Exploring Application Model Instances in System-Level Design

Todor Stefanov Ed Deprettere Bart Kienhuis
Leiden Institute of Advanced Computer Science, Leiden, The Netherlands
e-mail: {stefanov,edd,kienhuis}@liacs.nl

Abstract—**In the PROGRESS Artemis project [1] an architecture workbench is being developed. A case study [2] has been done in order to evaluate a set of concepts and techniques encapsulated in the workbench. In this case study we mapped a modified M-JPEG application onto a shared memory multi-processor architecture. We did a design space exploration for alternative architecture instances and mappings.**

One of the conclusions we draw from this case study is that in the exploration process we need techniques to refine a given application and to generate alternative model instances of the application. Such techniques are necessary for efficient system-level design space exploration. In this paper we present algorithmic transformation techniques for deriving a set of application model instances exploiting the task-level parallelism hidden in applications. We include these techniques in a tool called Compaan [3] and give some illustrative examples.

Keywords— **system level design, design space exploration, application model instances, algorithmic transformations**

I. Introduction

In the system-level design of embedded signal-processing systems, a system designer defines the target system as the pair *Application(s) - Architecture template*. An example of such pair is shown in the left part of Figure 1. The application specifies the functional behavior of the system. The architecture template specifies the resources of the system onto which the functional behavior is to be mapped. In this stage, a designer has to make some design decisions, for example, how to partition the application into tasks, how to map the tasks onto the architecture template, what kind of communication structure to be used in the architecture template, etc. In order to evaluate different design decisions, a system designer uses a model of the target system and does performance analysis for alternative application instances, architecture instances and mappings thereby exploring the design space of the *Application - Architecture template* pair. A general scheme for a design space exploration is the Y-chart paradigm [4]. Tools like SPADE [5] and ORAS [6] implement techniques that support the Y-chart paradigm. These techniques are efficient for the exploration of alternative architecture instances and mappings.

In this paper, we focus on techniques that support efficient exploration of alternative *application instances* in system level design. An *application instance* is every partitioning of an application into a composition of tasks that can operate in parallel. We use the Kahn Process Network (KPN) model of computation [7] to describe application instances. In the Kahn model, parallel processes communicate via unbounded FIFO channels. In Figure 1, we show a simple application and a set of model instances of this application described as KPNs (KPN_1 to KPN_5). Each application model instance differs from the others in the degree of exploited *task-level* parallelism. The latter means that the performance of the *Application - Architecture template* pair significantly depends on the application model instance mapped onto the architecture template. So, a system designer needs support to generate and explore a set of model instances of an application in order to evaluate the performance of the system and to choose an application partitioning that satisfies requirements the target system has to meet.

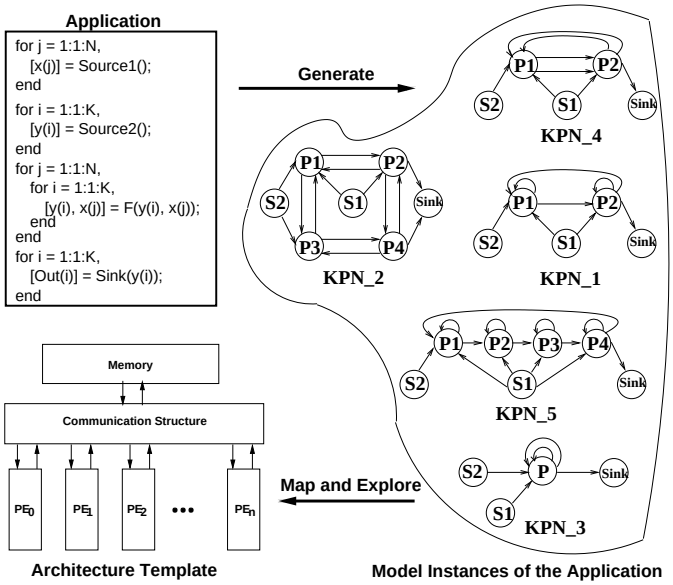

Fig. 1. Alternative model instances of the application have to be generated, mapped onto the architecture template and explored in order to evaluate the performance of the *Application-Architecture template* pair.

Generating alternative application model instances from a single application is costly in terms of the time needed for analyzing and modeling the application. Nevertheless, a huge amount of model instances of a single application exist that have to be derived and explored. Therefore, we present in this paper an *Application Transformation Layer* that automatically generates a set of application model instances (Kahn Process Networks) from an application described as an affine nested loop program (NLP) [8][9]. The application transformation layer encapsulates techniques that we have developed and implemented, namely algorithmic transformations - *Unfolding* and *Skewing* and an aggressive parallel compiler called Compaan.

In the next section we show the position of the *Application Transformation Layer* in the Y-chart paradigm. The algorithmic transformations are given in Section III. The Compaan tool is briefly described in Section IV. Finally, we show some examples and draw conclusions in Section V and Section VI, respectively.

II. The Application Transformation Layer

In this section, we discuss the application transformation layer in the context of the design space exploration process. We use this layer as an extension to the *Y-chart environment* [4].

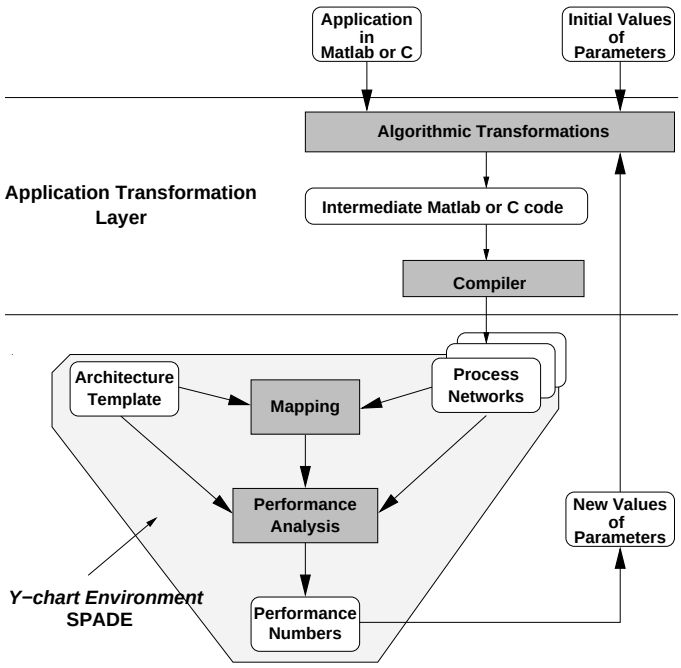

Fig. 2. The Y-chart extended with the *Application Transformation Layer*.

The positioning of the transformation layer is illustrated in Figure 2. We start with an application written in an imperative language like Matlab or C and we have to generate and explore a set of model instances (Kahn Process Networks) functionally equivalent to the application.

First, algorithmic transformations - *Unfolding* and *Skewing* are applied to the application. The transformations are controlled by a set of parameters. At the beginning some initial values are assigned to the parameters depending on the structure of the architecture template. According to these values the original code of the application is transformed and structured in a particular way in order to enhance the *task-level* parallelism in the application or to make the parallelism that is inherently available in the application explicit.

Second, the transformed code is converted automatically to a KPN description by an aggressive parallel compiler called COMPAAN. Next, we use a Y-chart environment like SPADE [5] to map the KPN onto an architecture template and to do performance analysis. The result of this performance analysis can be used to change the values of the parameters if the architecture performance is not satisfactory. Then, we repeat again the procedure described above which is actually a design space exploration of alternative model instances of the application. This is shown in Figure 2 as a feed-back arrow to the transformation layer.

By changing only the values of the parameters, the application transformation layer automatically generates a set of KPNs corresponding to a single application. The difference among the KPNs is the degree of the task-level parallelism which is exploited. Till the end of this paper we describe in more details the techniques and tools we have developed and incorporated in the transformation layer.

III. Algorithmic Transformations

In this section, we present two algorithmic transformations, namely *Unfolding* and *Skewing*. These transformations take as an input an affine nested loop program (NLP) [8] and a set of parameters. As an output of the unfolding transformation an affine nested loop program is generated which is functionally equivalent to the input program but with enhanced task-level parallelism. The skewing transformation makes the parallelism inside the input affine nested loop program explicit.

First, we explain what *unfolding* and *skewing* mean in the context of our algorithmic transformations. Next, we define the unfolding and skewing transformations as procedures that operate on an affine nested loop program. For convenience, in our further explanations, we assume that affine nested loop programs (NLP) are expressed as a Matlab code.

A. Unfolding and Skewing

Consider the application program (NLP) and its dependence graph shown in Figure 3-a).

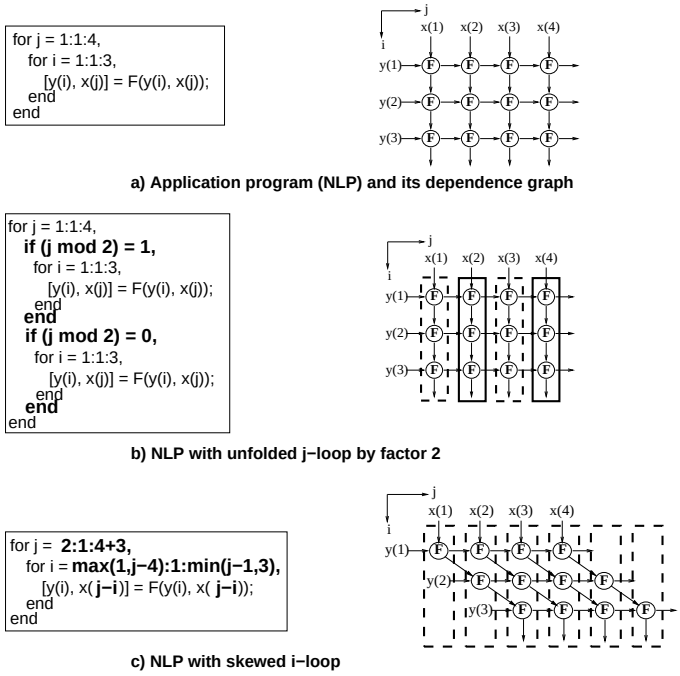

Fig. 3. Simple example illustrating the unfolding and skewing transformations.

The NLP has two loops (with iterators j, i) which can be unfolded. For example, unfolding the j-loop of the NLP by a factor of 2 creates a new NLP that has two copies of the j-loop body bounded by some "if"-statements as it is shown in Figure 3-b). In the conditions of these "if"-statements we use the *modulo* operation in a particular way to make the execution of the two pieces of code inside the "if"-statements mutually exclusive. This property can be exploited by an aggressive parallel compiler to partition the program into two processes (tasks) that can operate in parallel. The graphical interpretation is given

by the dependence graph in Figure 3-b). The first process will execute the nodes bounded by the dashed boxes. The second process will execute the nodes bounded by the solid boxes. An example of the network connecting these two processes is show in Figure 8 - see KPN_1.

Now, we consider the same application program (NLP) shown in Figure 3-a), but we skew the i-loop. The skewing creates a new NLP in which the bounds of the loops and the indexes of the variables are changed in a particular way to make the parallelism inside the original NLP explicit. The resulting NLP and its dependence graph are shown in Figure 3-c). The dependence graph shows that the nodes inside a dashed box can be executed in parallel because there is no data-dependence among these nodes. This property can be exploited by an aggressive parallel compiler in combination with the unfolding described above to partition the program into processes (tasks) that run in parallel. Also, inside these processes some pieces of code can be executed in parallel or in pipeline fashion.

B. Unfolding procedure

Let NLP be an N-deep affine nested loop program with an iteration vector $I = \{i_1, i_2, ..., i_N\}$. For each $i_k \in I \mid k = 1, 2, .., N$ a parameter $u_k \in \mathbb{N}$ is associated. All these parameters form a parameter vector $U = \{u_1, u_2, ..., u_N\}$ which we call *unfolding vector*. We define a transformation *UNFOLD(NLP,U,I)* which is described in Figure 4.

```
1  UNFOLD(NLP, U, I) {

   if (I is empty set) {

5      print(NLP);
       return();

   } else {

10   a = first element of the set I;
     b = first element of the set U;

     loop = take the code from the beginning of NLP
     till the "for" statement with loop iterator a,
15   including;
     body = take the body of loop a from NLP;

     print(loop);

20   for (k = 1; k <= b; k++) {

       println("if("+a+"mod"+b+")="+b-k+',');

       temp1 =  the set U without the first element;
25     temp2 =  the set I without the first element;
       UNFOLD(body, temp1, temp2);

       println("end");

30   }

     println("end");
     return();
   }
35 }
```

Fig. 4. Pseudo code describing the **UNFOLD** transformation.

The pseudo code in Figure 4 describes the unfolding transformation as a recursive procedure. This procedure operates on the affine nested loop program NLP with its iteration vector I using the value of the unfolding vector U. In order to explain the behavior of the procedure *UNFOLD* we consider the following simple example. Let NLP be the program shown in the left part of Figure 5. NLP has only one loop with an iterator (index) i. Hence, the iteration vector I corresponding to NLP has only one element $I = \{i\}$ and the unfolding vector U has also one element $U = \{u\}$. In our example the parameter u is equal to 10.

Following the procedure *UNFOLD*, first we check whether I is an empty set. In our example we start with $I = \{i\}$ which is not an empty set. Then, we initialize four variables, see lines 10, 11, 13 and 16 in Figure 4. As a result we have: variable a takes the character 'i'; variable b = 10; variable *loop* takes the string "$for\ i = 1 : 1 : N$," and *body* takes the code in the body of the loop with iterator 'i'. This code is marked in Figure 5 as a rectangle. The line 18 in Figure 4 prints to the output the variable *loop*. The result is shown in Figure 5 - the first line in the unfolded NLP. Executing lines 20 till 32 in Figure 4 will generate the rest of the code of the unfolded NLP in Figure 5. As a result the unfolded NLP in Figure 5 has ten copies of the loop body bounded by "if" statements with a "mod" statement making them mutually exclusive.

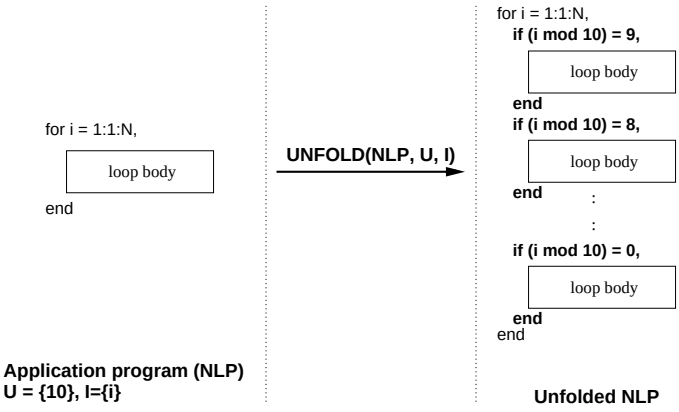

Fig. 5. Simple example illustrating the UNFOLD() transformation shown in Figure 4.

The example in Figure 5 shows that the input NLP is transformed to a functionally equivalent NLP which we call an unfolded NLP. The unfolded NLP can be easily converted into ten tasks that operate in parallel. That is why we say that the unfolded NLP has enhanced task-level parallelism compared with the input NLP.

C. Skewing procedure

Let NLP be an N-deep affine nested loop program with an iteration vector $I = \{i_1, i_2, ..., i_N\}$. For each $i_k \in I \mid k = 1, 2, .., N$ a parameter vector $D_k = \{m_1, m_2, ..., m_N\}$ is associated, where each $m_p \in \mathbb{N} \mid p = 1, 2, .., N$. All parameter vectors form a parameter matrix

$$M = \{D_1^T, D_2^T, ..., D_N^T\} = \begin{bmatrix} m_{11} & .. & m_{1N} \\ .. & .. & .. \\ m_{N1} & .. & m_{NN} \end{bmatrix}$$

which we call *skewing matrix*. We require M to be unimodular. We define a transformation *SKEW(NLP,M)* described below:

- **STEP1 -** Represent the iteration space of NLP as a polytope

$P = \{I \in Z^n \mid A.I \geq b\}$, where A is an integer matrix and b is an integer vector;

- **STEP2** - Use the *skewing matrix* M to transform P as follows:
  $A.M^{-1}.M.I \geq b \implies A^{'}.I^{'} \geq b$,
  where $A^{'} = A.M^{-1}$ and $I^{'} = M.I$;
- **STEP3** - Use the Fourier-Motzkin (FM) procedure [10] to represent the iteration space, described by $A^{'}.I^{'} \geq b$, in terms of nested loops. This is the new iteration space of NLP with iteration vector $I^{'}$;
- **STEP4** - Change all indexes of the variables in NLP according to the equation $I = M^{-1}.I^{'}$.

The four steps described above are illustrated in Figure 6 in the context of a simple example. We start with a 2-deep affine nested loop program and a skewing matrix $M = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$. In STEP1, the ranges of the loop indexes j and i are represented as a system of linear inequalities in a matrix form $A.I \geq b$. Next, we use the skewing matrix M to do the mathematical manipulations described in STEP2. As a result we have a new iteration space for the input NLP, defined by the loop indexes j' and i' and bounded by the system $A'.[j', i']^T \geq b$. The Fourier-Motzkin (FM) procedure is used to represent the new iteration space as nested loops as it is shown in Figure 6 - STEP3. After this step all variables inside the loops are still indexed by the old indexes j and i. We have to replace them with the new indexes j' and i'. In order to do this we know from STEP2 that $[j', i']^T = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} .[j, i]^T$, which implies that $[j, i]^T = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} .[j', i']^T$. So, we have to replace index j with $j' - i'$ and index i with i' in all variables. This is illustrated in Figure 6 - STEP4.

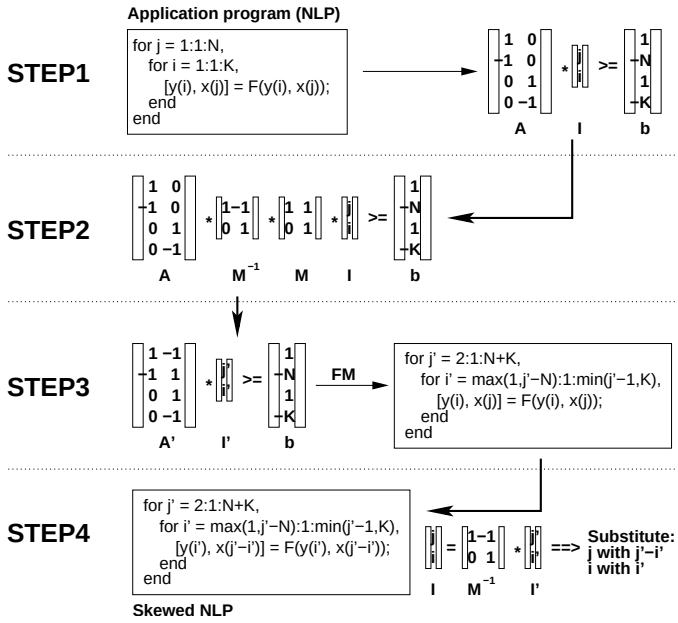

Fig. 6. Simple example illustrating the four steps in the SKEW(NLP,M) procedure.

The *Unfolding* and *Skewing* transformations presented above are related to the unfolding and retiming transformation techniques used in the Signal-Processing community [11]. Also, they are related to the loop unrolling and loop skewing techniques used in compiler design [12]. However, there are some important differences:

- we use our transformations for generating a set of Kahn Process Networks corresponding to an application (nested loop program);
- we developed a procedure to do this transformations on the algorithmic (source code) level, whereas in [11] similar transformations are applied on signal-flow graphs, data-flow graphs or dependence graphs corresponding to an algorithm.
- our transformations aim at exposing and exploiting the task-level parallelism available in an application, whereas the transformations in [12] aim at exploiting fine-grain instruction-level parallelism.

IV. COMPAAN

COMPAAN (Compilation of Matlab to Process Networks) [3] is a method and tool set for transforming affine nested loop programs (NLP) [8][9] written in Matlab into a Kahn Process Network (KPN) specification. COMPAAN consists of a number of steps shown in Figure 7, where a box represents a result and an ellipsoid represents an action or tool.

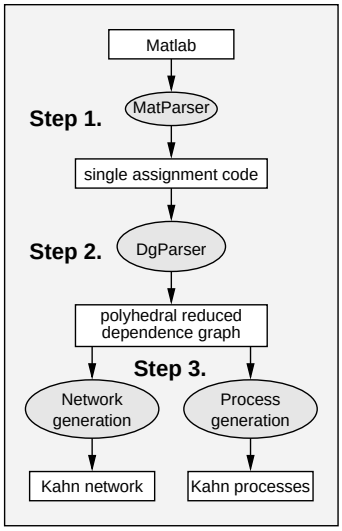

Fig. 7. The three major steps in COMPAAN.

COMPAAN starts the transformation by converting a Matlab specification into a *single assignment code* (SAC) specification. SAC describes all parallelism available in the original Matlab specification. The tool which does the Matlab-to-SAC transformation is MATPARSER [13]. MATPARSER is an *array dataflow analysis* compiler that finds all parallelism available in NLPs written in Matlab using a very aggressive *data-dependency analysis* technique. This technique is based on *parametric integer linear programming* [14]. We focus on Matlab since many signal-processing algorithms are written in this language. Just by writing another front-end language pre-processor, MatParser can also operate on NLPs written in other languages, for example C.

In the second step shown in Figure 7, a tool called DG-PARSER [9] converts the SAC description into a *Polyhedral Reduced Dependence Graph* (PRDG) [15] description. The PRDG is a compact representation of a dependence graph (DG) using parameterized polyhedra, making a DG description more suitable for mathematical manipulations.

Finally, the PRDG description is used to generate the Kahn Process Network description and the individual processes. The PANDA tool [16] does this - Step 3 in Figure 7. Deriving the network description is straightforward as it follows the topology of the PRDG. Each node of the PRDG corresponds to a Kahn process and each edge represents an unbounded FIFO channel. Generating the process description of each individual process is done in 3 steps: *domain scanning*, *domain reconstruction*, and *linearization*. For more details we refer the reader to [16].

V. EXAMPLES

In this section, we demonstrate the use of our algorithmic transformations in combination with the COMPAAN tool. We show how, merely by changing the values of the parameters, a set of Kahn Process Networks (KPN) can be easily generated from a single application.

Consider the application shown in the top-left corner of Figure 8. It is a 2-deep affine nested loop program written in Matlab. In Figure 8 first we apply the unfolding transformation on our application and then we use COMPAAN to convert the transformed code into a KPN description.

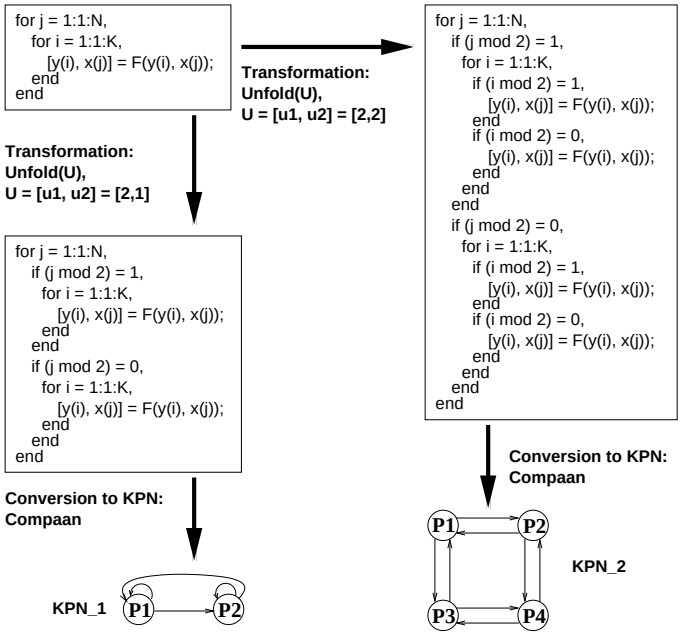

Fig. 8. An example of generating two possible Kahn Process Networks from a single application using the *unfolding* transformation and the COMPAAN tool.

We assign two different values to the parameter vector U, namely $U = [2, 1]$ and $U = [2, 2]$. As a result we obtain two different KPNs. They have different numbers of processes and different communication structures (see Figure 8- KPN_1 and KPN_2).

In Figure 9, we show another example in which we use the same application as in Figure 8. We obtain KPN_3, which has only one process, by applying the skewing transformation with a parameter matrix $M = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$. Also, we show that the skewing transformation and the unfolding transformation can be applied both in combination. KPN_4 in Figure 9 is derived by applying first the skewing transformation with $M = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ and then the unfolding transformation with $U = [2, 1]$.

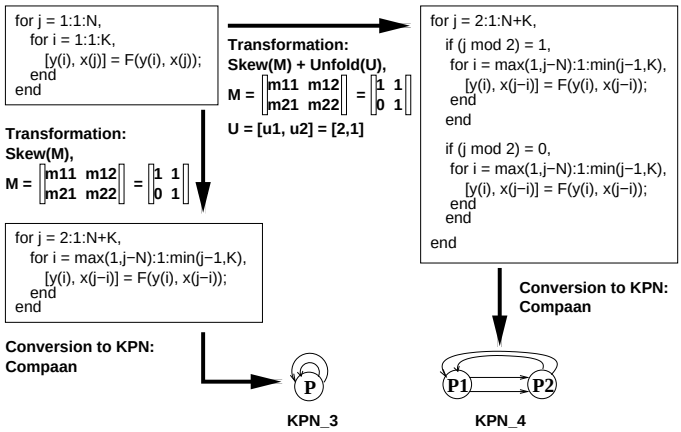

Fig. 9. An example of generating two possible Kahn Process Networks from a single application using the *skewing* and *unfolding* transformations and the COMPAAN tool.

VI. CONCLUSIONS

In this paper, we presented algorithmic transformation techniques - *Unfolding* and *Skewing* for deriving a set of application model instances (Kahn Process Networks) from a single application. These techniques support a system designer in exploring alternative model instances of the application mapped onto an architecture template. We developed and implemented the *Unfolding* and *Skewing* transformations in the context of the COMPAAN tool set. The latter means that the process of deriving alternative model instances is fully automated for applications described as affine nested loop programs.

We are currently experimenting the presented techniques on some algorithms used in signal processing applications. The preliminary results show that we are able to generate up to 30 alternative model instances (Kahn Process Networks) from a single algorithm in 8 hours. In comparison, a system designer can generate manually less than one model instance for the same period of time. Therefore, the presented techniques support a system designer to speedup significantly the process of exploring alternative application instances in system level design.

REFERENCES

[1] Andy Pimentel, Pieter van der Wolf, Bob Hertzberger, Ed Deprettere, Jos T.J. van Eijndhoven, and Stamatis Vassiliadis, "The Artemis architecture workbench," in *Progress Workshop 2000*, Utrecht, The Netherlands, Oct. 13 2000.

[2] Todor Stefanov, Paul Lieverse, Ed Deprettere, and Pieter van der Wolf, "Y-chart Based System Level Performance Analysis: An M-JPEG Case Study," in *Proc. 2000 Workshop on Embedded Systems (Progress'00)*, Utrecht, The Netherlands, Oct. 13 2000, pp. 113–124.

[3] Bart Kienhuis, Edwin Rijpkema, and Ed F. Deprettere, "Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Ar-

chitectures," in *Proc. 8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, CA, USA, May 3-5 2000.

[4] Bart Kienhuis, "Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools," Jan. 1999, PhD thesis, Delft University of Technology, The Netherlands.

[5] Paul Lieverse, Pieter van der Wolf, Ed Deprettere, and Kees Vissers, "A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems," in *Proc. 1999 Workshop on Signal Processing Systems (SiPS'99)*, Taipei, Taiwan, Oct. 20-22 1999, pp. 181–190.

[6] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "The Construction of a Retargetable Simulator for an Architecture Template," in *Proc. 6-th Int. Workshop on Hardware/Software Codesign (CODES'98)*, Seattle, Washington, Mar. 15-18 1998.

[7] Gilles Kahn, "The semantics of a simple language for parallel programming," in *Proc. of the IFIP Congress 74*. 1974, North-Holland Publishing Co.

[8] Ed F. Deprettere, Peter Held, and Paul Wielage, "Model and Methods for Regular Array Design," *Int. Journal of High Speed Electronics and Systems*, vol. 4, no. 2, pp. 133–201, 1993.

[9] Peter Held, "Functional Design of Data-Flow Networks," 1996, PhD thesis, Delft University of Technology, The Netherlands.

[10] C. Ancourt and F. Irigoin, "Scanning polyhedra with DO loops," in *Proc. ACM SIGPLAN'91*, June 1991, pp. 39–50.

[11] Keshab Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley & Sons, Inc., 1999.

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

[13] Bart Kienhuis, "MatParser: An array dataflow analysis compiler," Tech. Rep., University of California at Berkeley, 2000, UCB/ERL M00/9.

[14] Paul Feautrier, "Parametric integer programming," Operations Research, 22(3):243-268, 1988.

[15] Edwin Rijpkema, Ed F. Deprettere, and Bart Kienhuis, "Deriving Process Networks from Nested Loop Algorithms," *Parallel Processing Letters*, vol. 10, no. 2, pp. 165–176, 2000.

[16] Ed F. Deprettere, Edwin Rijpkema, Paul Lieverse, and Bart Kienhuis, "High Level Modeling for Parallel Executions of Nested Loop Algorithms," in *Proc. IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'2000)*, Boston Massachusetts, USA, July 10-12 2000.