On Compile-time Evaluation of Process Partitioning Transformations for Kahn Process Networks

Sjoerd Meijer
Leiden Institute of Advanced Computer Science,
Leiden University,
The Netherlands
{smeijer}@liacs.nl

Hristo Nikolov
Leiden Institute of Advanced Computer Science,
Leiden University,
The Netherlands
{nikolov}@liacs.nl

Todor Stefanov
Leiden Institute of Advanced Computer Science,
Leiden University,
The Netherlands
{stefanov}@liacs.nl

ABSTRACT

Kahn Process Networks is an appealing model of computation for programming and mapping applications onto multi-processor platforms. Autonomous processes communicate through unbounded FIFO channels in absence of a global scheduler. We derive Kahn process networks from sequential applications using the `pn` compiler, but the derived networks do not necessarily meet the performance requirements. Process partitioning transformations can achieve a more balanced network improving the performance results significantly. There are a number of process partitioning transformations that can be used, but no hints are given to the designer which transformation should be applied to minimize, for example, the execution time. Therefore, we investigate a compile-time approach for selecting the best transformation candidate and show results on a Xilinx Virtex 2 FPGA and the Cell BE processor.

Categories and Subject Descriptors

I.6.3 [**Simulation and Modeling**]: Applications; J.6 [**Computer-aided Engineering**]: Computer-aided design (CAD)

General Terms

Design, Performance

Keywords

Programming of MPSoC, Transformations, Kahn Process Networks

1. INTRODUCTION

Mapping application specifications onto multiprocessor systems is a difficult and time consuming task as it involves careful partitioning and assignment of partitions to processing elements. After identification of the different tasks, they must be mapped onto different processing elements and proper synchronization and data communication must ensure correct execution.

The Daedalus framework [9] helps the designer with the difficult task of programming and mapping applications onto Multi-Processor Systems on Chip. In this framework, the Kahn Process Network (KPN) model of computation is used as a programming model. KPNs are automatically derived from sequential nested-loop programs by using the `pn` compiler [15, 14]. In the derived parallel KPN specification, the following partitioning strategy is used: each process in the KPN corresponds to a function call statement in the sequential program. The control code for process synchronization and data communication is automatically derived thereby relieving the designer of this error-prone task. Figure 1 A) shows a two node network with 4 FIFO channels and the nested loop program from which this network 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 node over multiple nodes to better balance the network. We can achieve this, for example, as shown in Figure 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 nodes executing the `F` function resulting in a more balanced network. In [12], 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 algorithmic transformations have been defined, but the designer does not know when to apply which transformation. In our motivating examples (Section 1.2) 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 paper's contributions therefore include:

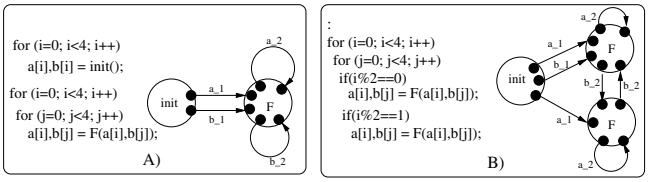


Figure 1: Kahn Process Network

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.

CODES+ISSS'09, October 11–16, 2009, Grenoble, France.

We show results for 3 different applications with different properties mapped onto the Cell processor and a Xilinx Vertex 2 FPGA.

1.1 Background and Notations

In [12], a number of parametric source-code transformations have been presented that can be used to partition processes. Two of these algorithmic transformations are the modulo unfolding and the plane-cut transformation. The former transformation is defined 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. The latter transformation is defined as `planecut(I,P)` where parameter `I` is the iteration vector and parameter `P` is a set of plane-cuts.

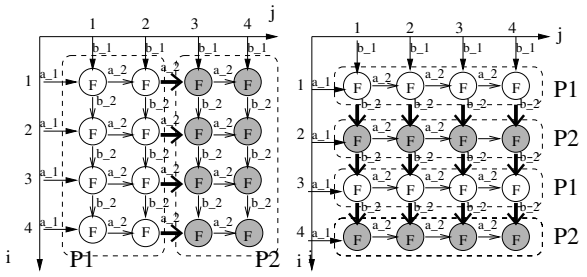


Figure 2: Partitioning Examples

Figure 2 shows the dependence graph of the program depicted in Figure 1 A) which is characterized by a two dimensional iteration space and horizontal and vertical dependencies. Loop iterator i corresponds to the outer loop and iterator j corresponds to the inner loop such that the scanning order is from top to bottom and from left to right. The arrows denote dependencies. The dependence 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 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 2 B) which corresponds to the modulo unfolding transformation presented in Figure 1 B) and is formally specified as `unfold({i,j},{2,0})`. All even iterations are assigned to $P1$ and all odd iteration points are assigned to $P2$. 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, inter-process communication occurs 4 times and the first iteration $(1, 3)$ of $P2$ must wait for 2 iterations $(1, 1)$ and $(1, 2)$ of $P1$ before it can start executing. In the modulo unfolding partitioning, $P2$ starts after 1 iteration of $P1$, but then 12 inter-process data transfers are performed. To analyze the effects of different transformations we use a formal framework to which we refer as *polyhedral process networks*. We consider polyhedral process networks that are input-output equivalent to static affine nested loop programs and use the `pn` compiler [15] to derive them. Thus, process iteration spaces, input and output port domains are polytopes that can be efficiently manipulated and analyzed. To give an example, the internal structure of one of the unfolded `F` processes and source process `init` from Figure 1 B) are given in Figure 3.

A function call statement has a number of input and output arguments and therefore the corresponding process in the Kahn Process

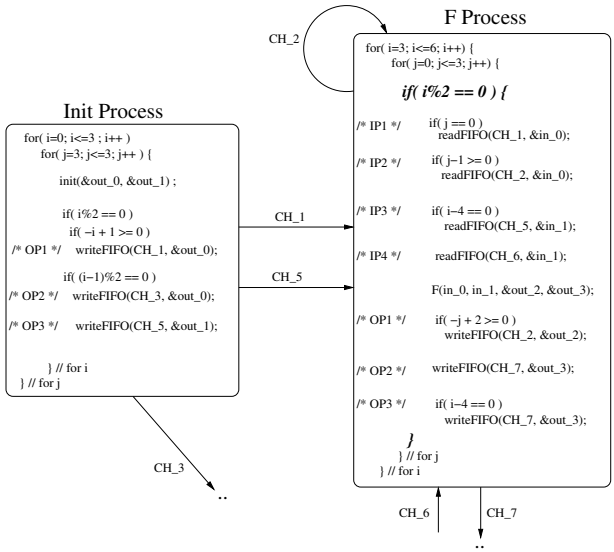


Figure 3: Structure of Unfolded Process F

Network (KPN) has a number of input/output ports to read/write data. The structure of the process consists of a list of input ports, a function call, and list of output ports. The iteration space domain of a process P_k corresponds to all iterations of statement k in the nested loop program and is defined, in general, as $D_{P_k} = \{x \in \mathbb{Z}^d \mid Ax + b \geq 0\}$. For process F in Figure 3, the iteration space domain is a two dimensional space defined as $D_F = \{(i, j) \mid 3 \leq i \leq 6 \wedge 0 \leq j \leq 3 \wedge i\%2 = 0\}$. The n-th input port domain. $IP^n_{P_k}$ of process P_k is defined as a subset of the process iteration space where data is read from an incoming FIFO channel: $IP^n_{P_k} \subseteq D_{P_k}$. Similarly, we define an output port domain $OP^n_{P_k} \subseteq D_{P_k}$ as the subset where data is written to an outgoing FIFO channel. In Figure 3, the FIFO read/write primitives are guarded by if-statements defining the input/output port domains. So, the first input port domain is defined as $IP1 = \{(i, j) \mid 3 \leq i \leq 6 \wedge i\%2 = 0 \wedge j = 0\}$. A mapping function M^l maps a point of the consumer process to a producer iteration point such that $OP_{P_k} = M^l(IP^l_{P_k})$. For example, for a producer/consumer pair in a two dimensional loop structure $\begin{pmatrix} i \\ j \end{pmatrix}$, which produces and consumes data at the same iteration point, the mapping function is specified as $M = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$. For a given domain and iteration point x, the $rank$ function returns the number of iteration points smaller than x by using an expansion of the lexicographical ordering. For example, the rank of point x in domain $D = \{(i, j) \in \mathbb{Z}^2\}$ is described by the number of points in the following set $\{(i', j') \mid i' < i \vee (i' = i \wedge j' < j)\}$. For example, the rank of point $(2, 3)$ in Figure 2, includes all points in the set $\{(i, j) \mid i < 2 \vee (i = 2 \wedge j < 3)\}$, which contains 6 points in total. The lexicographical minimum point of a set is denoted by $lexmin$ and corresponds to point $(1, 1)$ in Figure 2.

1.2 Motivating Examples

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.

The first bar in Figure 4 corresponds to the performance result for the unmodified application and its derived KPN in Figure 1 A) mapped on the ESPAM platform [7, 8]. The application is executed

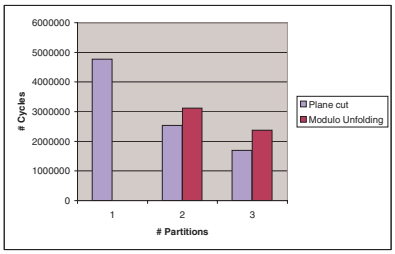


Figure 4: Results on the ESPAM platform

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 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, 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. 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.

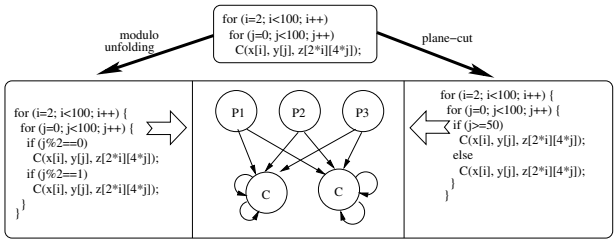


Figure 5: Modulo unfolding vs. Plane-cut

Another application is shown in Figure 5. The original 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 KPN is topologically the same for both transformation, but internally the processes are different. In Figure 6, the performance results for the original network and both transformed networks are shown. The first bar corresponds to the original network and it

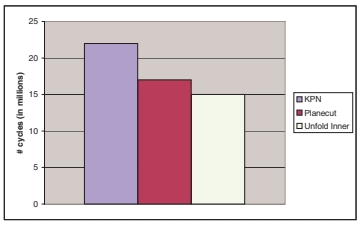


Figure 6: Results on the ESPAM platform

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 5), the consumer processes `C` in 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 2.2 and the results section). 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 2, we define the metrics that should be taken into account in applying and evaluating different transformations.

1.3 Problem Definition

There are many possibilities to partition processes as we have shown in the previous 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 4 and 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 application?
- For a chosen transformation, what should be the parameter values? For the `unfold` transformation, for example, one should choose one or more loop iterators to unfold and corresponding unfolding factor.

2. PARTITIONING METRICS

A process P_k has an iteration space D_{P_k} and is transformed by transformation H into m disjoint partitions $H(D_{P_k}) = \{D^1_{P_k}, .., D^m_{P_k}\}$. 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.

2.1 Computation and Communication Costs

At each firing of a process, a function is executed as illustrated in Figure 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, the workload, i.e., the **computation cost**, of a function for partition $D^n_{P_k}$ is taken into account and is denoted by $W(D^n_{P_k})$. An accurate costs description is thus crucial for selecting the best possible partitioning strategy and inaccurate descriptions 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. Besides the firing 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 the costs for the read/write primitives on the target platforms.

2.2 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. We define the **initial delay** as the number of iterations a producer executes before it generates the first data for a partition and denote it by $Y(D^n_{P_k})$ for a partition $D^n_{P_k}$. For example, the second partition $P2$ in Figure 2 A) must wait 2 iterations for the producer before it can start its execution and in Figure 2 B) the second partition can start after 1 iteration. For each partition $D^n_{P_k}$ 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 7 shows the function call statement `F` from Figure 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*.

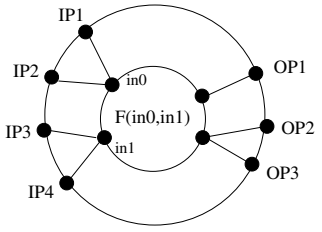


Figure 7: Function Input Arguments and its Delay Calculation

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:

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

We apply the mapping function of each input port to obtain all producer points p. 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 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, for all producer points $p \in D_P$ of the last partition D^n_C as follows:

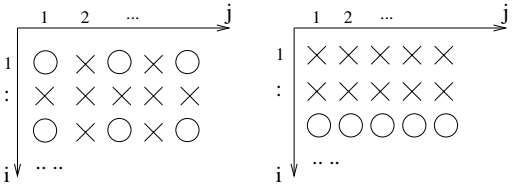
$$Y(D^n_C) = \begin{cases} rank(p, D_P) & if P \neq C \\ rank(p, D_P) + \sum_{x=0}^{n-1} Y(D^x_C) & otherwise \end{cases} \tag{2}$$

It shows that if the producer P and consumer C 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 $Y(D^x_C)$ is taken into account. The initial time T_{init} a consumer is waiting for initial data, is determined by the slowest producer. To calculate this time, we consider all $Y(D^n_C)$ values as defined above. These values are multiplied by the workload of the corresponding producer P_x and the maximum value is taken:

$$T_{init} = max_j\{Y(D^j_C) \cdot W(D_{P_j})\} \tag{3}$$

2.3 Production Period

The calculation of the initial delay is not enough to accurately estimate the execution 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 which we denote by d. A more elaborate example is given in Figure 8. Both the circles and crosses denote 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 production period d as the average number of iterations that is required to generate new data. The production period



A) Production Period P1 B) Production Period P2

Figure 8: Production Period

is calculated by dividing the total number of iteration points of a producer process by the total number of generated data tokens:

$$d = \frac{domain\ size}{\#\ generated\ tokens} \tag{4}$$

In our analytical framework, we compute this by considering the input port domains of a process/partition. We use the mapping function to obtain all corresponding producer points. The total number of points in the partition domain divided by all producer points, gives the average production period:

$$d = \frac{|D_P|}{|M(IP_C)|} \tag{5}$$

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 is the iteration space of the producer process P. To illustrate the production period, we consider the example in Figure 8 and assume the iteration spaces 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_{period} required to generate new data is the production period multiplied by the cost for executing the function of the producer P:

$$T_{period} = d \cdot W(D_P) \tag{6}$$

2.4 Data Transfers

Different partitionings can lead to a different number of inter- and intra-process **data transfers** which is denoted by DT. We already considered the example in Figure 2 A), where the plane-cut results in 4 data transfers (the bold arrows) from one process to the other process and 20 transfers to/from the same process. In Figure 2 B), the partitioning strategy results in 12 inter-process data transfers and 12 intra-process data transfers. The number of data transfers is important. For the example discussed above, it is clear that the plane-cut is better than the modulo unfolding if inter-process communication is costly, because there are only 4 inter-process communication compared to 12 transfers for the modulo unfolding transformation. For a process P_k, we calculate the number of intra and inter process communications by considering all input port domains of this process and check, in the process network, if the corresponding output port domain belongs to the same process P_k. If this is the case, then we classify the channel as intra-process communication, and inter-process communication otherwise. For this reason, the formula to calculate the number of inter- and intra-process communication is the same, just the set of ports is different:

$$DT_{inter} = \sum_i |M^i(IP_i)|$$
$$DT_{intra} = \sum_j |M^j(IP_j)|, \quad j \neq i \tag{7}$$

Equation 7 shows that the size of all input port domains determine the total number of intra/inter process data transfers.

2.5 Additional Control Overhead

The process partitioning transformations are source-code transformation as already indicated and also described in [12]. In Figure 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, that the control for reading/writing from/to FIFO channels becomes more complex as a result of the source-code 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 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.

3. SOLUTION APPROACH

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. Given an application, the decision to apply a particular transformation is made using the decision tree shown in Figure 9. The transformations listed in the leaf nodes are considered, the corresponding execution times are calculated using the analytical model, and the minimum value is selected. To balance the network, the designer starts with select-

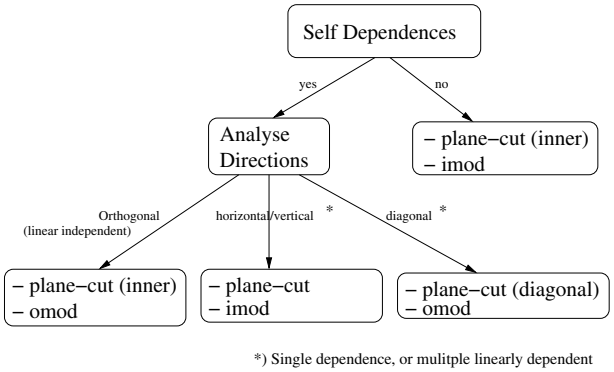


Figure 9: Decision Tree

ing the most computationally intensive process which will be partitioned using the unfolding or plane-cut transformation. In the partitioning process, 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-dependence exists, then inter-process communication can be introduced by a transformation if not chosen carefully. For example, if there exists a single horizontal dependency, 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. The first branch in the tree checks if there are any dependencies. If not, then only the plane-cut and modulo unfolding

on the inner most loop iterator (indicated by *imod* in Figure 9) are compared. The modulo unfolding on the outer loop is not considered because the initial delay will always be significantly bigger 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. If there are self-dependencies, then the dependence directions are analyzed. 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 this case, the plane-cut transformation must be compared with unfolding the outer loop (refered to as *omod*).

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

$$T_{transformation} = T_{init} + T_{exec} \tag{8}$$

The initial delay T_{init} is defined in Formula (3) and represents the maximum time before the first initial data is produced by a producer process. The execution time T_{exec} for a partitioning is defined and calculated as follows:

$$T_{exec} = |D_{{P_i}^n}| \cdot max(T_{avg\_period}, T_{iter}) \tag{9}$$

In this formula, T_{iter} is the execution time that is required to execute a single iteration of the last partition. The costs for firing a function includes reading all its input arguments, firing of the 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 $T_{avg\_period} \geq T_{iter}$ and use this time, if necessary, multiplied by the number of points in the domain to calculate the execution time T_{exec}. The time required to execute a single iteration T_{iter} in this formula is approximated by considering the workload W of the partition P_k^n, and the average time when inter- and intra-process communication occurs:

$$T_{iter} = W({P_k}^n) + \frac{DT_{inter}}{|D_{{P_k}^n}|} \cdot C_{inter}^{Rd} + \frac{DT_{intra}}{|D_{{P_k}^n}|} \cdot C_{intra}^{Rd} \tag{10}$$

where C_{inter}^{Rd} and C_{intra}^{Rd} are the costs for reading data through inter and intra-process communication as defined in Section 2.1. DT_{inter} and DT_{intra} are, respectively, the total number of inter and intra process communications as defined in Formula (7). Note that the costs for writing to FIFO channels is not taken into account as the processes do not block on writing; the consumer process is computationally much more expensive than the producer. If the computation of a process is not dominated by its own execution T_{iter}, 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 (9) 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:

$$T_{avg\_period} = \sum_{i=1}^{n} T_{period}^{i} \cdot \frac{|OP_i|}{\sum_{j=1}^{n} |OP_j|} \tag{11}$$

where T_{period} corresponds to the production period as defined in Formula (6).

4. EXPERIMENTS AND RESULTS

In this section we present 3 different applications. The first application is an application with a single diagonal dependence for the compute node, 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 [7, 8] prototyped on a Xilinx Virtex 2 FPGA and the CELL processor [3]. For programming the Xilinx Virtex 2 Pro FPGA, we use the Daedalus tool-flow [9] 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 as described in [6] 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.

4.1 Diagonal Dependence

In this experiment we consider a kernel as also used in [2]. 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 the previous section. The application is characterized by a compute node with a two dimensional iteration space and a single diagonal self-dependence as shown in Figure 10. The application has three

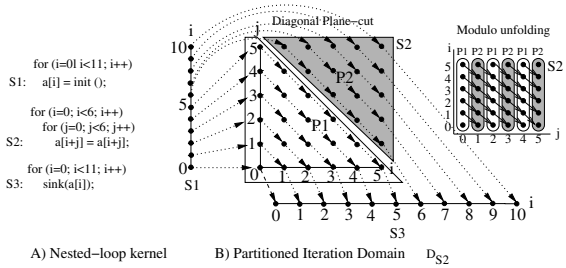


Figure 10: Nested-loop Program and Partitioned Dependence Graph

statements $S1$, $S2$, and $S3$ and the corresponding iteration spaces and dependencies are shown in Figure 10 as well. In this example, a triangular assignment of nodes to partitions using a diagonal plane-cut results in two partitions $P1$ and $P2$ free of any inter-process communication. Moreover, there is no initial delay for the partition $P2$ with respect to the first partition $P1$. The modulo assignment on the other hand, as also illustrated in Figure 10, would introduce many inter-process communications and has a small initial delay of 2 iterations. 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 spaces, compared to Figure 10, have been increased in the experiments to 20 iterations points for producer $S1$, and a 2-dimensional iteration space of 10×10 for the compute node $S2$.

To evaluate and determine the transformation to be applied for this example, the decision tree is checked as presented in Section 3. There is a self-dependence for compute node $S2$, so the left branch is taken and the dependence directions are analyzed. It is a single diagonal self-dependence and thus the decision tree indicates that the transformations plane-cut and modulo unfolding on the outer loop must be evaluated. These transformations are evaluated using Formula (8) and the metric values are required to do so. Table 1 displays the metrics and their corresponding values of the *second partition* $P2$ for the plane-cut and the modulo unfolding on the Cell platform. The first row shows that the plane-cut transformation has

Metric	planecut	unfold (outer)
Producer Delay: $Y(D_{S_1})$,$Y(D_{P_1})$	11, 0	0, 2
Production Period: d	$\frac{20}{10}(S1)$	$\frac{50}{45}(P1)$
Data transfers: DT_{inter}, DT_{intra}	10, 40	5, 45
Workload: $W(P_2)$	5680	5680
Comm. Costs: C_{inter}, C_{intra}	1000, 18	1000, 160

Table 1: Partition P2 and its Metric Values on the Cell

an initial delay of 11 iteration caused by producer $S1$. The modulo transformation has an initial delay of 2 iterations caused by the first partition $P1$. For the plane-cut experiment, 10 data tokens are read from $S1$, which produces 20 tokens in total. Therefore, the production period is $\frac{20}{10}$. The number of inter-process communications is 40 tokens, and 10 tokens of intra-process communication. We measured that the function call statement on the CELL is completed in 5680 cycles. For both transformations, the inter-process communication deals with data communication from the PowerPC to the SPE and a single transfer is finished in 1000 cycles. The intra-process communication cost is different though: for the unfolding it involves communication between 2 different SPEs (160 cycles), and for the plane-cut a single SPE (18 cycles). Given the metric values, we calculate the execution time of the modulo unfolding transformation T_{omod} as follows:

$$
\begin{aligned}
T_{omod} &= T_{init} + T_{exec} \\
T_{init} &= 2 \cdot 5680 = 11360 \\
T_{exec} &= 50 \cdot 6311 = 315550, \quad \text{since } T_{iter} < T_{avg\_period} \\
T_{avg\_period} &= \tfrac{50}{45} \cdot 5680 = 6311 \\
T_{iter} &= 5680 + \tfrac{45}{50} \cdot 160 + \tfrac{5}{50} \cdot 1000 = 5924, \quad \text{and thus:} \\
T_{omod} &= 11360 + 315550 = 326910
\end{aligned}
$$

We see that the execution time is estimated by multiplying the initial delay of 2 iterations with the workload of 5680. Then, the total number of 50 iterations are multiplied by the costs for a single iteration and added to the initial delay. If we do the same for the plane-cut, then we obtain $T_{plane} = 296880$. Because $T_{plane} < T_{omod}$, i.e., the execution time for the plane-cut transformation is smaller than the modulo unfolding, our solution approach indicates that the plane-cut transformation must be applied to obtain the best results. This compile-time hint is correct according to the performance results shown in Figure 11. 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 performance results on the Cell, for example, can be explained by the initialization and termination of SPE threads.

The first bar of Figure 11 shows the result for the original KPN on the Cell. The application executes in just over 1 million cycles. The second and third bar show the results for the plane-cut and modulo unfolding where the plane-cut is significantly better than

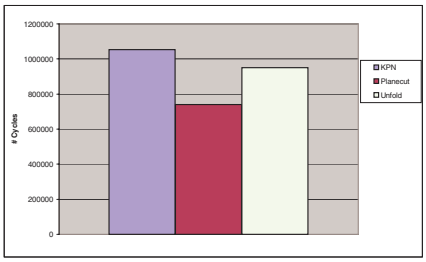


Figure 11: Diagonal dependencies: results on the Cell

the modulo, which corresponds to the compile-time hints as calculated above. For the ESPAM platform we perform the same calculations and predictions. The metrics are different only for the computation and communication costs as shown in Table 2. The workload cost is 5000 cycles, and the cost for inter- and intra-processes communication is 10 cycles.

Metric	planecut	unfold (outer)
Producer Delay: $Y(D_{S_1})$,$Y(D_{P_1})$	11, 0	0, 2
Production Period: d	$\frac{20}{10}(S1)$	$\frac{50}{45}(P1)$
Data transfers: DT_{inter}, DT_{intra}	10, 40	5, 45
Workload: $W(P_2)$	5000	5000
Comm. Costs: C_{inter}, C_{intra}	10,10	10,10

Table 2: Partition P2 and its Metric Values on ESPAM

Using these metric values, we calculate and predict the execution time for both transformations on the ESPAM platform in the same way as we have shown above. We find that $T_{mod} \approx 260000$ and $T_{plane} \approx 310000$, such that $T_{mod} < T_{plane}$. Thus, the prediction is that the modulo unfolding transformation is better and Figure 12, indeed, shows that this prediction is correct.

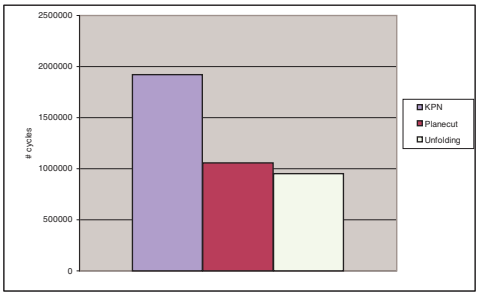


Figure 12: Diagonal dependencies: results on ESPAM

The first bar shows the results for the original KPN, and the second and third bars the performance results for plane-cut and unfolding transformations. The unfolded network is finished in less cycles compared to the plane-cut transformation. 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. For the ESPAM platform namely, the communication costs are cheap thereby making the initial delay the crucial factor.

4.2 Matrix Multiplication

We consider a matrix multiplication kernel implemented with a 3 dimensional loop nest structure. A single plane and its depen-

Metric	planecut	unfold (outer)
$\gamma(D_{P1})$, $\gamma(D_{P2})$, $\gamma(D_{P3})$, $\gamma(D_{P4})$	0, 100, 100, 100	200, 200, 0, 1
Production Period: d	0, 2, 2, 100	2, 2, 0, 1
Data transf.: $DT_{inter,intra}$	$40 \cdot 10^3$, $12 \cdot 10^6$	$8 \cdot 10^6$, $8 \cdot 10^6$
Workload: $W(P4')$	5680	5680
Comm. Cost: C_{inter}, C_{intra}	160, 18	160, 18

Table 3: Partition $P4'$ and its Metric Values on the Cell

dencies are already shown in Figure 2. The matrix application is an extension of this as there are a number 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 original KPN 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 and see that there are multiple self-dependencies for this process; the horizontal and vertical dependencies are orthogonal to each other. Therefore, we evaluate the plane-cut on the inner most loop iterator and modulo unfolding transformation on the outer most loop iterator. Note that unfolding the inner loop is not necessary here as it would result in sequential execution of the different partitions. If we experiment with a kernel of $200 \times 200 \times 200$ 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 second. As a result, the 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, and it can be seen that there are a great number of inter and intra process communications.

Now we compute the time for both transformation 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. The analytical model gives as a result that $T_{plane} \approx 2.04 \cdot 10^{10}$ and $T_{omod} \approx 2.14 \cdot 10^{10}$. 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 13, the analytical model predicts correctly that the measured performance results on the target platform for the plane-cut transformation is better than the unfolding transformation. The first bar corresponds to the unmodified Kahn process network, which needs more than 200 million cycles to finish its execution. The plane-cut transformed network is finished in 145 million cycles and the unfolding transformation in 196 million cycles. Now we follow the same steps and predict the results for the ESPAM platform. The metrics are almost the same, except the costs for communication and computation. The communication costs are 10 clock cycles, the workload is 2000 cycles, and the iteration space is $20 \times 20 \times 20$. We calculate the values and we obtain $T_{plane} \approx 420000$ and $T_{omod} \approx 402000$. Since the communication costs on the ESPAM platform are very cheap, we observe that the initial delay of a partition is the determining factor here and the analytical model, therefore, predicts that the modulo unfolding transformation leads to better performance results. Figure 14, indeed, shows that for the measured performance results, the unfolding is better than the plane-cut.

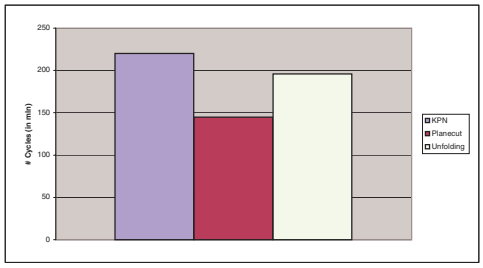


Figure 13: Matrix Multiplication on the Cell

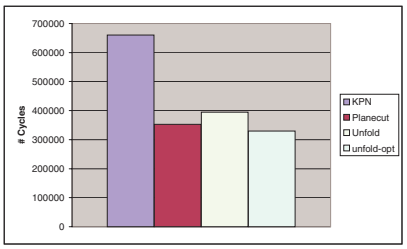


Figure 14: Matrix Multiplication on the ESPAM Platform

The first bar shows the results of the matrix multiplication mapped as a Kahn Process Network onto the ESPAM platform. It is finished in less than 700000 clock cycles. The second bar shows the result for the plane-cut transformation, which is finished in 350000 cycles. The third bar corresponds to the modulo unfolding, which is worse than the plane-cut, but this is caused by the introduced additional control overhead for the modulo statements. When this overhead is removed and the code optimized, denoted by *unfold-opt* in the fourth bar, indeed we see that the unfolding transformation is better. The model as we have defined it, captures well the influence of inter- and intra-process communication and is able to correctly predict which transformation is better for these cases.

4.3 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 and in addition to this, the producer process workload is increased. The experiment has been setup in this way, to see if the analytical model under these conditions still correctly predicts the trend. The Kahn process network 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 self-dependences for this function call statement. The third input argument array z is indexed with expressions $2 \cdot i$ and $4 \cdot j$. 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 KPN is shown in Figure 15 A). It consists of 4 producer processes $P1$, $P2$, $P3$, $P4$ and a single consumer C. The networks for the unfolding and plane-cut transformations are shown in Figure 15 B) and C), respectively.

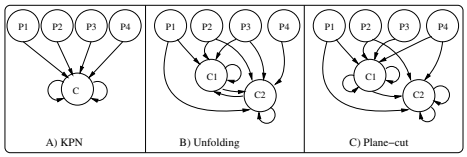


Figure 15: Consumer(s) with 4 Producers

To determine which transformation is better, the solution tree indicates that the transformations plane-cut on the inner loop and unfolding on the outer loop must be compared as there are orthogonal dependences in this example. It can be seen in Figure 15 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 $2 \cdot i$ and $4 \cdot j$ at the consumer $C2$. Therefore, the initial delay is $4 \cdot 400 + 200 = 1800$ iterations with regards to producer process $P1$. 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 4. For the unfolding transformation, we see in Figure 15 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 $6 \cdot 400 + 1 = 2401$. The other delays are 1201, 4, 1 and 1 iterations with respect to the remaining 4 producer processes, which is also shown in Table 4.

Metric	planecut	unfold (outer)
$Y(D_{P1}), .., Y(D_{P4}), Y(D_{C1})$	1800, 850, 0, 3, 3	2401, 1201, 4,1,1
Production Periods d	16, 16, 0, 2, 50	16, 16, 2, 1, 2
DT_{inter}, DT_{intra}	9652, 98	9700,4851
$W(C2)$	5680	5680
C_{inter}, C_{intra}	160,18	160,18

Table 4: Partition C2 and its Metric Values on the Cell

If we use these metric values to calculate and predict the execution times of the transformed KPNs, we obtain $T_{plane} \approx 37$ million and $T_{mod} \approx 39$ million.

The measured performance results on the Cell platform confirm that the compile-time hint is correct. The first bar in Figure 16 shows that the KPN is finished in 86 million cycles. The second bar corresponds to the plane-cut transformation and is finished in 70 million cycles, and the third bar corresponds to the unfolding

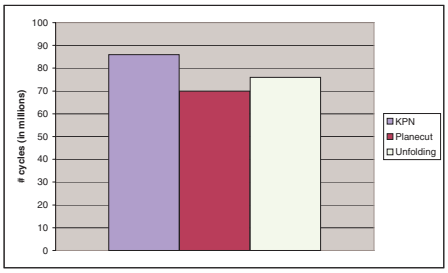


Figure 16: Results on the Cell

transformation which is finished in 76 million cycles. We observe that, indeed, the plane-cut transformation is better compared to the unfolding 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, and therefore we omit the metric values for this example. If we compute the time for both transformations, we find $T_{plane} \approx 34.75$ million and $T_{mod} \approx 34.69$ million. This prediction indicates that the unfolding transformation should be applied to minimize the execution time.

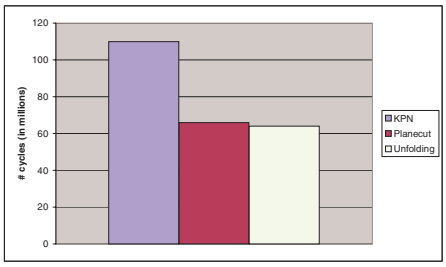


Figure 17: Results on ESPAM Platform

The measured performance results on the ESPAM platform are shown in Figure 17. The Kahn process network is finished in 110 million cycles, the plane-cut transformed network in 66 million cycles, and the unfolded network in 64 million cycles. This confirms the prediction that the unfolding transformation leads to better performance results than the plane-cut.

5. RELATED WORK

In this paper, we presented the approach we have developed to facilitate the systematic and automated derivation of alternative KPN specifications from static affine nested loop programs (SANLPs) by applying *Unfolding* and *Plain cut* algorithmic transformations. Our unfolding transformation is related to the loop unrolling techniques used in compiler design [5]. The relation is in that both transformations aim at enhancing parallelism in a sequential program. However, loop unrolling enhances instruction level parallelism by copying a loop body several times and re-indexing the variables in the body, thus creating more parallel instructions and reducing the loop control overhead. A prologue or epilogue is generated to guarantee that the unrolled version executes the correct number of iterations of the original loop. In contrast, our unfolding transformation enhances task-level parallelism by copying a loop body a number of times in such a way that these copies are mutually exclusive, thus these copies can be encapsulated in concurrent

processes. Also, our unfolding transformation does not generate prologue or epilogue code.

In [11], Sriram and Bhattacharyya describe an unfolding and retiming transformations used for improving block schedules for Homogeneous Synchronous Data Flow (HSDF) graphs by exploiting inter-iteration parallelism. This is related to our unfolding and plain cut transformations in the sense that the latter also facilitate the exploitation of inter-iteration parallelism available in a SANLP when such program is converted to a set of KPN specifications. In [10], Parhi and Messerschmitt describe a unfolding transformation developed to be applied on iterative data-flow programs. This transformation is similar to our unfolding in that both transformations increase the number of tasks in a program and unravel the hidden concurrency for static programs. The main difference between our work and the work presented in [11, 10] however, is that we have devised an approach to evaluate the quality achieved by applying the transformations when targeting a particular MPSoC platform. As we showed in this paper, there are several factors that must be taken into account when deciding what transformation to apply in order to improve performance. In contrast, in [11] the transformations are applied on the HSDF graph corresponding to an application where no information about the target implementation platform is considered.

In [13], Teich and Thiele propose an approach to partition affine dependence algorithms for mapping onto reduced/fixed size processor arrays. Their approach is based on two transformations called *Expand* and *Reduce*. Their work relates to our work presented in this paper in the sense that our approach to generate Kahn Process Networks (KPNs) using our unfolding and plane cut transformations is also an approach to partition algorithms. However, there are two important differences. First, the result of the partitioning, i.e., the generated KPNs are suitable for mapping onto heterogeneous multi-processor platforms. Second, by using our unfolding and plane cut transformations to generate KPNs we do a *reverse* partitioning compared to the approach of Teich and Thiele. They start with a dependence graph (DG) representation of an algorithm which is the partitioning of an algorithm that exploits the maximum parallelism available in an algorithm. Then they apply tiling (grouping) on the DG representation to obtain a desired partitioning in which less parallelism is exploited. In contrast, we start with a SANLP where no parallelism is exploited and by unfolding or plane cutting, we partition the computational workload of the SANLP onto several processes. That is, in the proposed approach we take into account the characteristics of a particular MPSoC target platform and evaluate the quality of different (possible) transformations, thereby, obtaining a desired partitioning in which more parallelism is exploited.

Kahn Process Networks are supported by the Ptolemy II framework [4] and the YAPI environment [1] for concurrent modeling and design of applications and systems. The designer has to specify manually the application as a Kahn Process Network and to give this network as an input to the Ptolemy II or YAPI simulation and verification engines. In many cases, manually specifying an application as a Kahn Process Network is a very time consuming and an error-prone process. Our work, presented in this paper, relates to Ptolemy II and YAPI in the sense that it can be used as a front-end tool by Ptolemy II or YAPI. This will significantly speedup the modeling effort when Kahn Process Networks are used, and modeling errors will be avoided because the proposed approach and the techniques for applying our unfolding and plain cut transformations guarantee correct-by-construction generation of Kahn Process Networks.

6. CONCLUSION

In this paper we have investigated a compile-time approach to select a 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. In the experiments we have seen that our model correctly predicts which transformation can be applied best.

Acknowledgements

The authors would like to thank Ed Deprettere and Sven Verdoolaege for their useful suggestions and comments on this work. This work is partially funded by the MEDEA+ 2A703 NEVA project.

7. REFERENCES

[1] 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.

[2] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[3] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[4] E. Lee et al. PtolemyII: Heterogeneous Concurrent Modeling and Design in Java. Technical report, University of California at Berkeley, 1999. UCB/ERL M99/40.

[5] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[6] D. Nadezhkin, S. Meijer, T. Stefanov, and E. Deprettere. Realizing fifo communication when mapping kahn process networks onto cell. In *SAMOS IX: International Symposium on Systems, Architectures, MOdeling and Simulation*, 2009.

[7] H. Nikolov, T. Stefanov, and E. Deprettere. Multi-processor system design with espam. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 211–216, New York, NY, USA, 2006. ACM.

[8] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):542–555, 2008.

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

[10] K. K. Parhi and D. G. Messerschmitt. Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding. *IEEE Transaction on Computers*, 40(2):178–195, Feb. 1991.

[11] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[12] T. Stefanov, B. Kienhuis, and E. Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 7–12, New York, NY, USA, 2002. ACM.

[13] J. Teich and L. Thiele. Exact Partitioning of Affine Dependence Algorithms. *Lecture Notes in Computer Science (LNCS), Springer*, 2268:133–151, 2002.

[14] A. Turjan. Compiling nested loop programs to process networks, 2007. PhD thesis, Leiden University, The Netherlands.

[15] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: a tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007(1):19–19, 2007.