Universiteit
Leiden
The Netherlands

# Estimation and Optimization of the Performance of Polyhedral Process Networks

Haastregt, S. van

**Citation**
Haastregt, S. van. (2013, December 17). *Estimation and Optimization of the Performance of Polyhedral Process Networks*. Retrieved from https://hdl.handle.net/1887/22911

| | |
|---|---|
| Version: | Corrected Publisher's Version |
| License: | Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden |
| Downloaded from: | https://hdl.handle.net/1887/22911 |

**Note:** To cite this publication please use the final published version (if applicable).

Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/22911 holds various files of this Leiden University dissertation.

**Author:** Haastregt, Sven Joseph Johannes van
**Title:** Estimation and optimization of the performance of polyhedral process networks
**Issue Date:** 2013-12-17

CHAPTER 2

# BACKGROUND

In this chapter, we introduce concepts and notations that are used throughout this thesis. In Section 2.1, we introduce the polyhedral model which we employ for analysis of programs. In Section 2.2, we review various models of computation that are widely employed to represent applications. We focus on the polyhedral process network model of computation employed by Daedalus and review in Section 2.3 how such networks can be derived from a particular class of sequential programs. In Section 2.4, we review how processes of polyhedral process networks can be implemented in hardware.

## 2.1 Polyhedral Model

The streaming applications that we consider in this thesis are data-driven: a sequence of computations is repeatedly applied on an incoming data stream, such as a stream of images produced by a video camera. These streaming applications spend most of their execution time in loops that perform computations on data stored in arrays. For example, edge detection algorithms consist of loop nests that iterate over all pixels of the input image that is stored in a 2-dimensional array. These loop nests are the primary candidates for optimization, since most of the time is spent there. To select and apply optimizations, one needs means to reason about iterations of loops and relations between statements contained in loop nests. This is possible with the polyhedral model [Pug91, Fea96] which is employed by modern compilers like GCC [PCB+06] and LLVM/Polly [GZA+11]. The polyhedral model allows a compact representation of loop nests while providing sufficient means to express advanced optimizations such as loop skewing [SKD02]. We use polyhedra to compactly represent loop nests

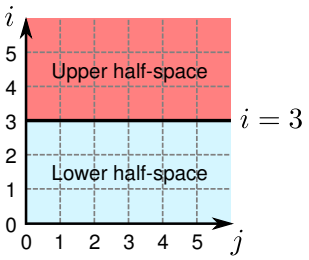



Figure 2.1: A 1-dimensional hyperplane (i.e., a line) $H_1 = \{(j, i) \in \mathbb{Q}^2 \mid i = 3\}$ dividing a 2-dimensional space.

in the polyhedral model. A polyhedron can be defined using hyperplanes.

**Definition 2.1** (Hyperplane)**.**
A *hyperplane* $H$ is a subspace of dimension $d - 1$ inside a $d$-dimensional space, that is,

$$H = \{\mathbf{x} \in \mathbb{Q}^d \mid \mathbf{a}^T\mathbf{x} = c\},$$

where $\mathbf{a}$ is a non-zero vector of size $d$ and $c$ is a constant [Rij02].

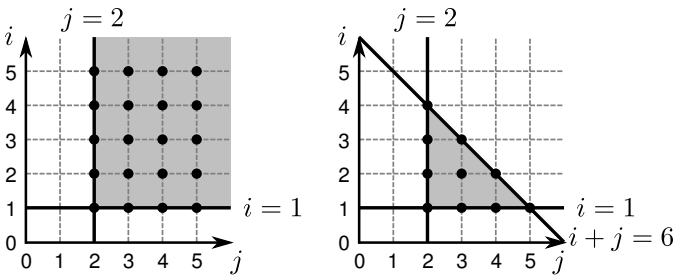
A hyperplane is a generalization of a conventional 2-dimensional plane to $n \in \mathbb{N}$ dimensions. A 1-dimensional hyperplane dividing a 2-dimensional space is shown in Figure 2.1. A hyperplane divides a space into an upper and a lower *half-space*. We distinguish *open half-spaces* which do not include the dividing hyperplane itself, and *closed half-spaces* which include the dividing hyperplane. We use hyperplanes to define subspaces of $\mathbb{Q}^d$, known as rational polyhedra:

**Definition 2.2** (Rational Polyhedron)**.**
A *rational polyhedron* $\mathcal{P}$ is a subspace of $\mathbb{Q}^d$ that is bounded by a finite set of $m$ hyperplanes, that is,

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

where $A$ is an integral $m \times d$ matrix and $\mathbf{c}$ is an integral vector of size $m$ [Ver10].

The shaded rectangular area in Figure 2.2a represents a 2-dimensional rational polyhedron that is bounded by the closed upper half-spaces of two 1-dimensional hyperplanes $i = 1$ and $j = 2$. This rational polyhedron extends into infinity in both dimensions. By adding the closed lower half-space of the hyperplane $i + j = 6$ to the bounds, we obtain a rational polyhedron that is fully enclosed by its bounding hyperplanes, as shown in Figure 2.2b. Such an enclosed rational polyhedron containing a finite number of integral points is called a *rational polytope*.




a) Rational polyhedron.

b) Rational polytope.

```
for (j=2;j<=5;j++) {
  for (i=1;i<=6-j;i++) {
    F(...);
  }
}
```

c) Loop nest represented by b).

```
for (j=2;j<=N;j++) {
  for (i=1;i<=6-j;i++) {
    F(...);
  }
}
```

d) Loop with parametric bound.

```
for (j=2;j<=5;j++) {
  for (i=1;i<=6-j;i++) {
    if (i%2 == 0)
      F(...);
  }
}
```

e) Statement with modulo guard.

Figure 2.2: a) A 2-dimensional rational polyhedron; b) a 2-dimensional rational polytope; c) a loop nest of depth two that can be represented by the 2-dimensional rational polytope given in b); d) a loop nest where the outer loop has a parametric upper bound; and e) a statement with a modulo guard.

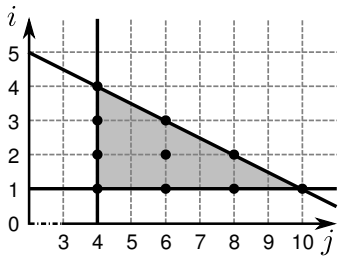
**Definition 2.3** (Parametric Rational Polyhedron)**.**
A *parametric rational polyhedron* $\mathcal{P}(\mathbf{s})$ is a family of rational polyhedra in $\mathbb{Q}^d$ that is parametrized by parameters $\mathbf{s} \in \mathbb{Q}^n$:

$$\mathbf{s} \mapsto \mathcal{P}(\mathbf{s}) = \{\mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} + B\mathbf{s} \geq \mathbf{c}\},$$

where $A$ is an integral $m \times d$ matrix, $B$ is an integral $m \times n$ matrix, and $\mathbf{c}$ is an integral vector of size $m$ [Ver10].

A parametric rational polyhedron can represent a loop nest that iterates over a finite, possibly parameterized set of iterations. By assuming that the iterators of such a loop nest are integers, we can represent a loop nest as a set of integral points in a (parametric) rational polyhedron. For example, the loop nest shown in Figure 2.2c can be represented by the rational polytope shown in Figure 2.2b. Each iteration of the loop nest has a corresponding point in the rational polytope. The loop nest shown in Figure 2.2d can be represented by a parametric rational polytope.

When for example a statement is guarded with an expression containing a modulo operator, we are interested in only a subset of the points of a parametric rational polyhedron. In the example shown in Figure 2.2d, function `F` is called only for even values of iterator `i`. We define the polyhedral set to represent a subset of points in a parametric rational polyhedron.




$$\begin{aligned}\mathcal{S} &= \left\{(j,i) \in \mathbb{Z}^2 \mid \exists e \in \mathbb{Z} : \begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ -1 & -2 \end{pmatrix} \cdot \begin{pmatrix} j \\ i \end{pmatrix} \right. \\ &\qquad \left. + \begin{pmatrix} 2 \\ -2 \\ 0 \\ 0 \\ 0 \end{pmatrix} \cdot (e) \geq \begin{pmatrix} 0 \\ 0 \\ 4 \\ 1 \\ -12 \end{pmatrix} \right\} \\ &= \{(j,i) \in \mathbb{Z}^2 \mid \exists e \in \mathbb{Z} : 2e = j \wedge j \geq 4 \wedge i \geq 1 \wedge \\ &\qquad 2i + j \leq 12\}\end{aligned}$$

Figure 2.3: Example polyhedral set.

**Definition 2.4** (Polyhedral Set)**.**
A *polyhedral set* $\mathcal{S}$ is a finite union of basic integer sets, $\mathcal{S} = \bigcup_i \mathcal{S}_i$, of type $\mathbb{Q}^n \rightarrow 2^{\mathbb{Q}^d}$, where each basic integer set $\mathcal{S}_i$ is defined as

$$\mathcal{S}_i = \mathbf{s} \mapsto \mathcal{S}_i(\mathbf{s}) = \{\mathbf{x} \in \mathbb{Z}^d \mid \exists \mathbf{z} \in \mathbb{Z}^e : A\mathbf{x} + B\mathbf{s} + D\mathbf{z} \geq \mathbf{c}\},$$

where $A$ is an integral $m \times d$ matrix, $B$ is an integral $m \times n$ matrix, $D$ is an integral $m \times e$ matrix, and $\mathbf{c}$ is an integral vector of size $m$. The *parameter domain* of $\mathcal{S}$, $\{\mathbf{s} \in \mathbb{Z}^n \mid \mathcal{S}(\mathbf{s}) \neq \emptyset\}$, is a polyhedral set containing all parameter values $\mathbf{s}$ for which $\mathcal{S}$ is non-empty. A polyhedral set with an empty parameter domain (i.e., $n = 0$) is called a *non-parametric* polyhedral set, and denoted with "$\mathbf{s} \mapsto$" omitted. The parameter domain of a polyhedral set is always non-parametric [Ver10].

The polyhedral set depicted in Figure 2.3 contains only a subset of the integral points of its bounding rational polytope. In particular, it only contains the integral points for even values of $j$, which can be expressed as "$j \bmod 2 = 0$". Such constraints are enforced using the existentially quantified variables $\mathbf{z}$ in Definition 2.4. For example, the constraint "$j \bmod 2 = 0$" is represented by a condition $2e = j$ and the requirement that $e$ is integral.

To allow reasoning about the execution order of different iterations of a program, we define the lexicographic order on the points of a polyhedral set:

**Definition 2.5** (Lexicographic Order)**.**
The *lexicographic order* is a total order on the elements of a polyhedral set. An element $\mathbf{a}$ is lexicographically smaller than an element $\mathbf{b}$, denoted as $\mathbf{a} \prec \mathbf{b}$, if

$a_i < b_i$ for the first dimension $i$ in which both elements differ, or, equivalently,

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

For example, an element $a = (2, 3, 5)$ is lexicographically smaller than an element $b = (2, 4, 0)$, because the first difference between both elements is in the second dimension, and the value 3 in the second dimension of $a$ is less than the value 4 in the second dimension of $b$.

Loop optimizations such as skewing transform iteration domains that we represent using polyhedral sets. A transformation of a polyhedral set can be expressed as a relation between the original polyhedral set and the transformed polyhedral set. We define the polyhedral map to express such relations:

**Definition 2.6** (Polyhedral Map)**.**
A *polyhedral map* $\mathcal{M}$ is a finite union of basic polyhedral maps, $\mathcal{M} = \bigcup_i \mathcal{M}_i$, of type $\mathbb{Q}^n \to 2^{\mathbb{Q}^{d_1+d_2}}$, where each basic polyhedral map is defined as

$$\begin{aligned}\mathcal{M}_i &= \mathbf{s} \mapsto \mathcal{M}_i(\mathbf{S}) \\ &= \{(\mathbf{x_1}, \mathbf{x_2}) \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \exists \mathbf{z} \in \mathbb{Z}^e : A_1\mathbf{x_1} + A_2\mathbf{x_2} + B\mathbf{s} + D\mathbf{z} \geq \mathbf{c}\},\end{aligned}$$

where $A_1$ is an integral $m \times d_1$ matrix, $A_2$ is an integral $m \times d_2$ matrix, $B$ is an integral $m \times n$ matrix, $D$ is an integral $m \times e$ matrix, and $\mathbf{c}$ is an integral vector of size $m$ [Ver10].

The polyhedral set

$$\mathbf{s} \mapsto \{\mathbf{x_1} \in \mathbb{Z}^{d_1} \mid \exists \mathbf{x_2} \in \mathbb{Z}^{d_2} : (\mathbf{x_1}, \mathbf{x_2}) \in M(\mathbf{s})\}$$

is the *domain* of a polyhedral map $M$. The polyhedral set

$$\mathbf{s} \mapsto \{\mathbf{x_2} \in \mathbb{Z}^{d_2} \mid \exists \mathbf{x_1} \in \mathbb{Z}^{d_1} : (\mathbf{x_1}, \mathbf{x_2}) \in M(\mathbf{s})\}$$

is the *range* of a polyhedral map $M$. In this thesis, we denote polyhedral maps as

$$\mathcal{M} = \mathbf{s} \mapsto \{\mathbf{x_1} \to \mathbf{x_2} \mid \dots\}.$$

An example polyhedral map consisting of only one basic polyhedral map is

$$\mathcal{M}_1 = \{(j_1, i_1) \to (j_2, i_2) \mid j_2 = 2j_1 \wedge i_2 = i_1\}. \tag{2.1}$$

We use polyhedral maps to manipulate points or polyhedral sets by *application* of the

polyhedral map. For example, applying $\mathcal{M}_1$ to a point $(2, 1)$ yields $(4, 1)$, denoted as

$$\mathcal{M}_1(2, 1) = (4, 1).$$

If we apply this polyhedral map to the polyhedral set of Figure 2.2b, that is, if we compute $\mathcal{M}_1(\mathcal{S}_1)$, we obtain the polyhedral set depicted in Figure 2.3. The points in this new polyhedral set result from application of $\mathcal{M}_1$ to each point in the original polyhedral set $\mathcal{S}_1$.

We sometimes need to know the size of a polyhedral set or map, for example to judge whether a certain transformation is beneficial to a given program. The number of elements in a polyhedral set or polyhedral map is given by the cardinality:

**Definition 2.7** (Cardinality)**.**
The *cardinality* of a polyhedral set $\mathcal{S}$, denoted as $|\mathcal{S}|$, represents the number of elements in $\mathcal{S}$.
The cardinality of a polyhedral map $\mathcal{M}$, denoted as $|\mathcal{M}|$, represents the number of elements in the range of $\mathcal{M}$ associated to any element in the domain of $\mathcal{M}$.

We use the `barvinok` library to analytically determine the cardinality of polyhedral sets and maps [VSB+07, Ver03a]. The cardinality is expressed as a piecewise quasipolynomial. A piecewise quasipolynomial consists of one or more quasipolynomials:

**Definition 2.8** (Quasipolynomial)**.**
A *quasipolynomial* $q(\mathbf{x})$ is a polynomial expression in greatest integer parts of affine expressions of variables in $\mathbf{x}$. The coefficient of each term may include a constant integer division [Ver10].

**Definition 2.9** (Piecewise Quasipolynomial)**.**
A *piecewise quasipolynomial* $q(\mathbf{x})$ consists of one or more quasipolynomials. Each quasipolynomial $q_i(\mathbf{x})$ is defined only for a disjoint piece $\mathcal{D}_i$ of a domain $\mathcal{D}$. For a given point $\mathbf{x} \in \mathcal{D}$, the piecewise quasipolynomial evaluates to

$$q(\mathbf{x}) = \begin{cases} q_i(\mathbf{x}) & \text{if } \mathbf{x} \in \mathcal{D}_i, \\ 0 & \text{otherwise} \end{cases} \quad \text{[Ver10]}.$$

For example, the cardinality of the polyhedral set $\mathcal{S}_2$ of Figure 2.3 is expressed using the piecewise quasipolynomial

$$|\mathcal{S}_2| = \begin{cases} 10 & \text{if } 1 \geq 0 \end{cases}.$$

The cardinality of $\mathcal{S}_2$ is constant because all bounding hyperplanes are constant. Therefore, the cardinality is not dependent on any parameters or variables and con-

sists of only one piece that is selected using the tautology $1 \geq 0$. The quasipolynomial has a constant value of 10, as $\mathcal{S}_2$ consists of 10 points.

The cardinality of the polyhedral map $\mathcal{M}_1$ of Equation (2.1) is expressed using the piecewise quasipolynomial

$$|\mathcal{M}_1|(j_1, i_1) = \begin{cases} 1 & \text{if } (j_1, i_1) \in \mathbb{Z}^2, \\ 0 & \text{otherwise.} \end{cases}$$

This means that applying $\mathcal{M}_1$ to any point $(j_1, i_1)$ that is in $\mathbb{Z}^2$ always yields exactly one new point $(j_2, i_2)$.

## 2.2 Models of Computation

Designers specify the behavior of a system in a structured way using a *Model of Computation* (*MoC*). To facilitate programming of multi-processor systems, a parallel MoC is needed such that the tasks for each processor and the communication and synchronization mechanisms can be specified. Different MoCs have been proposed and evaluated for their use in design automation in literature [LSV98, JS05]. For example, HDL simulators often employ a timed discrete-event MoC in which all events are ordered globally in time. A global ordering is often not desired for a multi-processor system because different parts of the system may execute in parallel. Our interest is in untimed dataflow process network based MoCs such as Kahn Process Networks (KPNs) defined by Kahn [Kah74]. The dataflow-based MoCs that we consider in this thesis have several properties that make them attractive for specification of multi-processor systems [SZT+04]. One desirable property is deterministic behavior, such that a given input sequence always results in the same output sequence regardless of variations in computation or communication times. Another desirable property is that each task behaves autonomously, such that each processor of a multi-processor system can be considered in isolation. This allows designers to better cope with complex multi-processor systems.

Many different specializations of dataflow process network based MoCs have been proposed in literature for the design of streaming applications. A major reason for the abundance of different specializations is to allow different tradeoffs of expressiveness against analyzability. With *expressiveness* of a model we refer to the ability to express an application in that model in a succinct way. Although more general models often can be converted to more specialized equivalent models, such a conversion often increases the size of the application model making it no longer succinct. With *analyzability* of a model we refer to the existence and complexity of compile-time analysis algorithms to compute for example static schedules, buffer sizes, or

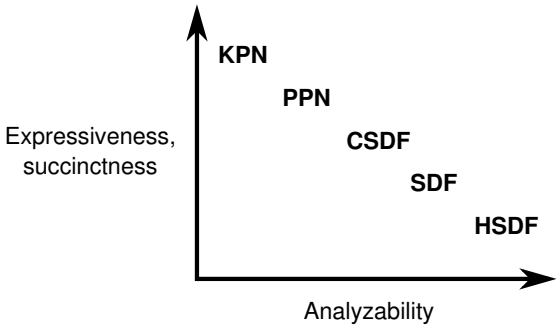



Figure 2.4: Different models and their expressiveness and analyzability.

throughput. In Figure 2.4, we depict five different models and compare their expressiveness and analyzability. For example, many applications can be expressed in the KPN MoC, but due to the genericity of the model, the compile-time analyzability is limited. In contrast, the HSDF model has a lower expressiveness but this allows for full analyzability. We now review four dataflow-based models that we use in the remainder of this thesis for specification and analysis of MPSoCs: the HSDF, SDF, CSDF, and PPN models of computation.

### 2.2.1 Homogeneous Synchronous Dataflow

The most restricted model of computation that we consider in this thesis is the homogeneous synchronous dataflow model, which is also known as the single-rate dataflow model [GGS$^+$06]. The more generic models that we discuss later extend the homogeneous synchronous dataflow model. We use the following definition, in line with the notation used by e.g. Moreira et al. [MBGS10]

**Definition 2.10** (Homogeneous Synchronous Dataflow Graph)**.**
A *Homogeneous Synchronous Data Flow* (*HSDF*) graph is a directed graph defined by a tuple $(V, E, t, d)$, where

- $V$ is a set of vertices representing computation *nodes*,
- $E$ is a set of *edges* representing communication channels that carry *tokens*,
- $t(i), i \in V$ represents the time needed for a single execution of node $i$, and
- $d(e), e \in E$ represents the number of *initial tokens* on edge $e$, also referred to as the *delay* of edge $e$.

An HSDF graph consisting of four nodes and six edges is shown in Figure 2.5. Shown in the upper half of each node is a label that we assign for convenient referencing. Shown in the lower half of each node is the node's execution time $t(i)$. For

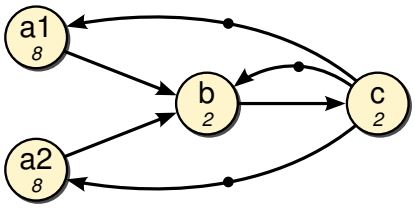



Figure 2.5: An HSDF graph.

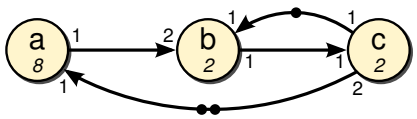
example, node $b$ has an execution time $t(b) = 2$ time units. Initial tokens $d(e)$ for each edge are shown as dots on the edges. For example, the edge connecting node $c$ to *a2* contains one initial token, that is, $d(c \rightarrow a2) = 1$. For clarity reasons, we may visualize multiple initial tokens by a single dot and a number above or below the dot.

Edges transfer units of data referred to as *tokens*. A node is said to be *enabled* if each of its incoming edges contains at least one token. An enabled node is said to *fire* when it consumes a token from each incoming edge, performs a computation on these tokens, and then produces a token on each of its outgoing edges. If none of the nodes is enabled, then the graph is in a *deadlock* state. If all nodes of a graph can fire infinitely often, then the graph is *live*. An HSDF graph is said to be *consistent* if every token written to an edge is eventually consumed, such that the graph can be executed under bounded memory conditions. An *iteration* of an HSDF graph is defined as each node executing exactly once.

Different firings of a node may start at the same time, such that overlapped execution between firings of the same node occurs. For example, if edge $c \rightarrow a1$ in Figure 2.5 would contain two initial tokens, then two firings of *a1* can start simultaneously. Such overlapped execution of firings of the same node is referred to as *auto-concurrency*. By adding an edge from a node to itself, referred to as a *selfloop*, we can regulate auto-concurrency of a node. The number of initial tokens on that selfloop limits the number of parallel firings. By putting one initial token on the selfloop, auto-concurrency is fully prevented. In such a case, the node consumes the initial token from the selfloop at the first firing, and only produces a new token on the edge once it finishes its firing. The node is not enabled for any subsequent firings until the first firing has finished, meaning no overlap between firings occurs.

### 2.2.2 Synchronous Dataflow

HSDF graphs are a special case of the more general synchronous dataflow graphs defined by Lee and Messerschmitt [LM87].




$$\Gamma = \begin{bmatrix} 1 & -2 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \\ -1 & 0 & 2 \end{bmatrix}$$

Figure 2.6: An SDF graph and its topology matrix $\Gamma$.

**Definition 2.11** (Synchronous Dataflow Graph)**.**
A *Synchronous Data Flow* (*SDF*) graph is a directed graph defined by a tuple $(V, E, t, d, p, c)$, where

- $V$, $E$, $t$, and $d$ follow those in Definition 2.10,
- $p(e), e \in E$ represents the number of tokens placed on edge $e$ when the corresponding source node fires, referred to as the *production rate*, and
- $c(e), e \in E$ represents the number of tokens consumed from edge $e$ when the corresponding destination node fires, referred to as the *consumption rate*.

An SDF graph consisting of three nodes and four edges is shown in Figure 2.6. The numbers depicted at the location where edges connect to nodes represent the production and consumption rates. For example, when node $c$ fires it consumes $c(b \rightarrow c) = 1$ token from edge $b \rightarrow c$, and it produces $p(c \rightarrow b) = 1$ token on edge $c \rightarrow b$ and $p(c \rightarrow a) = 2$ tokens on edge $c \rightarrow a$.

The structure and production and consumption rates of an SDF graph are compactly represented by a *topology matrix* $\Gamma$. The columns of $\Gamma$ represent the nodes and the rows of $\Gamma$ represent the edges. A positive entry $\Gamma(i, j)$ means that node $j$ produces $\Gamma(i, j)$ tokens on edge $i$. A negative entry $\Gamma(i, j)$ means that node $j$ consumes $-\Gamma(i, j)$ tokens from edge $i$. A zero entry $\Gamma(i, j)$ means that node $j$ does not read or write to edge $i$. A selfloop can be represented in $\Gamma$ by the net difference between production and consumption [LM87, p. 27].

An SDF graph can be converted into an equivalent HSDF graph [SB00, Chapter 3]. However, such a conversion may cause an exponential increase in the number of nodes in the worst case. The HSDF graph of Figure 2.5 is the result of converting the SDF graph of Figure 2.6. An *iteration* of an SDF graph is defined as each node of the equivalent HSDF graph executing exactly once. If an SDF graph is consistent, then a *repetition vector* $\mathbf{q}$ exists which contains for every node the number of times the node has to fire to return the SDF graph to its initial state. The repetition vector is the smallest non-trivial positive integer vector that is a valid solution to the *balance equation* $\Gamma \cdot \mathbf{q} = \mathbf{0}$.

For the graph of Figure 2.6, the smallest non-trivial solution to the balance equation is the repetition vector $\mathbf{q} = [2, 1, 1]^T$. This means that if node $a$ fires twice, node $b$

fires once, and node $c$ fires once, then the number of initial tokens on each edge is the same as before the execution of these four firings.

An SDF node always consumes tokens from all input edges and produces tokens on all output edges during a firing. Consequently, the SDF model cannot describe a node that for example reads from different input ports during different firings. This means that applications in which such behavior occurs cannot be modeled as an SDF graph.

### 2.2.3 Cyclo-Static Dataflow

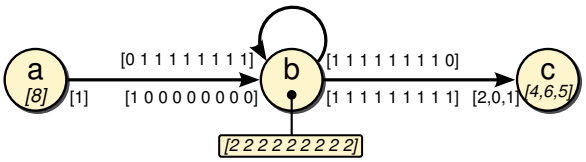
An extension to the SDF model that allows such behavior is the cyclo-static dataflow model [BELP96]. This model allows a compact representation of applications with a cyclically changing, but predefined behavior.

**Definition 2.12** (Cyclo-Static Dataflow Graph)**.**
A *Cyclo-Static Data Flow* (*CSDF*) graph is a directed graph defined by a tuple $(V, E, \mathbf{f}, \mathbf{t}, d, \mathbf{p}, \mathbf{c})$, where

- $V$, $E$, and $d$ follow those in Definition 2.10,
- $\mathbf{f}_j, j \in V$ represents the function repertoire for node $j$, which is a sequence of functions $[f_j(0), f_j(1), \cdots, f_j(S_j - 1)]$ of *phase length* $S_j$,
- $\mathbf{t}_j(i), j \in V$ represents the time needed for an execution of function $i$ in $\mathbf{f}_j$,
- $\mathbf{p}_e(i), e \in E$ is a sequence of integers representing the number of tokens produced on edge $e$ after $e$'s source node fires its $i$-th function, and
- $\mathbf{c}_e(i), e \in E$ is a sequence of integers representing the number of tokens consumed from edge $e$ before $e$'s destination node fires its $i$-th function.

Each node in a CSDF graph executes the functions in its function repertoire in a cyclic fashion. At the start of the $n$-th firing of node $j$, $\mathbf{c}_e(n \bmod S_j)$ tokens are consumed from incoming edge $e$. Then, function $f_j(n \bmod S_j)$ is executed which takes $\mathbf{t}_j(n \bmod S_j)$ time units. After the function finishes execution, $\mathbf{p}_e(n \bmod S_j)$ tokens are produced on outgoing edge $e$.

Similar to the topology matrix of an SDF graph, we can define a *topology matrix* $\Gamma$ for a CSDF graph. A positive entry $\Gamma(i, j)$ means that node $j$ produces in total $\Gamma(i, j)$ tokens on edge $i$ for a complete execution sequence, that is, $\Gamma(i, j) = \sum_{k=0}^{S_j - 1} \mathbf{p}_i(k)$. A negative entry $\Gamma(i, j)$ means that node $j$ consumes in total $\Gamma(i, j)$ tokens from edge $i$ for a complete execution sequence, that is, $\Gamma(i, j) = -\sum_{k=0}^{S_j - 1} \mathbf{c}_i(k)$. All other entries are zero.




$$\Gamma = \begin{bmatrix} 1 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 9 & -3 \end{bmatrix}$$

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

Figure 2.7: A CSDF graph, its topology matrix $\Gamma$, and its phase matrix $S$.

To obtain the repetition vector of a CSDF graph, one first solves the *balance equation* $\Gamma \cdot \mathbf{r} = \mathbf{0}$. The repetition vector then equals

$$\mathbf{q} = S \cdot \mathbf{r}, \qquad \text{where } S(i,j) = \begin{cases} S_j & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

Matrix $S$ in Equation (2.2), whose diagonal contains the phase lengths of all nodes, is referred to as the *phase matrix*.

A CSDF graph consisting of three nodes and three edges is shown in Figure 2.7. The function repertoire of node *c* contains three functions with latencies 4, 6, and 5, as shown in the bottom part of the node. Thus, the phase length of node *c* $S_c = 3$. Node *c* has one incoming edge $b \rightarrow c$. In the $0 \pmod 3$-th execution of node *c*, two tokens are consumed from this edge; in the $1 \pmod 3$-th execution of node *c*, no tokens are consumed; and in the $2 \pmod 3$-th execution of node *c*, one token is consumed from this edge.

The topology matrix of the CSDF graph is shown in the upper right part of Figure 2.7. Since the CSDF graph contains a selfloop, the second row of $\Gamma$ consists entirely of zeros. The phase matrix of the CSDF graph is shown in the lower right part of Figure 2.7. For example, the lower right element of this matrix equals node *c*'s phase length $S_c = 3$. The smallest non-trivial solution to the balance equation is $\mathbf{r} = [1, 1, 3]^T$. Hence, the repetition vector of the CSDF graph $\mathbf{q} = [1, 9, 9]^T$.

The phase lengths and production and consumption patterns $\mathbf{p}$ and $\mathbf{c}$ may be large for applications that have mainly regular, but occasionally irregular behavior. This is for example found in image edge detection algorithms, whose behavior is regular for most pixels, but irregular for pixels at the image borders. Large phase lengths make a CSDF representation impractical for analysis and synthesis tools. We therefore present another model in which complex patterns can be captured in a compact way using the polyhedral model.

### 2.2.4 Polyhedral Process Networks

The Daedalus system-level design tool set that was introduced in Section 1.1 (cf. Figure 1.1) employs polyhedral process networks as its application model. The polyhedral process network model was first coined by Meijer et al. [MNS10] and was later formally defined by Verdoolaege [Ver10]. The definition of Verdoolaege differs from the classical definitions employed by the Compaan and Daedalus tools, as presented by for example Turjan [Tur07], Nikolov et al. [NSD08b], and Rijpkema [Rij02]. Throughout this thesis, we use the definition of the latter references. A conversion from the definition of Verdoolaege to the definition used by Daedalus is possible and is extensively used in the Daedalus tool flow [Ver03b].

**Definition 2.13** (Polyhedral Process Network)**.**
A *Polyhedral Process Network* (*PPN*) is a directed graph $(\mathcal{P}, \mathcal{E})$ where $\mathcal{P}$ is a set of vertices representing processes and $\mathcal{E}$ is a set of edges representing communication channels. Each process $p_i \in \mathcal{P}$ is characterized by:

- a function $F_i$,
- a process dimensionality $d_i$,
- a polyhedral set $D_i \subseteq \mathbb{Z}^{d_i}$ defining the process' domain.
- a set of *input ports* $IP_i$, where the $k$-th input port $IP_i^k$ is bound to an input argument of $F_i$ and has an associated *Input Port Domain* (*IPD*) $IPD_i^k \subseteq D_i$, and
- a set of *output ports* $OP_i$, where the $k$-th output port $OP_i^k$ is bound to an output argument of $F_i$ and has an associated *Output Port Domain* (*OPD*) $OPD_i^k \subseteq D_i$.

Each channel $c_i \in \mathcal{E}$ is characterized by:

- a source process $\sigma_i \in \mathcal{P}$,
- a destination process $\delta_i \in \mathcal{P}$,
- a source process' output port $OP_{\delta_i}^j$,
- a destination process' input port $IP_{\sigma_i}^k$,
- a polyhedral map $M_i \subseteq D_{\sigma_i} \times D_{\delta_i}$ mapping iterations from the destination process domain back to the source process domain.
- a channel type $T_i$, which is *FIFO*, *sticky FIFO*, or *out-of-order* (cf. Section 2.3.1), and
- a piecewise quasipolynomial $S_i$ representing the buffer size.

The parameters that occur in the process domains, channel maps and buffer sizes are *static*, meaning that their values are fixed at run-time. A more general model which

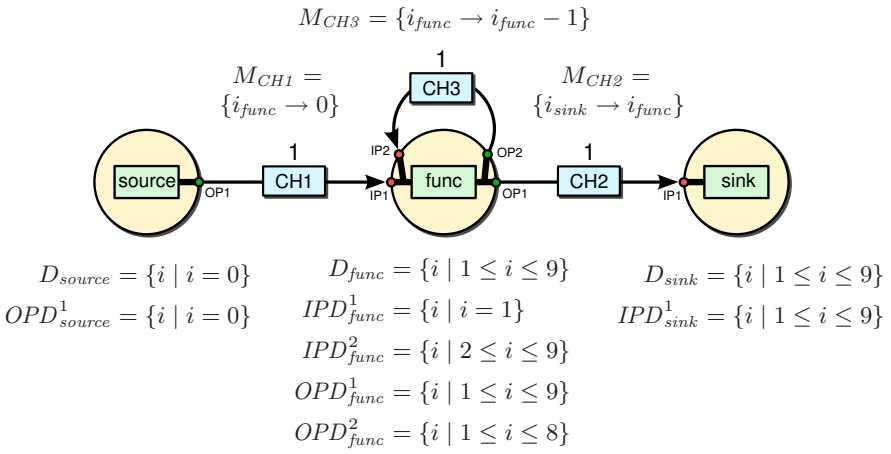



Figure 2.8: A polyhedral process network.

also includes *dynamic* parameters is the Parameterized Polyhedral Process Network ($P^3N$) model [ZNS11]. Such dynamic parameters enable the $P^3N$ model to cope with applications that adapt their behavior at runtime. Another related model is the Approximated Dependence Graph (ADG) [SD03]. The ADG model supports the class of weakly dynamic programs, which is more generic than the class of static affine nested loop programs that we consider in this thesis.

In this thesis, we are dealing with instances of PPNs for which all static parameters have known fixed values. We replace the static parameters by their fixed values, thereby removing the paramters, for the sake of simplicity.

An example PPN consisting of three processes and three channels is depicted in Figure 2.8. In this thesis, we only consider PPNs that consist of exactly one *connected component*. That is, if one replaces all directed edges in the graph by undirected edges, then a path from $u$ to $v$ exists for every pair of vertices $u, v$. The PPNs that we consider may contain zero or more *strongly connected component*s. A strongly connected component is a subgraph in which a path from any vertex in the subgraph to any other vertex in the subgraph exists.

If a process does not have any input ports, that is, $IP_i = \emptyset$, then the process is called a *source process*. Likewise, if a process does not have any output ports, that is, $OP_i = \emptyset$, then the process is called a *sink process*. The function of a process should be a *pure function*, that is, it should always yield the same output for a given input and it should not have any side effects. Exceptions to this requirement are source and sink processes, which often serve as an abstraction for the input and output interfaces of a system. As such, input and output operations are desired side-effects for functions of

source and sink processes.

In the PPN of Figure 2.8, *source* is a source process with one output port and *sink* is a sink process with one input port. The ports of a process are depicted by the dots on the border of each process. The output argument of the *source* function is connected to the output port of the process. Similarly, the input port of the *sink* process is connected to the input argument of the function. The *func* process has two input ports and two output ports. The *func* function has one input argument and one output argument. Both input ports connect to the same input argument and the output argument is connected to both output ports. Port multiplexing and demultiplexing is performed at run-time, as computations are distributed as a result of data flow analysis. Input and output tokens of a process need to be communicated from and to different processes at different iterations through process input and output ports.

The process and port domains are depicted below the processes in Figure 2.8. For example, the domain of the *sink* process consists of the integral points from 1 to 9. The IPD of its input port is identical to the process domain, which means that in every iteration a token is read from this input port.

The channels in the PPN of Figure 2.8 are shown as rectangles. All channels in this PPN are FIFO channels of size one, as denoted by the number above each channel. The map for each channel is shown above the channel sizes. Channel *CH1* maps an iteration of the *func* process to iteration $i = 0$ of the *source* process. Channel *CH2* maps iterations of the *sink* process to iterations of the *func* process. Channel *CH3* maps iterations of the *func* process to its previous iteration. In the remainder of this thesis, we depict channels in a more compact way as a single arrow with a number specifying the buffer size.

### Operational Semantics

Each process of a PPN executes autonomously according to a three-stage program that is executed for each point in the process domain: a read stage, an execute stage, and a write stage [ZI08]. This is an important property that we exploit throughout this thesis. First, in the *read stage*, the input arguments to the process function are read from the input ports whose IPD contains the current iteration. If the channel connected to an input port does not contain any tokens, then the process blocks until a token becomes available. Second, in the *execute stage*, the process function is executed with the input data obtained during the read stage. Third, in the *write stage*, the output arguments of the process function are written to the output ports whose OPD contains the current iteration. If the channel connected to an output port does not have sufficient room to store another token, then the process blocks until a free slot becomes available in the channel.

A process traverses the points in its iteration domain $D_i$ in the lexicographical order, in a sequential fashion. Thus, two iterations cannot start at the same time.

## 2.3 Derivation of PPNs from Sequential Programs

Polyhedral process networks can be derived automatically from sequential programs known as static affine nested loop programs [RDK00, VNS07].

**Definition 2.14** (Static Affine Nested Loop Program)**.**
A *static affine nested loop program* (*SANLP*) is a program consisting of statements enclosed by zero or more loops, where:

- all loops have a constant integral stride,
- loop bounds, if-conditions, and array index expressions are affine combinations of constants and enclosing loop iterators, and
- communication between statements is explicit, that is, statements do not exchange data through hidden variables.

The SANLP for the example of Figure 2.8 is shown in Figure 2.10. This PPN can be derived from the SANLP using the `c2pdg`, `pn`, and `pn2adg` tools from the `isa` tool set [Ver03b]. The tool flow is depicted in Figure 2.9. First, the `c2pdg` tool converts the SANLP into a Polyhedral Dependence Graph (PDG). This PDG contains the statements of the SANLP, the iteration domain of each statement, and the variable and array accesses performed by each statement. The `pn` tool extends the PDG with dependence information [VNS07] obtained using exact dataflow analysis [Fea91]. The `pn2adg` tool converts the extended PDG into an Approximated Dependence Graph (ADG). The PPN model that we introduced in Section 2.2.4 is a subset of this ADG model, as we do not handle dynamic parameters. We therefore consider the output of `pn2adg` as the actual PPN, assuming the input C code does not result in an ADG that lies beyond our PPN model. In this thesis, we refer to the consecutive execution of the `c2pdg`, `pn`, and `pn2adg` tools as PNGEN.

For each of the three function calls in the SANLP of Figure 2.10, PNGEN constructs a process. The domain of each process is derived from the for-loops and if-statements

Figure 2.9: PNGEN: Tool flow to convert a SANLP written in C into a PPN.

surrounding the function call. For function calls not enclosed in any for-loop, such as *source*, a 1-dimensional domain containing a single point is constructed.

PNGEN determines which processes should be interconnected by channels using exact dataflow analysis [Fea91] which is based on parametric integer programming techniques [Fea88]. For each read operation of a variable or array element, exact dataflow analysis reports the latest write operation that wrote the variable or array element. For example, at line 3 of Figure 2.10 we read array element `a[0]` during iteration $i = 1$. This array element is written in line 1. Therefore, PNGEN adds channel *CH1* to the PPN which connects the write operation (that is, the *source* process) to the read operation (that is, the *func* process). As another example, consider the read operations of array elements `a[1]` to `a[8]` at line 3. The read operations are performed during iterations $2 \leq i \leq 9$. Exact dataflow analysis reports that these array elements are written in line 3 during iterations $1 \leq i \leq 8$. Therefore, PNGEN adds channel *CH3* to the PPN which connects the *func* process to itself. The corresponding OPD contains the iterations $1 \leq i \leq 8$ during which the array elements are written. The corresponding IPD contains the iterations $2 \leq i \leq 9$ during which the array elements are read. This corresponds to the domains shown in Figure 2.8.

### 2.3.1 Channel Type Determination

Channels in a PPN are not all FIFOs, but need to be further classified [TKD07]. Each channel is either of type FIFO, sticky FIFO, or out-of-order, as defined in Defintion 2.13. To distinguish between out-of-order and (sticky) FIFO channel types, PNGEN first verifies if the values written to a channel are read in the same order as the order in which they were written. That is, communication over a channel is *in-order* if for any pair of write operations $(\mathbf{w_1}, \mathbf{w_2})$, the corresponding read operations $(\mathbf{r_1}, \mathbf{r_2})$ execute in the same order. If a pair of write operations exists for which the corresponding read operations occur in the opposite order, then the channel is marked as *out-of-order*.

```
1 source(&a[0]);
2 for (i=1; i<=9; i++) {
3   func(a[i-1], &a[i]);
4 }
5 for (i=1; i<=9; i++) {
6   sink(a[i]);
7 }
```

Figure 2.10: SANLP for the polyhedral process network of Figure 2.8.

In the example of Figure 2.10, all array elements are written and read exactly once. All communication is in-order, which causes the channels to be classified as FIFOs. If we would surround the for-loop containing the *sink* function call by another for-loop, then the elements of array `b` are written once and read multiple times. PNGEN employs a *reuse detection* technique to identify channels from which a single token is read multiple times. Reuse detection results in the construction of a *data reuse channel pair*, consisting of two FIFO channels. The first FIFO channel propagates data from the write operation to the first read operation. The second FIFO channel is a selfloop which propagates data from a read operation to a later read operation by the same process. If the same token is used by multiple subsequent iterations, then the reuse channel pair can be optimized further into a *sticky FIFO*. This means the selfloop is replaced by a register. We refer to Section 3.3 for an example of reuse detection, and we refer to Section 3.4 for an example of a sticky FIFO.

### 2.3.2 Buffer Size Computation

Each channel of a PPN has an associated buffer size specifying the number of tokens that can be stored. The buffer size has to be chosen under the following constraints. Choosing a buffer size that is too small results in an *artificial deadlock*, a condition in which none of the process can make progress because one or more processes are blocked on a write operation. Choosing an arbitrary large buffer size prevents artificial deadlocks, but increases memory cost. Therefore, careful selection of buffer sizes is required.

The buffer size computation performed by PNGEN consists of the following steps. First, PNGEN computes a global schedule for all processes. That is, it determines a single execution sequence containing all iterations of all processes. PNGEN ensures that the schedule is valid, meaning that each value is always written before it is read. Next, for each channel a buffer size is determined for the computed schedule. The schedule specifies a relative order on any pair of iterations from the same or a different process. Therefore, for a read iteration $\mathbf{r}$ (i.e., an iteration performing a read operation), the number of read iterations $nR(\mathbf{r})$ and the number of iterations performing a write iteration $nW(\mathbf{r})$ preceding $\mathbf{r}$ is known. The buffer size is then the maximal value of $nW(\mathbf{r}) - nR(\mathbf{r})$ over all read iterations $\mathbf{r}$. For the non-parametric PPNs that we consider, this maximal value can be computed symbolically or obtained using simulation. The symbolic approach works by computing an upper bound on a quasi-polynomial [CFGV09]. The simulation-based approach works by simulating the write and read iterations according to the schedule and tracking the maximal amount of tokens stored in the channel.

Computing minimal deadlock-free buffer sizes or a deadlock-free schedule is a non-

trivial optimization problem. PNGEN employs a greedy algorithm to compute a deadlock-free schedule. As a result, the buffer sizes computed by PNGEN are not guaranteed to be minimal, but at least a deadlock-free execution exists for the computed buffer sizes. The schedule is used for buffer size computation. Execution of a PPN is not bound to the schedule, as processes in a PPN only synchronize using blocking read and write operations.

## 2.4 Code Generation

We employ the ESPAM tool [NSD08b] to implement PPNs derived by PN. We illustrate the ESPAM tool flow in Figure 2.11. The input to ESPAM is a *System-Level Specification*, consisting of three components. First, we provide the *application specification* in the form of a PPN which is derived from a C program using PNGEN. Second, we provide a target *platform specification* describing the amount and types of processors and peripherals, and the type of interconnect. For example, the designer can populate a platform with programmable processors such as MicroBlazes and function-specific hardware IP cores. Third, we provide a *mapping specification* which maps the processes of the PPN onto the processors. The platform and mapping specifications are at a high level of abstraction, omitting low-level details such as the processor memory organization. ESPAM automatically elaborates the specifications to the required degree of detail. After elaboration and possible refinement, one of the backends at the right part of Figure 2.11 generates code which implements the specified system.

ESPAM offers different backends such that a given system-level specification can be implemented in different forms. We distinguish two classes of backends:

- **Implementation backends** produce a fully functional implementation of a

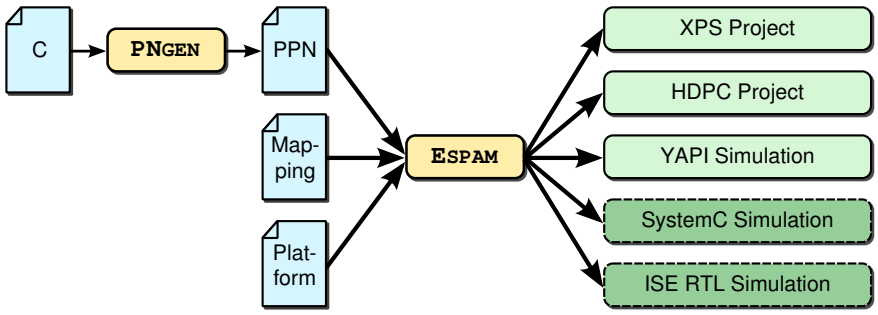



Figure 2.11: The ESPAM tool flow.

given system. ESPAM contains a Xilinx Platform Studio (XPS) backend which generates an FPGA project at the register transfer level, that can be synthesized using vendor-specific low-level synthesis tools to obtain a working prototype. ESPAM also contains a Heterogeneous Desktop Parallel Computing (HDPC) backend which generates a software implementation for a general purpose desktop computer containing for example a GPU device.

- **Simulation backends** produce an environment in which a given system can be simulated. For example, ESPAM contains a YAPI backend [KES+00] which enables fast functional verification of a parallelized application.

In Chapter 4 of this thesis, we present two new backends to ESPAM, which are depicted at the bottom right part of Figure 2.11. The SystemC simulation backend provides fast performance assessment. It works at a raised level of detail compared to RTL simulation, thereby increasing the simulation speed at the expense of lower accuracy. When more accurate performance and resource cost metrics are needed, the ISE backend can be employed. The ISE backend produces a Xilinx ISE project that implements the system entirely in VHDL. This project can be simulated and synthesized in the Xilinx ISE tool to obtain accurate execution time and resource usage metrics. Since the ISE simulation works at a more detailed level, obtaining metrics is more time-consuming compared to a SystemC simulation. For a small application like QR decomposition, a SystemC simulation takes a few seconds, whereas an ISE simulation may take about a minute.

### 2.4.1 Integrating Dedicated IP Cores

In systems with tight throughput constraints, performing all computations on programmable processors may not be feasible due to the limited performance of such processors. To increase the overall system throughput, designers offload the heaviest computations onto dedicated hardware *IP cores*. These IP cores are custom architectures that are optimized to perform a specific task. Such IP cores are traditionally written in RTL or may be generated from code written in a high-level language using a high-level synthesis tool. The LAURA Virtual Processor model was proposed to include such IP cores in the Daedalus tool flow [ZSKD03, NSD08a].

A representation of a process as C code is shown in the left part of Figure 2.12. The LAURA processor for this process is depicted in the right part of Figure 2.12. A LAURA processor consists of a read, execute, write, and control unit to implement the operational semantics of a PPN process. The read and write units iterate over the process domain $D_p$ and ensure at runtime that the proper channel is being read or written during each iteration. The execute unit contains an IP core which implements the process' functionality, that is, the function F. The read, execute, and write units

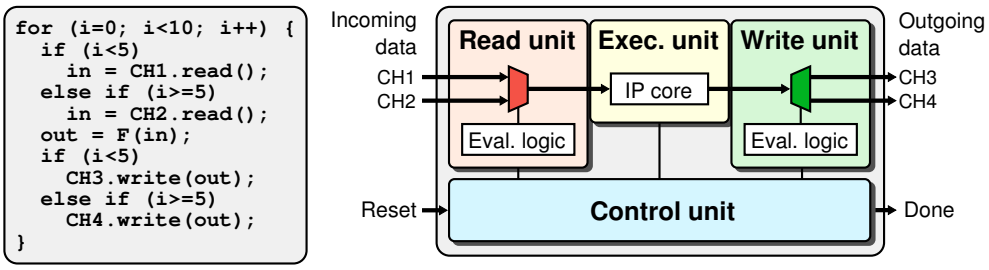



Figure 2.12: Example process code and the corresponding LAURA processor.

operate in a pipeline fashion, as depicted in Figure 2.13. For example, when iteration 0 enters the execute stage, the LAURA processor can initiate the read stage of the next iteration. The control unit orchestrates execution of the read, execute, and write units. For example, it stalls the read and execute units if the write unit reports a blocking write condition. In Figure 2.13, a blocking read condition occurs during iteration 3. In such a case, the controller ensures that previous iterations that are already in the pipeline continue executing, while iteration 3 is stalled until data is available. This leads to a bubble in the pipeline, which is indicated by a "–" in Figure 2.13.

The read unit is connected to the incoming channels of the process. For each input argument of the process' function F, a *read multiplexer* is instantiated. This multiplexer selects the incoming channels from which the argument is read. The selection is driven by the read unit's *evaluation logic* block. The evaluation logic employs a set of counters that iterate over the process domain. For each input port, the evaluation logic contains an expression in terms of the iterators that selects when that port has to be read. As such, data from the appropriate input ports is forwarded to the execute unit according to the current iteration.

The execute unit implements the process' function F. It provides an insertion slot for an IP core that implements the function F. The execute unit passes the argument values selected by the read unit to the IP core. The IP core processes the input data and produces output data after a delay that is specific to the IP core. The output data of the IP core is passed to the write unit. An IP core is often implemented in a pipeline fashion to provide high throughput. By employing pipelined IP cores, execution of subsequent independent process firings can overlap in time, thereby increasing the process' throughput.

The write unit is connected to the outgoing channels of the process. For each output argument of the process' function F, a demultiplexer selects the appropriate output channel to which the argument is written. The selection is driven by the write unit's evaluation logic block, which functions similarly to the evaluation logic of the read

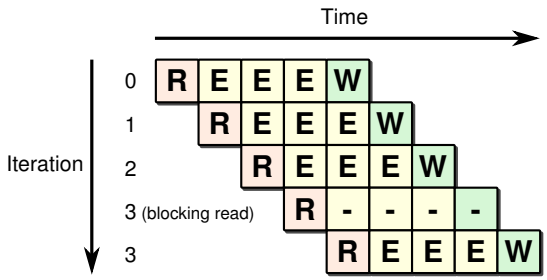



Figure 2.13: Pipelined execution of a LAURA processor containing a 3-stage IP core.

unit.

The control unit is responsible to implement the blocking read and write behavior of a PPN process. It enables or disables the read, execute, and write units based on information provided by those three units. The read unit reports a blocking read condition if one of the selected input channels does not contain any tokens. During a blocking read condition, the read unit is stalled while the execute and write units may still continue to process iterations pending in the pipeline. The write unit reports a blocking write condition if one of the selected output channels does not have sufficient room to store another token. During a blocking write condition, all units are stalled until the external consumer process clears the blocking write condition by reading a token from the full channel. The delay of an IP core may vary per firing of a process. For example, the delay of a variable length encoder IP core may depend on the input data. To integrate such IP cores in LAURA, the IP core should indicate when it is ready to accept or produce data. The execute unit forwards this information to the control unit, which then enables the read and write units accordingly.