An Integer Linear Programming Approach to Classify the Communication in Process Networks

Alexandru Turjan, Bart Kienhuis, Ed Deprettere

Leiden Institute of Advanced Computer Science (LIACS),
Leiden, The Netherlands

PUBLISHED IN THE PROCEEDINGS OF SCOPES'04, SRINGER LNCS 3199

Abstract. New embedded signal processing architectures are emerging that are composed of loosely coupled heterogeneous components like CPUs or DSPs, specialized IP cores, reconfigurable units, or memories. We believe that these architectures should be programmed using the Process Network model of computation. To ease the mapping of applications, we are developing the *Compaan* compiler that automatically derives a Process Network (PN) description from an application written in input Matlab. In this paper, we investigate a particular problem in nested loop programs, which is about classifying the interprocess communication in the PN representation of the nested loop program. The global memory arrays present in the Matlab code have to be replaced by a distributed communication structure used for sending data to the network processes. We will show that four types of communication exists, each exhibiting different requirements when realizing them in hardware of software. We present two compile time tests that decide the type of the communication corresponding to a particular static array. These tests are based on Integer Linear Programming and have become an important part of our Compaan compiler.

1 Introduction

Applications that are envisioned for the next decade in the area of multi-media, imaging, bioinformatics, and classical signal processing have a ferocious appetite for compute power. To satisfy this appetite, new embedded signal processing architectures are emerging. These are typically composed of loosely coupled heterogeneous components that exchange data using programmable interconnections such as a switch matrix or a network on chip (NoC). The components can be CPUs or DSPs, specialized IP cores, reconfigurable units, or memories. Also, a central control microprocessor is present for the configuration of the components at run-time using a low-bandwidth bus. An impression of such architecture is shown in Figure 1. Aside from the use of specialized heterogeneous components and instruction level parallelism on the CPUs, these architectures will employ more and more *task level parallelism* to deliver the required performance.

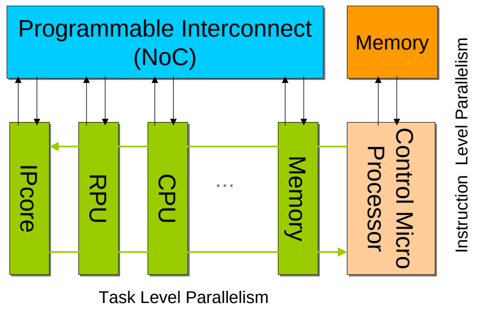


Fig. 1. Embedded Signal Processing Architecture consisting of loosely coupled heterogeneous components like CPUs, DSPs, Specialized IP cores, Reconfigurable Components, and Memories.

From a technology standpoint, companies and research institutions are already able to build instances of the presented architecture. Three examples are, for example, the *Picochip* from PicoChip, the *Virtex Pro* from Xilinx, and although still in research, the *SpaceCAKE* architecture from Philips. The PicoChip combines 430 simple RISC architectures on a single die [10]. Xilinx combines FPGA technology with four embedded PowerPCs on their Virtex-Pro chips [22]. Philips is researching the SpaceCAKE architecture which consists of a heterogeneous mix of memories, CPUs like the MIPS or the ARM and DSPs [15]. We observe that the problem with these heterogeneous architectures is not building them, but *programming* them, i.e., writing programs that take advantage of the offered heterogeneity and task-level parallelism. Writing a program for such an architecture means partitioning the application over the various components of the architecture and generating embedded software for each component. In case of a CPU or DSP, this means writing a piece of C code and in case of an IP core or reconfigurable unit, it means writing a VHDL or Verilog program. This partitioning and compilations are very time consuming and error prone, which makes the deployment of the heterogeneous architectures difficult.

We believe that the PN model of computation is suitable to cope with the heterogeneity of the new embedded architectures. Still, writing application in a PN format is a time consuming process. Therefore, we are developing the *Compaan* compiler [7, 14], which automatically derives a Process Network (PN) description from an application written in Matlab. Using Compaan, we can quickly derive PNs from existing applications and then map them on heterogeneous embedded architectures.

In this paper, we investigate a particular problem in the Compaan compiler regarding the ability to classify the interprocess communication in a PN. Within Compaan, we distribute the memory arrays present in the input program over a number of communication structures used for transmitting data between network's processes. We will show that four types of communication exist, each exhibiting different requirements when a hardware or software implementation is generated. Each communication structure in a process networks is classified to one of the four types, using two tests that are described in this paper. These tests can be performed at compile time and have become an important part of our Compaan compiler.

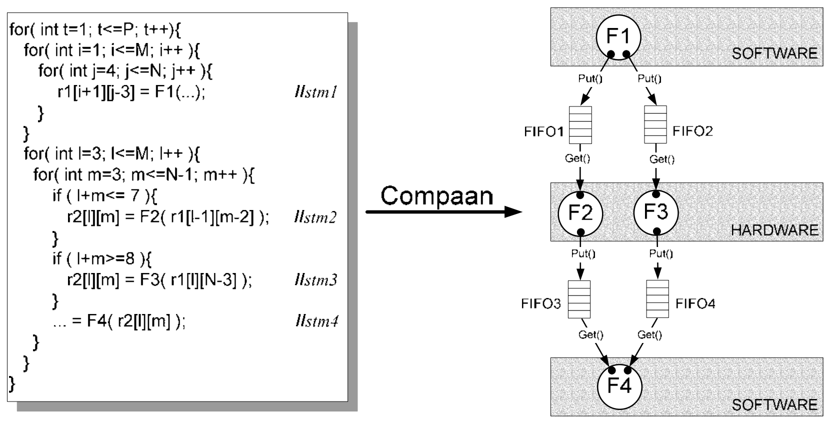
This paper is organized as follows. In Section 2, we explain our Compaan compiler project. The four kinds of communication in a process network are presented in Section 3. In Section 4 we presented related work. In Section 5 and Section 6, we present the two tests used for a compile time classification of the communication structures in a process network. In Section 7 we present some results and we conclude this paper in Section 8.

2 The Compaan Compiler

The aim of the *Compaan* compiler [7, 14] is to automatically derive a PN description from an application written in a standard language like C or Matlab as shown in Figure 2. It presents on the left-hand side a piece of Matlab code with four assignment statements (F1 .. F4). This piece of code is transformed in a PN, as given in the right-hand side. In Compaan, each assignment statement becomes a process and each static array (r1 and r2) is replaced by distributed communication structures; in this case FIFO buffers. Once the PN has been created, the individual processes can either be described as Java [3] or C++ code [5], or can be represented as synthesizable VHDL suitable for mapping onto a hardware platform [23]. As an example, we can map process F1 and F4 onto a CPU and processes F2 and F3 onto dedicated IP cores or reconfigurable block. The FIFOs are mapped either onto the communication structure or directly implemented as FIFOs.

An application written as a Process Network exhibits *distributed memory* and *distributed control* [14]. The distributed memory and control are essential when programming a distributed architecture as given in Figure 1, as the model of computation of a PN matches the way the architecture operates. Distributed memory means that not all components read and write to the same memory block, which usually leads to memory bottlenecks. Instead, components exchange data over separated communication channels, each channel being addressed by only two components acting in a Producer/Consumer manner. Distributed control means that each component can make progress on its own; it is not under the control of some global controller, which fits the loosely coupled components.

The Compaan compiler itself consists of three tools. The first tool, called *MatParser*, uses exact data dependence analysis to transforms the initial Matlab code into single assignment code (SAC). The second tool, called *DgParser*, converts the SAC into a Polyhedral Reduced Dependence Graph (PRDG) data struc-



```
for( int t=1; t<=P; t++){
  for( int i=1; i<=M; i++ ){
    for( int j=4; j<=N; j++ ){
      r1[i+1][j-3] = F1(...);                    //stm1
    }
  }
  for( int l=3; l<=M; l++ ){
    for( int m=3; m<=N-1; m++ ){
      if ( l+m<= 7 ){
        r2[l][m] = F2( r1[l-1][m-2] );   //stm2
      }
      if ( l+m>=8 ){
        r2[l][m] = F3( r1[l][N-3] );       //stm3
      }
      ... = F4( r2[l][m] );                      //stm4
    }
  }
}
```

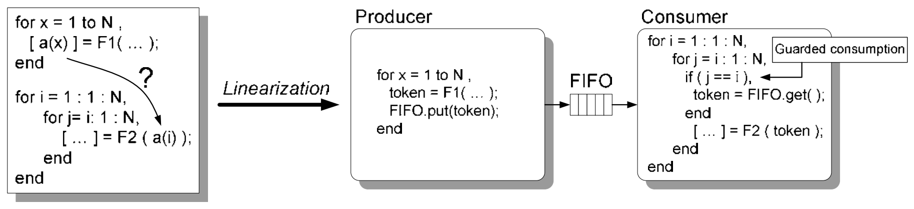
Fig. 2. From a sequential application to functionally equivalent Process Network

ture, which is a compact mathematical representation of the DG in terms of polyhedra. The third tool, called *Panda*, converts the PRDG into a Process Network, associating a process with each node of the PRDG.

3 Problem Description

In the single assignment code obtained after running MatParser, data is being communicating between assignment statements by sharing global single assignment arrays. In order to derive a PN, the single assignment code is broken into processes. Since we now have multiple processes accessing the global arrays, we need to replace them with distributed communication structures. We do not want to map the static array onto some shared memory, which is the typical approach. Shared memory contradicts with the desirable notion of distributed memory. Instead, we want to map the static array onto a truly distributed communication structure. This procedure represents an essential step in Panda that we call *Linearization* [12].

In Figure 3, we show two functions F1 and F2 which are respectively the producer of data in array $a(x)$ and the consumer of data in array $a(i)$. In Panda, these two functions F1 and F2 are split into separate processes to form a process network. Due to the splitting, we need to replace the static array a by a new communication structure. This would typically be an unbounded FIFO buffer that is accessed using a *blocking-read* primitive. However, the question is whether we can always do this replacement, as indicated by the question mark in Figure 3. To answer this question, we investigate the characteristics of accessing static arrays. As we will show, four different types of communicating data are possible.



```
for x = 1 to N ,
   [ a(x) ] = F1( ... );
end
for i = 1 : 1 : N,
      for j= i: 1 : N,
         [ ... ] = F2 ( a(i) );
      end
end
```

```
for x = 1 to N ,
   token = F1( ... );
   FIFO.put(token);
end
```

```
for i = 1 : 1 : N,
      for j = i : 1 : N,
         if ( j == i ),
            token = FIFO.get( );
         end
            [ ... ] = F2 ( token );
      end
end
```

Fig. 3. Linearization of static array a.

3.1 Producer/Consumer Model

The tool DgParser translates the single assignment program to a polyhedral reduced dependence graph (PRDG). This PRDG is accepted by Panda, which transforms the PRDG in such a way that each edge of the graph represents a point-to-point communication. In this process, the original static arrays have been partitioned over a number of edges. For example, in Figure 2, the static array r1 and r2 are both replaced by two FIFO buffers. This means that the content of the arrays is distributed over four FIFO buffers. This step is beyond the scope of this article but is an essential step in Panda. Once the point-to-point communication is established, each edge representing the point-to-point communication needs to be linearized between a producer and a consumer node. Each edge can be abstracted to a simple Producer/Consumer (P/C) pair, as shown in Figure 3. The Producer and the Consumer have parameterized polyhedral iteration domains that are related through an affine transformation. Depending on this transformation, described by the mapping matrix M and the structure of the for-loops in the original Matlab program, we have found that four different kinds of communicating data can be distinguished in process networks.

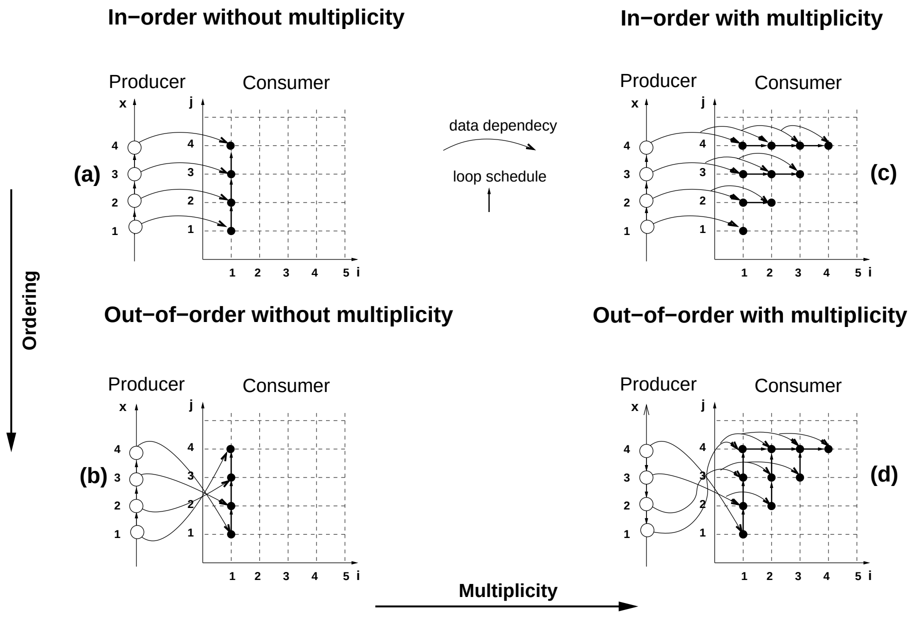

In−order without multiplicity
In−order with multiplicity
Out−of−order without multiplicity
Out−of−order with multiplicity
Producer
Consumer
(a)
(b)
(c)
(d)
data dependecy
loop schedule
Ordering
Multiplicity
x
j
i

Fig. 4. The four types of communication of a static array in a Producer/Consumer pairs.

These four types of communication are given in Figure 4. They result from the *ordering* of the iterations at the Producer and the Consumer processes and the existence of *multiplicity* for a given token, which means that a token that is sent by the Producer is read more than once at the Consumer side. In Figure 4, we represent the iterations with small circles. These iterations are ordered as indicated by the small arrows representing the loop schedule of the for-loops in the original Matlab program. We also show the relationship between the production of data at an iteration and the consumption of that data at one or more iterations, using a data-dependency function.

In Figure 4(a), we see that the schedule of both the producer and consumer are the same. The producer iterates from 1 to 4 using an iterator x. The consumer iterates in the same order from 1 to 4 using an an iterator j. We can see that data produced at one iteration is consumed at the consumer side in the same order, and by only one iteration. For example, the data produced at iteration x=3 is consumed at iteration j=3. In Figure 4(b), we see that although the schedule of both the Producer and Consumer is the same, the Consumer consumes data in the opposite order due to the dependency. We can see that data produced at one iteration is consumed at the consumer side in the opposite order by only a single iteration. For example, the data produced at iteration x=3 is consumed at iteration j=2. In Figure 4(c), we see that the schedule of the Consumer is two-dimensional (i,j). We see that for some iterations a token from the Producer is consumed more than once. This consumption takes place in the same order as it is produced. For example, we see that the token created at $x = 3$ is consumed multiple times at iterations $(i, j) = (1, 3), (2, 3), (3, 3)$.

Finally, In Figure 4(d), we see again that the schedule of the Consumer is still two-dimensional and that at some iterations a produced token is consumed more than once. However, in this case, the tokens are consumed in a different order from the one in which they are produced, due to the dependency function. For example, we see that the token produced at $x = 3$ is consumed out-of-order multiple times at iterations $(i, j) = (1, 2), (2, 2)$. We have named the four communication types as shown in Table 1.

Name	Type	Figure
IOM-	In Order without Multiplicity	4(a)
OOM-	Out Of Order without Multiplicity	4(b)
IOM+	In Order with Multiplicity	4(c)
OOM+	Out Of Order with Multiplicity	4(d)

Table 1. The four types of communication.

Looking at the example code given in Figure 3, we observe that the communication of a is of type IOM+. Now, by making small changes to the program, we will show how the other three types can be obtained. If we remove the `j` for-loop, we obtain a the linearization of a of type IOM-. If we take again the original example, but let iterator `x` run from 4 to 1, we get that the linearization of a is of type OOM+. If we remove again iterator `j`, we observe that the linearization of a is of type OOM-.

So we know that four types of linearization exist in any nested loop program when we want to replace the static array by a distributed communication structure. We need to investigate in which linearization case we can simply replace the static array with a FIFO buffer accessed by a put/get primitive. Only in the case of type IOM-, we can simply replace the variable by a FIFO. However, in the example given in Figure 3 where multiplicity takes place we can still rely on a simple FIFO linearization by adding a guard if-statement to take care of the life-time of a token read from the FIFO [17].

In the case of the two remaining communication types, two additional elements may be needed. If re-ordering is an issue, a controller is needed that re-orders a stream of tokens using some private memory to temporarily store tokens for reordering. If multiplicity is involved, a controller is needed that indicates when the life-time of a token comes to an end and the token can be released. The controller for reordering, the life-time controller, and the reordering memory define what we call the *Extended Linearization Model* (ELM), which has been introduced in [18, 17]. In these papers we also explain how to derive at compile time the re-ordering and the life-time controller. In the ELM we place the reordering memory and the controllers at the Consumer process, but between processes we still model the use of FIFO buffers, thereby adhering to the PN model of computation.

Depending on the type of communication, a static array is replaced in the following way:

– **IOM-** An in-order P/C pair is linearized using only a FIFO buffer.
– **IOM+** An in-order with multiplicity P/C pair is linearized using a FIFO buffer and controller to determine the life-time of a token.
– **OOM-** An out-of-order P/C pair is linearized using a FIFO buffer, reordering memory, and a controller to perform the reordering. Since multiplicity is not involved, each time the controller accesses the reordering memory for reading data, the corresponding memory location can be immediately released.
– **OOM+** An out-of-order with multiplicity P/C pair is linearized using a FIFO buffer, reordering memory, and a controller to perform reordering and to determine the life-time of a token.

Notice that the implementations of the linearization models described above increase in their complexity (both hardware and software), from IOM- to OOM+. Therefore, it is very important to select the correct linearization type. The implementation of IOM- and IOM+ are closely related, except that in IOM- a controller is needed to know when to release data from the FIFO. The implementation of OOM- and OOM+ requires additional reordering memory and at least one reorder controller. Of the four models identified, OOM+ is the most expensive linearization to be realized. It is also the generic linearization since it subsumes all other possible linearization.

3.2 Definition of Multiplicity and Re-ordering

We now give a formal definition of a P/C pair, multiplicity and reordering.

Definition 1 *A P/C pair is a tuple $< \mathcal{C}(p), f, P(p), \prec >$, where $\mathcal{C}(p) \subset R^n$ is a parameterized polytope, $f : R^n \to R^m$ is an affine function, $P(p) = f(\mathcal{C}(p) \cap Z^n)$, and $\prec$ is the lexicographical order.*

The set of integer points inside the parameterized polytope $\mathcal{C}(p)$ is called the *Consumer domain*, or the *Consumer iteration space*. The affine function f specifies the data dependencies and is represented by an $(m \times n)$ integral matrix M, and an m-dimensional offset vector O, i.e., $f(x) = Mx + O$. There are no additional constraints imposed to the mapping matrix M. As a consequence, the *Producer domain (iteration space)* is represented by a *linearly bounded lattice* (LBL) [16] given by the integer polyhedral image:

$$P(p) = f(\mathcal{C}(p) \cap \mathbb{Z}^n) = \{i \in \mathbb{Z}^m \mid i = Mk + O \wedge \ k \in (\mathcal{C}(p) \cap \mathbb{Z}^n)\}. \tag{1}$$

If $x \in \mathcal{C}(p)$ and $y \in P(p)$ are two elements (or *iteration points (IPs)*) such that $y = f(x)$, we say that the IP x consumes data produced by the IP y. Although $P(p)$ is by definition an LBL, in [17], we have shown that the Producer iteration space can be represented as union of polytopes. Therefore, without loss of generality, in the remainder of this paper we assume that the Producer is a polytope. We also assume that $O = 0$, such that f is represented only the matrix M.

Depending on the mapping matrix M, and on the schedule of producing and consuming data as given by the lexicographical order, four different kinds of P/C pairs can be distinguished (see Figure 4) based on the following two definitions:

Definition 2 *A P/C pair is* ***in-order*** *iff the mapping preserves the order, i.e. every two Consumer iteration points $x_1 \prec x_2$ are mapped onto two Producer iteration points $(y_1 = Mx_1)$ and $(y_2 = Mx_2)$ such that $y_1 \preceq y_2$. If the P/C pair is not in order it is called* **out-of-order**.

Definition 3 *A P/C pair is* **without multiplicity** *iff the mapping $M : (C \cap Z^n) \to P$ (i.e., the mapping M restricted to the Consumer domain) is injective. Otherwise we say that the P/C pair is* **with multiplicity**.

According to the previous definitions, an arbitrary P/C pair belongs to one of the four types as given in Table 1. The implementation cost of the linearization types increases from IOM- to OOM+. Therefore, to come to an optimal replacement of the static arrays, we need to be able to know at compile time to which type of communication an arbitrary P/C pair belongs.

4 Related Work

By replacing the shared arrays with FIFO buffers we get a flexible communication structure which allows to find a good balance between memory reuse and inter-process parallelism [9, 1]. There are several papers that are dealing with compile-time analysis of memory reuse in static nested loop programs. In [19] an approach is presented for a fixed linearization of the memory array. In [20, 11] a constructive approach is given in context of the single assignment language ALPHA, based on which the maximum life-time of an array can be derived for an optimal memory projection. In [8], in the process of parallelizing a static algorithm by removing output dependences, the authors propose a method of partial array expansion. In [4] an approach is presented to compute the bounding box for the elements that are simultaneously in use. The size of the original array is reduced to the bounding box and the array is accessed using modulo operations, improving in this way the memory usage. All the techniques presented so far rely on polytope manipulations and integer linear programming.

We have already presented a solution to determine the type of communication in [18]. In that paper, we presented a technique based on the Ehrhart theory [2] to determine the ordering and multiplicity. By comparing pseudo polynomial expressions, we could determine whether the schedule at the Consumer and Producer are the same. We could also determine if the pseudo polynomial was larger than one, indicating that multiplicity was involved. Although the presented procedure gives a definitive answer for the type of communication, the complexity of the pseudo polynomial calculations, the comparison of the pseudo polynomials, and the fact that the software implementation of Ehrhart cannot always derive a pseudo polynomial, made the approach unsuitable for our compiler.

5 Solution Approach

To classify a P/C pair to one of the four types, we need to be able to identify if data is reused more than once, and whether the ordering of producing data at the Producer side is the same as the order of consuming it at the Consumer. In this section, we present the techniques which allow us to determine the type of an arbitrary P/C pair. In Section 5.1, we provide a test that determines if a P/C pair is with or without multiplicity. In Section 5.2, we describe a test which determines whether a P/C pair is in-order or out-of-order. Using these two tests, we can determine at compile time to which of the four categories (IOM-, IOM+, OOM-, OOM+) an arbitrary P/C pair belongs.

5.1 The Multiplicity Test (MT)

The *Multiplicity Test* is used to check whether two Consumer iterations consume one and the same token. Consider an arbitrary P/C pair $< \mathcal{C}(p), M, P(p), \prec >$. According to *Definition 3,* this pair has multiplicity if there exist two different Consumer points x and y such that they are consuming the same token from the Producer as specified by the mapping M. These conditions can be captured in the following system that forms by definition the *Multiplicity Problem* (MP):

$$\mathbf{MP}: \quad \begin{cases} x \in (\mathcal{C}(p) \cap Z^n), \\ y \in (\mathcal{C}(p) \cap Z^n), \\ x \neq y, \\ Mx = My. \end{cases} \tag{2}$$

In the first two equations, we take a point x and y from the Consumer that according to the third equation, are not the same. However, if we can find more than one point that map x and y via M to the same producer point, we have found that data is reused and thus multiplicity is involved. Therefore, the problem of deciding whether a P/C pair has multiplicity is reduced to testing the existence of a solution for the MP.

5.2 The Reordering Test (RT)

The *Reordering Test* is used to determine whether the order of producing tokens at the Producer side is the same as the order in which they are consumed at the Consumer side. According to *Definition 2* a P/C pair $< \mathcal{C}(p), M, P(p), \prec >$ is out-of-order if two different Consumer points $y \prec x$ exist such that $Mx \prec My$. These conditions can be captured in the following system that forms by definition the *Reordering Problem* (RP):

$$\mathbf{RP}: \quad \begin{cases} x \in (\mathcal{C}(p) \cap Z^n), \\ y \in (\mathcal{C}(p) \cap Z^n), \\ x \prec y, \\ My \prec Mx. \end{cases} \tag{3}$$

In the first two equations, we take a point x and y from the Consumer, where x is lexicographically smaller than y according to Equation 3. However, If we can find more than one point that map x and y via M in such a way that x is still lexicographically smaller than y at the producer side, we have found that data is consumer in-order. Therefore, the problem of deciding whether reordering takes place in a P/C pair is reduced to testing the existence of a solution for the RP.

6 Realization of the Solution

In the previous section we introduced two tests. In this section we will show how they can be used in practice. In solving the systems, we exploit the fact that we can represent a Matlab program in terms of a PRDG. In such a PRDG, the iteration domains are represented by means of polytopes, allowing us to employ integer linear programming to find solutions to the MP and RP problems in an efficient way. Consequently, we can implement these tests and solve them to find the proper communication structure at compile time.

6.1 Empty Domain Test

Both the Multiplicity Test and the Reordering Test are so called *Existence tests* as we only need to determine whether a given system of constraints contains a solution. We are not interested in what the solution is, and therefore, do not need to compute this. A system of constraints has a solution if and only if it contains at least a single integer point. The procedure to determine if a polytope contains at least a single integer point is what we call the *Empty Domain Test* (ET). In both the RP and MP case, we build up a polytope and if such polytope contains at least a single integer point, a solution exists for the RP or MP.

To perform the Empty Domain Test, we can make use of integer linear programming (ILP) tools like *Pip* [6] or *Omega* [21]. These tools only work on systems of linear constraints, while the Multiplicity Problem and the Reordering Problem contain non-linear constraints. In case of the RP, the lexicographical order operator $\prec$ allows the following decomposition:

$$x \prec y \Leftrightarrow (x \prec_1 y) \vee (x \prec_2 y) \vee ... \vee (x \prec_n y), \tag{4}$$

where, $x \prec_i y$ means that $x_1 = y_1, x_2 = y_2, ..., x_{i-1} = y_{i-1}, x_i < y_i$. Suppose x and y are defined on a two-dimensional domain. Then the lexicographical expansions is represented by two polytope domains $\mathcal{D}_1 = \{(x_1, x_2) \in D \mid x_1 < y_1\}$ and $\mathcal{D}_2 = \{(x_1, x_2) \in D \mid x_1 = y_1 \wedge x_2 < y_2\}$.

In case of the MP, the negation is the non-linear operator. This negation can be rewritten to two inequalities, as follows:

$$x \neq y \Leftrightarrow y \prec x \vee x \prec y, \tag{5}$$

where we use again Equation 4 to convert the lexicographical operators to linear constraints.

Given the Empty Domain test and how to rewrite the appearance of the non-linear operation in the MP and RP, we now solve the MP and RP systems.

6.2 Solving the Multiplicity Test

By rewriting the negation in the Multiplicity problem given in Equation 2, using the substitution from Equation 5, the Multiplicity Problem is decomposed into two sub-problems: the *Primal Multiplicity Problem* and the *Dual Multiplicity Problem*:

$$\mathbf{PMP}: \begin{cases} x \in (\mathcal{C}(p) \cap Z^n), & (1) \\ y \in (\mathcal{C}(p) \cap Z^n), & (2) \\ x \prec y, & (3') \\ M\text{x} = My. & (4) \end{cases} \qquad \mathbf{DMP}: \begin{cases} x \in (\mathcal{C}(p) \cap Z^n), & (1) \\ y \in (\mathcal{C}(p) \cap Z^n), & (2) \\ x \succ y, & (3'') \\ Mx = My. & (4) \end{cases}$$

The two sub-problems have the property that if (x, y) is a solution for one of them, it is also a solution for the MP. As a consequence:

$$\text{ET(MP)} = \text{ET(PMP)} \ \vee \ \text{ET(DMP)}. \tag{6}$$

On the other hand, if (x, y) is a solution for PMP then (y, x) is also a solution for DMP. In this way the MT is reduced to solving the empty test only of the PMP or the DMP. Therefore, in the remainder of this section, we will describe the solution for just the PMP problem. Given the lexicographical decomposition procedure expressed in Equation 4, the PMP is further decomposed into a series of n disjoint subproblems PMP$_i$, where $i = \overline{1, n}$:

$$\mathbf{PMP_i}: \begin{cases} x = (x_1, x_2, ..., x_n)^T \in (\mathcal{C}(p) \cap Z^n), \\ y = (y_1, y_2, ..., y_n)^T \in (\mathcal{C}(p) \cap Z^n), \\ \begin{cases} (y_1, ..., y_{i-1}) = (x_1, ..., x_{i-1}), \\ x_i < y_i, \end{cases} \\ Mx = My \end{cases} \tag{4}$$

such that:

$$\text{ET(PMP)} = \bigvee_{i=1}^{n} \text{ET(PMP}_i).$$

If we find the existence of a solution for any of these systems, we thus find a solution for the MT. This signifies that multiplicity is involved in the linearization on the P/C pair under investigation. Hence, as soon as we have found the existence of a solution, we can stop.

Optimization A substantial reduction of Empty Domain test can be achieved by using the Hermite Normal Form of the mapping matrix M [13]. The *Hermite Normal Form* (HNF) can be used to decompose an integral matrix M, using an unimodular matrix C, into an unique lower triangle matrix H such that:

$$M = [H0]\,C. \tag{7}$$

The unimodular matrix C does not affect the multiplicity as it always has an inverse. Hence, the multiplicity information is still contained in H. Therefore, by taking the HNF of the mapping matrix M, we obtain H that still captures the multiplicity characteristics of M. Suppose we obtain t zero columns when taking the HNF of M. Thus, the size of H is $n - t$, which is less or equal than the size of M. Consequently, we need to test less MP systems for empty domains.

6.3 Solving the Reordering Test

By rewriting the negation in the Reordering Problem given in Equation 3, using the substitution from Equation 4, the Reordering test is decomposed in a number of systems that need to be tested using the Empty Domain test. Suppose the Consumer domain is contained into a n-dimensional iteration space such that $x = (x_1, x_2, ..., x_n)^T$ and $y = (y_1, y_2, ..., y_n)^T$ are two different points in the same Consumer domain. Suppose that the matrix M has k rows : $M = \begin{bmatrix} M_1 \\ \vdots \\ M_k \end{bmatrix}$. Using the lexicographical decomposition procedure, the RP is decomposed into $n \times k$ disjoint subproblems given as RP$_{ij}$, where $i = \overline{1,n}$ and $j = \overline{1,k}$:

$$\mathbf{RP_{ij}} : \begin{cases} x = (\mathrm{x}_1, \mathrm{x}_2, ..., \mathrm{x}_n)^T \in (C \cap Z^n), \\ y = (y_1, y_2, ..., y_n)^T \in (C \cap Z^n), \\ \begin{cases} (y_1, ..., y_{i-1}) = (\mathrm{x}_1, ..., \mathrm{x}_{i-1}), \\ \mathrm{x}_i < y_i, \\ (M_1 x, ...M_{j-1} x) = (M_1 y, ..., M_{j-1} y), \\ M_j y > M_j x, \end{cases} \end{cases}$$

such that:

$$\text{ET(RP)} = \bigvee_{i=1}^{n} \text{ET}(\text{RP}_{ij}).$$

This means that if we find the existence of a solution for any of these systems, we thus find a solution for the RP. This signifies that re-ordering is involved in the linearization on the P/C pair under investigation. Hence, as soon as we have found the existence of a solution, we can again stop.

Although the number of the subproblems is large, it can be reduced by choosing the order in which we test the systems in a smart way. In this way, we exploit the fact that we do not have to test the remaining systems, as soon as a solution is found. We have found that starting the empty domain tests in the following sequence of problems RP$_{1,1}$...RP$_{1,k}$...RP$_{n,1}$...RP$_{n,k}$ is a good strategy for minimizing the number of evaluations. This is due to the fact that the probability of finding a solution decreases with the polyhedron volume in which the search is being performed.

7 Experimental Results

In Table 2, we present the results for eight real applications written in Matlab for the domain of imaging and signal processing. The first column in the table shows the number of P/C pairs that appear in a particular algorithm. This is followed by the number of Empty Domain tests needed for the Reorder Test and the Multiplicity Test to classify these P/C pairs. The numbers for the MT are obtained without using the proposed optimization procedure. The last four columns give the four different communication types and how many of the P/C pairs belong to that type. Using functional simulation, we verified for each algorithm that the classification is indeed done correctly.

Table 2 clearly shows that the most occurring type is IOM- (80%) and the least occurring type is OOM+ (1%). This is good, as the IOM- requires only a simple FIFO buffer. The OOM+ requires also re-ordering memory and a controller, but hardly appears in networks. The second most occurring type is IOM+ (10%) which is also nice, as only a FIFO buffer is needed with some simple additional control to keep track of the life time of a token. Together with type IOM-, we can say that in 90% of the cases, a FIFO buffer can indeed be used to linearize a static array in a Matlab program.

Algorithm	P/C pairs	RT	MT	IOM-	IOM+	OOM-	OOM+
QR-Decomp	12	73	30	12	0	0	0
SVD	118	1283	565	84	4	30	0
Faddeev	28	205	78	24	3	1	0
Gauss-Elimin	11	17	6	7	0	1	3
DigBeamFormer	98	408	196	88	4	6	0
Motion Estim	98	882	294	98	0	0	0
M-JPEG	50	178	93	33	17	0	0
Stereo Vision	173	1470	518	172	0	1	0

Table 2. Experimental results: classification of the edges in real-life examples to one of the four communication types using the Reordering Test and the Multiplicity Test.

The number of Empty Domain tests performed to classify the P/C pairs of a network is quite large. Given the complexity of the Pip and Omega algorithms (integer linear programming is NP complete), the RT and MT tests are time and memory consuming procedures. We have presented already a possible optimization for the MT procedure. We are investigating a similar procedure for RT. Further optimizations to reduce the number of tests are currently subject to further research.

8 Conclusion

In this paper, we presented the problem of classifying a static array appearing in a Matlab program to one of four communication types. The linearization for these types is needed when we break down a Matlab program into processes, as the static array is now accessed by multiple processes. The array is replaced by a distributed communication structure such as a FIFO buffer to avoid the use of shared memory access. Shared memory contradicts the desired characteristics of distributed memory in a PN. Using the distributed communication structure, a static array is explicitly specified by communication between processes, thus really decoupling the processes.

To classify a P/C pair to one of the four types, we presented two tests. The Reorder test determines if re-order takes place on the communication between an arbitrary P/C pair. The Multiplicity test determines if reuse of data takes place on the communication between an arbitrary P/C pair. Both tests have been formulated as integer linear programming problems. This makes the solution suitable for implementation in a compiler.

The presented tests are not specific to the Compaan compiler. They work for any nested loop program in which static arrays need to be replaced by distributed communication structures. We have tested and classified eight real-life examples to validate our approach. We have shown that, on average, in 90% of the cases the static array can be replaced by a FIFO buffer. In 10% of the cases additional control logic and additional memory is needed at the Consumer process, besides a FIFO buffer, to make the linearization work. Although a FIFO buffer is placed between processes in this case, in practice the FIFO and reordering memory are combined in a single memory structure.

References

1. T. Basten and J. Hoogerbrugge. Efficient execution of process networks. In *Communicating Process Architectures - 2001, Proceedings*, pages 1–14, Bristol, UK, September 2001.
2. P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyse and transform scientific programs. In *10th International Conference on Supercomputing, Philadelphia*, May 1996.
3. J. Davis II, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong. Heterogeneous concurrent modeling and design in java. Technical Report Memorandum UCB/ERL M01/12, University of California, Dept EECS, Berkeley, CA USA 94720, Mar. 2001.
4. E. De Greef, F. Catthoor, and H. De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. In *Parallel Processing and Multimedia*, Geneva, Switzerland, July 1977.
5. E. de Kock, G. Essink, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. YAPI: Application modeling for signal processing systems. In *Proc. 37th Design Automation Conference (DAC'2000)*, pages 402–405, Los Angeles, CA, June 5-9 2000.
6. P. Feautrier. Parametric Integer Programming. In *RAIRO Recherche Op?rationnelle, 22(3):243-268*, 1988.
7. B. Kienhuis, E. Rypkema, and E. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES)*, San Diego, USA, May 2000.
8. V. Lefebvre and P. Feautrier. Automatic storage management in paralel programs. volume 24, pages 649 – 671. Parallel Computing, nov 1998.
9. T. Parks. Bounded scheduling of process networks. T. M. Parks, Bounded Scheduling of Process Networks, Technical Report UCB/ERL-95-105. PhD Dissertation. EECS Department, University of California, Berkeley CA 94720, December 1995.
10. http://www.picochip.com.
11. F. Quillere and S. Rajopadhye. Optimizing memory usage in the polyhedral model. In *ACM Transactions on Programming Languages and Systems*, volume 22, pages 773–815, September 2000.
12. E. Rijpkema. *From Piecewise Regular Algorithms to Dataflow Architectures*. PhD thesis, Delft University of Technology, 2001.
13. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons Ltd., 1986.
14. T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using kahn process networks: The compaan/laura approach. In *Proceedings of DATE2004*, Paris, France, Feb 16 – 20 2004.
15. P. Stravers and J. Hoogerbrugge. Homogeneous multiprocessoring and the future of silicon design paradigms. In *Proceedings of the Int. Symposium on VLSI Technology, Systems, and Applications*, Apr. 2001.
16. J. Teich and L. Thiele. Partitioning of processor arrays: A piecewise regular approach. *Integration, the VLSI journal*, 14:297–332, February 1993.
17. A. Turjan and B. Kienhuis. Storage management in process networks using the lexicographically maximal preimage. In *Proceedings of the IEEE 14th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP'03)*, The Hague, The Netherlands, June 24-26 2003.
18. A. Turjan, B. Kienhuis, and E. Deprettere. A compile time based approach for solving out-of-order communication in kahn process networks. In *Proceedings of the IEEE 13th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP'02)*, San Jose, California, July 2002.
19. V. Vanhoof, I. Bolsens, and H. De Man. Compiling multi-dimensional data streams into distributed dsp asic memory. In *In Proc. IEEE Int. Conf. Comp. Aided Design*, Santa Clara, CA, August 1989.
20. D. Wilde and S. Rajopadhye. Memory reuse in the polyhedral model. In *In Proc. Euro-Par96*, Lyon, France, August 2002.
21. P. William. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communications of the ACM*, 35(8):102–114, 1992.
22. http://www.xilinx.com.
23. C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, 2003.