Polyhedral Process Networks

Sven Verdoolaege

Abstract Reference implementations of signal processing applications are often written in a sequential language that does not reveal the available parallelism in the application. However, if an application satisfies some constraints then a parallel specification can be derived automatically. In particular, if the application can be represented in the polyhedral model, then a *polyhedral process network* can be constructed from the application. After introducing the required polyhedral tools, this chapter details the construction of the processes and the communication channels in such a network. Special attention is given to various properties of the communication channels including their buffer sizes.

1 Introduction

Signal processing applications are prime candidates for a parallel implementation. As we have seen in previous chapters, there are several parallel models of computations that can be used for specifying such applications. However, many programmers are unfamiliar with these parallel models of computation. Writing parallel specifications can therefore be a difficult, time consuming and error prone process. For this reason, many application developers still prefer to specify an application as a sequential program, even though such a specification may not be suitable for a direct mapping onto a parallel multiprocessor platform.

In this chapter, we present a technique for automatically extracting a parallel specification from a sequential program, provided these sequential programs satisfy some conditions. In particular, the control flow of these programs needs to be static, meaning that the control flow should not depend on the signals being processed. Furthermore, all loop bounds, conditions and array index expressions need to be

Sven Verdoolaege
Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Leuven, Belgium, e-mail: sven@cs.kuleuven.be

such that they can be expressed using affine constraints. All functions called should be pure. In particular, they should not change the values of loop iterators or arrays other than their output arguments. These requirements ensure that we can represent all relevant information about the program using mathematical objects called polyhedra. The resulting parallel model, a variation on Kahn process networks [8], can then also be described using such polyhedra, whence the name *polyhedral process networks*.

The concept of a polyhedral process network was developed in the context of the Compaan project [11, 16]. The exposition in this chapter closely follows that of [19].

2 Overview

This section presents a high-level overview of the process of extracting a process network from a sequential program. The extracted process network represents the task-level parallelism that is available in the program. The input program is assumed to consist of a sequence of nested loops performing various "operations". These operations may be calls to functions that can be arbitrarily complicated. The operations are performed on data that has been computed in some iteration of another operation or in a previous iteration of the same operation. The output process network consists of a set of processes, each encapsulating all iterations of a given operation, and communication channels connecting the processes and representing the dataflow. The processes in the network can be executed independently of each other, as long as data is available from the channels from which the process reads and as long as buffer space is available in the channels to which the process writes. That is, the communication primitives implement blocking reads and blocking writes.

```
1  for (i = 0; i < K; i++)
2    for (j = 0; j < N; j++)
3      a[i][j] = ReadImage();
4  for (i = 1; i < K-1; i++)
5    for (j = 1; j < N-1; j++) {
6      Sbl[i][j] = Sobel(a[i-1][j-1], a[i][j-1], a[i+1][j-1],
7                        a[i-1][  j], a[i][  j], a[i+1][  j],
8                        a[i-1][j+1], a[i][j+1], a[i+1][j+1]);
9      WriteImage(Sbl[i][j]);
10   }
```

Fig. 1 Source code of a Sobel edge detection program

Example 1. As a simple example, consider the code for performing Sobel edge detection in Figure 1. The first loop of this program reads the input image, while the second loop performs the actual edge detection and writes out the output image. A

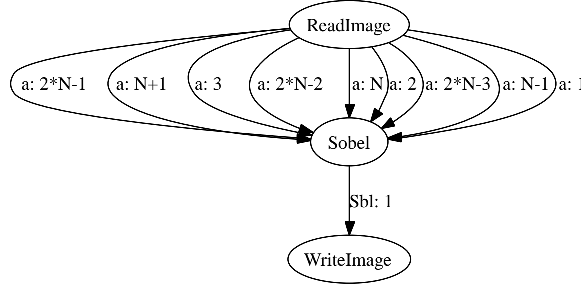

Fig. 2 A process network corresponding to the sequential program in Figure 1

process network that corresponds to this program is shown in Figure 2. There are three processes in the network, each corresponding to one of the three "operations" performed by the program, i.e., reading, edge detection and writing. Data flows from the reading process to the edge detection process and from the edge detection process to the writing process, resulting in the communication channels shown in the figure. The annotations on the edges will be explained later.

The extraction of a polyhedral process network consists of several steps summarized below and explained in more detail in the following sections:

- In a first step, a model is extracted from the input program on which all further analysis will be performed. In particular, the program is represented by a *polyhedral model*. This model allows for an efficient analysis, but imposes some restrictions on the input programs. The polyhedral model is explained in Section 3, while some basic components for the analysis of polyhedral models are explained in Section 4.
- Dataflow analysis is performed to determine which processes communicate with which other processes and how, i.e., to determine the communication channels. For example, the results of the call to `ReadImage` in Line 3 of Figure 1 are stored in the `a` array, which is read by the call to `Sobel` in Line 6. Dataflow analysis therefore results in one or more communication channels from the reading process to the edge detection process. Dataflow analysis is explained in Section 5.
- In the next step, the type of each communication channel is determined. For example, the channel may be a FIFO, in which case the processes connected to the channel simply need to write to and read from the channel, or it may not be a FIFO, in which case additional processing will be required. The classification of channels is discussed in Section 6.
- The communication channels may need to buffer some data to ensure a deadlock-free execution of the network. Especially for a hardware implementation of process networks, it is important to know how large these buffers may need to grow. The computation of buffer sizes is the subject of Section 8.
- The number of processes in the network may exceed the number of processing elements available. Some processes may therefore need to be merged. This

merging requires the construction of a combined schedule, which is the subject of Section 7. Depending on the kind of dataflow analysis that was performed in constructing the network, some of this analysis may need to be updated or redone based on the merging decisions.
- Finally, code needs to be written out for each of the processes in the network. This code needs to execute all iterations of the single or multiple (in case of merging) operations and needs to read from and write to the appropriate communication channels at the appropriate times. The main difficulty in this step is writing out code to scan overlapping polyhedral domains, which is discussed in Section 4.5.

The buffer size computation itself (Section 8) consists of several substeps. First a global schedule is computed (Section 7), assigning a global time point to each iteration of each process. This global schedule is only used during the buffer size computation and not during the actual execution. For each channel and each time point, the number of tokens in the channel at that time point is then computed (Section 4.3). Finally, for each channel, an upper bound is computed for the maximal number of tokens in the channel throughout the whole execution (Section 4.4).

3 Polyhedral Concepts

The key to an efficient transformation from sequential code to a process network is the polyhedral model used to represent the sequential program and the resulting process network. This section defines both models and related concepts.

3.1 Polyhedral Sets and Relations

Each statement inside a loop nest is executed many times when the sequential program is run. Each of these executions can be represented by the values of the iterators in the loops enclosing the statement. This sequence of iterator values is called the *iteration vector* associated to a given execution of the statement. The set of all such iteration vectors is called the *iteration domain* of the statement. Assuming that each iterator is an integer that is incremented by one in each iteration of the loop, this iteration domain can be represented very succinctly by simply collecting the lower and upper bounds of each of the enclosing loops.

Example 2. The iteration domain associated to the `ReadImage` statement in Line 3 of Figure 1 is

$$D_1 = \{\, (i, j) \in \mathbb{Z}^2 \mid 0 \le i < K \wedge 0 \le j < N \,\}. \tag{1}$$

The extraction of a process network requires several manipulations of iteration domains and related sets. To ensure that these manipulations can be performed efficiently or even performed at all, we need to impose some restrictions on how these

sets are represented. In particular, we require that the sets are described by integer *affine* inequalities and equalities over integer variables. An affine inequality is an inequality of the form $a_1x_1 + a_2x_2 + \ldots + a_dx_d + c \geq 0$, i.e., it expresses that some degree-1 polynomial over the variables is greater than or equal to zero. When dealing with several such inequalities in general, it will be convenient to represent them using a matrix notation $A\mathbf{x} + \mathbf{c} \geq \mathbf{0}$.

Sets of *rational* values described by affine inequalities have been the subject of extensive research and are called *polyhedra*.

Definition 1 (Rational Polyhedron). A *rational polyhedron* P is a subspace of $\mathbb{Q}^d$ bounded by a finite number of hyperplanes.

$$P = \{\, \mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} \geq \mathbf{c} \,\}, \tag{2}$$

with $A \in \mathbb{Z}^{m \times d}$ and $\mathbf{c} \in \mathbb{Z}^m$.

The sequential code from which we want to extract a process network may contain parameters such as the number of rows K and the number of columns N in Figure 1. In such cases, we do not want to extract a distinct process network for each value of the parameters, but instead a single parametric process network that is valid for all values of the parameters. We therefore also need the concept of a *parametric polyhedron*.

Definition 2 (Parametric Rational Polyhedron). A *parametric rational polyhedron* $P(\mathbf{s})$ is a family of rational polyhedra in $\mathbb{Q}^d$ parametrized by parameters $\mathbf{s} \in \mathbb{Q}^n$.

$$P : \mathbb{Q}^n \to 2^{\mathbb{Q}^d} : \mathbf{s} \mapsto P(\mathbf{s}) = \{\, \mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} + B\mathbf{s} \geq \mathbf{c} \,\}, \tag{3}$$

with $A \in \mathbb{Z}^{m \times d}$, $B \in \mathbb{Z}^{m \times n}$, $\mathbf{c} \in \mathbb{Z}^m$ and $2^{\mathbb{Q}^d}$ the power set of $\mathbb{Q}^d$, i.e., the set of all subsets of $\mathbb{Q}^d$. The *parameter domain* $D = \operatorname{pdom} P$ of a parametric polyhedron $P : \mathbb{Q}^n \to 2^{\mathbb{Q}^d}$ is a (non-parametric) polyhedron $D \subseteq \mathbb{Q}^n$ containing all parameter values $\mathbf{s}$ for which $P(\mathbf{s})$ is non-empty,

$$\operatorname{pdom} P = \{\, \mathbf{s} \in \mathbb{Q}^n \mid P(\mathbf{s}) \neq \emptyset \,\}.$$

Bounded polyhedra are called *polytopes*. In case of parametric polytopes, this means that each polyhedron in the family is a polytope, i.e., that $P(\mathbf{s})$ is a polytope for each value of the parameters $\mathbf{s}$.

Besides iterators and parameters, we may also need additional variables to accurately describe an iteration domain. These variables are not used to identify a given iteration, but rather to restrict the possible values of the iterators. This means that we do not care about the values of these variables, but rather that *some* integer value exists for these variables that satisfies the constraints. These variables may therefore be existentially quantified. The need for such variables arises especially when loop bounds or guards contain modulos or integer divisions. Expressions that contain such constructs but that can still be expressed using affine constraints are called *quasi-affine*.

```
1 for (i = 0; i < N; i++)
2   if (i % 3 == 0)
3     a[i] = f1();
4   else
5     a[i] = f2();
```

Fig. 3 Source code of a loop nest with iteration domains requiring existentially quantified variables to represent

Example 3. Consider the program in Figure 3. The modulo constraint "i % 3 == 0" is not in itself an affine constraint, but it can be represented as an affine constraint $i = 3\alpha$ by introducing an extra integer variable $\alpha \in \mathbb{Z}$. The iteration domain of the statement in Line 3 can therefore be represented as

$$D_1 = \{\, i \in \mathbb{Z} \mid \exists \alpha \in \mathbb{Z} : 0 \le i < N \wedge i = 3\alpha \,\}. \tag{4}$$

Similarly, the iteration domain of the statement in Line 5 can be represented as

$$D_2 = \{\, i \in \mathbb{Z} \mid \exists \alpha \in \mathbb{Z} : 0 \le i < N \wedge 1 \le i - 3\alpha \le 2 \,\}.$$

In general then, the "*polyhedral sets*" such as D_1 and D_2 that are used in the polyhedral representation of both the input program and the resulting process network, are defined as follows.

Definition 3 (Polyhedral Set). A *polyhedral set* S is a finite union of basic sets $S = \bigcup_i S_i$, each of which can be represented using affine constraints

$$S_i : \mathbb{Q}^n \to 2^{\mathbb{Q}^d} : \mathbf{s} \mapsto S_i(\mathbf{s}) = \{\, \mathbf{x} \in \mathbb{Z}^d \mid \exists \mathbf{z} \in \mathbb{Z}^e : A\mathbf{x} + B\mathbf{s} + D\mathbf{z} \ge \mathbf{c} \,\},$$

with $A \in \mathbb{Z}^{m \times d}$, $B \in \mathbb{Z}^{m \times n}$, $D \in \mathbb{Z}^{m \times e}$ and $\mathbf{c} \in \mathbb{Z}^m$. The *parameter domain* of S is the (non-parametric) polyhedral set $\operatorname{pdom} S = \{\, \mathbf{s} \in \mathbb{Z}^n \mid S(\mathbf{s}) \ne \emptyset \,\}$.

Note that any polyhedral set can be represented in infinitely many ways. When talking about polyhedral sets, we will usually have a specific representation in mind. In a similar fashion, we can define "*polyhedral relations*" over pairs of sets.

Definition 4 (Polyhedral Relation). A *polyhedral relation* R is a finite union of basic relations $R = \bigcup_i R_i$ of type $\mathbb{Q}^n \to 2^{\mathbb{Q}^{d_1+d_2}}$, each of which can be represented using affine constraints

$$R_i = \mathbf{s} \mapsto R_i(\mathbf{s}) = \{\, (\mathbf{x}_1, \mathbf{x}_2) \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \exists \mathbf{z} \in \mathbb{Z}^e : A_1\mathbf{x}_1 + A_2\mathbf{x}_2 + B\mathbf{s} + D\mathbf{z} \ge \mathbf{c} \,\},$$

with $A_i \in \mathbb{Z}^{m \times d_i}$, $B \in \mathbb{Z}^{m \times n}$, $D \in \mathbb{Z}^{m \times e}$ and $\mathbf{c} \in \mathbb{Z}^m$. The *parameter domain* of R is the (non-parametric) polyhedral set $\operatorname{pdom} R = \{\, \mathbf{s} \in \mathbb{Z}^n \mid R(\mathbf{s}) \ne \emptyset \,\} = \{\, \mathbf{s} \in \mathbb{Z}^n \mid \exists \mathbf{x}_1 \in \mathbb{Z}^{d_1}, \mathbf{x}_2 \in \mathbb{Z}^{d_2} : (\mathbf{x}_1, \mathbf{x}_2) \in R(\mathbf{s}) \,\}$. The *domain* of R is the polyhedral set $\operatorname{dom} R = \mathbf{s} \mapsto \{\, \mathbf{x}_1 \in \mathbb{Z}^{d_1} \mid \exists \mathbf{x}_2 \in \mathbb{Z}^{d_2} : (\mathbf{x}_1, \mathbf{x}_2) \in R(\mathbf{s}) \,\}$, while the *range* of R is the polyhedral set $\operatorname{ran} R = \mathbf{s} \mapsto \{\, \mathbf{x}_2 \in \mathbb{Z}^{d_2} \mid \exists \mathbf{x}_1 \in \mathbb{Z}^{d_1} : (\mathbf{x}_1, \mathbf{x}_2) \in R(\mathbf{s}) \,\}$. The polyhedral relation R can also be interpreted as being of type $\mathbb{Q}^n \to \mathbb{Q}^{d_1} \to 2^{\mathbb{Q}^{d_2}}$. Supplying two arguments then yields the *image* of an element $\mathbf{t} \in \operatorname{dom} R(\mathbf{s})$, i.e.,

$$R(\mathbf{s}, \mathbf{t}) = \{\, \mathbf{x}_2 \in \mathbb{Z}^{d_2} \mid (\mathbf{t}, \mathbf{x}_2) \in R(\mathbf{s}) \,\}.$$

Polyhedral sets and polyhedral relations have essentially the same type if we set $d = d_1 + d_2$. The difference is mainly a matter of interpretation. In polyhedral relations, we make a distinction between two sets of variables, whereas in polyhedral sets, there is no such distinction. Any polyhedral set can also be treated as a polyhedral relation with a zero-dimensional domain, i.e., by setting $d_1 = 0$, $d_2 = d$ and $A_2 = A$. Any statement about polyhedral relations will therefore also hold for polyhedral sets. We will usually only treat the general case of polyhedral relations.

3.2 Lexicographic Order

For a proper analysis of a sequential program, we not only need to know for which iterator values a given statement is executed, but also in which order these instances are executed. A given instance is executed before another instance if they are executed in the same iteration of zero or more outermost loops and if the second instance is executed in a later iteration of the next outermost loop. In other words, the execution order corresponds to the *lexicographic order* on iteration vectors.

Definition 5 (Lexicographic order). A vector $\mathbf{a} \in \mathbb{Z}^n$ is said to be *lexicographically smaller* than $\mathbf{b} \in \mathbb{Z}^n$ if for the first position i in which $\mathbf{a}$ and $\mathbf{b}$ differ, we have $a_i < b_i$, or, equivalently,

$$\mathbf{a} \prec \mathbf{b} \equiv \bigvee_{i=1}^{n} \left(a_i < b_i \wedge \bigwedge_{j=1}^{i-1} a_j = b_j \right). \tag{5}$$

Note that the lexicographic order can be represented as a polyhedral relation

$$L_n = \{ (\mathbf{a}, \mathbf{b}) \in \mathbb{Z}^n \times \mathbb{Z}^n \mid \mathbf{a} \prec \mathbf{b} \} \tag{6}$$

$$= \bigcup_{i=1}^{n} \{ (\mathbf{a}, \mathbf{b}) \in \mathbb{Z}^n \times \mathbb{Z}^n \mid a_i < b_i \wedge \bigwedge_{j=1}^{i-1} a_j = b_j \}. \tag{7}$$

3.3 Polyhedral Models

The polyhedral model of a sequential program mainly consists of the iteration domains of the statements (as explained in Section 3.1), and *access relations*. An access relation is a polyhedral relation $R \subseteq \mathbb{Z}^d \times \mathbb{Z}^a$ that maps the iteration vector of the corresponding statement to an array element. Here, d is the dimension of the iteration domain and a is the dimension of the array. A scalar can be considered as a zero-dimensional array. The access relation of an access to a scalar is therefore simply the Cartesian product of the iteration domain and the whole zero-dimensional space.

Example 4. The access relation of the first argument to the call to `Sobel` in Line 6 of Figure 1 is

$$\{(i,j) \to (a,b) \mid 1 \le i < K-1 \wedge 1 \le j < N-1 \wedge a = i-1 \wedge b = j-1\}. \quad (8)$$

Finally, the model should contain information about the relative execution order of any pair of statements in order to determine whether an iteration of one statement precedes or follows an iteration of another statement. One way of representing this relative position is to keep for each pair of statements the number of enclosing loops that they have in common and the relative ordering of the two statements in the program text.

Example 5. In Figure 1, `ReadImage` precedes `Sobel` in the program text, which in turn precedes `WriteImage`. `ReadImage` shares no enclosing loops with either of the other two statements, while `Sobel` and `WriteImage` share two enclosing loops.

Another way of representing the relative order is to record for each statement the position of the statement itself and each of its enclosing loops in the program text. That is, for a statement with d_i enclosing loops, we keep a vector of $d_i + 1$ "positions", e.g., line numbers, ordered from outermost to innermost. We call this vector the *statement location*.

Example 6. In Figure 1, `ReadImage` has statement location $(1,2,3)$, `Sobel` has statement location $(4,5,6)$ and `WriteImage` has statement location $(4,5,9)$.

Note that the first representation can easily be derived from the second representation.

Combining all this information, we have the following definition.

Definition 6 (Polyhedral Model). The *polyhedral model* of a sequential program consists of a list of statements, where each statement is in turn represented by

- an identifier,
- a dimension d_i,
- an iteration domain (a polyhedral set) $D_i \subseteq \mathbb{Z}^{d_i}$,
- a list of accesses and
- a statement location.

Finally, each array access is represented by an identifier, an access relation (a polyhedral relation) and a type (read or write).

The use of a polyhedral model imposes the following restrictions on the input program:

- static control flow,
- pure functions and
- loop bounds, conditions and index expressions are quasi-affine expressions in the parameters and the iterators of enclosing loops.

The extraction of a polyhedral model from a sequential program is available in several modern industrial and research compilers, e.g., [13, 15, 1].

3.4 Piecewise Quasi-Polynomials

As explained in Section 2, each channel in the process network has a buffer of a bounded size. If the network is parametric, then this bound will in general not simply be a number, but rather an expression in the parameters. Some of the intermediate steps in computing these bounds may also result in expressions involving both the parameters and the iterators, or just the iterators if the network is non-parametric. In both cases the expressions will be *piecewise quasi-polynomials*. Quasi-polynomials are polynomial expressions that may involve integer divisions of affine expressions in the variables. Piecewise quasi-polynomials are quasi-polynomials defined over polyhedral pieces of the domain. More formally, they can be defined as follows.

Definition 7 (Quasi-polynomial). A *quasi-polynomial* $q(\mathbf{x})$ in the integer variables $\mathbf{x}$ is a polynomial expression in greatest integer parts of affine expressions in the variables, i.e., $q(\mathbf{x}) \in \mathbb{Q}[\lfloor \mathbb{Q}[\mathbf{x}]_{\leq 1} \rfloor]$.

Definition 8 (Piecewise Quasi-polynomial). A *piecewise quasi-polynomial* $q(\mathbf{x})$, with $\mathbf{x} \in \mathbb{Z}^d$ consists of a finite set of pairwise disjoint polyhedra $K_i \subseteq \mathbb{Q}^d$, each with an associated quasi-polynomial $q_i(\mathbf{x})$. The value of the piecewise quasi-polynomial at $\mathbf{x}$ is the value of $q_i(\mathbf{x})$ with K_i the polyhedron containing $\mathbf{x}$, i.e.,

$$q(\mathbf{x}) = \begin{cases} q_i(\mathbf{x}) & \text{if } \mathbf{x} \in K_i \\ 0 & \text{otherwise.} \end{cases}$$

Note that the usual polynomials are special cases of quasi-polynomials as $\lfloor x_j \rfloor = x_j$ for x_j integer.

Example 7. Consider the statement in Line 3 of Figure 3. The number of times this statement is executed can be represented by the piecewise quasi-polynomial

$$\left\{ \left\lfloor \frac{N}{3} \right\rfloor \quad \text{if } N \geq 0. \right.$$

Note that there is only one polyhedral “piece” in this example.

3.5 Polyhedral Process Networks

Now we can finally define the structure of a polyhedral process network. For simplicity we assume that no merging of processes has been performed, i.e., that each process corresponds to a statement in the polyhedral model of the input.

Definition 9 (Polyhedral Process Network). A *polyhedral process network* is a directed graph with as vertices a set of *processes* $\mathscr{P}$ and as edges *communication channels* $\mathscr{C}$. Each process $P_i \in \mathscr{P}$ has the following characteristics

- a statement identifier s_i,

- a dimension d_i,
- an iteration domain $D_i \subseteq \mathbb{Z}^{d_i}$.

Each channel $C_i \in \mathscr{C}$ has the following characteristics

- a source process $S_i \in \mathscr{P}$,
- a target process $T_i \in \mathscr{P}$,
- a source access identifier corresponding to one of the accesses in the statement s_{S_i},
- a target access identifier corresponding to one of the accesses in the statement s_{T_i},
- a polyhedral relation $M_i \subseteq D_{S_i} \times D_{T_i}$ mapping iterations from the source domain to the target domain,
- a type (e.g., FIFO),
- a piecewise quasi-polynomial buffer size.

The identifiers in the process network can be used to obtain more information about the statements and the accesses when constructing a hardware or software realization of the network. The mapping M_i identifies which iterations of the source process write to the channel, which iterations of the target process read from the channel and how these iterations are related. The buffer sizes are such that the network can be executed without deadlocks.

Example 8. Consider the network in Figure 2, derived from the code in Figure 1. All processes have dimension $d_i = 2$. The iteration domain D_1 of the ReadImage process was given in Example 2. The iteration domains of the other two processes are

$$D_2 = D_3 = \{ (i, j) \in \mathbb{Z}^2 \mid 1 \leq i < K - 1 \wedge 1 \leq j < N - 1 \}. \tag{9}$$

There are nine communication channels between the first and the second process and one communication channel between the second and the third. All communication channels in this network are FIFOs. How these communication channels are constructed is explained in Section 5.1 and how their types are determined is explained in Section 6. The arrows in Figure 2 representing the communication channels are annotated with the name of the array that has given rise to the channel and the buffer size. These buffer sizes are computed in Section 8.

4 Polyhedral Analysis Tools

The construction of a polyhedral process network relies on a number of fundamental polyhedral operations. This section provides an overview of these operations. In each case, one or more tools are mentioned with which the operation can be performed. This list of tools is not exhaustive.

4.1 Parametric Integer Programming

By far the most fundamental step in the construction of a process network is figuring out which processes communicate data with each other and in what way. At the level of the input source program, this means figuring out for each value read in the program where the value was written. That is, for each read access to an array element, we need to know what was the last write access to the same array element that occurred before the given read access. In particular, if many iterations of the same statement write to the same array element, we need the (lexicographically) last iteration of that statement. Computing such a lexicographically maximal (or minimal) element of a polyhedral set can be performed using *parametric integer programming*.

Let us first define a lexmax operator on polyhedral relations.

Definition 10 (Lexicographic Maximum). Given a polyhedral relation R, the *lexicographic maximum* of R is a polyhedral relation with the same parameter domain and the same domain as R that maps each element $\mathbf{i}$ of the domain to the unique lexicographically maximal element that corresponds to $\mathbf{i}$ in R, i.e.,

$$\operatorname{lexmax} R = \{\, (\mathbf{i},\mathbf{j}) \in R \subseteq \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \neg(\exists \mathbf{j}' \in \mathbb{Z}^{d_2} : (\mathbf{i},\mathbf{j}') \in R \wedge \mathbf{j} \prec \mathbf{j}') \,\}. \tag{10}$$

Now we can define an operation for computing $\operatorname{lexmax} R$ as a polyhedral relation.

Operation 1 (Lexicographic Maximum).
Input:
- *a basic polyhedral relation* $R : \mathbb{Z}^n \to \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2}$
- *a basic polyhedral set* $S : \mathbb{Z}^n \to \mathbb{Z}^{d_1}$

Output:
- *a polyhedral relation* $M = \operatorname{lexmax}(R \cap (S \times \mathbb{Z}^{d_2}))$
- *a polyhedral set* $E = S \setminus \operatorname{dom} R$

The polyhedral relation M satisfies the following additional conditions:

- *every existentially quantified variable in M is explicitly represented as the greatest integer part of an affine expression in the parameters and the domain variables,*
- *every variable in the range of M is explicitly represented as an affine expressions in the parameters, the domain variables and the existentially quantified variables.*

Operation 1 can be performed using `isl` [17],[1] or, with some additional transformations, using `piplib` [6].[2] The use of the output set E will become clear in Section 5.

Example 9. Let us compute the lexicographically maximal element of the polyhedral set D_1 (4) from Example 3. Recall that a polyhedral set can be treated as a polyhedral relation with zero-dimensional domain. The input to Operation 1 is then $R = N \mapsto \mathbb{Z}^0 \times D_1(N)$. S may be taken as the universal zero-dimensional set with

[1] `http://freshmeat.net/projects/isl/`
[2] `http://www.piplib.org/`

universal parameter domain, i.e., $S = N \mapsto \mathbb{Z}^0$. The output is the lexicographically maximal element of the set D_1:

$$M = N \mapsto \mathbb{Z}^0 \times \operatorname{lexmax} D_1(N) = \{\, i \in \mathbb{Z} \mid \exists \alpha = \left\lfloor \frac{N+2}{3} \right\rfloor : i = 3\alpha - 3 \wedge N \geq 1 \,\}.$$

The set E describes the (parameter) values for which there is no element in the set D_1, i.e., $E = N \mapsto \{\, () \mid N \leq 0 \,\}$, with $()$ the single element of $\mathbb{Z}^0$.

Using Operation 1, we can also compute the domain of a polyhedral relation as a polyhedral set. We simply compute the lexicographic maximum relation of each basic relation and then drop the range variables and the equalities that define them. Similarly, the range of a relation can be computed as the domain of the inverse relation.

Alternatively, the lexicographic maximum can be computed using `Omega` [9],[3] by expressing the lexicographic order in (10) using linear constraints, as in (7). However, the result will not necessarily satisfy the two conditions of Operation 1. On the other hand, the `Omega` library provides built-in operations for computing domains and ranges of relations.

4.2 Emptiness Check

A very basic and frequently used operation is that of checking whether a given polyhedral set or relation contains any elements for any value of the parameters.

Operation 2 (Emptiness Check).
Input: *a polyhedral relation* $R : \mathbb{Z}^n \to \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2}$
Output: `true` *if* $\forall \mathbf{s} \in \mathbb{Z}^n : R(\mathbf{s}) = \emptyset$ *and* `false` *otherwise*

Operation 2 can be performed by applying Operation 1 (Lexicographic Maximum) on each of the basic polyhedral sets in a polyhedral set $S \in \mathbb{Z}^{n+d_1+d_2}$ with the same description as R, but where all parameters and input variables are treated as set variables. The relation R is empty iff S is empty iff in turn none of the basic polyhedral sets has a lexicographically maximal element. Since S is (a union of) Integer Linear Programming (ILP) problem(s), any other algorithm for testing the feasibility of an ILP problem will work as well.

4.3 Parametric Counting

An important step in the buffer size computation is the computation of the number of elements in the buffer before a given read. We will be able to reduce this com-

[3] `http://www.cs.umd.edu/projects/omega/`

putation to that of counting of the number of elements in the image of a polyhedral relation $R(\mathbf{s})$, denoted $\#R(\mathbf{s},\mathbf{t})$, for which we will use the following operation.

Operation 3 (Number of Image Elements).
Input: *a polyhedral relation* $R : \mathbb{Z}^n \to \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2}$
Output: *a piecewise quasi-polynomial* $q : \mathbb{Z}^n \times \mathbb{Z}^{d_1} \to \mathbb{Q} : (\mathbf{s},\mathbf{t}) \mapsto q(\mathbf{s},\mathbf{t}) = \#R(\mathbf{s},\mathbf{t})$

We already performed Operation 3 on a polyhedral set in Example 7.

Operation 3 can be performed using `barvinok` [20, 18].[4] Note that this library takes a basic polyhedral set as input, but it can be made to handle basic polyhedral relations by treating the domain variables as extra parameters. Unions can be handled by first computing a disjoint union representation and then summing the number of elements in each of the individual basic sets in this representation.

4.4 Computing Parametric Upper Bounds

As explained in the previous section, Operation 3 can be used to compute the number of elements in a buffer at any given time. When allocating memory for this buffer, we need to know the maximal number of elements that will ever have to reside in the buffer. As usual, we want to perform this computation parametrically. In general, computing the actual maximum may be too difficult, however. We therefore settle for computing an upper bound that is reasonably close to the maximum.

Operation 4 (Upper Bound on a Quasi-polynomial).
Input:
- *a piecewise quasi-polynomial* $q : \mathbb{Z}^n \times \mathbb{Z}^d \to \mathbb{Q}$
- *a bounded polyhedral set* $S : \mathbb{Z}^n \to \mathbb{Z}^d$*, the domain over which to compute the upper bound*

Output: *a piecewise quasi-polynomial* $u : \mathbb{Z}^n \to \mathbb{Q}$ *such that*

$$u(\mathbf{s}) \geq \max_{\mathbf{t} \in S(\mathbf{s})} q(\mathbf{s},\mathbf{t}) \qquad \forall \mathbf{s} \in \operatorname{pdom} S$$

Operation 4 can be performed using `bernstein` [3].[5] Although the basic technique only applies to polynomials, extensions to quasi-polynomials are also available [5]. Alternatively, the quasi-polynomials, which are usually the result of a counting problem such as Operation 3, can be approximated by a polynomial during the counting process [12].

Combining Operation 3 and Operation 4 results in the following operation, by taking $S = \operatorname{dom} R$.

Operation 5 (Upper Bound on the Number of Image Elements).
Input: *a polyhedral relation* $R : \mathbb{Z}^n \to \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2}$
Output: *a piecewise quasi-polynomial* $u : \mathbb{Z}^n \to \mathbb{Q}$ *such that*

[4] `http://freshmeat.net/projects/barvinok/`
[5] `http://icps.u-strasbg.fr/pco/bernstein.htm`

$$u(\mathbf{s}) \geq \max_{\mathbf{t} \in \operatorname{dom} R(\mathbf{s})} \#R(\mathbf{s},\mathbf{t}) \qquad \forall \mathbf{s} \in \operatorname{pdom} R$$

Fig. 4 A simple program reading the same array elements several times

```
1 for (i = 0; i < N; ++i)
2   for (j = 0; j < i; ++j)
3     b[i][j] = f(a[i + j]);
```

Example 10. Consider the program in Figure 4 and assume we want to know the maximal number of times an unspecified element of array a is read. This number is the maximal number of domain elements in the access relation that map to the same array element. In terms of the operations we have defined above, it is the maximal number of image elements in the inverse of the access relation. The access relation is

$$A = \{ (i, j) \to a \mid 0 \leq i < N \land 0 \leq j < i \land a = i + j \}.$$

Its inverse is

$$A^{-1} = \{ a \to (i, j) \mid 0 \leq i < N \land 0 \leq j < i \land a = i + j \}.$$

Applying Operation 3 yields the number of times a given array element is read:

$$\#A^{-1}(N,a) = \begin{cases} a - \left\lfloor \frac{a}{2} \right\rfloor & \text{if } 0 \leq a < N \\ N - \left\lfloor \frac{a}{2} \right\rfloor - 1 & \text{if } N \leq a \leq 2N - 3. \end{cases}$$

An upper bound on this number can then be computed using Operation 4, yielding,

$$\max_{a \in \operatorname{dom} A^{-1}} \#A^{-1}(N,a) \leq u(N) = \begin{cases} \frac{N}{2} & \text{if } N \geq 2. \end{cases}$$

4.5 Polyhedral Scanning

When writing out code for a process in a process network, we not only need to make sure that an operation is performed for each element of its iteration domain, but we also need to insert the appropriate reading and writing primitives in the appropriate places. In particular, if C_i is a communication channel with the given process as its source, then a write to the communication channel needs to be inserted in each element of the domain of its mapping relation M_i. Similarly, if C_j is a communication channel with the given process as its target, then a read from the communication channel needs to be inserted in each element of the range of its mapping relation M_j. Each of these domains and ranges is a polyhedral set and we see that we need to generate code for visiting each element of these sets. That is, we need the following operation.

Operation 6 (Code Generation).
Input: *A set of polyhedral sets* $\{S_i\}$
Output: *Code for visiting each element of each* S_i *in lexicographic order*

Operation 6 can be performed using `CLooG` [2][6] or `CodeGen` [10] (part of the `Omega` library).

Note that when generating code for a process, we cannot simply apply Operation 6 on the iteration domains and the domains and ranges of the communication channel mappings, as the lexicographic order does not make a distinction between several occurrences of the same element in two or more of these sets. Rather, we need to ensure that the reads happen *before* the actual operation and that the writes happen *after* the operation. To enforce this extra ordering constraint, we introduce an extra innermost dimension, assigning it the value 0 for reads, 1 for the iteration domain and 2 for writes.

Example 11. Consider a process with a one-dimensional iteration domain

$$D = \{ i \mid 0 \le i < N \}$$

that reads a value from some other process in its first iteration

$$M_1 = \{ () \to i \mid i = 0 \},$$

propagates a value from one iteration to the next

$$M_2 = \{ i \to i' \mid 0 \le i < N - 1 \wedge i' = i + 1 \}$$

and then sends a value to some other process in its last iteration

$$M_3 = \{ i \to () \mid i = N - 1 \}.$$

The process is shown in Figure 5. The code that results from scanning $D \times \{1\}$, $\operatorname{ran} M_1 \times \{0\}$, $\operatorname{dom} M_2 \times \{2\}$, $\operatorname{ran} M_2 \times \{0\}$ and $\operatorname{dom} M_3 \times \{2\}$ is shown schematically in Figure 6.

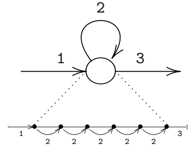

Fig. 5 A process with a one-dimensional iteration domain

[6] `http://www.cloog.org/`

```
for (i = 0; i < N; i++) {
  if (i == 0)
    Read(1);
  if (i >= 1)
    Read(2);
  f();
  if (i < N - 1)
    Write(2);
  if (i == N - 1)
    Write(3);
}
```

Fig. 6 Code generated for the process in Figure 5

5 Dataflow Analysis

This section describes how the communication channels in the process network are constructed using exact dataflow analysis [7]. We first discuss the standard dataflow analysis and then explain how some inter-process communication can be avoided by considering reuse.

5.1 Standard Dataflow Analysis

Standard exact dataflow analysis [7] is concerned with finding for each read of a value from an array element in the program, the write operation that wrote that value to that array element. By replacing the write to the array by a write to one or more communication channels (as many as there are accesses in the program where the value is read) and the read from the array to a read from the appropriate communication channel, we will have essentially constructed the communication channels. The effect of dataflow analysis can be described as the following operation.

Operation 7 (Dataflow Analysis).
Input:
- *a read access relation* $R \subseteq \mathbb{Z}^d \times \mathbb{Z}^a$
- *a list of [write] access relations* $W_i \subseteq \mathbb{Z}^{d_i} \times \mathbb{Z}^a$

Output:
- *a list of polyhedral relations* $M_i \subseteq \mathbb{Z}^{d_i} \times \mathbb{Z}^d$
- *a polyhedral set* $S \subseteq \mathbb{Z}^d$

The output satisfies the following constraints

- *each element in the domain of R is an element either of S or of the range of exactly one of the mappings M_i, i.e.,* $\{\operatorname{ran} M_i\}_i \cup \{S\}$ *partitions* $\operatorname{dom} R$,
- *if a particular element in the domain of R is in the range of one of the mappings, i.e.,* $\mathbf{j} \in \operatorname{dom} R \cap \operatorname{ran} M_k$, *then the corresponding [write] iteration of W_k, i.e.,* $M_k^{-1}(\mathbf{j})$, *was the last iteration in any of the domains of the input access relations W_i that accessed the same array element as* $\mathbf{j}$, *[i.e., it wrote the value read by* $\mathbf{j}$*] and was executed before* $\mathbf{j}$,

- *if a read iteration belongs to S, i.e.,* $\mathbf{j} \in \operatorname{dom} R \cap S$*, then this iteration accesses an array element that was not accessed by any of the* W_i*, [i.e., this iteration reads an uninitialized value].*

Operation 7 is applied for each read access in the program. The list of access relations required in the input is constructed from all write accesses in the program that access the same array. For each $M_i \neq \emptyset$, a communication channel is created from the process corresponding to the writing statement to the process corresponding to the reading statement, with the given M_i as mapping. The type and buffer size are defined in the following sections. The set S in the output is assumed to be empty.

Below, we briefly sketch how Operation 7 can be implemented using Operation 1, but first we define a global order on all iterations of all statements in the input program. Recall from Section 3.2 that within a single iteration domain, the execution order corresponds to the lexicographic order. In Section 3.3 we explained that the relative order of different statements can be expressed using statement locations. A global order can be obtained by combining these two pieces of information. In particular, let $d = \max_i d_i$. We define a $(2d+1)$-dimensional space where the even dimensions (0 to $2d$) correspond to the elements of the statement locations (in order) and the odd dimensions (1 to $2d-1$) correspond to the dimensions of the iteration domains (in order). We can map each iteration domain in this way to the $(2d+1)$-dimensional space and we call the result the *extended iteration domain*. For iteration domains with $d_i < d$, the remaining dimensions can be assigned an arbitrary value, say zero. The lexicographic order on this space corresponds to the execution order of the input program. In particular, if the first dimension in which two extended iteration vectors differ is $2n$, then the two corresponding statements share n loops, the iterators of these n loops have the same value in both vectors and the statements have a different location inside the nth loop. If the first dimension in which two extended iteration vectors differ is $2n+1$, then the two corresponding statements share at least $n+1$ loops, the iterators of first n loops have the same value in both vectors, but the iterator of the next loop has a different value in the two vectors. The access relations can similarly be extended by extending their domains.

Example 12. From Example 2, we know that the iteration domain associated to the `ReadImage` statement in Line 3 of Figure 1 is $\{\,(i,j) \in \mathbb{Z}^2 \mid 0 \le i < K \wedge 0 \le j < N\,\}$ (1), while from Example 6, we know that its statement location is $(1,2,3)$. The corresponding extended iteration domain is therefore

$$\{\,(1,i,2,j,3) \in \mathbb{Z}^2 \mid 0 \le i < K \wedge 0 \le j < N\,\}.$$

Let us now assume that the input to Operation 7 contains a single (extended) write access relation W. The composition of the read access relation with the inverse of the write access relation yields a mapping from read iterations to write iterations that wrote to the array element accessed by the read. The one that actually wrote the value that is read, is the one that was the (lexicographically) last to write to the element before the read. That is, we want to compute

$$\operatorname{lexmax}\left((W^{-1} \circ R) \cap L_{2d+1}^{-1}\right),$$

with L_{2d+1} the lexicographically smaller relation from (7). The result of this computation is the inverse of the relation M in the output.

Unfortunately, we cannot directly apply Operation 1 to compute this lexicographic maximum because L_{2d+1} (7) is a union of basic relations as soon as $d \geq 1$. (If W or R are unions then the basic relations in these unions can be treated as separate writes or reads, so we need not worry about unions in this part of the equation.) However, an extended write iteration that shares i_1 iterator values with the extended read iteration will always be executed after an extended write iteration that only shares $i_2 < i_1$ iterator values with the extended read iteration. We can therefore first apply Operation 1 to the writes that share $2d$ iterator values and with S set to the domain of the access relation. If the resulting set E is not empty then we can continue with the writes that share $2d-1$ iterator values, with S set to each of the basic sets in E. This process continues until all the resulting sets E are empty or until we have finished the case of 0 common iterator values.

Example 13. Consider once more the access relation (8) of the first argument to the call to `Sobel` in Line 6 of Figure 1 from Example 4, but now in its extended form,

$$R = \{\, (4,i,5,j,6) \to (a,b) \mid 1 \leq i < K-1 \wedge 1 \leq j < N-1 \wedge a = i-1 \wedge b = j-1 \,\}.$$

The only write to the a array occurs in Line 3, with extended access relation

$$W = \{\, (1,i,2,j,3) \to (a,b) \mid 0 \leq i < K \wedge 0 \leq j < N \wedge a = i \wedge b = j \,\}.$$

Composition of these two relations yields

$$W^{-1} \circ R = \{\, (4,i,5,j,6) \to (1,i',2,j',3) \mid 1 \leq i < K-1 \wedge 1 \leq j < N-1 \wedge i' = i-1 \wedge j' = j-1 \,\}.$$

We see that the range iterators are uniquely defined in terms of the domain iterators, so in this case there is no need to compute the lexicographic maximum as $\operatorname{lexmax} W^{-1} \circ R$ would be identical to $W^{-1} \circ R$. However, let us consider what would happen if we were to apply the above algorithm anyway. Since the first iterators in domain and range are distinct constants, the two iteration vectors never share any initial iterator values. The first $2d = 4$ applications of Operation 1 therefore operate on an empty basic polyhedral relation R' and simply return the input basic polyhedral set $S = \operatorname{dom} R$ as E. The final application returns $M_1^{-1} = R' = W^{-1} \circ R$ and $E = \emptyset$. Dropping the extra iterators again, we obtain

$$M_1 = \{\, (i',j') \to (i,j) \mid 1 \leq i < K-1 \wedge 1 \leq j < N-1 \wedge i' = i-1 \wedge j' = j-1 \,\}. \quad (11)$$

If there is more than one write access relation in the input of Operation 7, then the computation is a little bit more complicated. After computing the lexicographically maximal element of a write that shares i iterator values with the read, we still need to check that there is no other write in between. That is, we need to check if there is a write from a different write access that also shares i iterator values with the read, also occurs before the read, but occurs after the already found write. Again we have

to consider all possible cases of shared iterator values between the two writes, now from i to $2d+1$, as a write that occurs *after* a given iteration and that has fewer iterator values in common with the given iteration will be executed after an iteration that has more iterator values in common.

The above sketch of an algorithm can still be significantly improved by checking the write access relations in the appropriate order and by taking into account the fact that the dimensions that correspond to the statement locations have fixed values. For a more detailed description, we refer to [7], where a variant of the above algorithm is applied directly on "quasts", the output structure of `piplib` which we will not explain here.

5.2 Reuse

A process network constructed using the standard dataflow analysis of the previous section will always send data from the process that corresponds to the statement that wrote a given value in the sequential program to the process that corresponds to the statement that read the given value, even if the same value is read multiple times from the same process. Now, in practically any form of implementation of process networks, sending data from a process to itself (or simply keeping it) is much cheaper than sending data to another process. It is therefore better to only transmit data from one process to another the first time it is needed.

Obtaining a network that only transmits data to a given process once is actually fairly simple. When applying Operation 7 from the previous section, instead of only supplying a list of write accesses to the array read by the read access R, we extend the list with all read accesses to the given array (including R itself) from the same statement. Any value that was already read by the given process will then be obtained from the last of those reads from the same process instead of from the process that actually wrote the value.

Fig. 7 A process network corresponding to the sequential program in Figure 1 exploiting reuse

Example 14. In Example 1, we have shown a process network (Figure 2) derived from the code in Figure 1 without exploiting reuse inside the Sobel process. Figure 7 shows a process network derived from the same code, but in this case with reuse. The first network was created by only considering the write to array a in Line 3 of Figure 1 as a possible source for the reads in Line 6, while the second was created by also considering all the reads in Line 6 as possible sources. It is clear from the figures that in the first network the Sobel process does not communicate with itself, while in the second network it does. What may not be apparent, however, is that in both networks there are 9 channels from the ReadImage process to the Sobel process. The second figure only shows 3 channels because the software used to generate these process networks automatically combines some communication channels if this combination does not affect the type of the channels, as explained at the end of Section 6.1. The number of channels from the first process to the second has not changed because each read in Line 6 reads at least one value that is not read by any of the other reads. However, even though the number of channels may not have changed, the number of values transmitted over these channels has been drastically reduced. In particular, the image read by the ReadImage process is now transmitted only once, instead of nearly 9 times.

6 Channel Types

In the previous section, we showed how to determine the communication channels between the processes in the network. The practical implementation of these channels in hardware depends on certain properties that may or may not hold for the channels. In particular, it will be important to know if a communication channel can be implemented as a FIFO. Other properties include multiplicity and special kinds of internal channels. In this section, we describe how to check these properties.

6.1 FIFOs and Reordering Channels

The communication channels derived in Section 5 are characterized by a mapping M (see Definition 9) relating write iterations in one process to the corresponding read iterations in the same or another process. The channel needs to ensure that when the second process reads from the channel it is given the correct value and therefore needs to keep track of which iteration in the first process wrote a given value. There is, however, no need to keep track of this extra information if it can be shown that the values will always be read in the same order as that in which they were written. In such cases, the communication channel can simply be implemented as a FIFO.

To check if the writes and reads are performed in the same order, we consider what happens when the order is not the same, i.e., when reordering occurs on the channel. In this case, there is a pair of writes $(\mathbf{w}_1, \mathbf{w}_2)$ such that the corresponding

reads $(\mathbf{r}_1, \mathbf{r}_2)$ are executed in the opposite order, i.e., $\mathbf{w}_1 \succ \mathbf{w}_2$ while $\mathbf{r}_1 \prec \mathbf{r}_2$. That is, reordering occurs if the relation

$$T = \{\, \mathbf{w}_1 \to \mathbf{w}_2 \mid \exists \mathbf{r}_1, \mathbf{r}_2 : (\mathbf{w}_1, \mathbf{r}_1), (\mathbf{w}_2, \mathbf{r}_2) \in M \wedge \mathbf{w}_1 \succ \mathbf{w}_2 \wedge \mathbf{r}_1 \prec \mathbf{r}_2 \,\} \tag{12}$$

is non-empty, which can be verified by applying Operation 2 (Emptiness Check). Note that T will only be considered empty if there is no reordering for any value of the parameters. Also note that we only need the lexicographic order within iteration domains and not across iteration domains, as we only compare reads to other reads and writes to other writes. If we identify two (or more) communication channels C_1 and C_2 as FIFOs, we can check if we can combine them into a single FIFO by performing two emptiness check on a relation T as in (12), except that in one relation $(\mathbf{w}_1, \mathbf{r}_1)$ is taken from M_1 and $(\mathbf{w}_2, \mathbf{r}_2)$ is taken from M_2 and vice versa in the other relation.

Fig. 8 A program resulting in a network with a reordering channel

```
1  for (i = 0; i <= N; i++)
2    a[i] = g(i);
3  for (i = 0; i <= N; i++)
4    b[i] = f(a[N-i]);
```

Example 15. Consider the program in Figure 8. The corresponding process network has two processes and one communication channel connecting them. The mapping on the channel is

$$M = \{\, w \to r \mid 0 \le w \le N \wedge r = N - w \,\}$$

and we have

$$\begin{aligned} T &= \{\, w_1 \to w_2 \mid \exists r_1, r_2 : (w_1, r_1), (w_2, r_2) \in M \wedge w_1 > w_2 \wedge r_1 < r_2 \,\} \\ &= \{\, w_1 \to w_2 \mid 0 \le w_1, w_2 \le N \wedge w_1 > w_2 \wedge N - w_1 < N - w_2 \,\} \\ &= \{\, w_1 \to w_2 \mid 0 \le w_2 < w_1 \le N \,\}. \end{aligned}$$

For $N > 0$, this relation is clearly non-empty. We conclude that we are dealing with a reordering channel. See Example 20 for a variation on this example where we find a FIFO.

6.2 Multiplicity

The standard dataflow analysis from Section 5.1 may result in communication channels that are read more often than they are written to when the same value is used in multiple iterations of the reading process. Such channels require special treatment and this multiplicity condition should therefore be detected by checking whether

there is any write iteration $\mathbf{w}$ that is mapped to multiple read iterations through the mapping M. In particular, let T be the relation

$$T = \{\, \mathbf{w}_1 \to \mathbf{w}_2 \mid \exists \mathbf{r}_1, \mathbf{r}_2 : (\mathbf{w}_1, \mathbf{r}_1), (\mathbf{w}_2, \mathbf{r}_2) \in M \wedge \mathbf{w}_1 = \mathbf{w}_2 \wedge \mathbf{r}_1 \prec \mathbf{r}_2 \,\}.$$

This relation contains all pairs of identical write iterations that correspond to a pair of distinct read iterations. If this relation is non-empty, then multiplicity occurs. As in the previous section, we can check the emptiness using Operation 2 (Emptiness Check).

It should be noted that if we use the dataflow analysis from Section 5.2, i.e., taking into account possible reuse, then the resulting communication channels will never exhibit multiplicity. Instead, two channels would be constructed, one for the first time a value is read and one for propagating the value read to later iterations. In general, it is preferable to detect reuse rather than multiplicity, because the analysis results in fewer types of channels and because taking advantage of reuse may split channels that are both reordering and with multiplicity into a pair of FIFOs.

```
1  for (i = 0; i < N; ++i)
2      a[i] = A(i);
3  for (j = 0; j < N; ++j)
4      b[j] = B(j);
5  for (i = 0; i < N; ++i)
6      for (j = 0; j < N; ++j)
7          c[i][j] = a[i] * b[j];
```

Fig. 9 Outer product source code.

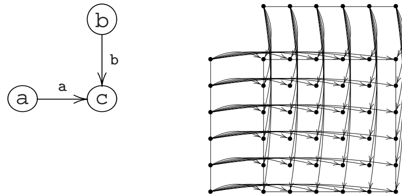

Fig. 10 Outer product network with multiplicity

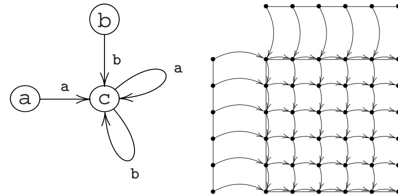

Fig. 11 Outer product network without multiplicity

Example 16. Consider the code for computing the outer product of two vectors shown in Figure 9. Figures 10 and 11 show the results of applying standard dataflow analysis (Section 5.1) and dataflow analysis with reuse (Section 5.2) respectively. Each figure shows the network on the left and the dataflow between the individual iterations of the processes on the right. The iterations are executed top-down,

left-to-right. In the first network, the channel between b and c has mapping

$$M_{\mathtt{b}\to\mathtt{c}} = \{\, w \to (r_1, r_2) \mid 0 \leq r_1, r_2 < N \wedge w = r_2 \,\}.$$

For testing multiplicity, we have the relation

$$T = \{\, r_{1,2} \to r_{2,2} \mid \exists r_{1,1}, r_{2,1} \in \mathbb{Z} : 0 \leq r_{1,1}, r_{1,2}, r_{2,1}, r_{2,2} < N \wedge r_{1,2} = r_{2,2} \wedge r_{1,1} < r_{2,1} \,\},$$

which is clearly non-empty, while for testing reordering we use the relation

$$T' = \{\, r_{1,2} \to r_{2,2} \mid \exists r_{1,1}, r_{2,1} \in \mathbb{Z} : 0 \leq r_{1,1}, r_{1,2}, r_{2,1}, r_{2,2} < N \wedge r_{1,2} > r_{2,2} \wedge r_{1,1} < r_{2,1} \,\},$$

which is also non-empty. In the second network, the channel between b and c has mapping

$$M_{\mathtt{b}\to\mathtt{c}} = \{\, w \to (r_1, r_2) \mid 0 \leq r_2 < N \wedge r_1 = 0 \wedge w = r_2 \,\}. \tag{13}$$

There is no multiplicity since each write corresponds to exactly one read. There is also no reordering, since the writes and reads occur for increasing values of $w = r_2$ in both processes.

6.3 Internal Channels

Internal channels, i.e., channels from a process to itself, can typically be implemented more efficiently than external channels. In particular, since processes are scheduled independently there is no guarantee that when a value is read, it is already available, or that when a value is written, there is still enough room in the channel to hold the data. The external channels therefore need to implement both blocking on read and blocking on write. For internal channels, there is no need for blocking. In fact, blocking would only lead to deadlocks. Besides the blocking issue, there are some special cases that allow for a more efficient implementation than a general FIFO buffer.

6.3.1 Registers

The first special case is that where the FIFO buffer holds at most one value. The FIFO buffer can then be replaced by a register. In Section 8 we will see how to compute a bound on a FIFO buffer in general, but the case where this bound is one can be detected in a simpler way. If each write $\mathbf{w}_1$ to a channel is followed by the corresponding read $\mathbf{r}_1$, without any other read occurring in between, then we indeed only need room for one value. On the other hand, if there *is* an intervening read, $\mathbf{w}_1 \prec \mathbf{r}_2 \prec \mathbf{r}_1$, then we will need more storage space. As usual, we can detect this situation by performing an emptiness check. Consider the set

$$T = \{ \mathbf{w}_1 \mid \exists \mathbf{r}_1, \mathbf{r}_2 : (\mathbf{w}_1, \mathbf{r}_1) \in M \wedge \mathbf{r}_2 \in \operatorname{ran} M \wedge \mathbf{w}_1 \prec \mathbf{r}_2 \prec \mathbf{r}_1 \}.$$

This set contains all writes $\mathbf{w}_1$ such that there exists a read $\mathbf{r}_2$ that occurs before the read $\mathbf{r}_1$ that corresponds to the write $\mathbf{w}_1$. Note that it is legitimate to compare read and write iterations because we are dealing with internal channels and so these iterations belong to the same iteration domain.

6.3.2 Shift Registers

Another special case occurs when the number of iterations between a write to an internal channel and the corresponding read from the channel is constant. If so, the FIFO can be replaced by a shift register, shifting the data in the channel by one in every iteration, independently of whether any data was read or written in that iteration. Such shift registers can be implemented more efficiently than FIFOs in hardware.

Checking whether we can use a shift register is as easy as writing down the relation

$$R = \{ (\mathbf{w}, \mathbf{i}) \in \mathbb{Z}^d \times \mathbb{Z}^d \mid \exists \mathbf{r} \in \mathbb{Z}^d : (\mathbf{w}, \mathbf{r}) \in M \wedge \mathbf{i} \in D \wedge \mathbf{w} \prec \mathbf{i} \prec \mathbf{r} \},$$

where M is the mapping on the channel and D is the iteration domain of the process, and applying Operation 3 (Number of Image Elements). The result is a piecewise quasi-polynomial q of type $\mathbb{Z}^n \times \mathbb{Z}^d \to \mathbb{Q}$, where n is the number of parameters. If the expression is independent of the last d variables, i.e., if q is defined purely in terms of the parameters and not in terms of the write iterators, then we can use a shift register. Note that we also count iterations in which no value is read from or written to the channel. This means that the size of the shift register may be larger than the buffer size of the FIFO, as computed in Section 8.

7 Scheduling

Although the processes in a process network are scheduled independently during their execution, there are two occasions where we may want to compute a common schedule for two or more processes. The first such occasion is when there are more processes in the network than there are processing elements to run them on. The second is when we want to compute safe buffer sizes as will be explained in Section 8. In principle, we could use the extended iteration domains from Section 5.1 as the common schedule, resulting in the same execution order as that of the input sequential program. This schedule may lead to very high overestimates of the buffer sizes, however, and we therefore prefer constructing a schedule that is more suited for the buffer size computation. There are many ways of obtaining such a schedule. Below we discuss a simple incremental technique.

7.1 Two Processes

Let us first consider the case where there are only two processes, P_1 and P_2, that moreover have the same dimension $d = d_1 = d_2$. We will relax these conditions in Section 7.2 and Section 7.4. Since the two iteration domains have the same dimension, we can consider the two iteration domains as being part of the same iteration space. However, if there are any communication channels between the two processes, then we need to make sure that that the second does not read a value from any channel before it has been written. Let $M_i \subseteq D_{P_1} \times D_{P_2}$ be the mapping of one of these channels and consider the delays between a write to the channel and the corresponding read

$$\Delta_i = \{\, \mathbf{t} \in \mathbb{Z}^d \mid \exists (\mathbf{w}, \mathbf{r}) \in M_i : \mathbf{t} = \mathbf{r} - \mathbf{w} \,\}.$$

Note that the concept of a delay, in particular the subtraction $\mathbf{r} - \mathbf{w}$, is only valid because we now consider both iteration domains as being part of the same iteration space. If any of these delays is lexicographically negative, i.e., if the smallest delay

$$\delta_i = \operatorname{lexmin} \Delta_i \tag{14}$$

is lexicographically negative, then some reads do occur before the corresponding writes and the schedule would be invalid. The solution is to delay the whole second process by just enough to make all the delays over the communication channels lexicographically non-negative. In particular, let

$$\delta^{1\to 2} = \operatorname{lexmin}\{\, \delta_i \,\}_i, \tag{15}$$

with i ranging over all communication channels between processes P_1 and P_2. Then, replacing the iteration domain D_2 of P_2 by $D_2 - \delta^{1\to 2}$ (and adapting all mappings M_i accordingly) will ensure that all delays over channels are lexicographically non-negative.

In principle, we could choose any delay on the second iteration domain that is lexicographically larger than $-\delta^{1\to 2}$. However, delaying the reads more than strictly necessary would only increase the time the values stay inside the communication channel and could therefore only increase the buffer size needed on the channel. Furthermore, if there are also communication channels from P_2 to P_1, then we need to make sure the delays on these channels are also non-negative. Fortunately, if $\delta^{2\to 1}$ is the minimal delay over any such channel, then

$$\delta^{1\to 2} + \delta^{2\to 1} \succ \mathbf{0}. \tag{16}$$

Otherwise there would be a read operation in process P_1 that indirectly depends on a write operation from the same process that occurs later or at the same time, which is clearly impossible if the process network was derived from a sequential program. By choosing a delay on process P_2 of $-\delta^{1\to 2}$, the minimal delay on any channel from P_2 to P_1 becomes exactly $\delta^{1\to 2} + \delta^{2\to 1}$ and is therefore safe.

There are still some details we glanced over in the discussion above. First, it should be clear that δ_i (14) can be computed using a variation of Operation 1 (Lexicographic Maximum) or even directly as $-\text{lexmax}(-\Delta_i)$. The minimal delay over all communication channels $\delta^{1\to 2}$ (15) can be computed as

$$\text{lexmin}\{\,\delta_i\,\}_i = \bigcup_i \{\,\delta_i \mid \delta_i \prec \delta_j \text{ for all } j < i \text{ and } \delta_i \preccurlyeq \delta_j \text{ for all } j > i\,\},$$

where both i and j range over all communication channels between the two processes. That is, we select δ_i for the values of the parameters where it is strictly smaller than all previous δ_j and smaller than or equal to all subsequent δ_j. The result is a polyhedral set that contains a single vector for each value of the parameters.

Finally, we need to deal with the fact that the above procedure can set the delay over a channel to zero, which means that the write and corresponding read would happen simultaneously. The solution is to assign an order of execution to the statements *within* a given iteration by introducing an extra innermost dimension as we did in Section 4.5. The values for the extra dimension can be set based on a topological sort of a direct graph with as only edges those communication channels with a zero delay. The absence of cycles in this graph is guaranteed by (16).

Example 17. Consider a network composed of the first two processes in Figure 2. There are nine channels between the two processes. For one of these channels, we computed the mapping M_1 (11) in Example 13. The delay between writes and reads on this channel is constant and we obtain $\{\,\delta_1\,\} = \Delta_1 = \{\,(1,1)\,\}$. The other channels also have constant delays and the smallest of these yields $\delta^{1\to 2} = (-1,-1)$. We therefore replace the second iteration domain D_2 (9) by $D_2 + (1,1)$, resulting in a new $\delta^{1\to 2}$ of $(0,0)$. To make this delay lexicographically positive (rather than just non-negative), we introduce an extra dimension and assign it the value 0 in P_1 and 1 in P_2. The new iteration domains are therefore

$$\begin{aligned} D_1 &= \{\,(i,j,0) \in \mathbb{Z}^3 \mid 0 \le i < K \wedge 0 \le j < N\,\}, \\ D_2 &= \{\,(i,j,1) \in \mathbb{Z}^3 \mid 2 \le i < K \wedge 2 \le j < N\,\}. \end{aligned}$$

7.2 More than Two Processes

If there are more than two processes in the network, then we can still apply essentially the same technique as that of the previous section, but we need to be careful about how we define the minimal delay $\delta^{1\to 2}$ (15) between two processes P_1 and P_2. It will not be sufficient to consider only the communication channels between P_1 and P_2 themselves. Instead, we need to consider all paths in the process network from P_1 to P_2, compute the sum of the minimal delays on the communication channels in each path and then take the minimum over all paths. In effect, this is just the shortest path between P_1 and P_2 in the process network with as weights the minimal delays on the communication channels. As in the case of a two process network

(16), the minimal delay over a cycle is always (lexicographically) positive. We can therefore apply the Bellman-Ford algorithm (see, e.g., [14, Section 8.3]) to compute this shortest path.

If the network not only contains more than two processes, but we also want to combine more that two processes, then we can apply the above procedure incrementally, in each step combining two processes into one process until we have performed all the required combinations. If many or all processes need to be combined, it may be more efficient to compute the all-pairs shortest paths using the Floyd-Warshall algorithm [14, Section 8.4] and then to update the minimal delays in each combination step.

Example 18. Consider once more the network in Figure 2. There are three processes, so we will need two combination steps. Let us for illustration purposes combine the first and the last process first, even though there is no direct edge between these two processes. (Normally, we would only combine processes that are directly connected.) The delay between P_2 and P_3 is constant and equal to $\delta^{2\to3} = (0,0)$, while the minimal delay between P_1 and P_2, as computed in Example 17, is $\delta^{1\to2} = (-1,-1)$. The minimal delay between P_1 and P_3 is therefore $\delta^{1\to3} = \delta^{1\to2} + \delta^{2\to3} = (-1,-1)$. We therefore replace the third iteration domain D_3 by $D_3 + (1,1)$, resulting in a new $\delta^{1\to3}$ of $(0,0)$. $\delta^{1\to2}$ remains unchanged, but $\delta^{2\to3}$ changes to $(1,1)$. In the next combination step, D_2 is again replaced by $D_2 + (1,1)$ and $\delta^{1\to2}$ and $\delta^{2\to3}$ both become $(0,0)$. There are now two edges with a zero minimal delay, so we assign the processes an increasing value in a new innermost dimension. The final iteration domains of the first two processes are as in Example 17, while that of the third process is

$$D_3 = \{\, (i,j,2) \in \mathbb{Z}^3 \mid 2 \le i < K \wedge 2 \le j < N \,\}.$$

Figure 12 shows the code for the completely merged process. For simplicity, we have written out this code in terms of the original statements and their original accesses rather than using read and writes to communication channels. If we were to compute buffer sizes based on the original schedule (Figure 1), then each channel between the first and the second process would get assigned a buffer size of $(K-2)(N-2)$ because in this original schedule the ReadImage process runs to completion before the Sobel process starts running. The significantly smaller buffer sizes that we obtain using the procedure of Section 8 based on the schedule computed here are shown in Figure 2.

7.3 Blocking Writes

In deriving combined schedules, we have so far only been interested in avoiding deadlocks. When merging several processes into a single process this is indeed all we need to worry about. However, when using a global schedule in the computation

```
1  for (i = 0; i < K; i++)
2    for (j = 0; j < N; j++) {
3      a[i][j] = ReadImage();
4      if (i >= 2 && j >= 2) {
5        Sbl[i-1][j-1] = Sobel(a[i-2][j-2], a[i-1][j-2], a[i][j-2],
6                              a[i-2][j-1], a[i-1][j-1], a[i][j-1],
7                              a[i-2][  j], a[i-1][  j], a[i][  j]);
8        WriteImage(Sbl[i-1][j-1]);
9      }
10   }
```

Fig. 12 Merged code for the process network of Figure 2

of valid buffer sizes, we may end up with buffer sizes that, while not introducing any deadlocks, may impose blocking writes, impacting negatively on the throughput of the network. In particular, iterations of different processes that are mapped onto the same iteration in the global schedule (ignoring the extra innermost dimension), can usually be executed in a pipelined fashion, except when non-adjacent processes in this pipeline communicate with each other. In this case, the computed buffer sizes may be so small that the first of these non-adjacent processes needs to wait until the second reads from the communication channel before proceeding onto the next iteration.

The solution is to enforce a delay between a writing process and the corresponding reading process that is equal to one iteration of the writing process. This delay ensures that the delay between non-adjacent processes in a pipeline will be large enough to avoid blocking writes that hamper the throughput. The required delay is the smallest variation in the iteration domain D of a process. Let P be the relation that maps a given iteration of D to the previous iteration of D, i.e.,

$$P(D) = \operatorname{lexmax}\{\, \mathbf{i} \to \mathbf{i}' \mid \mathbf{i}, \mathbf{i}' \in D \wedge \mathbf{i}' \prec \mathbf{i} \,\}. \tag{17}$$

The relation $P(D)$ can be computed using Operation 1 (Lexicographic Maximum). The required delay η is then the smallest difference between two subsequent iterations, i.e.,

$$\eta = \operatorname{lexmin}\{\, \mathbf{t} \mid \exists (\mathbf{i}, \mathbf{i}') \in P(D) \wedge \mathbf{t} = \mathbf{i} - \mathbf{i}' \,\}.$$

The delay is enforced by subtracting the η corresponding to the writing process from each channel delay δ_i (14). Note, however, that in the presences of cycles, we may not always be able to enforce this extra delay. In particular, subtracting the η's may result in negative total delays over these cycles. If this happens, we have to resort to the scheduling of Section 7.2, without the additional delays.

Example 19. Consider the code in Figure 13. There are three statements, f, g, and h with three FIFO communication channels between them: one from f to g, one from g to h and one from f to h. The delays on these channels are all zero. The scheduling of Section 7.2 therefore leaves everything in place and the resulting code is identical to the original. It is clear that in this schedule the size of each of the FIFOs can

```
1 for (i = 0; i < N; ++i) {
2   a[i] = f(i);
3   b[i] = g(a[i]);
4   c[i] = h(a[i], b[i]);
5 }
```

Fig. 13 A program illustrating blocking writes

be set to 1. However, if we try to run the resulting process network, then we see that the second iteration of process f needs to wait for the first iteration of h to empty the FIFO before it can write to the FIFO of size 1. Because of this blocking write, the three processes cannot be fully pipelined, as illustrated in Figure 14. Since all domains are 1-dimensional, the internal delay in each process is 1. Subtracting this internal delay from each channel delay, we see that the minimal delay between process f and h is $(-1)+(-1) = -2$. Process h is therefore scheduled at position 2 relative to process f. This means that the channel between these two process needs to be of size at least 2. This size in turn then allows a fully pipelined execution of the three processes, as illustrated in Figure 15.

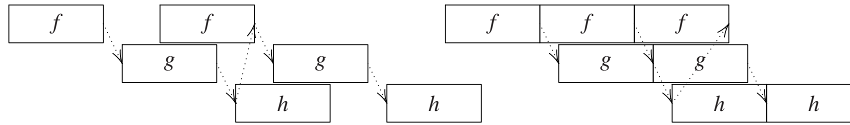

Fig. 14 Time diagram with blocking

Fig. 15 Time diagram without blocking

7.4 *Linear Transformations*

The schedules that we have seen so far are all of the form $\theta(\mathbf{j}) = I_{d+1,d}\mathbf{j} + \mathbf{b}$, where $I_{d+1,d}$ is $(d+1) \times d$ matrix with $i_{k,k} = 1$ and $i_{k,l} = 0$ for $k \neq l$, and $\mathbf{b} \in \mathbb{Z}^{d+1}$. That is, we add an extra dimension ($I_{d+1,d}\mathbf{j}$) and apply a shift ($\mathbf{b}$). Note that we have also assumed so far that all iteration domains have the same dimension. In principle, we could apply more general affine schedules $\theta(\mathbf{j}) = A\mathbf{j} + \mathbf{b}$, i.e., including an arbitrary linear transformation $A\mathbf{j}$. Not only would this allow us to compute potentially tighter bounds on the buffer sizes, but it may even change the types of the communication channels.

Example 20. Consider once more the program in Figure 8. In Example 15 we have shown that the only channel in the corresponding network has reordering. If we reverse the second loop, however, i.e., if we apply the transformation $-\mathbf{j} + N$, then the mapping on the channel becomes

$$M = \{ w \to r \mid 0 \leq w \leq N \wedge r = w \}$$

and we have

$$T = \{ w_1 \to w_2 \mid 0 \le w_1, w_2 \le N \wedge w_1 > w_2 \wedge w_1 < w_2 \}.$$

This set is clearly empty and so the communication channel has become a FIFO.

The example shows that if we were to apply general linear transformations, then we would need to do this before the detection of channel types of Section 6. In fact, if we want to detect reuse, as we did in Section 5.2, then we would need to do this reuse detection *after* any linear transformation. The reason is that a linear transformation may change the internal order inside an iteration domain, meaning that what used to be a previous iteration, from which we could reuse data, may have become a later iteration. On the other hand, the dataflow in the sequential program imposes some restrictions on which linear transformations are valid, so we need to perform dataflow analysis (Section 5) *before* any linear transformation. The solution is to first apply standard dataflow analysis (Section 5.1), then to perform linear transformations and finally to detect reuse inside the communication channels that were constructed in the first step. A full discussion of general linear transformations is beyond the scope of this chapter.

There is however one case where we are forced to apply a linear transformation, and that is the case where not all iteration domains have the same dimension. Before we can merge two iteration domains, they have to reside in the same space. In particular, they need to have the same dimension. Let d be the maximal dimension of any iteration domain, then we could could expand a d_i-dimensional iteration domain with $d_i \le d$ using the transformation I_{d,d_i}. This transformation effectively pads the iteration vectors with zeros at the end. This may, however, not be the best place to insert the zeros. A simple heuristic to arrive at better positions for the zeros is to look at the communication channels between a lower-dimensional domain and a full-dimensional domain and to insert the zeros such that iterators that are related to each other through the communication channels are aligned. Note that inserting zeros does not change the internal order inside the iteration domain. The same holds for any scaling that we may want to apply based on essentially the same heuristic.

Example 21. Consider the communication channel between processes `b` and `c` in the network of Figure 11 (see Example 16). The first process has a one-dimensional iteration domain, while the second process has a two-dimensional iteration domain. The mapping on the channel (13) shows that there is a relation $w = r_2$ between the single iterator of the writing process and the second iterator of the reading process. We therefore insert a zero before the iteration vectors of the writing process such that the original iterator ends up in the second position. For the channel between `a` and `c` the relation is $w = r_1$, so in this case we insert a zero *after* the original iterator. These choices are reflected by the orientations of these two one-dimensional iteration domains in Figure 11. With these orientations, the writes may be moved on top of the corresponding reads by simply shifting the iteration domains, meaning that a buffer size of at most 1 is needed. Had we chosen a different orientation for the two one-dimensional iteration domains, then this would not have been possible.

8 Buffer Size Computation

This section describes how to compute valid buffer sizes for all communication channels. The buffer sizes are valid in the sense that imposing them does not cause deadlocks. The first step is to compute a global schedule for all processes in the network, e.g., using the techniques of Section 7. This scheduling step effectively makes all communication channels internal (to the single combined process). We then compute buffer sizes for this particular schedule. The schedule itself is not used during the actual execution of the network. However, we know that there is at least one schedule for which the buffer sizes are valid. The blocking reads and writes on the communication channels will therefore not introduce any deadlocks.

8.1 FIFOs

We are given an internal communication channel that has been identified as a FIFO using the technique of Section 6.1 and we want to know how much buffer space is needed on the FIFO. Recall that any channel may be considered internal after a global scheduling step that maps all iteration domains to a common iteration space. The buffer should be large enough to hold the number of tokens that are in transit between a write and a read at any point during the execution. We therefore first count this number of tokens in terms of an arbitrary iteration and then compute an upper bound over all iterations.

Let us look at these steps in a bit more detail. The communication channel is described by a mapping M from the write iteration domain D_{w} to the read iteration domain D_{r}. The maximal number of tokens in the buffer will occur after some write to the buffer and before the first subsequent read from the buffer. It is therefore sufficient to investigate what happens right before a token is read from the buffer. Let W be the relation mapping any read iteration $\mathbf{r}$ to all write iterations that occur before this read and let R be the relation mapping the same read iteration to all previous read iterations, i.e.,

$$W = \{\, \mathbf{r} \to \mathbf{w}' \mid \mathbf{r} \in \operatorname{ran} M \wedge \mathbf{w}' \in \operatorname{dom} M \wedge \mathbf{w}' \prec \mathbf{r} \,\} \tag{18}$$

$$R = \{\, \mathbf{r} \to \mathbf{r}' \mid \mathbf{r}, \mathbf{r}' \in \operatorname{ran} M \wedge \mathbf{r}' \prec \mathbf{r} \,\}. \tag{19}$$

Then the number $n(\mathbf{s},\mathbf{r})$ of elements in the buffer right before the execution of read $\mathbf{r}$ is the number of writes to the buffer before the read, i.e., $\#W(\mathbf{s},\mathbf{r})$, minus the number of reads from the buffer before the given read, i.e., $\#R(\mathbf{s},\mathbf{r})$, where as usual, $\mathbf{s}$ are the parameters. Both of these computation can be performed using Operation 3 (Number of Image Elements). Finally, we apply Operation 4 (Upper Bound on a Quasi-polynomial) to the piecewise quasi-polynomial $n(\mathbf{s},\mathbf{r}) = \#W(\mathbf{s},\mathbf{r}) - \#R(\mathbf{s},\mathbf{r})$ and the polyhedral set $\operatorname{ran} M$, resulting in a piecewise quasi-polynomial $u(\mathbf{s})$ that is an upper bound on the number of elements in the FIFO channel during the whole execution.

Example 22. Consider once more the network in Figure 2. A global schedule for this network was derived in Examples 17 and 18, and code corresponding to this schedule is shown in Figure 12. Writing the mapping (11) on the channel constructed from the first argument to the call to `Sobel` in Line 6 of Figure 1 in Example 13 in terms of the new iterators, we obtain

$$M = \{\, (i', j', 0) \to (i, j, 1) \mid 2 \le i < K \wedge 2 \le j < N \wedge\ i' = i - 2 \wedge j' = j - 2 \,\}.$$

From this relation we derive,

$$\begin{aligned} W = \{\, (i, j, 1) \to (i', j', 0) \mid {} & 2 \le i < K \wedge 2 \le j < N \wedge \\ & 0 \le i' < K - 2 \wedge 0 \le j' < N - 2 \wedge \\ & ((i' < i) \vee (i' = i \wedge j' \le j)) \,\}. \end{aligned}$$

The number of image elements of this relation can be computed as

$$\#W(K, N, i, j) = \begin{cases} i(N-2) + j + 1 & \text{if } (i, j, 1) \in \operatorname{ran} M \wedge i < K - 2 \wedge j < N - 2 \\ (i+1)(N-2) & \text{if } (i, j, 1) \in \operatorname{ran} M \wedge i < K - 2 \wedge j \ge N - 2 \\ (K-2)(N-2) & \text{if } (i, j, 1) \in \operatorname{ran} M \wedge i \ge K - 2. \end{cases}$$

For the number of reads before a given read $(i, j, 1)$, we similarly find

$$\#R(K, N, i, j) = \begin{cases} (i-2)(N-2) + j - 2 & \text{if } (i, j, 1) \in \operatorname{ran} M. \end{cases}$$

Taking the difference yields

$$n(K, N, i, j) = \begin{cases} 2(N-2) + 3 & \text{if } (i, j, 1) \in \operatorname{ran} M \wedge i < K - 2 \wedge j < N - 2 \\ 3(N-2) - j + 2 & \text{if } (i, j, 1) \in \operatorname{ran} M \wedge i < K - 2 \wedge j \ge N - 2 \\ (K-i)(N-2) - j + 2 & \text{if } (i, j, 1) \in \operatorname{ran} M \wedge i \ge K - 2. \end{cases}$$

The maximum over all reads in $\operatorname{ran} M$ occurs in the first domain and is equal to $2(N-2) + 3 = 2N - 1$, which is the value shown on the first edge in Figure 2.

8.2 Reordering Channels

For channels that have been identified as exhibiting reordering using the technique of Section 6.1, we need to choose where we want to perform the reordering of the tokens. One option is to perform the reordering inside the channel itself. In this case we can apply essentially the same technique as that of the previous section to compute an upper bound on the minimal number of locations needed to store all elements in the buffer at any given time. However, since the tokens now have to be reordered, we need to be able to address them somehow. An efficient mapping from read or write iterators to the internal buffer of the channel may require more

space than strictly necessary. We refer to [4] for an overview and a mathematical framework for finding good memory mappings.

Another option is to perform the reordering inside the reading process. In this case, a process wanting to read a value from the reordering channel first looks in its internal buffer associated with the channel. If the value is not in the buffer, it reads values from the channel in the order in which they were written, storing all values it does not need yet in the local buffer until it has read the value it is actually looking for. In other words, the original reordering channel is split into a FIFO channel and a local reordering buffer. There are therefore two buffer sizes to compute in this case.

Before embarking upon the computation of these buffer sizes, it should be noted that nothing really interesting happens to the buffers during iterations that do not read from the FIFO. In particular, the maximal number of elements in the FIFO buffer will be reached right before a read from the FIFO and the maximal number of elements in the reordering buffer will be reached right before the read of the value from the FIFO that the process is actually interested in, i.e., after it has copied all intermediate data to the reordering buffer. The uninteresting iterations U are those reads $\mathbf{r}$ for which there is an earlier read $\mathbf{r}'$ that reads something that was written *after* the value read by $\mathbf{r}$. This latter value will have been put in the reordering buffer at or perhaps even before read $\mathbf{r}'$. That is,

$$U = \{\, \mathbf{r} \mid \exists \mathbf{w}, \mathbf{r}', \mathbf{w}' : (\mathbf{w}, \mathbf{r}) \in M \wedge (\mathbf{w}', \mathbf{r}') \in M \wedge \mathbf{w} \prec \mathbf{w}' \wedge \mathbf{r}' \prec \mathbf{r} \,\}$$

and

$$S = \operatorname{ran} M \setminus U$$

is the set of “interesting” iterations.

For the first of these interesting iteration, i.e., the first read $\mathbf{r}^*$ of a value from the FIFO, the number of tokens in the FIFO is equal to the number of writes that have occurred before that read, i.e., $\#W(\mathbf{s}, \mathbf{r}^*)$, where W is as defined in (18). For any other read $\mathbf{r} \in S$, we will also have read some values from the FIFO. In particular, we will have read all values that were written up to and including the value that we needed in the previous read $\mathbf{r}'$. Let W' map a given read $\mathbf{r}$ to all these previously read values. Then the number of tokens in the FIFO right before $\mathbf{r}$ is $\#W(\mathbf{s}, \mathbf{r}) - \#W'(\mathbf{s}, \mathbf{r})$. The relation W' can be computed as

$$W' = \{\, \mathbf{r} \to \mathbf{w}'' \mid \exists \mathbf{r}' : \mathbf{r} \in S \wedge (\mathbf{r}, \mathbf{r}') \in P \wedge (\mathbf{r}', \mathbf{w}') \in M \wedge \mathbf{w}'' \preccurlyeq \mathbf{w}' \,\},$$

where $P = P(S)$ is the relation that maps a read in S to the previous read in S, as defined in (17). As before, the relation P can be computed using Operation 1 (Lexicographic Maximum). As a side effect, we obtain a polyhedral set E containing the first read $\mathbf{r}^*$. The counts can be computed using Operation 3 (Number of Image Elements) and finally Operation 4 (Upper Bound on a Quasi-polynomial) needs to be applied to obtain a bound on the FIFO buffer size that is valid for all reads.

As for the internal buffer, the number of tokens in this buffer after a read $\mathbf{r}$ from the FIFO is equal to the number of subsequent reads that read something (from the internal buffer) that was written (to the FIFO) before the token that was actually

needed by $\mathbf{r}$, i.e.,

$$\#\{\, \mathbf{r} \to \mathbf{r}' \mid \exists \mathbf{w}, \mathbf{w}' : (\mathbf{w}, \mathbf{r}), (\mathbf{w}', \mathbf{r}') \in M \wedge \mathbf{w}' \prec \mathbf{w} \wedge \mathbf{r} \prec \mathbf{r}' \,\}.$$

Again, this number can be computed using Operation 3 and a bound on the buffer size can then be computed using Operation 4.

8.3 Accuracy of the Buffer Sizes

Although the buffer sizes computed using the methods described above are usually fairly close to the actual minimal buffer sizes, there are some possible sources of inaccuracies that we want to highlight in this section. These inaccuracies will always lead to over-approximations. That is, the computed bounds will always be safe. For internal channels, the only operation that can lead to over-approximations is Operation 4 (Upper Bound on a Quasi-polynomial). There are three causes for inaccuracies in this operation: the underlying technique [3] is defined over the rationals rather than the integers; in its basic form it only handles polynomials and not quasi-polynomials; and the technique itself will only return the actual maximum (rather than just an upper bound) if the maximum occurs at one of the extremal points of the domain. For external channels, we rely on a scheduling step to effectively make them internal. Since we only consider a limited set of possible schedulings, the derived buffer sizes may in principle be much larger than the absolute minimal buffer sizes that still allow for a deadlock-free schedule.

9 Summary

In this chapter we have seen how to automatically construct a polyhedral process network from a sequential program that can be represented in the polyhedral model. The basic polyhedral tools used in this construction are parametric integer programming, emptiness check, parametric counting, computing parametric upper bounds and polyhedral scanning. The processes in the network correspond to the statements in the program, while the communication channels are computed using dependence analysis. Several types of channels can be identified by solving a number of emptiness checks and/or counting problems. Safe buffer sizes for the channels can be obtained by first computing a global schedule and then computing an upper bound on the number of elements in each channel at each iteration of this schedule.

Acknowledgements This work was supported by FWO-Vlaanderen, project G.0232.06N. The author would like to thank Maurice Bruynooghe and Sjoerd Meijer for their feedback on earlier versions of this chapter.

References

1. Amarasinghe, S.P., Anderson, J.M., Lam, M.S., Tseng, C.W.: The SUIF compiler for scalable parallel machines. In: Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing (1995)
2. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 7–16. IEEE Computer Society, Washington, DC, USA (2004). DOI 10.1109/PACT.2004.11
3. Clauss, P., Fernández, F.J., Gabervetsky, D., Verdoolaege, S.: Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. IEEE Transactions on Very Sarge Scale Integration (VLSI) Systems (2009). Accepted
4. Darte, A., Schreiber, R., Villard, G.: Lattice-based memory allocation. IEEE Trans. Comput. **54**(10), 1242–1257 (2005). DOI 10.1109/TC.2005.167
5. Devos, H., Verdoolaege, S., Van Campenhout, J., Stroobandt, D.: Bounds on quasi-polynomials for static program analysis (2010). Manuscript in preparation
6. Feautrier, P.: Parametric integer programming. Operationnelle/Operations Research **22**(3), 243–268 (1988)
7. Feautrier, P.: Dataflow analysis of array and scalar references. International Journal of Parallel Programming **20**(1), 23–53 (1991)
8. Kahn, G.: The semantics of a simple language for parallel programming. In: Proc. of the IFIP Congress 74, pp. 471–475. North-Holland Publishing Co. (1974)
9. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The Omega library. Tech. rep., University of Maryland (1996)
10. Kelly, W., Pugh, W., Rosser, E.: Code generation for multiple mappings. In: Frontiers'95 Symposium on the frontiers of massively parallel computation. McLean (1995)
11. Kienhuis, B., Rijpkema, E., Deprettere, E.: Compaan: deriving process networks from matlab for embedded signal processing architectures. In: CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign, pp. 13–17. ACM Press, New York, NY, USA (2000). DOI 10.1145/334012.334015
12. Meister, B., Verdoolaege, S.: Polynomial approximations in the polytope model: Bringing the power of quasi-polynomials to the masses. In: J. Sankaran, T. Vander Aa (eds.) Digest of the 6th Workshop on Optimization for DSP and Embedded Systems, ODES-6 (2008)
13. Pop, S., Cohen, A., Bastoul, C., Girbal, S., Jouvelot, P., Silber, G.A., Vasilache, N.: Graphite: Loop optimizations based on the polyhedral model for gcc. In: 4th GCC Developper's Summit. Ottawa, Canada (2006)
14. Schrijver, A.: Combinatorial Optimization - Polyhedra and Efficiency. Springer (2003)
15. Schweitz, E., Lethin, R., Leung, A., Meister, B.: R-stream: A parametric high level compiler. In: J. Kepner (ed.) Proceedings of HPEC 2006, 10th annual workshop on High Performance Embedded Computing. Lincoln Labs, Lexington, MA (2006)
16. Turjan, A.: Compaan - A Process Network Parallelizing Compiler. VDM Verlag (2008)
17. Verdoolaege, S.: An integer set library for program analysis (2009). ACES symposium, Edegem, 7-8 September
18. Verdoolaege, S., Beyls, K., Bruynooghe, M., Catthoor, F.: Experiences with enumeration of integer projections of parametric polytopes. In: R. Bodik (ed.) Proceedings of 14th International Conference on Compiler Construction, Edinburgh, Scotland, *Lecture Notes in Computer Science*, vol. 3443, pp. 91–105. Springer-Verlag, Berlin (2005). DOI 10.1007/b107108
19. Verdoolaege, S., Nikolov, H., Stefanov, T.: pn: A tool for improved derivation of process networks. EURASIP Journal on Embedded Systems, special issue on Embedded Digital Signal Processing Systems **2007** (2007). DOI 10.1155/2007/75947
20. Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V., Bruynooghe, M.: Counting integer points in parametric polytopes using Barvinok's rational functions. Algorithmica **48**(1), 37–66 (2007). DOI 10.1007/s00453-006-1231-0