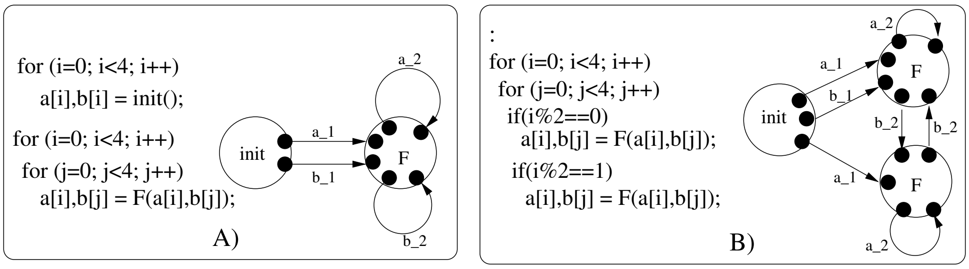
Transformations for polyhedral process networks

Meijer, S.

Chapter 3

Process Splitting Transformations

In this chapter, we present an approach how the process splitting transformation, introduced in Chapter 1, can be applied to transform a Polyhedral Process Network in order to select and obtain the best performance results from different splitting alternatives. Recall that the Polyhedral Process Network (PPN) model of computation is used as a programming model in the Daedalus framework [62] to help with the difficult task of programming and mapping applications onto Multi-Processor Systems on Chip. PPNs are automatically derived from sequential nested-loop programs by using the `pn` compiler [95] as we have illustrated with an example in Chapter 2. In the derived parallel PPN specification, the following partitioning strategy is used: each process in the PPN corresponds to a function call statement in the sequential program.



```
for (i=0; i<4; i++)
   a[i],b[i] = init();
for (i=0; i<4; i++)
  for (j=0; j<4; j++)
     a[i],b[j] = F(a[i],b[j]);
```

```
:
for (i=0; i<4; i++)
  for (j=0; j<4; j++)
    if(i%2==0)
      a[i],b[j] = F(a[i],b[j]);
    if(i%2==1)
      a[i],b[j] = F(a[i],b[j]);
```

Figure 3.1: Polyhedral Process Networks

Figure 3.1 A) shows a PPN consisting of two process with 4 FIFO channels, and also the nested loop program from which this PPN is derived. Deriving the network using the `pn` partitioning strategy, as described above, does not necessarily lead to

optimal performance results as the network may not be well balanced. Therefore, process partitioning transformations can distribute the workload of a single process over multiple processes to better balance the network. We can achieve this, for example, as shown in Figure 3.1 B). The function call statement `F` is duplicated and assigned to odd and even iterations of the outer loop iterator. The corresponding network has now two processes executing the `F` function resulting in a more balanced network. In [79], a number of algorithmic transformations have been presented which a designer can apply on the source code to balance the network. However, no hints are given to the designer when a particular transformation can be applied to minimize, for example, the execution time. So, a number of transformations have been defined, but the designer does not know when to apply which transformation. In our motivating examples (Section 3.2) we show that it is not straightforward to select the best transformation for the best performance results. In order to select the best partitioning transformation, the different alternatives must be evaluated and metrics are required to do so. This chapter, therefore, deals with:

1. Definition of evaluation metrics;

2. Calculation of the metric values using an analytical framework;

3. A compile-time evaluation approach to select a particular transformation based on the metric values.

We show results for 3 different applications with different properties mapped onto the Cell processor [39] and the ESPAM platform prototyped on a Xilinx Vertex 2 FPGA [61].

3.1 Process Splitting: Definitions, Notations, and Examples

First, it is important to note that process splitting is a general term referring to transformations duplicating program code to obtain more processes. In Figure 3.1 B), we have shown one example of process splitting, but there are many other possibilities to duplicate the program code. In [79], a number of parametric transformations have been presented that can be used to split up processes. Two of these splitting transformations are the modulo unfolding and the plane-cut transformation:

Notation 1: we refer to the **modulo unfolding** transformation as `unfold(I,U)`, where parameters `I` and `U` are respectively the iteration vector of the function of a process and the vector of unfolding factors for each loop iterator.

Notation 2: we refer to the **plane-cut** transformation as `planecut(I,P)` where parameter `I` is the iteration vector and parameter `P` is a set of affine hyperplanes (see Section 2.1).

Definition 11 *A **process partition**, or partition in short, is a new instance of an original process that is created by applying a process splitting transformation* `unfold` *or* `planecut`*. Thus, the different process partitions execute the same function, possibly, in parallel.*

In the remainder of this chapter, we focus on the unfolding and plane-cutting transformations. In [79], some more (algorithmic) transformation techniques have been presented. An example is the skewing transformation, which re-times the process iterations. However, only the unfolding and plane-cutting split-up a process, i.e., assign process iterations to different partitions. To illustrate the difference in the unfolding and plane-cutting transformations, we consider the example shown in Figure 3.2.

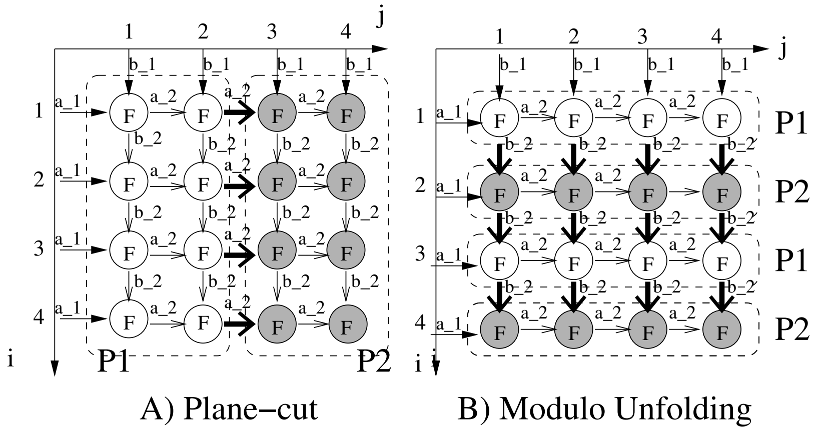
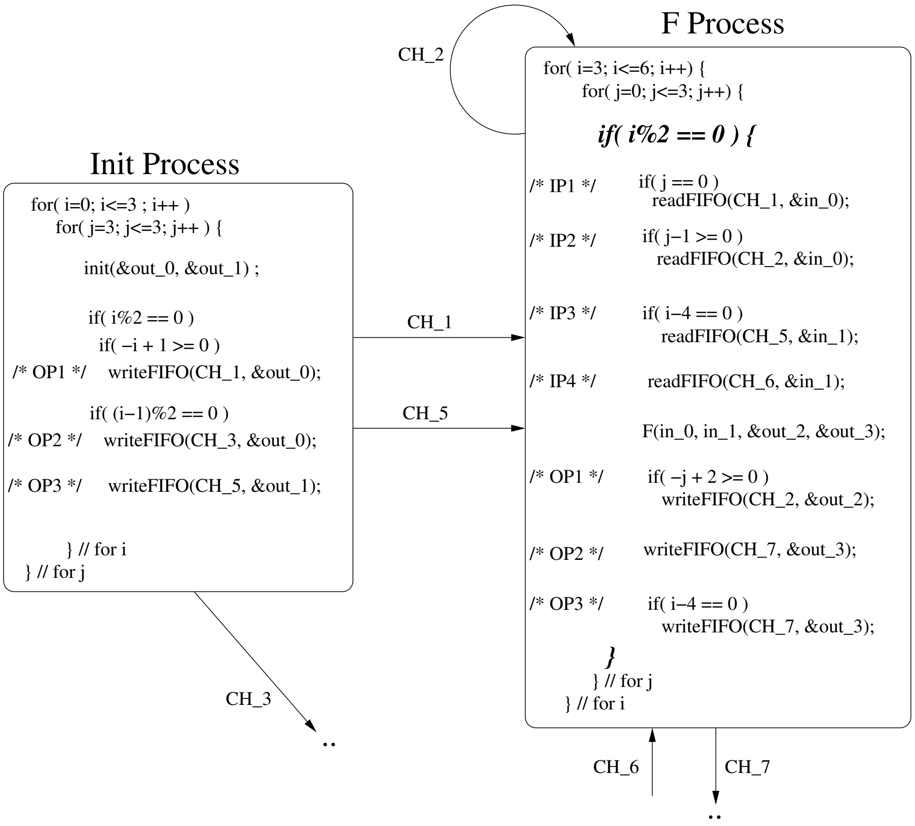


Figure 3.2: Examples of Process Splitting Transformations

Figure 3.2 shows the dependency graph of the program depicted in Figure 3.1 A) and is characterized by a two dimensional process iteration domain and horizontal and vertical dependencies. Loop iterator i corresponds to the outer loop and iterator j corresponds to the inner loop such that the lexigraphical order is from top to bottom and from left to right. The arrows denote dependencies. The dependency graphs are annotated with two possible partitionings which are the result of applying transformations. The plane-cut transformation `planecut({i,j},{j=2})` has been applied in Figure 3.2 A) such that partition $P1$ executes all points with $j \leq 2$ (the white iteration points) and $P2$ executes all points with $j \geq 3$ (grey points). Another partitioning is shown in Figure 3.2 B) which corresponds to the modulo unfolding transformation presented in Figure 3.1 B) and is formally specified as `unfold({i,j},{2,0})`. All even i iterations are assigned to $P2$, and all odd i iteration points are assigned to $P1$. The plane-cut and unfolding transformations and partitions differ in terms of the amount of inter-process communication (as indicated with the bold arrows) and initial delay of the partitions. In the plane cutting example in Figure 3.2 A), inter-process communication occurs 4 times and the first iteration point of $P2$, i.e., point $(1, 3)$, must wait for 2 iterations $(1, 1)$ and $(1, 2)$ of $P1$ before it can start executing. In the modulo unfolding partitioning in Figure 3.2 B), $P2$ starts after 1 iteration of $P1$, but then 12 inter-process data transfers are performed. This makes clear that different transformations lead to different behavior of the partitioned processes.

To give a more elaborate example of the internal structure of processes, we consider the processes in Figure 3.3. It shows one of the unfolded `F` processes and source process `init` from Figure 3.1 B).



```
for( i=0; i<=3 ; i++ )
    for( j=3; j<=3; j++ ) {

        init(&out_0, &out_1) ;

        if( i%2 == 0 )
          if( −i + 1 >= 0 )
/* OP1 */   writeFIFO(CH_1, &out_0);

        if( (i−1)%2 == 0 )
/* OP2 */   writeFIFO(CH_3, &out_0);

/* OP3 */   writeFIFO(CH_5, &out_1);

        } // for i
  } // for j
```

```
for( i=3; i<=6; i++) {
    for( j=0; j<=3; j++) {

        if( i%2 == 0 ) {

/* IP1 */     if( j == 0 )
                readFIFO(CH_1, &in_0);

/* IP2 */     if( j−1 >= 0 )
                readFIFO(CH_2, &in_0);

/* IP3 */     if( i−4 == 0 )
                readFIFO(CH_5, &in_1);

/* IP4 */     readFIFO(CH_6, &in_1);

              F(in_0, in_1, &out_2, &out_3);

/* OP1 */     if( −j + 2 >= 0 )
                writeFIFO(CH_2, &out_2);

/* OP2 */     writeFIFO(CH_7, &out_3);

/* OP3 */     if( i−4 == 0 )
                writeFIFO(CH_7, &out_3);
        }
    } // for j
} // for i
```

Figure 3.3: Structure of Unfolded Process F

It can be seen that process F has one so called *self-channel* or *self-edge*, i.e., channel CH_2 from/to process F. Self-channels are important in determining how to split-up processes as will be discussed later. Furthermore, it can be seen that the splitting transformation introduces a control statement inside the process (i.e., the bold modulo statement) to partition and ensure that an iteration point is executed by one partition only, and not by two partitions for example.

3.2 Challenges of Applying the Process Splitting Transformation

In this section we show performance results for two applications. These two motivating examples show that the question which transformation to apply contains many subtle parts, based on the interplay of many factors which may not be evident at first sight. This makes it difficult to select the proper process splitting transformation.

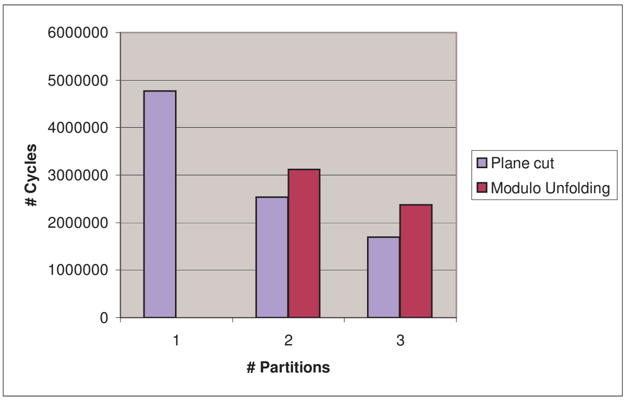
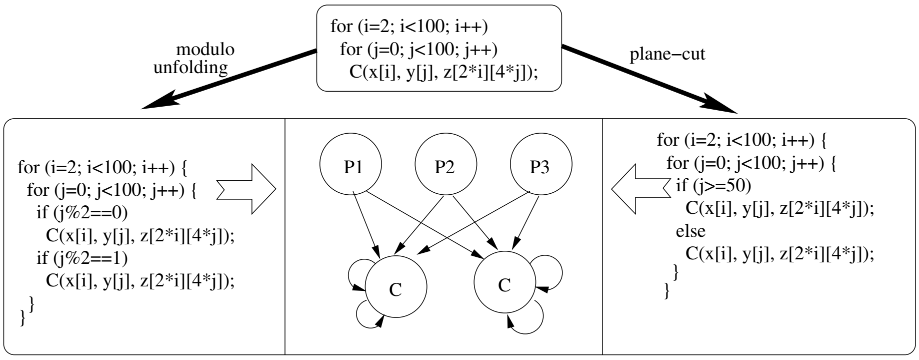


Figure 3.4: Results of Different Splittings on the ESPAM platform

The first bar in Figure 3.4 corresponds to the performance result for the unmodified application and its derived PPN in Figure 3.1 A) mapped on the ESPAM platform [60, 61]. The application is executed in 4.8 million cycles. Then, the network is balanced by applying the modulo unfolding and plane-cut transformations and thus two partitions are created for function call statement F. The second bar corresponds to the plane cut transformation and the third bar to the two times unfolded version shown in Figure 3.1 B). The fourth and fifth bars display results for creating three partitions using the same transformations. It can be seen that the plane-cut transformation is better than the modulo unfolding: 2.5 million vs. 3.1 million cycles for creating 2 partitions and 1.8 million vs. 2.2 million cycles for creating 3 partitions.

These results are surprising as the initial producer delay for the plane-cut is larger than for the modulo unfolding, but still the plane-cut transformation leads to better performance results. In this example, the number of intra and inter-process communication is not important as the cost for intra and inter-process communication are the same on the ESPAM platform. Therefore, the measured performance results can only be explained by a non-constant cost for the communication when different transformations are applied, which involves a FIFO read/write primitive and a control part when to read/write (the function workload cannot change and is constant). We observe that by introducing modulo statements, the communication (the control part) becomes more costly as the modulo expressions will appear in the definitions of the input/output ports. An example is the bold modulo statement in the `F` process in Figure 3.3. The modulo statement is introduced as a result of the transformation and is evaluated every iteration. In general, the if-conditions for reading/writing from/to FIFO channels are more expensive as more complex expressions must be evaluated.



```
for (i=2; i<100; i++)
  for (j=0; j<100; j++)
    C(x[i], y[j], z[2*i][4*j]);
```

```
for (i=2; i<100; i++) {
  for (j=0; j<100; j++) {
    if (j%2==0)
      C(x[i], y[j], z[2*i][4*j]);
    if (j%2==1)
      C(x[i], y[j], z[2*i][4*j]);
  }
}
```

```
for (i=2; i<100; i++) {
  for (j=0; j<100; j++) {
    if (j>=50)
      C(x[i], y[j], z[2*i][4*j]);
    else
      C(x[i], y[j], z[2*i][4*j]);
  }
}
```

Figure 3.5: Modulo unfolding vs. Plane-cut

Another application is shown in Figure 3.5. The initial application source-code at the top (the producer processes $P1$, $P2$, and $P3$ are omitted for the sake of brevity) is transformed by unfolding the inner loop two times: `unfold({i,j},{0,2})`, and a plane-cut on the inner loop: `planecut({i,j},{j=50})`. The PPN is topologically the same for both transformations, but internally the processes are different. In Figure 3.6, the performance results for the initial network and both transformed networks are shown. The first bar corresponds to the initial network and it shows that the application requires 22 million cycles to finish its execution. The second and third bar correspond to the plane-cut and modulo unfolding and require, respectively, 17 million and 15 million cycles. We observe that the plane-cut method is slightly worse compared to the modulo unfolding. Although there are no dependencies between the two processes executing function `C` (see Figure 3.5), the consumer processes `C` in

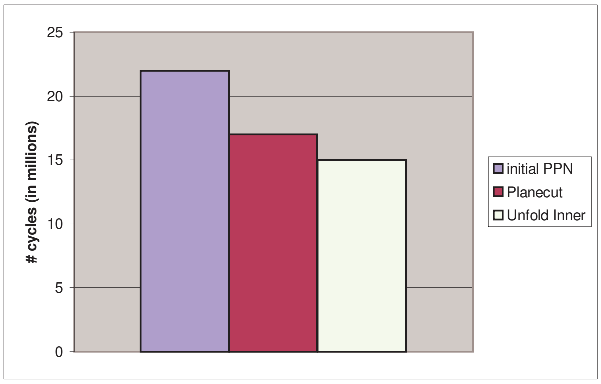


Figure 3.6: Measured Performance Results of the PPNs in Figure 3.5 on ESPAM

the plane-cut example must wait more iterations before the producer processes generate the first data compared to the modulo unfolding example (this is discussed in detail in Section 3.3.2 and in the case-studies in Section 3.5). From this example we learn that it is not enough to consider only inter-process communication and initial delay caused by other partitions, but also the delay caused by all other producers. In Section 3.3, we define the metrics that should be taken into account in applying and evaluating different transformations.

Problem Statement

There are many possibilities to partition processes as we have shown in this section. Different partitioning strategies have a significant impact on performance results and thus selecting the best partitioning strategy is crucial in achieving the best possible results. Figure 3.4 and 3.6, for example, show that it is not straightforward to select the best partitioning candidate. The challenge is to find a compile-time solution to predict the best possible partitioning and thus minimize the execution time. Therefore, one should be able to answer the following two questions:

- Given the two parameterized transformations `unfold(I,U)` and `planecut(I,P)`, which transformation should one apply for a given process to be split-up?

- For a chosen transformation, what should the parameter values be? For the `unfold` transformation, for example, one should choose one or more loop iterators to unfold and corresponding unfolding factors.

3.3 Partitioning Metrics

A process P_i has a process iteration domain D_{P_i} and is transformed by transformation H into n disjoint partitions $H(D_{P_i}) = \{D_{P_i^1}, .., D_{P_i^n}\}$. Different partitioning transformations result in partitions with different properties and in this section we discuss six metrics we have identified to evaluate different partitionings. The metrics we discuss are *i*) computation costs, *ii*) communication costs, *iii*) initial delays, *iv*) production period, *v*) data transfers, *vi*) additional control overhead.

3.3.1 Computation and Communication Costs

In each process iteration, a function is executed as illustrated in Figure 3.3 (function `F`). The complexity of this function can vary from a simple multiply-accumulate operation in a matrix multiplication kernel to a coarse grain task such as a DCT in a JPEG encoder application. The complexity of this function contributes, among other factors, to the delay at which data is produced. In determining the total execution time of a process P_i, the workload, i.e., the **computation cost**, of a process function is taken into account and is denoted by W_{P_i} (see also Section 2.5). An accurate costs estimation is thus crucial for selecting the best possible partitioning strategy and inaccurate estimations can lead to wrong decisions. We consider the function cost as an input parameter for our algorithm that can be obtained by running the function once on the target platform. We consider the function cost to be a constant value, see Section 3.6 for a discussion on this. Besides the execution of a function, a process reads from a number of input channels to get all function input arguments at each iteration. Similarly, it writes the result to a number of output channels. The FIFO read/write primitives can be supported by hardware (e.g., the ESPAM platform), or must be supported with a software implementation (e.g., the CELL). Clearly, the **communication cost** of data communication depends on the target platform and can influence the partitioning significantly. With a software implementation of FIFOs, for example, data communication can easily become more costly than the computation itself. The ratio of computation and communication is an important metric to evaluate different partitionings. To the costs for inter-process communication we refer as C_{inter} and for intra-process communication we use C_{intra}. These are constant costs to transfer a single token from a producer to a consumer process and are obtained by checking/measuring the costs for the read/write primitives on the target platforms. The reader is referred to Section 3.6 for a more in-depth discussion on using constant values for the cost of process functions and FIFO communication.

3.3.2 Initial Delay

A partition may not directly start executing its first iteration as a result of dependencies. In that case, a producer process, or another partition, is responsible for generating the required initial data.

Definition 12 *We define the **initial delay** as the number of iterations a producer executes before it generates the first data for a partition, and we denote it by $Y(D_{P_i^n})$ for a partition $D_{P_i^n}$.*

For example, the second partition $P2$ in Figure 3.2 A) must wait 2 iterations for producer $P1$ before it can start its execution and in Figure 3.2 B) the second partition can start after 1 iteration. For each partition $D_{P_i^n}$ we calculate the initial delay, which may be caused by a producer process or another partition. Each partition has a number of input ports and we determine the lexicographical minimum point of each function input argument. This point corresponds to the iteration point where data is read for the first time with respect to that function argument. Figure 3.7 shows the function call statement `F` from Figure 3.3. It has two input arguments `in0` and `in1`. At different iterations, argument `in0` is read from input ports *IP1* or *IP2*, and the second argument from input ports *IP3* or *IP4*.

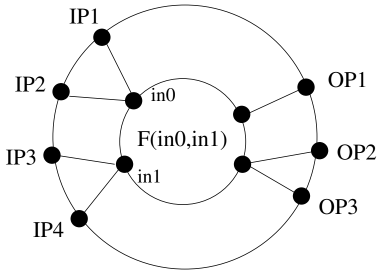


Figure 3.7: Function Input Arguments and its Delay Calculation

For each input argument, we determine the first read action by considering the lexicographical minimum point of all associated input ports. For the example above, we calculate the minimum of *IP1* and *IP2*, and then we do the same for *IP3* and *IP4*. In general, when there are x input arguments with y input ports associated to the first function argument and z ports to the last argument, we calculate the producer points as follows:

$$
\begin{aligned}
p_1 &= M(a), \;\; where \;\; a = lexmin(\bigcup_{j=1}^{y} IP_j) \\
&\vdots \qquad\qquad\qquad\quad \vdots \\
p_x &= M(b), \;\; where \;\; b = lexmin(\bigcup_{j=1}^{z} IP_j)
\end{aligned}
\tag{3.1}
$$

We apply the mapping function M (see Section 2.5) of each input port to obtain all producer points p_t where $1 \leq t \leq x$. The initial data is generated at these producer iteration points, which means that the consumer is waiting for all preceding producer iteration points to receive its initial data. Now, to calculate this initial delay, the `rank` function (see Section 2.2) is applied to a producer point returning the number of preceding iterations for a given iteration point. We calculate this offset, the initial delay Y_t, for all producer points $p_t \in D_{P_t}$ of the last partition D_{C^n} as follows:

$$
Y_t(D_{C^n}) = \begin{cases} rank(p_t, D_{P_t}) & if P_t \neq C^n \\ rank(p_t, D_{P_t}) + \sum_{x=0}^{n-1} Y(D_{C^x}) & otherwise \end{cases} \tag{3.2}
$$

It shows that if the producer P_t and consumer C^n are different processes, then the offset is calculated based only on the number of iterations of the producer process. If the producer point belongs to the same process but to a different partition, then the delay of the preceding partitions $Y(D_{C^x})$ are taken into account. The initial time $T_{C^n}^{init}$ a consumer C^n is waiting for initial data, is determined by the slowest producer. To calculate this time, we consider all $Y_t(D_{C^n})$ values as defined above. These values are multiplied by the estimated time $T_{P_t}^{iter}$ required for one process iteration, which we define with Formula 3.9 in Section 3.4, of the corresponding producer and the maximum value is taken:

$$
T_{C^n}^{init} = max_t\{Y_t(D_{C^n}) \cdot T_{P_t}^{iter}\} \tag{3.3}
$$

3.3.3 Production Period

The calculation of the initial delay is not enough to accurately estimate the execution time of a partition. For example, a producer can generate data for a consumer at its first iteration, but then it may take a number of iterations before it generates new data. This illustrates that the **production period** of a producer process is another import metric.

Definition 13 *The* ***production period*** *of a process is the number of process iterations between two consecutive data productions.*

A more elaborate example is given in Figure 3.8. Both the circles and crosses denote process iteration points. The circles indicate that data is produced for a particular consumer at that point, and the crosses indicate that no data is produced. A consumer process receiving data from these two producers is waiting 1 iteration for producer $P1$ and 10 iterations for $P2$ to generate the initial data so that the consumer can start executing. After this initial delay, producer $P2$ is producing data at each iteration, while $P1$ is producing data in either 2 or 5 iterations. We define the average production period d_{P_i} as the average number of iterations that is required to generate new data by producer P_i.

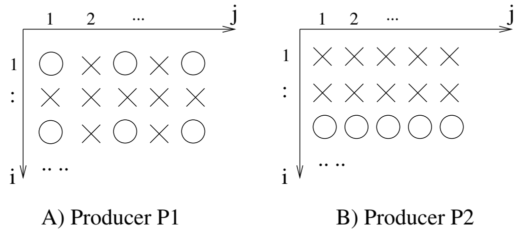


Figure 3.8: Production Period Examples

Definition 14 *The* ***average production period****, denoted by d_{P_i}, is calculated by dividing the total number of iteration points of a producer process P_i by the total number of generated data tokens:*

$$d_{P_i} = \frac{|D_{P_i}|}{|M(IP_C)|} \tag{3.4}$$

where IP_C is the input port domain of consumer process C, M is the mapping function which is used to obtain the producer iteration points for this input port domain, and D_{P_i} is the process iteration domain of the producer process P_i.

To illustrate the production period, we consider the example in Figure 3.8 and assume the iteration domain consist of 3 rows and 5 columns. The production period is $\frac{15}{6} = 2.5$ and $\frac{15}{5} = 3$ for producer $P1$ and $P2$, respectively. The time $T_{P_i}^{period}$ required to generate new data is the average production period multiplied by the required time $T_{P_i}^{iter}$ that is needed for one process iteration of a producer:

$$T_{P_i}^{period} = d_{P_i} \cdot T_{P_i}^{iter} \tag{3.5}$$

In Section 3.4, we explain how the time $T_{P_i}^{iter}$ for a process iteration is calculated.

3.3.4 Data Transfers

Different partitionings can lead to a different number of inter- and intra-process **data transfers** which is denoted by DT. A data transfer occurs when data is read/written to/from a FIFO channel. We already considered the example in Figure 3.2 A), where the plane-cut results in $4 + 4 = 8$ data transfers (the bold arrows) from one process to the other process and 40 transfers to/from the same process. In Figure 3.2 B), the partitioning strategy results in $12 + 12 = 24$ inter-process data transfers and $12 + 12 = 24$ intra-process data transfers. The number of data transfers is important. For the examples in Figure 3.2, it is clear that the plane-cut is better than the modulo unfolding if inter-process communication is costly, because there are only 8 inter-process communication compared to 24 transfers for the modulo unfolding transformation.

For a process P_i, we calculate the number of intra and inter process data transfers by considering all input/output port domains of this process and check, in the polyhedral process network, if the corresponding output/input port domains belong to the same process P_i. If this is the case, then we classify the input/output port and corresponding channel as intra-process communication, and inter-process communication otherwise. We compute the number of intra and inter process data transfers as follows:

$$\begin{aligned}
DT^{Rd}_{inter} &= \sum_i |M^i(IP_i)| \\
DT^{Rd}_{intra} &= \sum_j |M^j(IP_j)| \\
DT^{Wr}_{inter} &= \sum_k |OP_k| \\
DT^{Wr}_{intra} &= \sum_l |OP_l| \qquad (3.6)
\end{aligned}$$

Equation 3.6 shows that the size of all input port domains determine the total number of intra/inter process data transfers for data that is read. In a similar way, we define data that is written as inter/intra process data transfers by considering the output port domains.

3.3.5 Additional Control Overhead

The process partitioning transformations are equivalent to source-code transformations as already indicated and also described in [79]. In Figure 3.1 B), a function call statement is duplicated and assigned to even/odd iterations of the outer loop iterator. We have shown in Figure 3.3, that the control for reading/writing from/to FIFO channels becomes more complex as a result of the transformation. This **additional control overhead** can change the computation-communication ratio. If this is not taken into account, then execution times cannot be accurately estimated leading to incorrect predictions which transformation is better. It is very difficult however, to predict this additional control overhead as the nesting level of the if-statements are different for each application and transformation. As a result, costs for the control overhead cannot be accurately estimated at compile-time. Furthermore, it is not feasible to ask the designer to provide the costs as there may be many ports to be checked. However, there are cases when the control overhead can be safely ignored. The additional control can only change significantly the computation-communication ratio if the computational process workload is small. With coarse grain tasks, the additional control will not change significantly this ratio and it is not necessary to take this into account in the cost function. Another approach to avoid the additional control overhead is a manual modification of the generated code. In case of the modulo unfolding for example, the introduced modulo statements can be manually removed from the generated code by adjusting the loop step-size and corresponding conditions in the input/output port domains. The conditions for the plane-cut are usually much simpler and thus can be ignored in many cases. In our approach we consider examples with compute intensive tasks and change manually the generated code to remove the additional control overhead.

3.4 Compile-time Selection of Splitting Transformation

In this section, we present a solution approach and analytical model to predict, at compile-time, which transformation should be applied to obtain the best performance results. To compare different transformations, we estimate the execution time of a transformation.

Definition 15 *The **execution time of a transformation**, denoted by $T_{transformation}$, is defined as the estimated total execution time, i.e., the time required to execute all process iterations of the last processes partition which is obtained after applying the process splitting transformation.*

One solution to evaluate the different splitting transformations is simply to evaluate all possibilities. This is possible, because we define in this section a compile-time model that allows a designer to estimate the execution of a transformation. However, evaluating all possibilities is not a very attractive solution as the number of possibilities to check and evaluate can be large. Here we present an approach that does

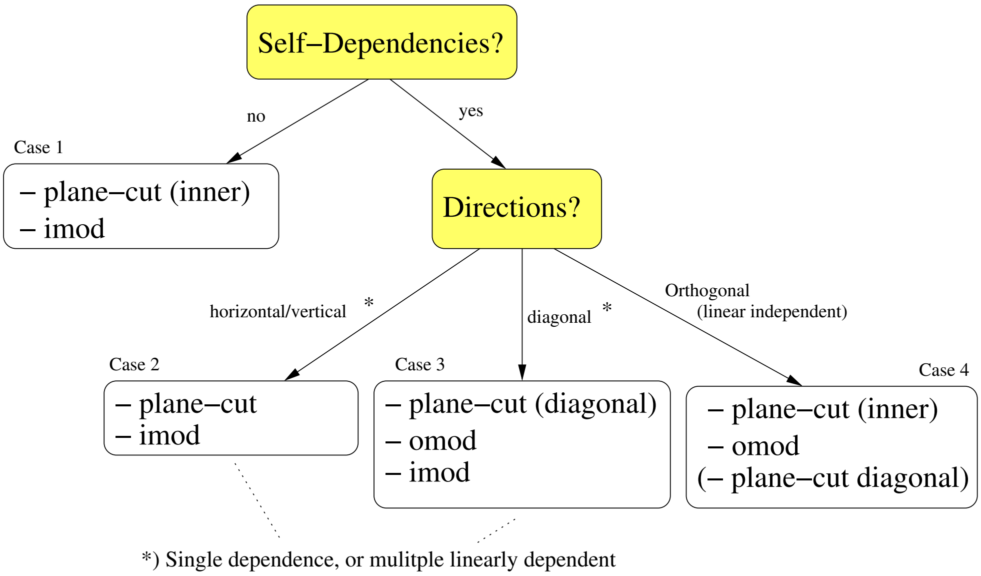


Figure 3.9: Decision Tree

not require to evaluate all possible transformations, i.e., some transformations can be excluded beforehand. To achieve this, the decision to evaluate and apply a particular transformation for a given process is made using the decision tree shown in Figure 3.9. The transformations listed in the leaf nodes of the decision tree are considered, the corresponding execution times $T_{transformation}$ are calculated using the analytical model, and the minimum value is selected. There are 5 possibilities to apply a process splitting transformation: a horizontal, vertical, diagonal plane-cut, and modulo unfolding on the inner- and outermost loop. Thus, the advantage of using the decision tree is that some possibilities do not need to be evaluated.

To balance the network, the designer starts with selecting the most computationally intensive process which will be split-up using the unfolding or plane-cut transformation. Following the decision tree, inter-process communication is avoided as much as possible by analyzing the self-dependencies of that process. If there are no self-dependencies at all before the partitioning, then a partitioning cannot introduce inter-process communication. If a single self-dependency exists, then inter-process communication can be introduced by a transformation if the transformation is not chosen carefully. Thus, the idea of the decision-tree is to avoid inter-process communication as much as possible by creating partitions that ”follow the directions” of these dependencies. In other words, producer-consumer pairs are clustered into the same partition, and not assigned to different partitions, such that the communication remains local. For example, if there exists a single horizontal dependency in a 2-dimensional process iteration domain, then vertical partitions will introduce inter-process communication, while horizontal partitions will not. For multiple dependencies that are orthogonal to each other, a partitioning with inter-process communication cannot be avoided. These cases are captured in the decision tree shown in Figure 3.9 and we discuss each of these cases in more detail. Please note that we illustrate below our approach with 2-dimensional process iteration domains, while the approach also works for processes with n-dimensional domains where $n > 2$. This is shown with a case-study in Section 3.5.2. For higher dimensional iteration domains (i.e., $n > 2$), the principle of the decision tree in Figure 3.9 remains the same, only the space spanned by the dependencies are different. Consider, for example, `case 2` of the decision tree shown in Figure 3.9. A horizontal dependency in a 2-dimensional domain is a line, while in the a 3-dimensional domain it can also be a plane. Thus, independent partitions can be created as long as the dependencies do not span the entire iteration domain.

Case 1

The first branch in the tree checks if there are any self-dependencies. If not, then only the plane-cut and modulo unfolding on the inner most loop iterator (indicated by *imod* in Figure 3.9) are compared. Thus, `case 1` is the easiest case because inter-process communication cannot be introduced by the splitting transformation since the process does not have any self-dependencies. In this case, the most important factor is the initial delay which we illustrate with the example shown in Figure 3.10.

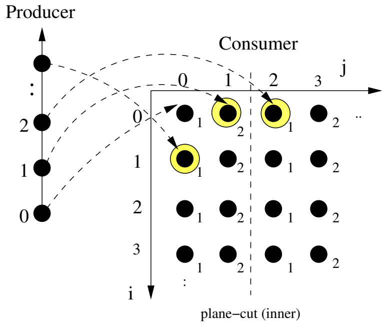


Figure 3.10: Decision Tree: Case 1

Recall from Section 3.3.2 , that the initial delay represents the number of process iterations, which a producer process needs to execute before it generates the first input data for a consumer process. This initial delay is the reason that only the plane-cutting on the inner-loop j and the modulo unfolding on the inner-loop are compared in `case 1`. For other transformations, such as modulo unfolding on the outer-loop i, the initial delay will always be larger. To illustrate this, take into account that the lexicographical order of the Consumer process iterations is from from top to bottom and from left to right, in Figure 3.10. Process iteration $(i = 0, j = 2)$ is, therefore, the *first* process iteration to be executed by the *second* process partition after applying the plane-cut transformation. Similarly, process iteration $(i = 1, j = 0)$ is the first iteration to be executed by the second partition after applying modulo unfolding on the outer-loop i, and iteration $(i = 0, j = 1)$ is the first for the unfolding transformation on the inner loop j. When data is produced in the same order as it is consumed, then it should be clear that iteration $(1, 0)$ must always wait more iterations than iterations $(0, 1)$ and $(0, 2)$ before its input data is generated by the producer. Hence, unfolding on the outer-loop i is not considered. The plane-cut is the preferred transformation to apply, because the introduced overhead of the transformation is less than modulo unfolding on the inner-loop j. However, the initial delay can be much larger and therefore the plane-cut and modulo unfolding (inner) are the two transformations that are evaluated and compared at compile-time.

Case 2 & Case 3

In case the selected process has self-dependencies, then the dependency directions are analyzed. We have identified 3 different cases as shown in Figure 3.9. For `case 2` and `case 3`, inter-process communication can still be avoided: i.e., when the process has a horizontal/vertical self-dependency, or a diagonal self-dependency. For these cases, the dependent iterations are assigned to the same partition and the communication remains local to each partition.

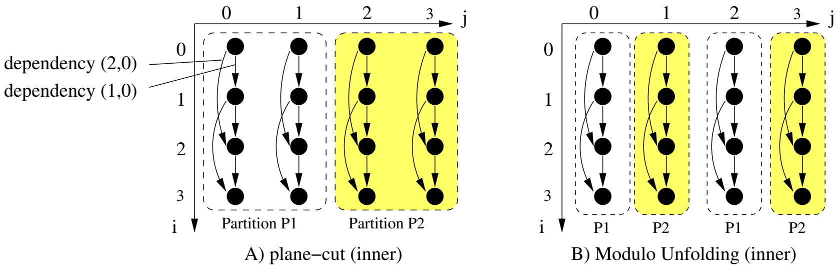


Figure 3.11: Decision Tree: Case 2

The reason to consider a single self-dependency and multiple linearly dependent self-dependencies as one case, is because inter-process communication can be avoided. Therefore, it is crucial that the multiple self-dependencies are linearly dependent, which is illustrated with the example in Figure 3.11. Intuitively, the idea is to split-up a process in such a way that the plane-cut or modulo unfolding follows "the same direction" as the linearly dependent self-dependencies. Figure 3.11 shows such an example with two different dependencies: one in the direction of $(i + 1, j + 0)$, or in short $(1, 0)$, and the other one in the direction of $(2, 0)$. These dependencies allow a partitioning that creates independent partitions, with the dependent iterations assigned to a same partition. This is illustrated with the plane-cut transformation shown in Figure 3.11 A), and the modulo unfolding on the inner loop j shown in Figure 3.11 B). It is clear that for these cases there is no difference if there is only one self-dependency, or multiple linearly dependent: the partitions will be free of any inter-process communication. In `case 2`, the modulo unfolding on the outer loop is not considered because the initial delay will always be significantly larger than the other two partitionings and therefore it will never be better. The best transformation is obtained by evaluating the execution times of the plane-cut and modulo unfolding on the inner loop iterator. While Figure 3.11 shows two processes with *vertical* self-dependencies, another possibility are *horizontal* self-dependencies, i.e., in the direction $(i+0, j+1)$. We do not further elaborate on this case as the analysis is the same as for the vertical dependencies.

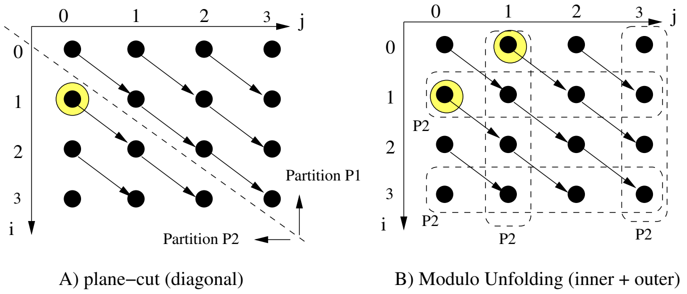
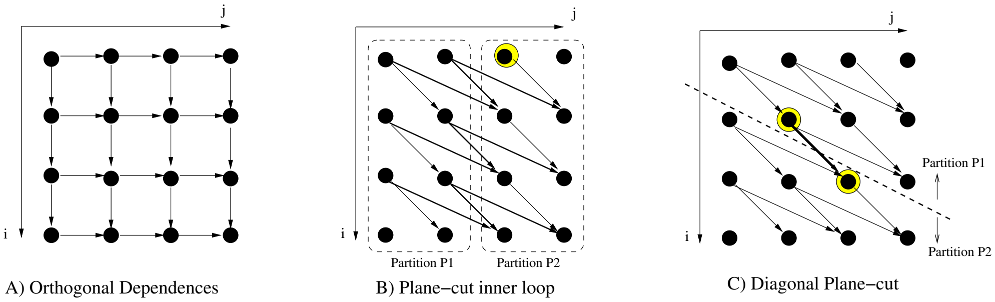


Figure 3.12: Decision Tree: Case 3

For diagonal self-dependencies, i.e., `case 3` of the decision tree, different splitting transformations should be evaluated compared to the horizontal/vertical dependencies. Figure 3.12 shows an example of a diagonal self-dependency. In this case, it is clear that a diagonal plane-cut results in partitions that do not need to communicate, as shown in Figure 3.12 A). On the other hand, the initial delay can be quite large. The *first* iteration of the *second* partition corresponds to iteration $(1, 0)$. If a producer processes first generates data for all points on the first line with $i = 0$, then the second partition cannot directly start executing. In that case, a modulo unfolding on the inner/outer loop as shown in Figure 3.12, will have much smaller initial delays: the first iterations of the second partition correspond to iterations $(0, 1)$ and $(1, 0)$ for the modulo unfolding on the inner and outer loop, respectively. Note that in this example, the first iterations of the second partition for the diagonal plane-cut and unfolding on the outermost loop i are the same, i.e., iteration $(1, 0)$, but this does not need to be the case in general. Although the modulo unfolding can have a smaller initial delay than the plane-cut transformation, the different partitions must synchronize and communicate data, which is not the case for the plane-cut. The transformation that results in the best performance results, therefore, depends on the costs for FIFO communication and the process workload, and thus the plane-cut and modulo unfolding transformations should be evaluated and compared.

Case 4

When a process has multiple linearly independent self-dependencies, it is not possible to create partitions without any inter-process communication. This corresponds to `case 4` of the decision tree. For example, when a process with a 2-dimensional process iteration domain and 2 self-dependencies that are perpendicular, i.e., they are orthogonal as shown in Figure 3.13 A), any process splitting transformation will result in inter-process communication between the different partitions.



A) Orthogonal Dependences B) Plane–cut inner loop C) Diagonal Plane–cut

Figure 3.13: Decision Tree, Case 4: Linear Independent Self-Dependencies

In Figure 3.13 A), a 2-dimensional process iteration domain is shown where the arrows denote dependencies, i.e., the dependencies are orthogonal to each other. The lexicographical order of the iteration points is from top to bottom and from left to right, (i.e., i is the outer loop and j the inner loop). Thus, for dependencies that are orthogonal to each other, unfolding on the inner most loop is not considered because this transformation leads to sequential execution of the partitions. In addition to the unfolding on the inner most loop, we also do not consider the diagonal plane-cut. The reason is that the delay for the iteration points at the diagonal of the second partition, is always much larger than the initial delay for the plane-cut on the inner loop and the modulo unfolding on the outer loop. Therefore, the plane-cut transformation on the inner loop must be compared with unfolding the outer loop (refered to as *omod*).

While orthogonal dependencies are one example of linearly independent dependencies, there are many other possibilities for two dependencies to be linearly independent. An example is shown in Figure 3.13 B). Although the dependencies are not orthogonal, they are linearly independent and a plane-cut on the inner loop $j = 2$, as shown in Figure 3.13 B), would result in 9 inter-process communications. A diagonal plane-cut, however, as shown in Figure 3.13 C), would result in only 1 inter-process communication. Furthermore, we see that for both plane-cuts, that there is no initial delay. However, there is a small delay for the first synchronization point in the diagonal plane-cut, i.e., the two highlighted iteration points in Figure 3.13 C). That is, the synchronization point is the 5th iteration point of *partition 1*, and the consumer point is the 4th iteration of *partition 2*. This means that *partition 2* is waiting 1 iteration for *partition 1* to receive its data, which does not occur in the plane-cut on the inner loop. Despite this small delay, the diagonal plane-cut can possibly be better than a plane-cut on the inner loop, depending on the costs for communication and the workload of the process function, because it has less inter-process communications. Therefore, the diagonal plane-cut, plane-cut on the inner loop, and modulo unfolding should be evaluated and compared.

Calculating the Execution Time of a Transformation

Now we present how the execution time of a transformation can be estimated and thus how transformations can be evaluated and compared. The execution time of a transformation is calculated by summing the initial time $T^{init}_{P^n_i}$ the last partition is waiting for data and the time $T^{exec}_{P^n_i}$ required for executing that last partition P^n_i:

$$T_{transformation} = T^{init}_{P^n_i} + T^{exec}_{P^n_i} \tag{3.7}$$

The initial delay $T^{init}_{P^n_i}$ is defined in Formula (3.3) and represents the maximum time before the first initial data is produced by producer processes. The execution time $T^{exec}_{P^n_i}$ for a partitioning is defined and calculated as follows:

$$T^{exec}_{P^n_i} = |D_{P^n_i}| \cdot max(T_{avg\_period}, T^{iter}_{P^n_i}) \tag{3.8}$$

In this formula, $T^{iter}_{P^n_i}$ is the execution time that is required to execute a single iteration of the last partition. The costs for executing a single process iteration includes reading all the process function input arguments, execution of the process function, and writing of the result(s) to the output port(s). If this time is less than the time required by a producer to generate data, then the execution of an iteration is dominated by the producer process. For this reason, we check if $T_{avg\_period} \geq T^{iter}_{P^n_i}$ and use this time, if necessary, multiplied by the number of process iteration points in the domain to calculate the execution time $T^{exec}_{P^n_i}$. The time required to execute a single iteration $T^{iter}_{P^n_i}$ in this formula is approximated by considering the workload $W_{P^n_i}$ of the partition P^n_i, and the average time for inter- and intra-process data transfers:

$$\begin{aligned} T^{iter}_{P^n_i} = W_{P^n_i} + \frac{DT^{Rd}_{inter}}{|D_{P^n_i}|} \cdot C^{Rd}_{inter} + \frac{DT^{Rd}_{intra}}{|D_{P^n_i}|} \cdot C^{Rd}_{intra} + \\ + \frac{DT^{Wr}_{inter}}{|D_{P^n_i}|} \cdot C^{Wr}_{inter} + \frac{DT^{Wr}_{intra}}{|D_{P^n_i}|} \cdot C^{Wr}_{intra} \end{aligned} \tag{3.9}$$

where C^{Rd}_{inter}, C^{Rd}_{intra}, C^{Wr}_{inter}, C^{Wr}_{intra} are the costs for reading and writing data for inter and intra-process communication as defined in Section 3.3.1. DT^{Rd}_{inter}, DT^{Rd}_{intra}, DT^{Wr}_{inter}, and DT^{Wr}_{intra} are, respectively, the total number of inter and intra process data transfers as defined in Formula 3.6.

If the computation of a process is not dominated by its own execution $T^{iter}_{P^n_i}$, but by the producer(s) and its large production period(s), then the average period T_{avg_period} from the producers is used to calculate the execution time of a single iteration. T_{avg_period} in Formula (3.8) corresponds to the execution time a partition is waiting for data considering its producer process. The average time is approximated taking into account the number of tokens transfered between a producer-partition pair with respect to the total number of data transfers. This number is used as a weight for the production period of a producer. The average period T_{avg_period} is calculated by summing the production period multiplied by the weight factor for all n producers:

$$T_{avg\_period} = \sum_{i=1}^{n} T^{period}_{P_i} \cdot \frac{|OP_i|}{\sum_{j=1}^{n} |OP_j|} \tag{3.10}$$

where $T^{period}_{P_i}$ corresponds to the production period as defined in Formula (3.5).

3.5 Case-Studies

In this section we present 3 different applications. The first application is an application with a single diagonal dependency for the compute process, the second application is a matrix multiplication, and the third is an application with four different producers and (initial) delays. We map the applications on the ESPAM platform [60, 61] prototyped on a Xilinx Virtex 2 FPGA and the CELL processor [34]. For programming the Xilinx Virtex 2 Pro FPGA, we use the Daedalus tool-flow [62] to implement a multi-processor system on chip. Each process from the network is mapped onto a MicroBlaze softcore processor and the processes are point-to-point connected. The FIFO channels are implemented using FSL channel components provided by Xilinx. We measured that writing/reading to/from FIFOs is completed in just 10 clock cycles. The second platform is the CELL BE processor and we use the code generator presented in [58] to map applications on the Cell processor of a `Playstation 3` console. We map the compute processes to different SPEs and source/sink processes to the PPU. The FIFO channels are implemented in local memories of both the producer and consumer process. Synchronization with signals/mailboxes ensures mutual exclusive access, which makes the read/write primitives much more expensive compared to the ESPAM platform. In these case-studies, we will not exhaustively explore all cases and transformations. Instead, we focus on `case 3` and `case 4` of the decision tree shown in Figure 3.9, because they are the most interesting from the dependencies point of view. For these two cases, we experiment with different intial delays, production periods, and inter-process communication. For each experiment, we show our approach applied on different transformations to verify that our model correctly captures these differences and thus predicts correctly the execution times.

3.5.1 Single Diagonal Dependence

In this experiment we consider a kernel as also used in [25]. This example is used to check if we can correctly predict which transformation is better by using the analytical model as we have defined in Section 3.4. The application is characterized by a compute process with a two dimensional iteration domain and a single diagonal self-dependency as shown in Figure 3.14. The application has three statements $S1$, $S2$, and $S3$ and the corresponding iteration domains and dependencies are shown in Figure 3.14 as well. In this example, a triangular assignment of process iterations to partitions using a diagonal plane-cut results in two partitions $P1$ and $P2$ free of any inter-process communication. The second partition $P2$ does not have any initial delay with respect to the first partition $P1$, but it does have a relatively large initial delay with respect to producer $S1$, i.e., 6 process iterations of $S1$, see Figure 3.14. The modulo assignment on the other hand, as also illustrated in Figure 3.14, would introduce many inter-process communications, but it has a small initial delay of only 2 iterations with respect to partition $P1$. With this experiment, we investigate if the model captures well the trade-off of having inter-process communication at low costs, or a case without any inter-process communication but with a relatively large initial delay. For testing purposes only, the iteration domains, compared to Figure 3.14, have been increased in the experiments to 20 iterations points for producer $S1$, and a 2-dimensional iteration domains of 10×10 for the compute process $S2$.

To evaluate and determine the transformation to be applied for this example, the

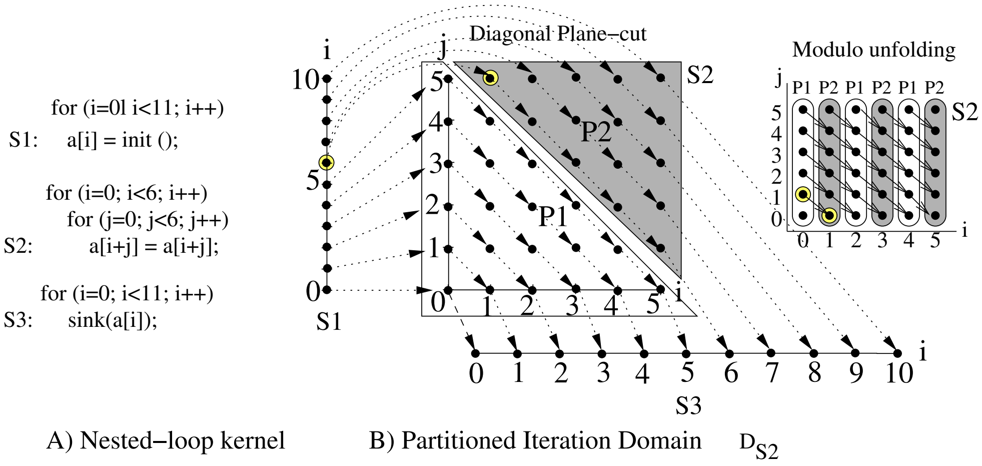


Figure 3.14: Nested-loop Program and Partitioned Dependence Graph

decision tree is checked as presented in Section 3.4. There is a self-dependency for compute process $S2$, so the right branch is taken and the dependency directions are analyzed. It is a single diagonal self-dependency and thus the decision tree indicates that we should consider the transformations in `case 3`, i.e., the transformations plane-cut and modulo unfolding on the inner and outer loop must be evaluated using Formula 3.7.

Communication	**Cost**
$C^{Rd}_{inter} : PPE \leftrightarrow SPE_i$	4000
$C^{Rd}_{inter} : SPE_i \leftrightarrow SPE_j$	160
$C^{Rd}_{intra} : SPE_i \leftrightarrow SPE_i$	10
$C^{Wr}_{inter} = C^{Wr}_{intra}$	10

Table 3.1: Communication Costs on the Cell

Table 3.1 shows the costs for communication on the Cell platform. It can be seen that there are two different costs for C^{Rd}_{inter}, because inter-process communication in the Cell can occur between the PPE and an SPE (the cost is 4000 cycles), but also between different SPEs (the cost is 160 cycles). Reading data from the same SPE, and also the writing of data, costs 10 cycles. There is no difference in the costs for writing data via inter/intra process communication, because data is always written to a local FIFO buffer of a producer process.

The two partitions $P1$ and $P2$ have a process workload of $W_{P1} = W_{P2} = 5000$.

The producer $S1$ does not have any workload such that $W_{S1} = 0$. These computation costs are shown in Table 3.2.

Computation	Cost
$W_{P1} = W_{P2}$	5000
W_{S1}	0

Table 3.2: Computation Costs on the Cell

Next, we consider the specific metric values of the *second partition* $P2$ for the different process splitting transformations as shown in Table 3.3.

Metric	planecut	unfold (outer)	unfold (inner)
Prod. Delays $Y_{S1}(D_{P2})$,$Y_{P1}(D_{P2})$	11, 0	0, 3	2, 0
Production Periods $d_{S1},\ d_{P1}$	$\frac{20}{10}(S1)$	$\frac{50}{45}(P1)$, $\frac{20}{5}(S1)$	$\frac{50}{45}(P1)$, $\frac{20}{5}(S1)$
DT^{Rd}_{inter}	9	45 + 5 = 50	45 + 5 = 50
DT^{Rd}_{intra}	36	0	0
DT^{Wr}_{inter}	9	45 + 5 = 50	45 + 5 = 50
DT^{Wr}_{intra}	36	0	0

Table 3.3: Partition P2 and its Metric Values

The first row shows that the plane-cut transformation has an initial delay of 11 iteration caused by producer $S1$. The modulo transformation on the outer loop has an initial delay of 3 iterations: the second partition $P2$ needs to wait 2 iterations for the first partition $P1$, which on its turn needs to wait 1 iteration for producer $S1$. The modulo transformation on the inner loop has an initial delay of 2 iterations, which is caused only by only one process, i.e., producer $S1$. For the plane-cut experiment, 10 data tokens are read from $S1$, which produces 20 tokens in total. Therefore, the production period is $\frac{20}{10}$. Furthermore, 9 tokens are read/written via inter-process communication, and 36 tokens are read/written via intra-process communication. For both the unfolding transformations, 50 tokens are read via inter-process communication and 0 tokens via intra-process communication. The writing of tokens is performed with 50 tokens via inter-process communication, and 0 tokens via intra-process communication. We use these metric values to calculate the execution time of the modulo unfolding transformation T_{omod} using the model defined in Formula 3.7 as follows:

$$\begin{aligned}
T_{omod} &= T^{init}_{P2} + T^{exec}_{P2} = 11108 + 305450 = 316558 \\
T^{iter}_{P2} &= 5000 + \tfrac{45}{50} \cdot 160 + \tfrac{5}{50} \cdot 4000 + \tfrac{50}{50} \cdot 10 = 5554 \\
T^{period}_{S1} &= \tfrac{20}{5} \cdot 10 = 40
\end{aligned}$$

$$
\begin{aligned}
T_{P1}^{period} &= \tfrac{50}{45} \cdot 5554 = 6788 \\
T_{avg\_period} &= \tfrac{5}{50} \cdot T_{S1}^{period} + \tfrac{45}{50} \cdot T_{P1}^{period} = \tfrac{5}{50} \cdot 40 + \tfrac{45}{50} \cdot 6788 = 6109 \\
T_{P2}^{exec} &= 50 \cdot max(5554, 6109) = 305450 \\
T_{P2}^{init} &= d_{P1} \cdot T_{P1}^{iter} = 2 \cdot 5554 = 11108
\end{aligned}
$$

If we do the same for the plane-cut and unfolding on the inner loop, then we obtain $T_{plane} = 301248$ and $T_{imod} = 304736$. Thus, we find that $T_{plane} < T_{imod} < T_{omod}$ which indicates that the plane-cut transformation can be applied best because its estimated execution time is smaller compared to the other 2 transformations. In other words, our solution approach finds that the plane-cut transformation must be applied to obtain the best performance results. This compile-time hint is correct according to the measured performance results shown in Figure 3.15.

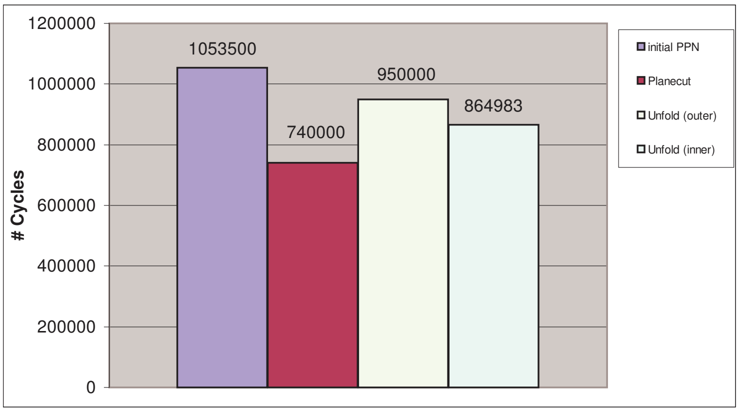


Figure 3.15: Diagonal Dependencies: Measured Performance Results on the Cell

The first bar in Figure 3.15 shows the result for the initial PPN on the Cell. The application executes in just over 1 million cycles. The second, third and fourth bar show the measured performance results for the plane-cut, and modulo unfolding on the outer and inner loop, respectively. We observe that the plane-cut is better than the 2 modulo unfolding transformations, which corresponds to the compile-time hints as calculated above. The purpose of calculating the execution time is not to estimate the real absolute performance results as close as possible, but to capture the trend of the transformations instead. The difference of the calculated execution times and the measured performance results on the Cell, for example, can be explained by the initialization and termination of SPE threads.

For the ESPAM platform we perform the same calculations and predictions. The metrics are different only for the computation and communication costs. These costs are both shown in Table 3.4, i.e., the process workload of the compute process is 5000 cycles, and the cost for reading/writing data through inter- and intra-processes communication is 10 cycles. Note that the costs for all communication types are the same on the ESPAM platform, whereas on the Cell they are different and more expensive.

Metric	**Cost**
Workload W_{P2}	5000
Comm. Costs: $C^{Rd}_{inter}, C^{Wr}_{inter}$	10
Comm. Costs: $C^{Rd}_{intra}, C^{Wr}_{intra}$	10

Table 3.4: Workload and Communication Costs on ESPAM

Using the metric values in Table 3.3 and 3.4, we calculate and predict the execution time for the three transformations on the ESPAM platform in the same way as we have shown above. We find that $T_{omod} \approx 252240$, $T_{plane} \approx 276200$, and $T_{imod} \approx 251220$ and observe that $T_{imod} < T_{omod} < T_{plane}$. Thus, the prediction is that the modulo unfolding transformation on the inner loop is better than the plane-cut and unfolding on the outer loop. The measured performance results shown in Figure 3.16 illustrate that this predictions are correct.

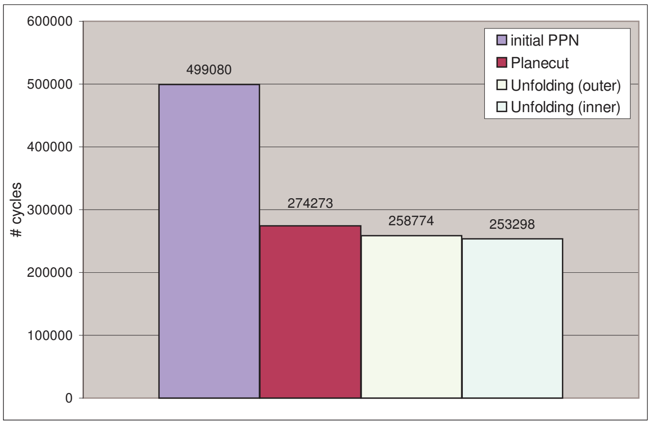


Figure 3.16: Diagonal Dependencies: Measured Performance Results on ESPAM

The first bar shows the measured performance results for the initial PPN, the second bar corresponds to the plane-cut transformation, and the third and fourth bars correspond to the results for the modulo unfolding on the outer and inner loop, respectively. It can be seen that the differences in the measured performance for the different transformations are very small, as also predicted by the estimated execution times. Despite these small differences, the predictions are correct and unfolding on the inner loop results in the best performance results. We see that the plane-cut transformation gives the worst performance results on the ESPAM platform, while it is the best alternative on the Cell. From this experiment, we conclude that the analytical model captures well the fact that the initial delay can be the dominating factor even if there is inter-process communication, i.e., for the ESPAM platform the communication costs are cheap thereby making the initial delay the crucial factor.

Note that on the ESPAM platform the estimated execution times approximate very well the actual measured execution times. For the Cell platform, the estimated execution times are less than the measured execution times for this particular experiment, because we do not take into account the overhead in SPE thread creation, synchronization, and termination, and the absolute execution times are small. For PPNs with large execution times, this overhead will not be significant and, thus, the estimated execution times will approximate better the performance results as we show in the next experiments.

3.5.2 Matrix Multiplication with Multiple Dependencies

We consider a matrix multiplication kernel implemented with a 3 dimensional loop nest structure. A single plane and its dependencies are already shown in Figure 3.2. The matrix application is an extension of this as there are a number of these planes with dependencies from each point in a plane to the same point in the next plane. The matrix multiplication application is considered because both transformations will lead to a great number of inter- and intra-process communication, such that the same transformation may have a completely different impact on the Cell than on the ESPAM platform. We verify that the analytical model and solution approach correctly predicts this behavior. The initial PPN consists of 4 processes. Processes $P1$, $P2$, $P3$ initialize, respectively, the matrix where the result is stored and the two matrices that are multiplied. Process $P4$ is the compute process and with the plane-cut and unfolding transformations we create a second process $P4'$. We consider compute process $P4$, check the decision tree in Figure 3.9 and see that there are multiple self-dependencies for this process; the horizontal and vertical dependencies are orthogonal to each other, i.e., `case 4` of the decision tree. Thus, the transformations *plane-cut on the inner loop*, and *unfolding on the outermost loop* should be evaluated. Note that we do not evaluate the diagonal plane-cut, which is taken into account when the dependencies are linearly independent and not orthogonal, see the discussion on `case4` in Section 3.4. If we experiment with a kernel of 200 × 200 × 200 iterations and apply the plane-cut transformation on the inner loop, then the first 100 iterations of the inner loop are assigned to the first partition and the remaining 100 to the sec-

Metric	**planecut**	**unfold (outer)**
$Y_{P1}(D_{P4'}), Y_{P2}(D_{P4'}), Y_{P3}(D_{P4'}), Y_{P4}(D_{P4'})$	0, 100, 100, 100	200, 200, 0, 1
Production Periods $d_{P1}, d_{P2}, d_{P3}, d_{P4}$	0, 2, 2, 100	2, 2, 0, 1
DT^{Rd}_{inter}	$40 \cdot 10^3$	$4 \cdot 10^6$
DT^{Rd}_{intra}	$12 \cdot 10^6$	$8 \cdot 10^6$
DT^{Wr}_{inter}	0	$4 \cdot 10^6$
DT^{Wr}_{intra}	$12 \cdot 10^6$	$8 \cdot 10^6$

Table 3.5: Partition $P4'$ and its Metric Values on the Cell

ond. As a result, the initial delay of the second partition is 100 iterations. In the modulo unfolding all iterations of the outer loop $i\%2 = 1$ are assigned to the first partition, and $i\%2 = 0$ to the second. As a result, the delay is 1 for the second partition. The metric values for this example are shown in Table 3.5, and it can be seen that there is a great number of inter and intra process data transfers.

Now we compute the time for both transformations by using these values in the formulas as we have presented before. We do no repeat all intermediate steps to calculate these numbers, but just give the final outcome. Note that the costs for FIFO communication is the same as in the previous experiment, see Table 3.1. The workload is also the same, i.e., 5000 cycles for the compute process(es).

The analytical model gives as a result that $T_{plane} \approx 20.4 \cdot 10^9$ and $T_{omod} \approx 21.4 \cdot 10^9$. Because the estimated time for the plane-cut transformation is less than the modulo unfolding, we conclude that the plane-cut transformation results in better performance results. As can be seen in Figure 3.17, the analytical model predicts correctly that the measured performance results on the Cell platform for the plane-cut transformation is better than the unfolding transformation. The first bar corresponds to the initial Polyhedral Process Network, which needs more than 4000 million cycles to finish its execution. The plane-cut transformed network is finished in 20071 million cycles and the unfolding transformation in 20445 million cycles.

Now we follow the same steps and predict the results for the ESPAM platform. Recall that the costs for communication and computation on the ESPAM platform is 10 clock cycles for both intra and inter process communication. The workload of the compute process(es) is 5000 cycles, and the process iteration domain is $20 \times 20 \times 20$. Thus, the total number of process iterations is 8000. After splitting the compute process, 4000 process iterations are executed by one partition, and the other 4000 process iterations by the other partition. We calculate the values and we obtain $T_{plane} \approx 20.29 \cdot 10^6$ and $T_{omod} \approx 20.24 \cdot 10^6$. Since the communication costs on the ESPAM platform are very cheap and the same for intra or inter process data transfers, we observe that the initial delay of a partition (i.e., $Y_{P4}(D_{P4'})$, see Table 3.5) is the

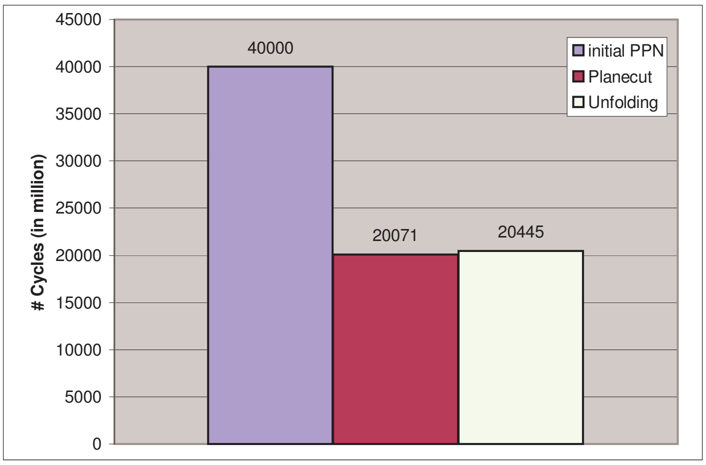


Figure 3.17: Measured Performance Results of Matrix Multiplication on the Cell

determining factor in this experiment. The analytical model predicts that the modulo unfolding transformation leads to better performance results. Figure 3.18, indeed, shows that for the measured performance results, the unfolding is better than the plane-cut.

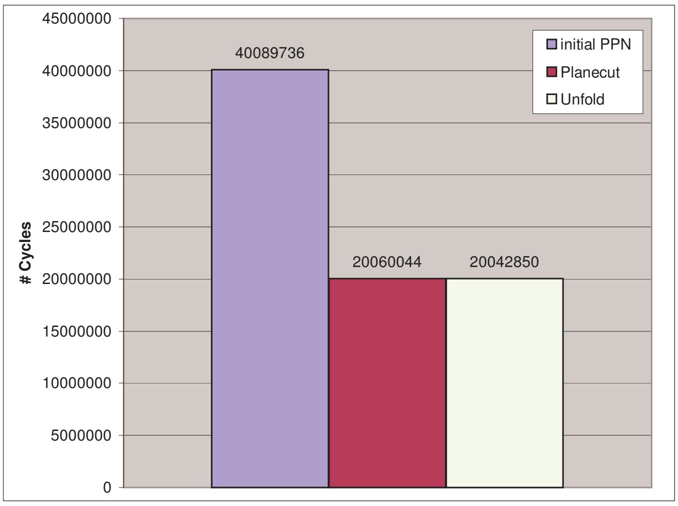


Figure 3.18: Measured Performance Results of Matrix Multiplication on ESPAM

The first bar shows the results of the matrix multiplication mapped as a Polyhedral Process Network onto the ESPAM platform. It is finished in a bit more than 40 million clock cycles. The second bar shows the result for the plane-cut transformation, which is finished in 20060044 cycles. The third bar corresponds to the modulo unfolding and we see that the unfolding transformation is slightly better than the plane-cut, i.e., it is finished in 20042850 cycles.

3.5.3 Four Producers with Delays

In this experiment, we investigate the effects of production periods on different transformations. The production period of one producer process is chosen to be much larger than the other producers. The experiment has been setup in this way, to see if the analytical model under these conditions still correctly predicts the trend. The Polyhedral Process Network (PPN) used in this experiments is derived from the nested loop program below:

```
for (i=2; i<100; i++)
  for (j=0; j<100; j++)
    x[i], y[j] = C(x[i], y[j], z[2*i][4*j], w[i][j]);
```

At each iteration, function `C` is executed and data is read from different arrays. Arrays x and y are read at each iteration and also new values are written into it. Thus, there are two (orthogonal) self-dependencies for this function call statement. The third input argument array z is indexed with expressions $2 * i$ and $4 * j$. Consecutive read accesses at the consumer process, map to iteration points at the producer process which are not consecutive. For example, iterations $(2, 0)$ and $(2, 1)$ of the consumer map to iterations $(4, 0)$ and $(4, 4)$ at the producer. In this way, we model a producer process with a production period that is different from the other processes. The fourth input argument is array w, which is written and read at each iteration of the producer and consumer. Furthermore, the first iteration of i starts at 2, such that there is an initial delay for each of the producers. The corresponding PPN is shown in Figure 3.19 A). It consists of 4 producer processes $P1, P2, P3, P4$ and a single consumer C.

To determine which transformation is better, the decision tree (see Figure 3.9) indicates that the transformations plane-cut on the inner loop and unfolding on the outer loop must be compared, i.e., it is `case 4`, as the dependencies are orthogonal in this example. The networks for the unfolding and plane-cut transformations are shown in Figure 3.19 B) and C), respectively. It can be seen in Figure 3.19 C) that, for the plane-cut transformation, the second partition $C2$ receives data from processes $P1, P2, P4, C1$. The first iteration to be executed by the second partition $C2$ is iteration point $(2, 50)$. Producer process $P1$ generates data for this point at iteration $(4, 200)$ as a result of index expressions $2*i$ and $4*j$ at the consumer $C2$. Therefore, the initial delay is $4 * 400 + 200 = 1800$ iterations with regards to producer process

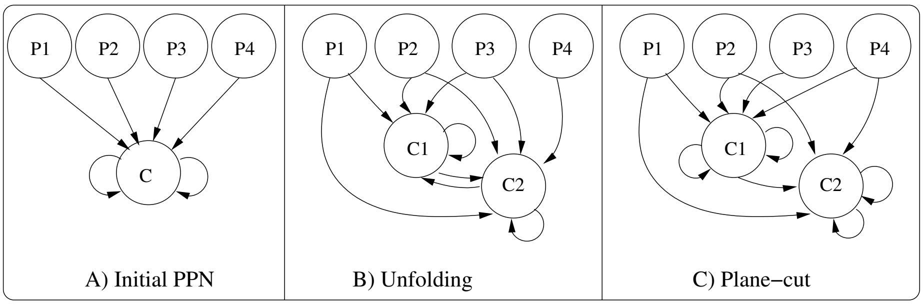


Figure 3.19: Consumer(s) with 4 Producers

$P1$. To calculate the production period, we find that producer $P1$ executes 80.000 iterations and that consumer $C2$ reads 4900 tokens from it. Therefore, the production period is $\frac{80000}{4900} \approx 16$ iterations. For the other producer process, the initial delays and production periods are calculated in a similar way and are also shown in Table 3.6. For the unfolding transformation, we see in Figure 3.19 B) that partition $C2$ depends on 5 producers. To give an example of the initial delay calculation for this transformation, we consider the first iteration point $(3, 0)$ of partition $C2$. This point is mapped to iteration point $(6, 0)$ of the producer $P1$, and hence the initial delay is $6 * 400 + 1 = 2401$. The other delays are 1201, 4, 1 and 1 iterations with respect to the remaining 4 producer processes, which is also shown in Table 3.6.

Metric	**planecut**	**unfold (outer)**
$Y_{P1}(D_{C2}), .., Y_{P4}(D_{C2}), Y_{C1}(D_{C2})$	1800, 850, 0, 3, 3	2401, 1201, 4,1,1
$d_{P1},\ d_{P2},\ d_{P3},\ d_{P4},\ d_{C1}$	16, 16, 0, 2, 50	16, 16, 2, 1, 2
DT^{Rd}_{inter}	98	4800
DT^{Rd}_{intra}	9652	4851
DT^{Wr}_{inter}	0	4800
DT^{Wr}_{intra}	9652	4851

Table 3.6: Partition C2 and its Metric Values on the Cell

The communication costs and the process workload are the same as in the previous experiments, i.e., the communication costs are shown in Table 3.3 and the workload is 5000 cycles for the compute process. If we use these metric values to calculate and predict the execution times of the transformed PPNs, we obtain that $T_{plane} \approx 39$ million cycles and $T_{omod} \approx 37$ million cycles.

The measured performance results on the Cell platform confirm that the compile-time hint is correct. The first bar in Figure 3.20 shows that the PPN is finished in

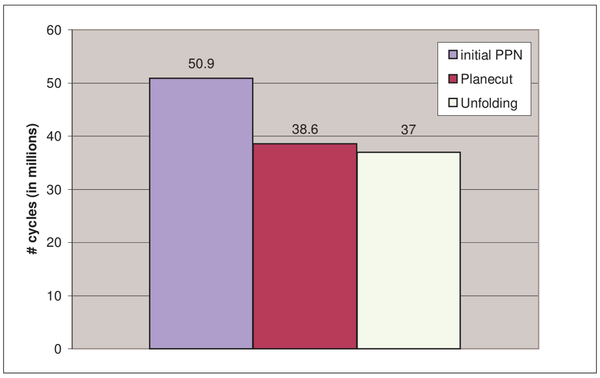


Figure 3.20: Measured Performance Results on the Cell

50.9 million cycles. The second bar corresponds to the plane-cut transformation and is finished in 38.6 million cycles, and the third bar corresponds to the unfolding transformation which is finished in 37 million cycles. We observe that, indeed, the unfolding transformation is better compared to the plane-cut transformation.

If we want to predict which transformation is better for the ESPAM platform, we repeat all steps. The only difference are the metric values for writing/reading to/from FIFO channels, which are shown in Table 3.4. If we compute the execution time for both transformations, we find $T_{plane} \approx 27.8$ million cycles and $T_{omod} \approx 25.6$ million cycles . This prediction indicates that the unfolding transformation should be applied to minimize the execution time.

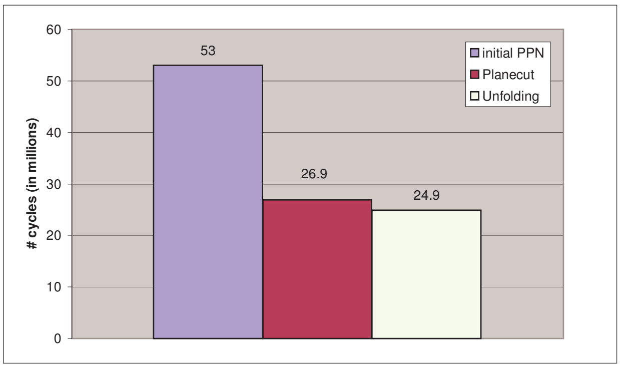


Figure 3.21: Measured Performance Results on ESPAM Platform

The measured performance results on the ESPAM platform are shown in Figure 3.21. The initial polyhedral process network is finished in 53 million cycles, the plane-cut transformed network in 26.9 million cycles, and the unfolded network in 24.9 million cycles. This confirms the prediction that the unfolding transformation leads to better performance results than the plane-cut.

3.6 Discussion and Summary

We have presented a compile-time approach to select a particular splitting transformation in order to achieve the best possible performance results. We defined the metrics that are required to make such a decision, showed how the metric values can be calculated, and presented a solution approach that uses these metric values to evaluate the different transformations to give hints to the designer. With the experiments, we have shown that our model correctly predicts which transformation can be applied best. In order to correctly predict which transformation is better, the designer needs to provide the following parameters: the workload of all functions, the costs for FIFO reading/writing on the target platform, and on which process the process splitting should be applied. A designer may therefore still have the following questions:

1. Which process should be split-up for the best performance results?
2. What if the process workload is not constant?
3. What if the cost for FIFO reading/writing is not constant?

The first two questions are related, because the process splitting transformation has the largest positive impact when it is applied on the process with the largest workload, i.e., the computationally most intensive process. To obtain the process workload, the designer has to run the functions on the target platform, or generate a profile of the application. Thus, not only the workload is obtained, but also a first indication which process can possibly be the bottleneck process of the system. For simple polyhedral process networks, i.e., if they behave like SDF graphs [47] and always read/write from/to the FIFO channels, the workload is enough to identify the bottleneck processes. However, when the process network has complicated communication patterns, it becomes very difficult to identify a single bottleneck process. The reason is that different processes can dominate the throughput at different stages of the execution of the application. This could imply that the designer needs to apply splitting on different processes in order to obtain a balanced PPN that meets the performance requirements, i.e, following the Y-chart approach, and in an iterative way, splitting can be applied consecutively on different processes. In Chapter 5, we show an example of different processes that dominate the throughput at different stages of the execution of the PPN. Moreover, an approach is presented how to apply the process splitting and merging transformation in combination that relieves the designer from the task to select a particular process. In this approach, the results of this chapter are used to decide how a process must be split up. Thus, question 1 posed here is solved as discussed in Chapter 5.

Besides selecting the best process on which the splitting transformation should be applied, a designer can have process functions with *non-constant* execution times. In the experiments discussed in Section 3.5, the workload is constant because the functions internally do not have any branches. In other words, the process workload consists of one sequence of instructions, without any branches with a varying number of instructions. Executing such functions will always require the same number of time units, i.e., it is constant. However, if a function does have branches then the execution time of that process can vary depending on which branches are taken. To model the workload of a process in this case, two options are possible: to take the worst-case execution time of the function, or to calculate an average value. It should be clear, however, that the model becomes less precise regardless whatever option the designer chooses as a solution to set the workload. The main question is: will this result in incorrect predictions what transformation should be applied? We have not investigated this with experiments, but it is not difficult to imagine that this can actually happen. If the error in the workload is significant, then the wrong value can be chosen in calculating the execution time of one process iteration as shown in Formula 3.8. On the other hand, if an imprecise workload value is used, then it is used in all evaluations of the different splitting transformations. So, in the end the trend may still be correct, but as already mentioned above, this has not been investigated. The reason is that we consider a class of applications, i.e., streaming applications, that does not expose this behavior in its process functions. Typically, data is streamed in and a series of computations are performed on the data before data is written back. In the unlikely case the process functions have some branches, then these different branches have similar computational workload.

Similar to the process workload, the costs for FIFO communication has also been modeled with a constant value. The problem is that imprecise cost estimations make evaluating the model less precise. The communication costs can have non-constant values when the platform interconnect, for example, is designed to provide a “best effort” service, instead of a “guaranteed service”. We assumed the latter and thus created platform instances that provide constant costs for FIFO communication. For embedded platforms this is a realistic assumption, because these platforms should be predictable and analyzable. In the ESPAM platform for example, the FIFO communication is implemented with hardware components and the processors can be point-to-point connected. In this case, the costs for FIFO communication is truly constant. However, if a crossbar is chosen as the interconnect for the different processors, then the FIFO costs are not constant anymore as it depends on the number of requests and the arbitration scheme of the crossbar. In [38], a performance model is introduced for different crossbar configurations, which can serve as a basis to model the FIFO costs, but we did not investigate this in the experiments. The other platform used in the experiments is the Cell platform, which uses the so called Element Interconnect Bus (EIB) [3] to connect the different processing elements. It is a bus consisting of 4 data rings and a shared command bus and multiple data transfers can be in process concurrently on each ring. We implemented FIFO communication on this provided communication infrastructure [58] and modeled the costs with a constant value. This could be inaccurate as a FIFO transfer on the CELL consists of 3 parts, i.e., 2 signals and 1 DMA transfers, and thus 3 factors influence the actual time for performing one data transfer. However, when we measure the costs for FIFO reading/writing on the real hardware, they are almost constant. Apparently, all request can be processed and no delays occur in processing them, i.e., the bus is not saturated with requests, and the costs for FIFO reading/writing are nearly constant. We were therefore able to also correctly predict the performance results for the different process splittings on the CELL platform.