presence of constructs that are not static affine. The approach is similar to that of fuzzy array dataflow analysis in that it also introduces parameters that represent information that is only available at run-time, but the parameters have a different meaning and are analyzed before they are introduced. The approach was motivated by our work on process networks, but should be generally useful since fewer parameters are introduced on larger inputs. We include some preliminary experimental results.

1. INTRODUCTION AND MOTIVATION

Array dataflow analysis [12] (also known as value-based dependence analysis [20]) is of crucial importance in the polyhedral framework with applications in array expansion [2, 10], scheduling [13], equivalence checking [5, 27] optimizing computation/communication overlap in MPI programs [18] and the derivation of process networks [24, 28] to name but a few. For program fragments that are static affine, dataflow analysis can be performed exactly [12, 15, 20]. For programs that contain certain dynamic and/or non-affine constructs, both exact [4] and approximate [21] approaches have been proposed. In particular, fuzzy array dataflow analysis (FADA) [4] introduces additional parameters, whose values depend on run-time information, that allow the dependences to be represented exactly. After all parameters have been introduced, certain properties on the parameters are derived that allow for a simplification of the result. In the end, the parameters may also be projected out, resulting in approximate but static dependences.

The initial motivation for our new approach stems from our work on the derivation of process networks. A process network derived from a sequential program is essentially a refinement of the dataflow graph of the program, where nodes in the graph correspond to processes and edges to communication channels, and is mainly used to represent the task-level parallelism available in the program. These process networks may then be mapped to various hardware implementations [17]. The parameters introduced by FADA can be used to construct control channels between the different processes [22]. Unfortunately, preliminary experiments with the only known publicly available implementation of FADA [6] have shown that this approach tends to introduce too many parameters to be practically useful, whence the need for an alternative approach. For our application, we are mainly interested in dynamic and/or non-affine conditions and this is currently the only extension beyond static affine programs that we support.

Our approach shares many similarities with FADA and is therefore also useful for other applications of this approach [3] (a description of these other applications is however beyond the scope of the present paper). In particular, we also introduce parameters whose values depend on run-time information. However, our parameters have a *different meaning*, essentially representing the last iteration of a potential source that was executed, and we analyze their effect *before* they are introduced. This allows us to (typically) introduce *fewer* parameters, resulting in simpler dependence relations that can be computed more efficiently. Unlike the FADA approach, our approach does not rely on a resolution engine, but instead performs operations on affine sets and relations to determine which parameters to add and which constraints they should satisfy.

We start with a description of our representation of static affine programs in Section 2 and an overview of standard dataflow analysis in Section 3. These sections also introduce notation that is used in later sections. The representation of dynamic conditions in the input is explained in Section 4, while the representation of dynamic dependence relations is explained in Section 5. Section 6 describes the computation of dynamic dependence relations. In Section 7, we discuss some extensions. We conclude with preliminary experimental results in Section 8 and a comparison to related work in Section 9.

2. PROGRAM REPRESENTATION

Each input program is represented using a polyhedral model [14], consisting of *iteration domains*, *access relations*, *dependence relations* and a *schedule*. Each statement has an associated iteration domain containing the values of the iterators of the enclosing loops for which the statement is executed. For example, the iteration domain of the statement in Line 14 of Listing 1 is

$$\{\, \mathtt{H}(k,i,j) \mid 0 \le k,j \le 99 \wedge 0 \le i < 99 \,\}. \qquad (1)$$

Note that `n` is modified inside the program and can therefore not be treated as a parameter. This iteration domain is therefore an overapproximation of the set of executed iterations, based on the bounds on `n` specified by the user in Line 3. The mechanism for filtering out the iterations that have not actually been executed is explained in Section 4. An access relation maps elements of an iteration domain to the element(s) of an array domain accessed by that iteration of the associated statement through some array reference. For example, the access to `a` in the same statement is

```
1       int n, m;
2       int a[100][100];
3       #pragma value_bounds n 0 99
4       #pragma value_bounds m 0 100
5
6   N1: n = f();
7       for (int k = 0; k < 100; ++k) {
8   M:      m = g();
9           for (int i = 0; i < m; ++i)
10              for (int j = 0; j <= n; ++j)
11  A:              a[j][i] = g();
12          for (int i = 0; i < n; ++i)
13              for (int j = 0; j < m; ++j)
14  H:              h(i, j, a[i + 1][j]);
15  N2:     n = f();
16      }
```

Listing 1: Code with locally static conditions

represented as $\{\, \mathtt{H}(k,i,j) \to \mathtt{a}(i+1,j) \,\}$, simplified with respect to the iteration domain. In this paper, a dependence relation maps a (nested) read access relation to the write access relation that wrote the value being read by the read access. For example, the dependence relation for the above read access is $\{\, (\mathtt{H}(k,i,j) \to \mathtt{a}(i+1,j)) \to (\mathtt{A}(k,j,i+1) \to \mathtt{a}(i+1,j)) \mid 0 \le k,j \le 99 \wedge 0 \le i < 99 \,\}$, while the dependence relation for the read of `n` in the bound of the enclosing loop (simplified with respect to the iteration domain) is

$$\begin{aligned} &\{\, (\mathtt{H}(0,i,j) \to \mathtt{n}()) \to (\mathtt{N1}() \to \mathtt{n}()) \,\} \cup \\ &\{\, (\mathtt{H}(k,i,j) \to \mathtt{n}()) \to (\mathtt{N2}(k-1) \to \mathtt{n}()) \mid k \ge 1 \,\}. \end{aligned} \tag{2}$$

Note that the above access and dependence relations are all single-valued functions. We will, however, also use the "$\to$" notations for other relations that may not represent single-valued functions. The schedule maps the union of all iteration domains to a common space where the execution order of the corresponding statement iterations is determined by the lexicographical order.

Each of the above sets and relations is represented in `isl` [25]. The iteration domains, the access relations and the initial schedule are extracted using `pet` [26], while the construction of the dependence relations is the topic of the present paper. In the base case, the input program consists of expression statements, `if` conditions and `for` loops with only static quasi-affine index expressions, conditions and bounds. The representation of dynamic (or non-affine) conditions, the main focus of the present paper, is explained in Section 4. Dynamic loop conditions and dynamic index expressions are briefly discussed in Section 7. The initial schedule describes an ordering that corresponds to the original execution order. To simplify the exposition, we will not take arbitrary initial schedules as input, but instead exploit the positions of the statements inside the abstract syntax tree, in particular the number of shared outer loops and the relative order of pairs of statements.

3. STANDARD DATAFLOW ANALYSIS

Several algorithms have been proposed in the literature for performing dataflow analysis in the case of static affine programs [12, 15, 20]. Our implementation in `isl` is a variation of these algorithms. We do not claim any novelty in this implementation and we will not describe the implementation in detail here. Instead, we only highlight those aspects that are important for understanding the remainder of this paper.

Dataflow analysis is performed separately for each read access (a.k.a., "sink" or "consumer") C. For each such read access we consider all the write accesses (a.k.a., "potential source" or "producer") P that access the same array, ordering them such that the "closest" are considered first. Let A_C and A_P be the corresponding access relations. Furthermore, let B represent the relative ordering of the statements. That is, if $\mathcal{I}$ is the union of all iteration domains, then B is a binary relation on $\mathcal{I}$ with $\mathbf{i} \to \mathbf{j} \in B$ iff $\mathbf{j}$ is executed before $\mathbf{i}$. For example, $\{\, \mathtt{H}(k,i,j) \to \mathtt{N2}(k') \mid 0 \le k,k',i,j \le 99 \wedge k' < k \,\}$ is a subset of B. Note that bold variable names are used to denote (named) integer tuples. Furthermore, if S is a set and if R, R_1 and R_2 are relations, then let 1_S be the identity relation on S, i.e,

$$1_S = \{\, \mathbf{s} \to \mathbf{s} \mid \mathbf{s} \in S \,\},$$

let R^{-1} be the inverse of R, i.e.,

$$R^{-1} = \{\, \mathbf{t} \to \mathbf{s} \mid \mathbf{s} \to \mathbf{t} \in R \,\},$$

let $R_2 \circ R_1$ be the composition of R_1 and R_2, i.e.,

$$R_2 \circ R_1 = \{\, \mathbf{s} \to \mathbf{u} \mid \exists \mathbf{t} : \mathbf{s} \to \mathbf{t} \in R_1 \wedge \mathbf{t} \to \mathbf{u} \in R_2 \,\},$$

let $R_1 \times R_2$ denote the cross product of R_1 and R_2 with

$$R_1 \times R_2 = \{\, (\mathbf{s} \to \mathbf{t}) \to (\mathbf{u} \to \mathbf{v}) \mid \mathbf{s} \to \mathbf{u} \in R_1 \wedge \mathbf{t} \to \mathbf{v} \in R_2 \,\}$$

and let $\underrightarrow{\operatorname{ran}}\, R$ map a nested copy of R to its range, i.e.,

$$\underrightarrow{\operatorname{ran}}\, R = \{\, (\mathbf{s} \to \mathbf{t}) \to \mathbf{t} \mid \mathbf{s} \to \mathbf{t} \in R \,\}.$$

If the value read by an element in the domain of A_C was written inside the program fragment under consideration, then it was written by one of the domain elements of one of those write access relations A_P. In particular, it is an element of the image of the relation $\left(A_P^{-1} \circ A_C\right) \cap B$ for some P. Since we want to keep track of the array elements being accessed, we can extend the above computation to

$$D_{C,P}^{\text{mem}} = \left((\underrightarrow{\operatorname{ran}}\, A_P)^{-1} \circ (\underrightarrow{\operatorname{ran}}\, A_C) \right) \cap (B \times 1_{\mathcal{A}}), \tag{3}$$

where $D_{C,P}^{\text{mem}}$ refers to the "memory based" dependences of C on P and $\mathcal{A}$ is the union of all array domains. For example, if $A_P = \{\, \mathtt{N2}(k) \to \mathtt{n}() \mid 0 \le k \le 99 \,\}$ then $\underrightarrow{\operatorname{ran}}\, A_P = \{\, (\mathtt{N2}(k) \to \mathtt{n}()) \to \mathtt{n}() \mid 0 \le k \le 99 \,\}$. Combining this with $A_C = \{\, \mathtt{H}(k,i,j) \to \mathtt{n}() \mid 0 \le k,i,j \le 99 \,\}$, we obtain $D_{C,P}^{\text{mem}} = \{\, (\mathtt{H}(k,i,j) \to \mathtt{n}()) \to (\mathtt{N2}(k') \to \mathtt{n}()) \mid 0 \le k,k',i,j \le 99 \wedge k' < k \,\}$.

If the iteration domain of the statement containing C has dimension d, then we first consider elements of the D_C^{mem} relations such that the first d iterators in domain and range have the same value, then $d-1$ iterators, continuing until we consider the case where 0 iterators have the same value. Let ℓ denote this number of equal iterators. For each value of ℓ, we consider the potential sources P that share at least ℓ outer loops, compute the maximum image element of $D_{C,P}^{\text{mem}} \cap (E_=^\ell \times 1_{\mathcal{A}})$, with $E_=^\ell$ expressing that exactly ℓ outer iterators have the same value, adding constraints that ensure this maximal element is executed after any previously computed maximum at this level. When moving from ℓ to $\ell - 1$, we only consider those elements from the sink

access relation for which no source has been found so far. The main operation in this computation is then that of the partial lexicographical maximum of a relation M on a domain U, which returns a relation mapping elements u of U to the lexicographically greatest element associated to u by M, along with a set of elements in U that do not have any images in M. Let us define some more operations on sets and relations. Specifically, if S_1 and S_2 are sets and if R is a relation, then the universal relation from S_1 to S_2 is denoted

$$S_1 \to S_2 = \{\, \mathbf{s} \to \mathbf{t} \mid \mathbf{s} \in S_1 \wedge \mathbf{t} \in S_2 \,\},$$

while the domain and range of R are denoted

$$\operatorname{dom} R = \{\, \mathbf{s} \mid \mathbf{s} \to \mathbf{t} \in R \,\}.$$

and

$$\operatorname{ran} R = \{\, \mathbf{t} \mid \mathbf{s} \to \mathbf{t} \in R \,\}.$$

The lexicographical maximum of a relation R is then denoted $\operatorname{lexmax} R$ and is equal to

$$\{\, \mathbf{s} \to \mathbf{t} \mid \mathbf{s} \to \mathbf{t} \in R \wedge (\forall \mathbf{t}' : \mathbf{s} \to \mathbf{t}' \in R \Rightarrow \mathbf{t}' \preccurlyeq \mathbf{t}) \,\},$$

"$\preccurlyeq$" representing the lexicographical order, Finally, the partial lexicographical maximum of M on U is

$$\operatorname*{lexmax}_U M = (\operatorname{lexmax}(M \cap (U \to \operatorname{ran} M)), U \setminus \operatorname{dom} M) \tag{4}$$

The partial lexicographical maximum can be computed using parametric integer programming [11]. In the example, the read of `n` in Line 12 can only have 0 equal loop iterators with the write in Line 15. We therefore compute $\operatorname{lexmax}_U D^{\text{mem}}_{C,P}$ with U the (nested) sink access relation. The result consists of the (simplified) relation $\{\, (\mathtt{H}(k,i,j) \to \mathtt{n}()) \to (\mathtt{N2}(k-1) \to \mathtt{n}()) \mid k > 0 \,\}$ and the set $\{\, (\mathtt{H}(0,i,j) \to \mathtt{n}()) \,\}$. Sources for this latter part of the sink relation can be found in Line 6. The final result is shown in (2).

4. FILTERS

As explained in Section 2, if there are any dynamic conditions in the program, then the iteration domain may be an affine overapproximation of the set of executed iterations. To mark those iterations that are actually executed, we apply one or more *filters* to the iteration domain. These filters encode the dynamic conditions that determine whether an iteration in the iteration domain is actually executed. For example, as shown before, the iteration domain (1) is an overapproximation of the executed iterations of the statement in Line 14 of Listing 1. The filter on this iteration domain then expresses the dynamic conditions `i < n` and `j < m`. Each filter consists of a sequence of *filter access relations*, accessing variables that may be updated during the execution of the program, and a *filter value relation*, mapping statement iterations to the possible values of the dynamic variables for which the iteration is executed. For simplicity of exposition, we will assume that all the filter access relations are functions and that there is only one filter. The more general case is discussed in Appendix A. Non-affine conditions are treated as dynamic conditions.

More formally, let S be a statement and I_S its iteration domain. Furthermore, let the filter on S consist of a filter value relation V^S and n^S filter access relations F^S_i. We have $F^S_i \subseteq I_S \to (I_S \to \mathcal{A})$ and $V^S \subseteq I_S \to \mathbb{Z}^{n^S}$. That is, each filter access relation maps an iteration to an access of an array element and the filter value relation maps an iteration to a tuple of values. Moreover, each F^S_i is a function on the iteration domain. The application of the relation R to the set S is defined as

$$R(S) = \{\, \mathbf{u} \mid \exists \mathbf{t} : \mathbf{t} \to \mathbf{u} \in R \wedge \mathbf{t} \in S \,\}.$$

Furthermore, let

$$(a_j)_{j=1}^{n}$$

be the n-tuple with elements a_j and let $\mathcal{V}(\{\, \mathbf{j} \to \mathbf{b} \,\})$ be the value of $\mathbf{b}$ at $\mathbf{j}$. Iteration $\mathbf{k} \in I_S$ is then executed iff $\mathit{executed}^S(\mathbf{k})$ holds, with

$$\mathit{executed}^S(\mathbf{k}) = \left(\mathcal{V}\left(F^S_j(\mathbf{k})\right)\right)_{j=1}^{n^S} \in V^S(\mathbf{k}), \tag{5}$$

where $F^S_j(\mathbf{k})$ is short for $F^S_j(\{\, \mathbf{k} \,\})$. An access $\mathbf{k} \to \mathbf{a}$ from an iteration $\mathbf{k}$ is executed iff the iteration is executed, so that $\mathit{executed}^S((\mathbf{k} \to \mathbf{a})) = \mathit{executed}^S(\mathbf{k})$. Let $\underrightarrow{\operatorname{dom}} R$ map a nested copy of R to its domain, i.e.,

$$\underrightarrow{\operatorname{dom}}\, R = \{\, (\mathbf{s} \to \mathbf{t}) \to \mathbf{s} \mid \mathbf{s} \to \mathbf{t} \in R \,\},$$

then, initially, each filter access relation is a subset of the relation $\left(\underrightarrow{\operatorname{dom}}(\mathcal{I} \to \mathcal{A})\right)^{-1}$. That is, each iteration is mapped to an access that takes place at the same iteration.

We consider two types of filters, one for the general case and one for the special case of "locally static affine" conditions. A condition is considered to be locally static affine if it is an affine expression in variables that are definitely not modified between the point where they are evaluated and all the statements that are guarded by the condition. Such variables are called "locally static". In this case, the accesses to the locally static variables themselves are used as filter accesses. Otherwise, a new statement is created that evaluates the condition and writes the resulting boolean value (0 or 1) to a virtual array, as in [26, Section 4.3]. This virtual array is then used as a filter access in a condition that simply evaluates the element of the virtual array. The first type of filter is allowed in both `if`-conditions and loop conditions, while the second type is in the current context only allowed in `if`-conditions. In Section 7.1, we explain how to also handle the second type in loop conditions.

Consider, for example, the code in Listing 1. The condition `i < n` in Line 12 references the variable `n` which is not a parameter because it is assigned in Line 6 and in Line 15. However, the value of `n` is clearly not changed between the condition in Line 12 and the statement in Line 14 governed by this condition. Similarly, `m` is also locally static for the statement. The filter for statement `H` therefore has filter access relations

$$F^{\mathtt{H}}_1 = \{\, \mathtt{H}(k,i,j) \to (\mathtt{H}(k,i,j) \to \mathtt{n}()) \,\} \tag{6}$$

and

$$F^{\mathtt{H}}_2 = \{\, \mathtt{H}(k,i,j) \to (\mathtt{H}(k,i,j) \to \mathtt{m}()) \,\}. \tag{7}$$

The filter value relation $V^{\mathtt{H}}$ is

$$\{\, \mathtt{H}(k,i,j) \to (n,m) \mid 0 \le k \le 99 \wedge 0 \le i < n \wedge 0 \le j < m \,\}. \tag{8}$$

For convenience to the reader, the dimensions that correspond to the filters have been named after the arrays (in this case scalars) that are being accessed by the corresponding filter access relations. However, the reader should keep

```
1    state = 0;
2    while (1) {
3        sample = radioFrontend();
4        if (t(state)) {
5  D:        state = detect(sample);
6        } else {
7  C:        decode(sample, &state, &value0);
8            value1 = processSample0(value0);
9            processSample1(value1);
10       }
11   }
```

Listing 2: Code with dynamic conditions, adapted from [7, Figure 8.1]

in mind that these names are completely arbitrary. Note that the condition of the loop in Line 12 is allowed since it is locally static affine.

The code in Listing 2 has a generic dynamic condition in Line 4. `pet` introduces a separate virtual array, say $\mathtt{t}_0$, for storing the result of the condition and a separate statement, say $\mathtt{S}_0$, for computing this result. The access relation that writes to the filter is of the form

$$\{\, \mathtt{S}_0(i) \to \mathtt{t}_0(i) \,\}.$$

Note that `pet` introduces an implicit iterator (called i in this relation) with non-negative values [26, Section 3.3] for the loop `while (1)` in Line 2. The filter value relation of the statement in Line 5 is

$$V^{\mathtt{D}} = \{\, \mathtt{D}(i) \to (1) \mid i \geq 0 \,\}$$

with filter access relation

$$F^{\mathtt{D}} = \{\, \mathtt{D}(i) \to (\mathtt{D}(i) \to \mathtt{t}_0(i)) \,\}.$$

That is, the statement is executed for values of i greater than or equal to 0 such that $\mathtt{t}_0(i)$ at $\mathtt{D}(i)$ is equal to 1. (Recall that $\mathtt{t}_0(i)$ is a boolean variable that only attains values 0 and 1.) The filter value relation of the statement in Line 7 is

$$V^{\mathtt{C}} = \{\, \mathtt{C}(i) \to (0) \mid i \geq 0 \,\} \tag{9}$$

with filter access relation

$$F^{\mathtt{C}} = \{\, \mathtt{C}(i) \to (\mathtt{C}(i) \to \mathtt{t}_0(i)) \,\}. \tag{10}$$

Most of the analysis in Section 6 is based on filter values being equal or different in different iterations. We therefore need to be able to identify that filter values accessed in different iterations are actually the same. This information can be obtained by applying dataflow analysis on the arrays accessed by the filters. As explained below, the result of this analysis may or may not depend on any parameters as defined in Section 5. If any such parameters are involved, then we keep the original filter access relations. If, on the other hand, the resulting dependence relations do *not* depend on any such parameters, then the original filter access relations can be replaced by a composition with these dependence relations. For the code in Listing 1, the dependence relation for `n` in Line 12 is shown in (2). Applying this dependence relation to the filter access relation in (6) yields

$$\begin{aligned} F_1^{\mathtt{H}} = {} & \{\, \mathtt{H}(0,i,j) \to (\mathtt{N1}() \to \mathtt{n}()) \,\} \cup \\ & \{\, \mathtt{H}(k,i,j) \to (\mathtt{N2}(k-1) \to \mathtt{n}()) \mid k \geq 1 \,\}. \end{aligned} \tag{11}$$

For the filter access relation in (10), no dataflow analysis needs to be performed since we know each element of the virtual array $\mathtt{t}_0$ is written by exactly one iteration of the statement $\mathtt{S}_0$. The filter access relation can therefore be replaced by

$$F^{\mathtt{C}} = \{\, \mathtt{C}(i) \to (\mathtt{S}_0(i) \to \mathtt{t}_0(i)) \,\}. \tag{12}$$

In principle, the meaning of $\mathcal{V}$ depends on whether sources have been found for the filter array or not, i.e., whether we have been able to compose the original filter access relations with dependence relations or not. In particular, if sources have been found, then $\mathcal{V}(\{\, \mathbf{j} \to \mathbf{b} \,\})$ is the value of $\mathbf{b}$ *after* the write in iteration $\mathbf{j}$, while if sources have not been found, then $\mathcal{V}(\{\, \mathbf{j} \to \mathbf{b} \,\})$ is the value of $\mathbf{b}$ *before* the read in iteration $\mathbf{j}$. In practice, however, this difference is not important since filter accesses for which no sources have been found will never be matched to any other filter accesses.

5. PARAMETER REPRESENTATION

If the iteration domain of a potential source P is affected by any filters, then we cannot simply compute the lexicographical maximum in (4) since the maximal element in the range of M may not actually be executed, even if some of the other elements are. We will therefore introduce parameters that represent the "last executed" source iteration. By equating the iterators in the range of M to these parameters, the lexicographical maximization operation will simply return these parameters, which are guaranteed to represent an executed iteration.

The exact form and meaning of the parameters depends on whether we are considering the final result of the dataflow analysis or intermediate results. Let us first consider the final result. Let $D_{C,P}$ be the final dependence relation, the description of which may involve some parameters $\beta_C^Q(\mathbf{k})$ and $\boldsymbol{\lambda}_C^Q(\mathbf{k})$, where Q may be either P or some other potential source and $\mathbf{k}$ is an element of the sink access relation A_C. Note that in principle the parameters depend on the sink access $\mathbf{k}$, but as we will explain below, we do not need to make this dependence explicit in our representation. Let $D'_{C,P}$ be the result of projecting out all these parameters. The parameters then have the following meaning. The parameter $\beta_C^Q(\mathbf{k})$ is a boolean variable that expresses whether any of the elements in $D'_{C,P}(\mathbf{k})$ is executed. If $\beta_C^Q(\mathbf{k}) = 1$, then $\boldsymbol{\lambda}_C^Q(\mathbf{k})$ represents the last element in $D'_{C,P}(\mathbf{k})$ that is executed. (If $\beta_C^Q(\mathbf{k}) = 0$, then $\boldsymbol{\lambda}_C^Q(\mathbf{k})$ is undefined.) In other words, we have

$$\begin{aligned} &\beta_C^P(\mathbf{k}) = 0 \Rightarrow \forall \mathbf{j} \in D'_{C,P}(\mathbf{k}) : \neg \mathit{executed}^{S_P}(\mathbf{j}) \\ &\beta_C^P(\mathbf{k}) = 1 \Rightarrow \mathit{executed}^{S_P}(\boldsymbol{\lambda}_C^P(\mathbf{k})) \wedge \\ &\qquad\qquad \forall \mathbf{j} \in D'_{C,P}(\mathbf{k}) : \mathbf{j} \succ \boldsymbol{\lambda}_C^P(\mathbf{k}) \Rightarrow \neg \mathit{executed}^{S_P}(\mathbf{j}), \end{aligned} \tag{13}$$

with S_P the statement containing access P and *executed* as defined in (5). For example, let C by the access to `state` in the statement $\mathtt{S}_0$ evaluating the condition in Line 4 of Listing 2. Let P by the access to the same variable (`state`) from statement `C`. The dependence relation for the dependence of C on P is of the form $(\beta_C^P, \lambda_C^P) \to \{\, (\mathtt{S}_0(i) \to \mathtt{state}()) \to (\mathtt{C}(\lambda_C^P(i)) \to \mathtt{state}()) \mid \beta_C^P(i) = 1 \wedge i = \lambda_C^P(i) + 1 \geq 1 \,\}$. That is, there is only a dependence if access P (from statement `C`) was ever executed (before $\mathtt{S}_0(i)$) and if the last execution was in the previous iteration. Note that if the last execution

was in some earlier iteration, then C would not depend on the access in statement C, but on that in statement D. This result is obtained in Section 6.4.

During the computation, the resulting dependence relation is obviously not known, but we do know that it is a subset of $D^{\text{mem}}_{C,P}$ (3). The relation M in the current maximization problem (4) is also a subset of $D^{\text{mem}}_{C,P}$. We can therefore temporarily treat $\boldsymbol{\lambda}^P_C(\mathbf{k})$ as the last element of $D^{\text{mem}}_{C,P}(\mathbf{k})$ that is executed. Note that because we still assume static affine index expressions, the array element accessed by this last executed element of $D^{\text{mem}}_{C,P}(\mathbf{k})$ is necessarily the same as that accessed by $\mathbf{k}$. We can also exploit additional information to reduce the number of elements in the $\boldsymbol{\lambda}^P_C(\mathbf{k})$ vector that need to be explicitly represented. In particular, we know that the first ℓ iterators in the domain and range of M (with ℓ as in Section 3) are pairwise equal and that the same property holds for the previously considered maximization problems within the current dataflow problem, i.e., for the same C. This allows us to avoid introducing elements of the $\boldsymbol{\lambda}^P_C(\mathbf{k})$ vector until we really need them. In particular, we keep track in $\sigma^P_C(\mathbf{k})$ of the number of initial elements in the domain of $\mathbf{k}$ and in $\boldsymbol{\lambda}^P_C(\mathbf{k})$ that are (implicitly) equal to each other. Values of $\sigma^P_C(\mathbf{k})$ smaller than ℓ then mean that there is no element in $D^{\text{mem}}_{C,P}(\mathbf{k})$ that is executed and that shares the values of the first ℓ iterators with $\mathbf{k}$. As a special case, $\sigma^P_C(\mathbf{k}) < 0$ means that no element in $D^{\text{mem}}_{C,P}(\mathbf{k})$ is executed. Summarizing, (13) is replaced by

$$\begin{aligned}
&\sigma^P_C(\mathbf{k}) < \ell \Rightarrow \forall \mathbf{j} \in D''_{C,P}(\mathbf{k}) : \neg \mathit{executed}^{S_P}(\mathbf{j}) \\
&\sigma^P_C(\mathbf{k}) \geq \ell \Rightarrow \mathit{executed}^{S_P}(\boldsymbol{\lambda}^P_C(\mathbf{k})) \wedge \\
&\qquad\qquad \forall \mathbf{j} \in D''_{C,P}(\mathbf{k}) : \mathbf{j} \succ \boldsymbol{\lambda}^P_C(\mathbf{k}) \Rightarrow \neg \mathit{executed}^{S_P}(\mathbf{j}),
\end{aligned} \tag{14}$$

where $D''_{C,P} = D^{\text{mem}}_{C,P} \cap E^{\ell}_{\geq}$, with $E^{\ell}_{\geq}$ expressing that at least ℓ outer iterators have the same value.

After the dataflow computation has finished, we need to convert the intermediate representation (14) to the final representation (13). If $\ell^P_{\leq}$ is the smallest value of ℓ for which we had to apply parametrization for a given potential source P, then we do not need to introduce dimensions of $\boldsymbol{\lambda}^P_C$ before $\ell^P_{\leq}$. Instead, we need to make the equalities implied by σ^P explicit and set $\beta^P_C(\mathbf{k}) = 1$ when $\sigma^P \geq \ell^P_{\leq}$ and $\beta^P_C(\mathbf{k}) = 0$ when $\sigma^P < \ell^P_{\leq}$. The parameter σ^P can then be projected out. Besides dimensions before $\ell^P_{\leq}$, we also do not need to introduce dimensions of $\boldsymbol{\lambda}^P_C$ for which $D^{\text{mem}}_{C,P}(\mathbf{k})$ attains a single value (for any given value of $\mathbf{k}$) or that correspond to loops inside the innermost condition that is not static affine.

Since $\boldsymbol{\lambda}^P_C$ and σ^P_C depend on the sink iteration, it may appear that we would need to treat them as uninterpreted functions. During the entire computation, there is however no interaction between different sink iterations. That is, any of the intermediate relations during the computation only refers to a single sink iteration and the parameters $\boldsymbol{\lambda}^P_C$ and σ^P_C (if present) always refer to that single sink iteration. This means that we can keep the relation between $\boldsymbol{\lambda}^P_C$ and σ^P_C on one hand and $\mathbf{k}$ on the other hand entirely implicit and simply treat $\boldsymbol{\lambda}^P_C$ and σ^P_C as parameters. The intended meaning of those parameters is then the value of the corresponding functions at the particular value of $\mathbf{k}$ involved in the given access or dependence relation. This reasoning is essentially the same as that of [1] for showing that his $\boldsymbol{\alpha}$ vectors can be treated as parameters.

Algorithm 1: Parametric partial lexicographical maximum

```
(type, S1, S2) = parametrization(M, U)
if type = Input then
    M := intersect_range(M, S1)
    U := intersect(U, S2)
else if type = Empty then
    M := empty(M)
end
(R,E) = partial_lexmax(M, U)
if type = Output then
    R := intersect_range(R, S1)
end
```

6. PARAMETRIZATION

Recall that during dataflow analysis (Section 3), we frequently compute a partial lexicographical maximum of the form (4). The inputs to this operation are based on the static affine iteration domains and so we may need to introduce parameters representing the last executed source iteration (Section 5) to take into account the filters (Section 4) on the iteration domains. In particular, we replace a call "(R,E) = partial_lexmax(M, U)", corresponding to (4), by the pseudocode in Algorithm 1. The different types of parametrization in this code are explained in Section 6.1, while the determination of which of these parametrizations should be applied is explained in Section 6.3. The sets S_1 and S_2 used during the parametrization are constructed in Section 6.2 and Section 6.4.

6.1 Types of Parametrization

We are presented with a maximization problem of the form (4), with U a set of iterations of a sink C and M a relation between sink iterations and iterations of a potential source P and we want to decide if we need to introduce parameters. In principle, the result is that either parametrization is required (type = Input) or it is not required (type = None). However, we also consider a couple of other special cases (type = Empty and type = Output). In particular, we may find that it is impossible for any of the source iterations to execute given that the corresponding sink is executed. In such cases we want to avoid introducing parameters since there is no dependence and we therefore do not need any extra parameters to represent the dependence. However, we cannot indicate to the dataflow analysis that no parametrization is required (type = None) as then it would treat the source iterations as definitely being executed. Instead, we communicate to the dataflow analysis that M should be replaced by an empty relation (type = Empty). Another special case occurs when we are able to determine that no parametrization is required, but that this detection depends on information available about *other* potential sources. For reasons beyond the scope of the present paper, the construction of process networks can in this case be facilitated by introducing parameters anyway, but only on the *result* of the maximization problem rather than on the input of the maximization problem (type = Output). A schematic overview of the determination of the type of parametrization to apply is shown in Algorithm 2. The process is explained in more detail in Section 6.3.

6.2 Application

If parametrization is required, then we need to equate the source iteration to the parameters λ^P representing the last executed iteration of the potential source P (14). That is, the range of M (or, more precisely, the domain of the nested relation in this range) is intersected with

$$(\boldsymbol{\lambda}_I^P, \sigma^P) \to \{\, \mathtt{S}_P(\mathbf{j}) \mid \mathbf{j}_I = \boldsymbol{\lambda}_I^P \wedge \sigma^P \geq \ell \,\}, \qquad (15)$$

where I selects the elements of $\boldsymbol{\lambda}^P$ that need to be explicitly represented as explained in Section 5 and ℓ the number of equal outer iterators in the domain and range of M as in Section 3. These constraints determine the set S_1 in Algorithm 1. For example, let C be the read of `A` in Line 14 of Listing 1 and let P be the write in Line 11. The corresponding statements share at most one loop iterator. When $\ell = 1$, then we have that M is equal to

$$\{\, (\mathtt{H}(k,i,j) \to \mathtt{a}(i+1,j)) \to (\mathtt{A}(k,j,i+1) \to \mathtt{a}(i+1,j)) \,\}, \qquad (16)$$

where the outer dimensions k are equal because $\ell = 1$ and the inner dimensions i' and j' are equal to j and $i+1$ because the same array element is accessed. As we will see in Section 6.3, no parametrization is required for this problem, but let us for illustrative purposes assume that we do want to apply parametrization. Dimension 0 of $\boldsymbol{\lambda}$ does not need to be introduced (yet) because $0 < \ell$. Since $D_{C,P}^{\text{mem}}$ is equal to

$$\{\, (\mathtt{H}(k,i,j) \to \mathtt{a}(i+1,j)) \\ \to (\mathtt{A}(k',j,i+1) \to \mathtt{a}(i+1,j)) \mid k' \leq k \,\},$$

the remaining dimensions of $\boldsymbol{\lambda}$ do not need to be introduced either because they are fully determined by $D_{C,P}^{\text{mem}}$ (and therefore also by $D'_{C,P} \subseteq D_{C,P}^{\text{mem}}$). In effect, we would only need to introduce σ^P with constraint $\sigma^P \geq 1$.

The parametrization of the source domain expresses that the source is (essentially) the last of the potential source iterations associated to the sink through $D_{C,P}^{\text{mem}}$. However, it only does so through the introduction of parameters that implicitly depend on the sink iteration $\mathbf{k}$. The parametrization of the sink takes care of expressing that this last iteration, if it exists, actually belongs to $D_{C,P}^{\text{mem}}(\mathbf{k})$. In particular, we first split the set of sink iterations U into two parts, one with associated potential source iterations and one without, i.e.,

$$U_1 = U \cap (\operatorname{dom} M) \qquad \text{and} \qquad U_2 = U \setminus (\operatorname{dom} M). \qquad (17)$$

The parametrization is only applied to U_1 and expresses that either none of the potential source iterations in $D_{C,P}^{\text{mem}}(\mathbf{k})$ that share the first ℓ iterators is executed or that there is such an executed potential source iteration and that it belongs to $D_{C,P}^{\text{mem}}(\mathbf{k})$. That is, the sink C is intersected with

$$(\boldsymbol{\lambda}^P, \sigma^P) \to \left\{\, \mathbf{k} \mid \sigma^P < \ell \vee \left(\boldsymbol{\lambda}^P \in D_{C,P}^{\text{mem}}(\mathbf{k}) \wedge \sigma^P \geq \ell\right) \right\}. \qquad (18)$$

These constraints, together with those of Section 6.4 below, determine the set S_2 in Algorithm 1. In practice, we only introduce the same set of dimensions of the $\boldsymbol{\lambda}^P$ vector as introduced by (15). In particular, the second disjunct is obtained by applying the parametrization of (15) to the range of $D_{C,P}^{\text{mem}}$ and computing the domain of the result.

6.3 Detection

Let us now explain in more detail the steps in the determination of which parametrization to apply as sketched in Algorithm 2. If M and/or U are empty or if P is not affected by any filter, then no parametrization is required (Line 2). Otherwise, we compute the possible values for the filter elements at the potential source, given that the sink is executed. Clearly, we can only do this if the sink is affected by a filter and if source and sink have some filter accesses in common. If the computed possible values are disjoint from the filter value relation on the source, then no source iteration is executed when the sink is executed and we indicate that M should be replaced by an empty relation (Line 6). Otherwise, we check if U references any parameters that were introduced by previous calls to the parametrization. If so, we use the constraints on those parameters and the filters of the associated (other) potential sources to derive extra information about the filters at the sink (Line 8). This derivation is explained in Section 6.3.2. The resulting information is then propagated again to potential source P and a new relation of possible values is computed. If this relation is disjoint from the filter value relation on the source, then we again indicate that M should be replaced by an empty relation (Line 10). Otherwise, if the computed relation is a subset of the filter value relation on the source, then we know the source is always executed and we indicate that either no parametrization is required (Line 13) or that parametrization is only required on the output of the maximization (Line 16), depending on whether the relation computed based on only information on the sink was already a subset of the filter value relation. Finally, if none of these cases apply, then parametrization is required (on the input of the maximization problem, Line 18).

Algorithm 2: Type of parametrization

```
1  if M is empty, U is empty or there are no filters on the source then
2  |  return None
3  end
4  F := filters on the sink
5  if filters on the source contradict F then
6  |  return Empty
7  end
8  F' := update(F, filters on other sources)
9  if filters on the source contradict F' then
10 |  return Empty
11 end
12 if filters on the source imply F then
13 |  return None
14 end
15 if filters on the source imply F' then
16 |  return Output
17 end
18 return Input
```

6.3.1 Filter Values implied by the Sink

Let us illustrate the derivation of filter value constraints on the potential source from those on the sink based on an example. The general case is explained in Section A.2. In particular, let us reconsider the example from Section 6.2. Let M_1 be the result of projecting out the array elements from M, i.e.,

$$M_1 = \{\, \mathtt{H}(k,i,j) \to \mathtt{A}(k,j,i+1) \mid 0 \leq i \leq 98 \,\},$$

where we omit the constraints on j and k for brevity. The filters on statement `A` are similar to those on `H` in (6) and (7). The first of these was updated in (11) to take into account the filter source. The second can be updated to

$$F_2^{\mathtt{H}} = \{\, \mathtt{H}(k,i,j) \to (\mathtt{M}(k) \to \mathtt{m}()) \mid 0 \le i \le 98 \,\}.$$

For statement `A`, we have, say,

$$F_1^{\mathtt{A}} = \{\, \mathtt{A}(k,i,j) \to (\mathtt{M}(k) \to \mathtt{m}()) \mid 0 \le j \le 99 \,\}.$$

We want to check whether the fact that the sink is executed tells us anything about the values of these filter variables at the source. Note that if the sink is not executed, then it does not need any values and so there is no need to compute dataflow dependences for sink iterations that are not executed.

The first step is to check whether any of the source filter arrays are also accessed by the corresponding sink iterations. To do so, we pull back the filter access relations over M_1, resulting in

$$F_1^{\mathtt{A}} \circ M_1 = \{\, \mathtt{H}(k,i,j) \to (\mathtt{M}(k) \to \mathtt{m}()) \mid 0 \le i \le 98 \,\},$$

where the constraints $0 \le i \le 98$ derive from $0 \le i+1 \le 99$ and the domain constraints of M_1. Since this relation is a subset of $F_2^{\mathtt{H}}$ i.e., $\forall \mathbf{j} \in M_1(\mathbf{k}) : F_1^{\mathtt{A}}(\mathbf{j}) \subseteq F_2^{\mathtt{H}}(\mathbf{k})$, and similarly for $F_2^{\mathtt{A}} \circ M_1 \subseteq F_1^{\mathtt{H}}$, we know that (5) also holds for the source filter accesses for all $\mathbf{j} \in M_1(\mathbf{k})$, i.e.,

$$\left(\mathcal{V}\left(F_i^{\mathtt{A}}(\mathbf{j})\right)\right)_{i=1}^{2} \in V_1(\mathbf{k})$$

with V_1 derived from $V^{\mathtt{H}}$ in (8) by changing the order of the range dimensions to match the matching of the filters, i.e.,

$$\{\, \mathtt{H}(k,i,j) \to (m,n) \mid 0 \le k \le 99 \wedge 0 \le i < n \wedge 0 \le j < m \,\}.$$

Note that in this particular example, we have $F_1^{\mathtt{A}} \circ M_1 = F_2^{\mathtt{H}}$ and $F_2^{\mathtt{A}} \circ M_1 = F_1^{\mathtt{H}}$ but equality is not required in the general case. There is also no need for every filter access relation on the source to match a filter access relation on the sink and vice versa. Besides reordering dimensions of $V^{\mathtt{H}}$, we may in the general case also have to project out dimensions and/or introduce unconstrained dimensions.

Precomposing V_1 with M_1^{-1}, we obtain a mapping from source iterations to filter values that allow one or more corresponding sink iterations to be executed. In the example, we obtain

$$\{\mathtt{A}(k,i,j) \to (m,n) \mid 0 \le k \le 99 \wedge 0 \le j-1 < n \wedge 0 \le i < m\}.$$

Since this relation is a subset of the filter value relation on `A`, i.e.,

$$\{\, \mathtt{A}(k,i,j) \to (m,n) \mid 0 \le k \le 99 \wedge 0 \le i < m \wedge 0 \le j \le n \,\},$$

we know that for each sink iteration in the domain of M' that is executed, the corresponding source iterations are also executed and therefore no parametrization is required.

6.3.2 Filter Values implied by Other Sources

The derivation of extra information from other potential sources is again illustrated based on an example. The general case is explained in Section A.3. In particular, let us consider the determination of the sources for the access to `state` in Line 4 of Listing 2. We first consider the write P in Line 7 as a potential source for $\ell = 0$. Parametrization is required and in particular (18) specializes to

$$(\lambda_0^P, \sigma^P) \to \left\{\, \mathtt{S}_0(i) \mid \sigma^P < 0 \vee \left(0 \le \lambda_0^P < i \wedge \sigma^P \ge 0\right) \,\right\}.$$

The dataflow analysis then continues looking for sources from the write Q in Line 5. Let us now consider the case where $\sigma^P < 0$. Since statement $\mathtt{S}_0$ is not affected by any filters, we need to turn to the other potential sources, in particular P, for any constraints that could help us determine whether parametrization is required.

We first construct a relation N mapping sink iterations to iterations of P that have definitely *not* been executed (according to (14) and given $\sigma^P < 0$), i.e.,

$$N = \{\, \mathtt{S}_0(i) \to \mathtt{C}(i') \mid 0 \le i' < i \,\}.$$

This relation can be constructed by subtracting iterations that may have executed from $D_{C,P}^{\text{mem}}$ (with the array elements projected out). In the example, there are no iterations that may have executed (since $\sigma^P < 0$) and so in this case $N = D_{C,P}^{\text{mem}}$. From (5), we know that the values of the corresponding filter elements (12) do *not* satisfy the filter access relation (9), i.e., for all $\mathtt{S}_0(i) \to \mathtt{C}(i') \in N$ we have

$$\mathcal{V}(F^{\mathtt{C}}(\mathtt{C}(i'))) \notin V^{\mathtt{C}}(\mathtt{C}(i')).$$

In other words,

$$\mathcal{V}(F^{\mathtt{C}}(\mathtt{C}(i'))) \in V_1(\mathtt{C}(i')),$$

with

$$V_1 = \{\, \mathtt{C}(i) \to (1) \mid i \ge 0 \,\}.$$

Note that `t_0` is a boolean variable, so if its value is not 0, it must be 1. Combining N and $F^{\mathtt{C}}$ (12) into a single relation, we obtain

$$N_1 = \{\, (\mathtt{S}_0(i) \to (\mathtt{S}_0(i') \to \mathtt{t}_0(i')) \to \mathtt{C}(i') \mid 0 \le i' < i \,\}.$$

Let $R_1 \rtimes R_2$ be the domain product of two relations, mapping nested pairs of domain elements from R_1 and R_2 to their shared images, i.e.,

$$R_1 \rtimes R_2 = \{\, (\mathbf{s} \to \mathbf{t}) \to \mathbf{u} \mid \mathbf{s} \to \mathbf{u} \in R_1 \wedge \mathbf{t} \to \mathbf{u} \in R_2 \,\}.$$

In general, we then have $N_1 = N \rtimes \left(F^{\mathtt{C}}\right)^{-1}$. The relation N_1 maps a pair of sink iteration and filter access to source iterations that have definitely not been executed and that perform the filter access, with value in V_1. The same filter element may be accessed by multiple source iterations, each of them imposing the V_1 constraints. We therefore compute the intersection of the V_1 images over all associated source iterations. In the example, only a single source iteration is associated to a given filter element and we obtain

$$V_2 = \{\, (\mathtt{S}_0(i) \to (\mathtt{S}_0(i') \to \mathtt{t}_0(i')) \to (1) \mid 0 \le i' < i \,\}.$$

Projecting out the filter access, we obtain the filter value relation

$$V_3 = \{\, \mathtt{S}_0(i) \to (1) \mid 1 \le i \,\},$$

which is valid for any element of $F^{\mathtt{C}}(N(\mathtt{S}_0(i)))$, with $i \ge 1$. This information can then be propagated to the potential source Q using the technique of Section 6.3.1, from which it can be concluded that Q is always executed (in the current case where $\sigma^P < 0$) and that therefore no parametrization is required. Note that the combined filter access relation $F^{\mathtt{C}} \circ N$ is no longer a function, but, as explained in Section A.2, the computations are also valid for multi-valued access relations. We just need to be careful about the domains of the access relations.

6.4 Additional Constraints

If we determine that we need to apply parametrization, then we may in some cases wish to impose additional constraints on the introduced parameters beyond those of Section 6.2. In particular, we may find that not all potential source iterations are executed (otherwise no parametrization would be required), but that *some* potential source iterations are definitely executed. If so, we add constraints that impose that there is a last execution ($\sigma^P \geq \ell$) and that this last execution is no earlier than any of the definitely executed iterations.

Additionally, we check for conflicts between the filters associated to the source of the newly added parameters and those of previously added parameters, introducing extra constraints that avoid the conflicts. As usual, we only show an example, while the details are explained in Section A.4. Continuing from the example in Section 6.3.2, let us consider the case where we are looking for an instance of the write Q in Line 5 of Listing 2 that is executed after the last execution of the write P in Line 7. Parametrization is required in this case, but the sink already contains parameters that refer to P, so we look for possible conflicts between the parameters of P and Q.

Based on dataflow analysis on the filter arrays, e.g., (12), we know that identical iterations of `D` and `C` access the same filter element, written in the same iteration. These filter elements therefore need to have the same value. Let us try to derive conflicts from iterations of both statements that are not executed. To ease the notation, we will only consider the case $\sigma^P, \sigma^Q \geq 0$ and omit these variables and constraints. The iterations in $D^{\text{mem}}_{C,P}$ that are not executed are given by

$$\lambda_0^P \to \{\, \mathtt{S}_0(i) \to \mathtt{C}(i') \mid 0 \leq \lambda_0^P < i' < i \,\}$$

with filter value $\{\, \mathtt{C}(i') \to (1) \,\}$ and similarly for Q. Pairing up the non-executed iterations of `C` and `D` and restricting them to the pairs that should have the same value, we obtain

$$(\lambda_0^P, \lambda_0^Q) \to \{\, \mathtt{S}_0(i) \to (\mathtt{C}(i') \to \mathtt{D}(i')) \mid 0 \leq \lambda_0^P, \lambda_0^Q < i' < i \,\}.$$

Considering now the pairs of iterations from the two statements that map to values that allow the iterations to not be executed and such that these values are the same, we find that there are no such pairs. Subtracting this set of pairs of iterations that have the same value from the range of the relation above, we obtain a mapping from sink iterations to pairs of source iterations that should have the same values for their filters but in fact do not. In this example, an empty set is subtracted so that the relation remains the same. The domain of this relation, i.e.,

$$(\lambda_0^P, \lambda_0^Q) \to \{\, \mathtt{S}_0(i) \mid 0 \leq \lambda_0^P, \lambda_0^Q \leq i - 2 \,\}.$$

represents conflicting values of the parameters. In particular, it is impossible for the last executed iterations of P and Q to both be before the previous iteration of the loop. These impossible cases are removed from the sink parametrization S_2.

7. EXTENSIONS

In this section, we briefly discuss two extensions, dynamic loop bounds, which are supported by our dataflow analysis implementation, and dynamic index expressions, which are currently not supported.

```
1     for (int x1 = 0; x1 < n; ++x1) {
2  S1:    s = f();
3         for (int x2 = 0; P(x1, x2); ++x2) {
4  S2:        s = g(s);
5         }
6  R:     h(s);
7     }
```

Listing 3: C version of example E1 from [1, Section 3.2.2]

7.1 Dynamic Loop Conditions

In principle, dynamic loop conditions can be handled by introducing a filter that represents the last executed iteration of the loop. Consider, for example, the loop in Line 3 of Listing 3. The loop condition depends on some unknown function `P` applied to the loop iterator and is therefore not (locally) static affine. We could introduce a virtual scalar, say `m`, that represents the final iteration of the loop and that implicitly depends on `x1`. The filter value relation would then be of the form $(n) \to \{\, \mathtt{S2}(x_1, x_2) \to (m) \mid 0 \leq x_1 < n \wedge 0 \leq x_2 \leq m \,\}$. Unfortunately, the resulting dataflow dependence relations would not be practically useful for the construction of process networks since this last executed iteration is not known in advance.

Instead, we record the result of the dynamic loop condition in a virtual array and make the body of the loop depend on the value of the current *and all previous* iterations of the loop being 1. That is, the statement on Line 4 has filter value relation

$$V^{\mathtt{S2}} = (n) \to \{\, \mathtt{S2}(x_1, x_2) \to (1) \mid 0 \leq x_1 < n \wedge x_2 \geq 0 \,\}$$

with filter access relation

$$F^{\mathtt{S2}} = n \to \{\, \mathtt{S2}(x_1, x_2) \to \mathrm{t}_0(x_1, a) \mid 0 \leq a \leq x_2 \,\}.$$

Note that this filter access relation is not a function, but a multi-valued filter access relation and therefore requires the treatment of Appendix A. The statement evaluating the condition is made to depend on all previous iterations. Note that we currently only handle dynamic loop conditions for the purpose of dataflow analysis and not for the actual construction of process networks.

7.2 Dynamic Index Expressions

Our implementation currently does not support dynamic index expressions. We would have to allow the access relations to be approximate as well and the parameters would change to mean that the iteration is executed *and* that the element being accessed is the same as that accessed by the sink. This change in meaning would limit the conclusions we could draw from the new parameters.

8. EXPERIMENTS

Our approach has been implemented in the `da` and `pn` tools of the `isa` prototype tool suite (`git://repo.or.cz/isa.git`). The `da` tool performs dataflow analysis and supports dynamic loop conditions, while the `pn` tool additionally constructs a process network and currently does not support dynamic loop conditions. Table 1 shows the results of a preliminary experimental comparison of our `da` tool against that of [6], which is an implementation of the

input	da			fadatool			fadatool -s		
	time	p	d	time	p	l	time	p	l
Lst 1	0.01s	0	5	0.01s	6	6	0.01s	6	6
Lst 2	0.01s	4	9	0.01s	6	16	incorrect		
fuzz4	0.06s	3	9	0.02s	4	9	0.01s	0	9
for1	0.02s	2	3	0.01s	4	46	0.02s	2	3
for2	0.03s	2	3	0.09s	12	5k	0.04s	4	3
for3	0.04s	2	3	42s	24	1M	0.08s	6	3
for4	0.06s	2	3				0.16s	8	3
for5	0.08s	2	3				0.25s	10	3
for6	0.14s	2	3				0.42s	12	3
c_if1	0.02s	2	3	0.01s	2	4	0.01s	2	4
c_if2	0.02s	2	10	0.02s	4	52	0.02s	2	8
c_if3	0.03s	2	22	0.03	6	723	0.36s	3	16
c_if4	0.02s	2	10	0.17s	8	9k	1m	4	28
whil1	0.01s	0	4	0.00s	1	4	0.01s	0	4
whil2	0.03s	3	4	0.01s	5	6	incorrect		
if_var	0.03s	4	3	0.01s	2	8	0.01s	2	4
if_wh	0.04s	2	14	0.01s	5	58	0.02s	4	58
if2	0.02s	2	2	0.46s	12	29k	0.04s	4	2

Table 1: Experimental Results

approach that most closely resembles our own. In particular, we use version isa-0.11-319-gead5e27 of `da` and version fda6009 of `fadatool`. We use `fadatool` both with and without the `-s` option, since the results can be wildly different. It should be noted that both tested tools are prototypes. We should therefore be careful about drawing conclusions from these results, especially since `fadatool -s` sometimes produces incorrect results. Since the inputs in the table are relatively simple, correctness was determined through visual inspection.

For each tool and for each test case, we report the time taken by the analysis, the number of parameters introduced and the number of disjuncts in the dependence relations (for `da`) or the number of leaves in the quasts (for `fadatool`). Note that as explained in Section 9 below, the internal representation of dependence relations inside `fadatool` includes an additional parameter similar to our β. Since these internal parameters are not explicitly printed in the output of the tool, they are not included in the parameter count for `fadatool`. By contrast, the β parameters *are* included in the parameter count for `da`. The first two inputs are those of Listing 1 and Listing 2, modified to pass through the default `fadatool` parser. In particular, the parser does not support multiple writes in a single statement. It was therefore difficult to convert our more extensive test cases. The remaining cases (with some of the names abbreviated) come from the `fadalib` distribution. We omit those test cases that contain index expressions that are not static affine since we cannot handle them. Of note is that `fadatool` fails to recognize that no parameters need to be introduced on Listing 1 and that without the `-s` option, it quickly runs out of control on the `for` test cases. The `cascade_if` results may be somewhat misleading since `pet` recognizes that the filter variables used in the `if` conditions are the same and that some of the inner tests are implied by the outer test, greatly simplifying the input to `da`.

9. RELATED WORK

Despite its name, FADA is to the best of our knowledge the only alternative approach that allows for an exact (but run-time dependent) dataflow analysis in the presence of dynamic and/or non-affine conditions or index expressions. The main differences are that FADA introduces different parameters (with a different meaning), that they are only analyzed *after* all parameters have been introduced and that the analysis is performed using resolution on general first order logic formulas. A new vector of parameters $\boldsymbol{\alpha}$ is introduced for every maximization problem similar to (4), meaning that potentially many more parameters are introduced. The absence of a solution (similar to our $\beta = 0$) is represented as $\perp$ on paper and is reported to be represented by a scalar variable similar to our β inside the implementation of [6]. Note that it is sometimes suggested [8, 22] to use a value outside the iteration domain, but this may not always be easy to determine and is impossible for 0D iteration domains. Our dataflow analysis on filter arrays is a special case of the iterative approach of [1].

Other approaches to dataflow analysis [16, 21] produce approximate results in the presence of constructs that are not static affine. The approach of [16] in particular propagates values to discover static affine constraints in constructs that do not at first appear to be static affine. The authors of [9] propose an algorithm for computing reaching definitions for arrays that applies to both structured and unstructured programs. However, they only focus on how to collect constraints and do not explain how to solve them. Instead, they introduce uninterpreted functions and rely on `Omega` [19] for solving formulas containing such uninterpreted functions. Unfortunately, the support in `Omega` for uninterpreted functions is very limited and cannot handle the constraints they collect. The `iegenlib` library [23] has more extensive support for uninterpreted functions, but does not support a difference operation and can therefore not be used to perform value-based dependence analysis.

10. CONCLUSIONS AND FUTURE WORK

We have presented a novel approach for exact array dataflow analysis in the presence of constructs that are not static affine. Dynamic behavior in the input program is represented using filters. An analysis of these filters determines if the dependences are also run-time dependent. If so, parameters are introduced to represent this run-time dependence, where we are careful to introduce as few parameters as possible. This is made possible by a judicious definition of these parameters. We plan on working on a closer integration with `pet` so that we can perform the dependence analysis incrementally, allowing us to locally treat some variables in the input as symbolic constants (as advocated by [15]) and more easily detect some cases (such as that of Listing 1) where no parameters need to be introduced. Note that the techniques developed in this paper would still be useful on more complicated inputs.

11. ACKNOWLEDGMENTS

This work was partially funded by a gift received by LIACS from Intel Corporation and by the European Commission through the FP7 project CARP id. 287767.

12. REFERENCES

[1] D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints*. PhD thesis, PRiSM -

Laboratoire de recherche en informatique, Feb. 1998.

[2] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. *Int. J. Parallel Program.*, 28(3):213–243, 2000.

[3] D. Barthou, J.-F. Collard, and P. Feautrier. Applications of fuzzy array dataflow analysis. In *Euro-Par Conference*, volume 1123 of *Lect. Notes in Computer Science*, pages 424–427, Lyon, Aug. 1996. Springer-Verlag.

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

[5] D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. In *Euro-Par Conference*, volume 2400 of *Lect. Notes in Computer Science*, pages 309–313, Paderborn, Aug. 2002. Springer-Verlag.

[6] M. Belaoucha, D. Barthou, A. Eliche, and S.-A.-A. Touati. FADAlib: an open source C++ library for fuzzy array dataflow analysis. In *Intl. Workshop on Practical Aspects of High-Level Parallel Programming*, volume 1, pages 2075—2084, Amsterdam, The Netherlands, May 2010.

[7] T. Bijlsma. *Automatic parallelization of nested loop programs for non-manifest real-time stream processing applications*. PhD thesis, University of Twente, 2011.

[8] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proceedings of 5th ACM SIGPLAN Symp. on Principles and practice of Parallel Programming*, July 1995.

[9] J.-F. Collard and M. Griebl. A precise fixpoint reaching definition analysis for arrays. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 286–302, London, UK, 2000. Springer-Verlag.

[10] P. Feautrier. Array expansion. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 429–441. ACM Press, 1988.

[11] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

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

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

[14] P. Feautrier. *The Data Parallel Programming Model*, volume 1132 of *LNCS*, chapter Automatic Parallelization in the Polytope Model, pages 79–100. Springer-Verlag, 1996.

[15] V. Maslov. Lazy array data-flow dependence analysis. In H.-J. Boehm, B. Lang, and D. M. Yellin, editors, *POPL*, pages 311–325. ACM Press, 1994.

[16] V. Maslov. Enhancing array dataflow dependence analysis with on-demand global value propagation. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 265–269, New York, NY, USA, 1995. ACM.

[17] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia MP-SoC design. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 574–579, New York, NY, USA, 2008. ACM.

[18] S. Pellegrini, T. Hoefler, and T. Fahringer. Exact dependence analysis for increased communication overlap. *Recent Advances in the Message Passing Interface*, pages 89–99, 2012.

[19] W. Pugh and D. Wonnacott. Going beyond integer programming with the omega test to eliminate false data dependences. Technical Report Technical Report CS-TR-3191, Department of Computer Science, University of Maryland, College Park, Maryland, Dec. 1992. An earlier version of this paper appeared at the ACM SIGPLAN '92 Conference on PLDI.

[20] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 546–566. Springer-Verlag, 1994.

[21] W. Pugh and D. Wonnacott. Nonlinear array dependence analysis. Technical report, College Park, MD, USA, 1994.

[22] T. Stefanov. *Converting Weakly Dynamic Programs to Equivalent Process Network Specifications*. PhD thesis, Leiden University, Leiden, The Netherlands, Sept. 2004.

[23] M. M. Strout, G. George, and C. Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Sept. 2012.

[24] A. Turjan, B. Kienhuis, and E. Deprettere. Translating affine nested-loop programs to process networks. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 220–229, New York, NY, USA, 2004. ACM Press.

[25] S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010.

[26] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, Jan. 2012.

[27] S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In *Computer Aided Verification 21*, pages 599–613. Springer, June 2009.

[28] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: A tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems, special issue on Embedded Digital Signal Processing Systems*, 2007, 2007.

APPENDIX

A. MULTI-VALUED FILTER ACCESS RELATIONS

In this appendix, we describe how to extend the representation and manipulation of filters to handle multi-valued filter access relations.

A.1 Filters

In Section 4, we assumed that all filter access relations access a single element in each iteration. Here, we describe how to extend the definitions to handle multi-valued filter access relations. The trickiest part is not so much handling filter access relations that access more than one data element, but filter access relations that may access zero data elements for some iterations. We therefore allow several filters on the same iteration domain, with disjoint domains, and impose that in each filter the domain of the filter value relation is a subset of the domains of all the filter access relations. In particular, each statement S has μ^S filters on its iteration domain, with $\mu^S \geq 0$. Each of the filters $\mathcal{F}_i$, with $1 \leq i \leq \mu^S$, is represented by a sequence of filter access relations $F^S_{i,j}$ with $1 \leq j \leq n^S_i$ and n^S_i the number of filter access relations, and a filter value relation V^S_i. As in Section 4, we have $F^S_{i,j} \subseteq I_S \to (I_S \to \mathcal{A})$ and $V^S_i \subseteq I_S \to \mathbb{Z}^{n^S_i}$. Furthermore, we impose

$$\operatorname{dom} F^S_{i,j} \supseteq \operatorname{dom} V^S_i,$$

for all $1 \leq i \leq \mu^S$ and $1 \leq j \leq n^S_i$, to ensure that all the filter access relations are total on the domains of the corresponding filter value relations, and

$$\operatorname{dom} V^L_{i_1} \cap \operatorname{dom} V^L_{i_2} = \emptyset$$

for all $1 \leq i_1, i_2 \leq \mu^S$ such that $i_1 \neq i_2$, to ensure that the domains of the filter value relations are disjoint.

The definition of $\mathit{executed}^S(\mathbf{k})$ (5) is then replaced by

$$\forall(\mathbf{f}_1, \ldots, \mathbf{f}_{n^S_i}) \in \prod_{j=1}^{n^S_i} F^S_{i,j}(\mathbf{k}) : (\mathcal{V}(\mathbf{f}_j))_{j=1}^{n^S_i} \in V^S_i(\mathbf{k}) \quad (19)$$

if $\mathbf{k} \in \operatorname{dom} V^S_i$ and is simply true on those parts of the iteration domain I_S that do not intersect any of the $\operatorname{dom} V^S_i$. That is, an element of the iteration domain I_S is only executed if the tuples of values of all the tuples of accessed filter array elements satisfy the active filter value relation.

A.2 Filter Values implied by the Sink

In this section, we describe the derivation illustrated in Section 6.3.1. We will assume that only a single sink filter and a single potential source filter apply to the domain and range of M. That is, we will assume that $\operatorname{dom} M_1 \subseteq \operatorname{dom} V^C$ and $\operatorname{ran} M_1 \subseteq \operatorname{dom} V^P$, with M_1 the result of projecting out the array elements from M as in Section 6.3.1. In general, M needs to be split up according to the filters.

Let the sink filter consist of n filter access relations F_i and the potential source filter of m filter access relations G_j. Since we are going to compare the filters, we replace the filter access relations by their sources, as explained at the end of Section 4. In particular, $F_i \subseteq I_C \to (I \to A)$ and $G_i \subseteq I_P \to (I \to A)$. We construct a relation H between their possible values, initialized as

$$H = \mathbb{Z}^n \to \mathbb{Z}^m.$$

Let B_j represents the bounds on the values of the array elements accessed by G_j as specified by the user through a `pragma value_bounds` [26, Section 2], or $\mathbb{Z}$ if no such bounds have been specified. Let B be the Cartesian product of these bounds, i.e.,

$$B = \prod_{j=1}^{m} B_j. \quad (20)$$

The relation can then be refined to

$$H = \mathbb{Z}^n \to B.$$

In the next step, we iterate over the potential source filter access relations and check if we have any information about them in the sink filter value. In particular, we check if any of the sink filter access relations accesses "the same" value(s). If so, we equate the corresponding dimensions in the mapping between filter values. To check if "the same" value(s) are accessed, we pull back the filter access relation over M_1. This results in a relation between sink domain iterations and filter sources such that there is at least one potential source corresponding to the sink domain iteration that accesses that filter source. If this relation is a subset of the filter access relation at the sink, then we know that everything we know about the values of the filter accesses at the sink also applies to the values of the corresponding filter accesses at the corresponding potential source iterations. Note that there may be more than one potential source iteration associated to a given sink iteration and that each of these potential source iteration may access a different element from the filter array. The above process ensures that source and sink values are only equated if *all* of these filter array elements are covered by the sink filter access relation.

In particular, for any $\mathbf{k} \in \operatorname{dom} V^C$ that is executed, we know from (19),

$$\forall(\mathbf{f}_1, \ldots, \mathbf{f}_n) \in \prod_{j=1}^{n} F^C_j(\mathbf{k}) : (\mathcal{V}(\mathbf{f}_j))_{j=1}^{n} \in V^C(\mathbf{k}).$$

If we have $G^P_j \circ M_1 \subseteq F^C_i$, i.e., $\forall \mathbf{t} \in M_1(\mathbf{k}) : G^P_j(\mathbf{t}) \subseteq F^C_i(\mathbf{k})$, then we know

$$\forall \mathbf{t} \in M_1(\mathbf{k}) : \forall(\mathbf{f}_1, \ldots, \mathbf{f}_m) \in \prod_{j=1}^{n} G^P_j(\mathbf{t}) : (\mathcal{V}(\mathbf{f}_j))_{j=1}^{n} \in V_1(\mathbf{k}), \quad (21)$$

with $V_1 = H \circ V^C$. The relation H takes care of projecting out those dimensions in V^C for which we were unable to find a corresponding filter access G^P_j, reordering those for which we did find a correspondence and introducing dimensions for those filter accesses G^P_j for which we were unable to find a corresponding filter access F^C_i. The relation $V_1 \subseteq I_C \to \mathbb{Z}^m$ represents what we know about the filter values at the potential sources associated to a given sink domain iteration, given that the sink domain iteration is executed. A further composition with (the inverse of) $M_1 \subseteq I_C \to I_P$, yields a subset of $I_P \to \mathbb{Z}^m$. This relation maps potential source iterations to filter values that allow for one or more corresponding sink iterations to be executed. In particular, if there is an element $\mathbf{k} \in M_1^{-1}(\mathbf{t})$ that is executed, then $(\mathcal{V}(\mathbf{f}_j))_{j=1}^{m}$ is an element of $V_1(\mathbf{k})$. Given that there is such a $\mathbf{k}$, the tuple $(\mathcal{V}(\mathbf{f}_j))_{j=1}^{m}$ is therefore an element of the union of $V_1(\mathbf{k}')$ over all $\mathbf{k}' \in M_1^{-1}(\mathbf{t})$. That is, $(\mathcal{V}(\mathbf{f}_j))_{j=1}^{m}$ is an element of $(V_1 \circ M_1^{-1})(\mathbf{t})$. In other words, for values of the filters outside the relation $V_1 \circ M_1^{-1}$, no corresponding (according to

M_1) sink domain iterations are executed. Note that we cannot take the intersection of $V_1(\mathbf{k}')$ over all $\mathbf{k}'$ because (21) only applies to those $\mathbf{k}$ that are executed and we only know that at least one of the $\mathbf{k}' \in M_1^{-1}(\mathbf{t})$ is executed, not that all of them are executed.

A.3 Filter Values implied by Other Sources

In this section, we describe the derivation illustrated in Section 6.3.2. In particular, we describe the "update" function in Line 8 of Algorithm 2. In particular, during the computation of the partial lexicographical maximum of M on U (4), the set U may already involve parameters that refer to the last iterations of (other) potential sources. We describe how we can exploit the constraints on these parameters to derive extra information about the filter values at the sink. The procedure of Section A.2 then needs to be applied to the updated sink filter to actually derive information about the filter values at the original potential source.

Specifically, we will derive information from the fact that some potential source iterations have *not* been executed. The conditions on the filter values at the potential source that allow the sink to be executed, but not the potential source are mapped back to the sink, first by taking the intersection over all potential source iterations associated to a given sink iteration and filter element and then by taking the union over all accessed filter elements. We take the intersection over the potential source iterations since we know that all those iterations are not executed and so all of the corresponding constraints apply. We take the union over the accessed filter elements, since we need to obtain constraints that are valid for all of those elements. As in Section A.2, we assume that only a single sink filter and a single potential source filter applies to the domain and range of M, i.e., that $\operatorname{dom} M_1 \subseteq \operatorname{dom} V^C$ and $\operatorname{ran} M_1 \subseteq \operatorname{dom} V^P$. As before, M_1 is the result of projecting out the array elements from M. Let us similarly define a U_1 that is the result of projecting out the access array element from U. In the remainder of this section, we will take $D^{\text{mem}}_{C,Q}$ to have the array elements projected out, i.e., $D^{\text{mem}}_{C,Q} \subseteq I_C \to I_Q$.

Let us now look at the construction in more detail. We start with the construction of a mapping from sink iterations to iterations of some access Q that have *not* been executed (according to information in U). We first apply the parametrization of (15) to the iteration domain of Q and construct a relation $N_0 \subseteq U_1 \to I'_Q$ (with I'_Q the result of the parametrization) that is universal, except that the first ℓ dimensions in domain and range are equated. Note that some of the extra parameters in I'_Q also appear in U_1 (otherwise we would not consider this potential source). This means that the relation N_0 relates sink iterations to potential source iterations that include the last potential source iteration executed before the sink iteration. That is, $\{\, \mathbf{k} \to \lambda^Q_C(\mathbf{k}) \mid \mathbf{k} \in U_1 \wedge \sigma^Q_C(\mathbf{k}) \geq \ell \,\}$ is a subset of N_0. In particular, according to (14), the last element of $D^{\text{mem}}_{C,Q}(\mathbf{k})$, with $\mathbf{k} \in U_1$, that shares the first ℓ iterators and where the filter values satisfy the filtered iteration domain of Q is included in this relation. The relation may also contain additional elements since we may not have introduced a parameter for each dimension as explained in Section 5. Projecting out all parameters introduced in (15) (for any iteration domain), we obtain a relation between sink iterations and potential source iterations that include the last potential source iteration for any value of the other parameters. Further combining this relation with a relation mapping potential source iterations to earlier potential source iterations $\{\, \mathrm{S}_Q(\mathbf{i}) \to \mathrm{S}_Q(\mathbf{i}') \mid \mathbf{i} \succcurlyeq \mathbf{i}' \,\}$ results in a relation between sink iterations and potential source iterations that may have executed. In particular, the potential source iterations that are *not* related to a given sink iteration (and that share the first ℓ iterators) are *definitely not* executed. To obtain this relation between sink iterations and potential source iterations that are definitely not executed we subtract the relation computed above from the corresponding memory based dependences $D^{\text{mem}}_{C,Q}$ (with the first ℓ dimensions equated). Let us call the resulting relation $N \subseteq I_C \to I_Q$. Due to the construction, we have for each $\mathbf{k} \to \mathbf{j} \in N$ that $\neg \mathit{executed}^{S_Q}(\mathbf{j})$.

If N is empty, then we cannot use it to derive any information and the computation stops. Otherwise, the first step in our derivation is to apply the computation of Section A.2 to N. This assumes that $\operatorname{dom} N \subseteq \operatorname{dom} V^C$ and $\operatorname{ran} N \subseteq \operatorname{dom} V^Q$, for some filter of Q. The first condition can be enforced by intersecting N with $\operatorname{dom} M \to I_Q$, since we are only interested in sink iterations that belong to $\operatorname{dom} M$ in any case. If we cannot find a filter on Q such that the second condition holds, then the computation stops. During the course of the computation, we will remove filter access relations G^Q_j that are not single-valued. We therefore check if the filter on Q has any single-valued filter access relations. If not, the computation stops.

Applying the computation in Section A.2 (with M replaced by N and the potential source P by the other potential source Q), we obtain a relation $V_0 \subseteq I_Q \to \mathbb{Z}^{m'}$ such that for each $\mathbf{k} \to \mathbf{j} \in N$, we have

$$\forall (\mathbf{f}_1, \ldots, \mathbf{f}_{m'}) \in \prod_{i=1}^{m'} G^Q_i(\mathbf{j}) : (\mathcal{V}(\mathbf{f}_i))_{i=1}^{m'} \in V_0(\mathbf{j}).$$

For the same $\mathbf{j}$, since $\neg \mathit{executed}^{S_Q}(\mathbf{j})$, we also know

$$\exists (\mathbf{f}_1, \ldots, \mathbf{f}_{m'}) \in \prod_{i=1}^{m'} G^Q_i(\mathbf{j}) : (\mathcal{V}(\mathbf{f}_i))_{i=1}^{m'} \notin V^Q(\mathbf{j}).$$

Combining these results, we have

$$\exists (\mathbf{f}_1, \ldots, \mathbf{f}_{m'}) \in \prod_{i=1}^{m'} G^Q_i(\mathbf{j}) : (\mathcal{V}(\mathbf{f}_i))_{i=1}^{m'} \in \left(V_0 \setminus V^Q\right)(\mathbf{j}).$$

Unfortunately, knowing that there is *some* sequence of filter elements $\mathbf{f}_i$ will not allow us to derive any further information. We will therefore assume that that all filter access relations G^Q_i are single-valued. Effectively, this means that we remove those filter access relations that are not single-valued and project out the corresponding dimensions from $V_0 \setminus V^Q$. Let V_1 be the result of this projection. That is, for each $\mathbf{k} \to \mathbf{j} \in N$,

$$\left(\mathcal{V}(G^Q_i(\mathbf{j}))\right)_{i=1}^{m''} \in V_1(\mathbf{j}).$$

Let G be the range product of the single-valued filter access relations G^Q_i.

At this point we have a relation $N \subseteq I_C \to I_Q$ mapping sink iterations to corresponding non-executed potential source iterations, a relation $G \subseteq I_Q \to (I \to A)^{m''}$, mapping potential source iterations to (single) filter elements, and a relation $V_1 \subseteq I_Q \to \mathbb{Z}^{m''}$, mapping potential source

iterations to corresponding filter values that do not allow the potential source iteration to be executed, but do allow the corresponding sink iteration to be executed. We first combine N and G into a single relation

$$N_1 = N \rtimes G^{-1}$$

The result is a subset of $(I_C \to (I \to A)^{m''}) \to I_Q$. We have, for each $(\mathbf{k} \to (\mathbf{f})_i) \to \mathbf{j} \in N_1$,

$$(\mathcal{V}(\mathbf{f}_i))_{i=1}^{m''} \in V_1(\mathbf{j}).$$

Note that because they represent (part of) a filter, we have $\operatorname{dom} G \supseteq \operatorname{dom} V^Q$. Combined with our assumption that $\operatorname{ran} N \subseteq \operatorname{dom} V^Q$, this ensures that we do not remove any elements from the range of N. We now connect the pairs of sink iterations and filter elements to the possible values of those filter elements at the corresponding potential source iterations by computing

$$V_2 : \operatorname{dom} N_1 \to \mathbb{Z}^{m''} : V_2(\mathbf{k} \to \mathbf{f}) = \bigcap_{\mathbf{j} \in N_1(\mathbf{k} \to \mathbf{f})} V_1(\mathbf{j}). \quad (22)$$

That is, we consider the potential source iterations $\mathbf{j}$ that have definitely not been executed before a certain sink iteration $\mathbf{k}$ and compute the intersection of the possible values of the filter elements $\mathbf{f}$ over all those definitely not executed source iterations. We have, for each $\mathbf{k} \to (f)_i \in \operatorname{dom} V_2$,

$$(\mathcal{V}(\mathbf{f}_i))_{i=1}^{m''} \in V_2(\mathbf{k} \to (\mathbf{f})_i).$$

Note that different potential source iterations associated to the same sink iteration may access different elements from the filter arrays. We therefore need to be careful to only combine constraints on values associated to the same elements. If N_1 is single-valued, the intersection in (22) is computed over a single element and so we can simply compute V_2 as

$$V_2 = V_1 \circ N_1.$$

Otherwise, it is computed as

$$V_2 = N_1 \sqsubseteq V_1^{-1},$$

with $\sqsubseteq$ the non-empty subset operation on two relations, which constructs a relation between the domain elements of the two relations such that the image of the first domain element is a subset of the image of the second domain element. That is, $R_1 \sqsubseteq R_2$ is equal to

$$\{\, \mathbf{s} \to \mathbf{t} \mid \mathbf{s} \in \operatorname{dom} R_1 \wedge \mathbf{t} \in \operatorname{dom} R_2 \wedge R_1(\mathbf{s}) \subseteq R_2(\mathbf{t}) \,\}$$

The constraint $R_1(\mathbf{s}) \subseteq R_2(\mathbf{t})$ can be expressed as

$$\forall \mathbf{u} : \mathbf{s} \to \mathbf{u} \in R_1 \Rightarrow \mathbf{t} \to \mathbf{u} \in R_2$$

or

$$\neg \exists \mathbf{u} : \mathbf{s} \to \mathbf{u} \in R_1 \wedge \mathbf{t} \to \mathbf{u} \notin R_2$$

where $\mathbf{u}$ may be restricted to $\operatorname{ran} R_1$. The operation can therefore be computed as

$$R_1 \sqsubseteq R_2 = R_2^{-1} \circ R_1 \setminus \left(((\operatorname{dom} R_2 \to \operatorname{ran} R_1) \setminus R_2)^{-1} \circ R_1 \right).$$

Finally, we project out filter elements and compute

$$V_3 : I_C \to \mathbb{Z}^{m''} : V_3(\mathbf{k}) = \bigcup_{\mathbf{f} \in (\operatorname{dom} V_2)(\mathbf{k})} V_2(\mathbf{k} \to \mathbf{f}).$$

The resulting relation contains all values of all filter elements read by any non-executed iteration associated to a certain sink iteration. That is, for every $\mathbf{k} \in \operatorname{dom} V_3 = \operatorname{dom} N$,

$$\forall (\mathbf{f}_1, \ldots, \mathbf{f}_{m''}) \in \prod_{i=1}^{m''} G_i^Q(N(\mathbf{k}))) : (\mathcal{V}(\mathbf{f}_i))_{i=1}^{m''} \in V_3(\mathbf{k}).$$

V_3 can be computed as

$$V_3 = V_2 \circ (\underrightarrow{\operatorname{dom}}(\mathcal{W}^{-1}(\operatorname{dom} V_2)))^{-1},$$

with $\mathcal{W}^{-1} S$ extracting the nested relation from the set S, i.e.,

$$\mathcal{W}^{-1} S = \{\, \mathbf{s} \to \mathbf{t} \mid (\mathbf{s} \to \mathbf{t}) \in S \,\}.$$

The relation V_3 may only apply to a subset of the domain of M_1. Since we do not have any information about elements outside $\operatorname{dom} V_3 = \operatorname{dom} N$, we extend V_3 to the entire domain of M_1 as

$$V_4 = V_3 \cup \left((\operatorname{dom} M_1 \setminus \operatorname{dom} N) \to \mathbb{Z}^{m''} \right).$$

We now want want to intersect V^C with V_4, but as in Section A.2 we first need to align the filter access relations, by constructing a relation H mapping the filter values of V_4 to those of V^C. In this case, however, we also allow extra filter access relations to be added, both to the original set of filter access relations of the sink and to the filter access relations associated to V_4. We do this to be able to collect as much information as possible at the sink. In particular, some of the sources may involve the same filter access relations even if these filter access relations do not originally appear among those of the sink and we still want to combine the information from different sources at the sink.

More specifically, we look for filter access relations F_i^C that are identical to some $G_j^Q \circ N$. If we cannot find such an F_i^C, we add $G_j^Q \circ N$ to the sink filter access relations (adjusting V^C). In both cases, we express the correspondence in H. Additionally, we look for filter access relations F_i^C that form a (strict) subset of some $G_j^Q \circ N$. If so, we add F_i^C to the filter access relations associated to V_4, adjusting V_4 by duplicating the constraints on the corresponding dimension to the new dimension that corresponds to the extra filter access relation. Again, we express the correspondence in H. At the end, we apply H to V_4 and intersect V^C with the result.

A.4 Avoid Inconsistencies

As explained in Section 6.4, some values of the parameters expressing the last iteration of potential source P, introduced in Section 6.2 and constrained by (18), may not be consistent with the fact that the sink C is executed or with the values of parameters expressing the last iteration of other potential sources introduced before. In this section, we describe how we can remove some of these inconsistencies. Note that leaving in these inconsistencies does not lead to incorrect results, but only to less accurate results. We therefore do not need to remove every possible inconsistency, but instead try to remove those that we can easily discover.

Let us start with inconsistencies that arise from the fact that two potential sources, the current one (P) and one that was considered before (Q), are executed. In particular, assume that iteration $\mathbf{i}$ of P is executed and iteration $\mathbf{j}$ of Q

as well. According to (19), we then have

$$\forall(\mathbf{f}_1,\ldots,\mathbf{f}_m) \in \prod_{j=1}^{m} F_j^P(\mathbf{i}) : (\mathcal{V}(\mathbf{f}_j))_{j=1}^{m} \in V^P(\mathbf{i}) \qquad (23)$$

for some filter of P and

$$\forall(\mathbf{f}_1,\ldots,\mathbf{f}_{m'}) \in \prod_{j=1}^{m'} F_1^Q(\mathbf{j}) : (\mathcal{V}(\mathbf{f}_j))_{j=1}^{m'} \in V^Q(\mathbf{j}) \qquad (24)$$

for some filter of Q. If we can find a pair of subsequences of filter access relations that access some sequence of filter array elements in common, then the corresponding dimensions of $V^P(\mathbf{i})$ and $V^Q(\mathbf{j})$ need to have some value in common. Let $K \subseteq I_P \times I_Q$ contain those pairs of iteration that access the same filter array elements and let $H \subseteq \mathbb{Z}^m \times \mathbb{Z}^{m'}$ express the correspondence of values. That is, H is expressed as one or more equalities equating pairs of dimensions, one from $V^P(\mathbf{i})$ and one from $V^Q(\mathbf{j})$. The relation

$$T = \mathcal{W}^{-1}\left(\left(V^P \times V^Q\right)^{-1}(\mathcal{W}H)\right), \qquad (25)$$

where

$$\mathcal{W}R = \{\,(\mathbf{s} \to \mathbf{t}) \mid \mathbf{s} \to \mathbf{t} \in R\,\},$$

then contains those pairs of iterations that allow for a common value. Removing those from K, we obtain the relation

$$K' = K \setminus T$$

of pairs of iterations that *should* allow for a common value, but in fact do not. To map these inconsistencies to the parameters we construct a relation D_1 from $D_{C,P}^{\text{mem}}$ (with array elements projected out). In particular, we intersect the domain of $D_{C,P}^{\text{mem}}$ with the parametrization of (15) and equate the first ℓ dimensions of domain and range. We similarly construct a relation D_2 from $D_{C,Q}^{\text{mem}}$. The inconsistent values of the parameters are then obtained as

$$(D_1 \rtimes D_2)\,(\mathcal{W}K') \qquad (26)$$

and the result is removed from the sink parametrization.

The above procedure may be repeated for any appropriate pair of K and H. In our implementation, we currently only construct a single such pair in a greedy way. In particular, we start out with a universal K and H and consider pairs of filter access relations $F_i^P \subseteq I_P \to (I \to A)$ and $F_j^Q \subseteq I_Q \to (I \to A)$ that access the same array. For each such pair, we compute

$$K_{i,j} = \left(F_j^Q\right)^{-1} \circ F_i^P \subseteq I_P \to I_Q \qquad (27)$$

and check if $K_{i,j}$ has a non-empty intersection with the current value of K. If so, we replace K with this intersection and adjust H accordingly. Otherwise, we skip this pair of filter access relations.

We have only considered inconsistencies based on pairs of potential sources that *are* executed. It is also possible to derive inconsistencies based on one or both of the potential sources *not* being executed. Let us first consider the case where P is not executed and Q is executed. In this case, (23) is replaced by

$$\exists(\mathbf{f}_1,\ldots,\mathbf{f}_m) \in \prod_{j=1}^{m} F_j^P(\mathbf{i}) : (\mathcal{V}(\mathbf{f}_j))_{j=1}^{m} \notin V^P(\mathbf{i}).$$

Whereas in the case of (23) and (24) a conflict occurs as soon as there is *any* sequence of accessed elements for which no matching value can be found, in this case we only know something about *some* sequence of accessed elements at P and we can therefore only arrive at a conflict if *all* of the accessed elements are also accessed by Q. In particular, (27) needs to be replaced by

$$K_{i,j} = F_i^{L_P} \sqsubseteq F_j^{L_Q} \subseteq I_{L_P} \to I_{L_Q}.$$

Additionally, V^P in (25) should be replaced by

$$(I_{L_P} \to B) \setminus V^{L_P}$$

with B the bounds on the possible values on the array elements, as computed in (20), while D_1 in (26) should refer to iterations that are *not* executed. That is, rather than intersecting $D_{C,P}^{\text{mem}}$ with the parametrization of (15), we first map this parametrization to lexicographically later elements and take the union with the set

$$(\sigma^P) \to \{\,\mathrm{S}_{S_P}(\mathbf{j}) \mid \sigma^P < \ell\,\}.$$

This second part accounts for the fact that when $\sigma^P < \ell$, none of the iterations that share the first ℓ iterators have executed.

The case where P is executed and Q is not executed is handled similarly. For the case where both P and Q are not executed, we can only draw any conclusion for those iterations that access a single filter element. That is, $K_{i,j}$ of (27) is replaced by

$$\{\,\mathbf{i} \to \mathbf{j} \in (\operatorname{dom} F_i^P) \to (\operatorname{dom} F_j^Q) \mid \forall \mathbf{f}_1, \mathbf{f}_2 \in F_i^P(\mathbf{i}), \mathbf{f}_3, \mathbf{f}_4 \in F_j^Q(\mathbf{j}) : \mathbf{f}_1 = \mathbf{f}_2 = \mathbf{f}_3 = \mathbf{f}_4\,\}.$$

This relation can be computed by removing

$$\operatorname{dom}\left(\left(F_i^P \ltimes F_i^P\right) \setminus (I_P \to 1_{I \to A})\right)$$

from the domain of $\left(F_j^Q\right)^{-1} \circ F_i^P$ and

$$\operatorname{dom}\left(\left(F_j^Q \ltimes F_j^Q\right) \setminus (I_Q \to 1_{I \to A})\right)$$

from its range, with

$$R_1 \ltimes R_2 = \{\,\mathbf{s} \to (\mathbf{t} \to \mathbf{u}) \mid \mathbf{s} \to \mathbf{t} \in R_1 \wedge \mathbf{s} \to \mathbf{u} \in R_2\,\}.$$

That is, we consider pairs of image elements associated to the same domain element of F_i^P or F_j^Q and remove pairs of identical image elements. That only leaves pairs of different image elements and the domain of the relation represents those domain elements that have multiple image elements associated to them.

Inconsistencies between the potential source P and the sink C are removed in a similar way, except that C is always executed so that we only need to consider two cases, one where P is executed and one where P is not executed. Furthermore, the relation D_2 in (26) is replaced by the identity relation on the iteration domain of the sink, i.e., 1_{I_C}.