Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances

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

Bart Kienhuis
Leiden Institute of Advanced Computer Science
Leiden University
The Netherlands

Ed Deprettere
Leiden Institute of Advanced Computer Science
Leiden University
The Netherlands

ABSTRACT

Following the Y-chart paradigm for designing a system, an application and an architecture are modeled separately and mapped onto each other in an explicit design step. Next, a performance analysis for alternative application instances, architecture instances and mappings has to be done, thereby exploring the design space of the target system. Deriving alternative application instances is not trivially done. Nevertheless, many instances of a single application exist that are worth to be derived for exploration. In this paper, we present algorithmic transformation techniques for systematic and fast generation of alternative application instances that express task-level concurrency hidden in an application in some degree of explicitness. These techniques help a system designer to speedup significantly the design space exploration process.

Keywords

system-level design, design space exploration, application instances, algorithmic transformations

1. INTRODUCTION

In system-level design of embedded signal-processing systems, a system designer sees the target system as the pair *Application(s) specification - Architecture template*. An example of such a pair is shown in the left part of Figure 1. The application specification provides the functional behavior of the system. The architecture template specifies the organization of 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 use 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* pair.

A general scheme for a design space exploration is the Y-chart paradigm [4]. Tools like SPADE [9] and ORAS [6] implement techniques that support the Y-chart paradigm but they focus only on the exploration of alternative architecture instances and mappings [8]. In this paper, however, 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 concurrent tasks.* We use the Kahn Process Network (KPN) model of computation [3] to describe application instances. In the Kahn model, concurrent processes communicate via unbounded FIFO channels. In Figure 1, we show a simple application and a set of alternative KPN instances of this application (KPN_1 to KPN_5). Each application instance differs from the others in the degree of exploited *task-level* parallelism. The performance of the *Application - Architecture* pair can significantly depend on the application instance. So, a system designer needs support to generate and explore a set of 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.

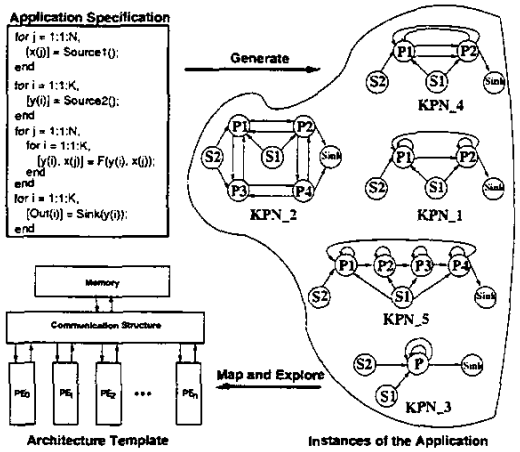

Figure 1: Alternative instances of the application have to be generated, mapped onto the architecture template and explored in order to evaluate the performance of the *Application-Architecture* pair.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES'02, May 6-8, 2002, Estes Park, Colorado, USA.
Copyright 2002 ACM 1-58113-542-4/02/0005...$5.00.

In general, a system designer is only able to derive at most a few alternative application instances. This is so because no systematic way to derive an application instance, let alone alternatives, from an application specification is known, as a result of which heuristic and time consuming approaches are taken in practice. Nevertheless, many instances of a single application exist that are worth to be derived for exploration. We present in this paper algorithmic transformations that we have developed and implemented in order to help a system designer to derive systematically and fast alternative application instances. These transformations together with an aggressive parallel compiler called COMPAAN are encapsulated in an *Application Transformation Layer* that automatically generates a set of application instances. The transformations and the tools presented in this paper are not generally applicable in the sense that the application specification has to be an affine nested loop program (NLP).

In the next section we show the position of the *Application Transformation Layer* in the Y-chart paradigm. In Section 3 two specific algorithmic transformations are given[1]. The COMPAAN tool is briefly described in Section 4. In Section 5 we show how our algorithmic transformations are used in practice. In section 6 we present a number of experiments and associated results. Finally, we discuss related work and draw conclusions in Section 7 and Section 8, respectively.

2. 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]. The position-

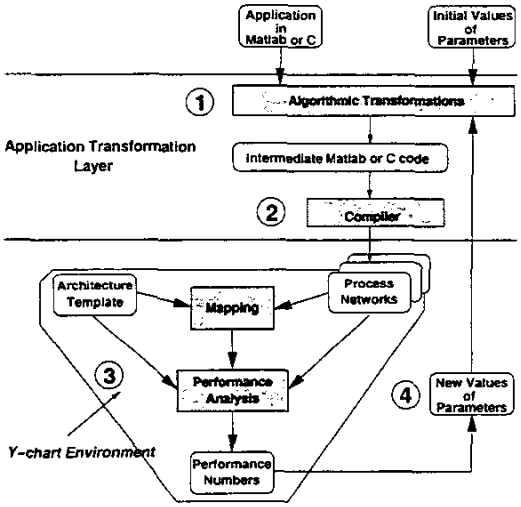

Figure 2: The Y-chart extended with the *Application Transformation Layer*.

[1]For lack of space we confine ourselves to only two such transformations. We have identified and implemented other transformations as well, e.g., plane-cutting, look-ahead, loop transformations. The approach and technique is uniform over all transformations.

ing of the transformation layer is shown in Figure 2. We start with an application specification written in an imperative language like Matlab or C and we have to generate and explore a set of instances (Kahn Process Networks) functionally equivalent to the application. First, algorithmic transformations are applied to the application specification. The transformations are controlled by a set of parameters. At the beginning some initial values are assigned to the parameters depending on the available resources in the architecture template. With these values, the original code of the application is automatically transformed and structured in a particular way in order to make the parallelism that is inherently available in the application explicit or to enhance the *task-level* parallelism in the application. Second, the transformed code is converted automatically to a KPN description by an aggressive parallel compiler called COMPAAN. Third, we use a Y-chart environment to map the KPN onto an architecture template and do performance analysis. The result of this performance analysis can be used to change the values of the parameters (step 4 in Figure 2) if the system performance is not satisfactory. Then, we repeat the procedure described above resulting in a *design space exploration* of alternative instances of the application. This is shown in Figure 2 as a feed-back arrow to the transformation layer.

By changing 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 that 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.

3. ALGORITHMIC TRANSFORMATIONS

In this section, we present two algorithmic transformations, namely *Unfolding* and *Skewing*. These transformations take as input an affine nested loop program (NLP) [2] and a set of parameters. The output of the unfolding transformation is an affine nested loop program which is functionally equivalent to the input program but with enhanced task-level parallelism. The skewing transformation makes the potential parallelism in the input affine nested loop program explicit. We have developed and implemented these and other transformations in a tool box called MATTRANSFORM. The transformations in this tool box operate directly on the NLP source code without using some intermediate representation like dependence graphs, signal-flow graphs or data-flow graphs corresponding to the NLP.

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 (NLPs) are expressed in Matlab code. The NLPs could also be expressed in other imperative programming languages like, for example, C.

3.1 Unfolding and Skewing

Consider the application program (NLP) and its dependence graph (DG) shown in Figure 3-a). The DG is a graphical representation of the NLP. The nodes in the DG represent the NLP functions that are executed in each loop iteration and the edges represent the data dependencies between the functions. The NLP has two loops (with iterators j, i) which can be unrolled to yield the DG. Unlike common approaches, in which either the loop control is removed through loop unrolling [10] or the DG is folded [11], our new approach to get the desired degree of parallelism - at the task level - is to copy

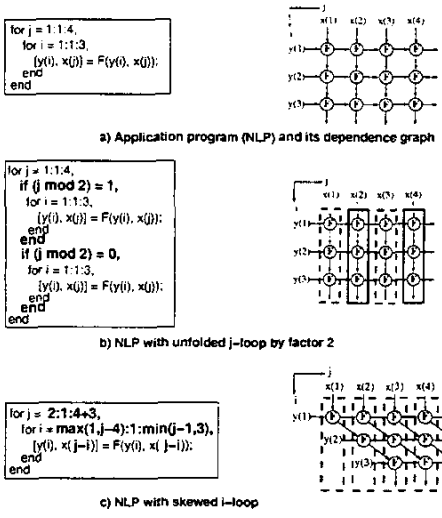

Figure 3: Simple example illustrating the unfolding and skewing transformations.

a loop body a number of times in such a way that these copies are *mutually exclusive*. We call this new approach *unfolding* and we have implemented it in our unfolding transformation. An example of our unfolding is shown in Figure 3-b), where the j-loop of the program in Figure 3-a) is unfolded by a factor of 2. The two pieces of code bounded by the "if" statements in Figure 3-b) are mutually exclusive. The mutually exclusiveness can be exploited by an aggressive parallel compiler to partition the program in Figure 3-b) into two processes (tasks) that can operate in parallel. The graphical interpretation of the unfolding transformation is given by the dependence graph in Figure 3-b). For this simple example the unfolding transformation *partitions* the computational workload over two parallel processes. 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 shown in Figure 7 - see KPN_1. In general, our unfolding transformation is used to partition an NLP in N processes, where N is equal to the unfolding factor. The process network corresponding to a fully unfolded NLP is equal to the dependence graph of this NLP.

Now, consider the same application program (NLP) shown in Figure 3-a). The transformation of skewing is to create 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 potential parallelism in the original NLP explicit. For example, skewing the i-loop of the program in Figure 3-a) leads to the NLP in Figure 3-c). The effect of our skewing transformation is visualized by the dependence graph (DG) in Figure 3-c). This DG explicitly shows that the nodes inside a dashed box can be executed in parallel because there are no data dependences between 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. An example of a network of such parallel processes corresponding to the NLP in Figure 3-c) is given in Figure 8 - see KPN_4. Moreover, inside these processes some pieces of code can be executed in parallel or in a pipeline fashion because of the skewing transformation.

```
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 }
```

Figure 4: Pseudo code describing the UNFOLD transformation.

Note that in both cases (unfolding and skewing), the transformations proceed along the NLP code in Figure 3. The dependence graphs are only shown to visualize the effect of the transformations.

3.2 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. 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 and 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 "$for\ i = 1 : 1 : N_i$" and *body* takes the code in the body of the loop with iterator 'i'. This code is marked in Figure 5 as a rectangle. 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

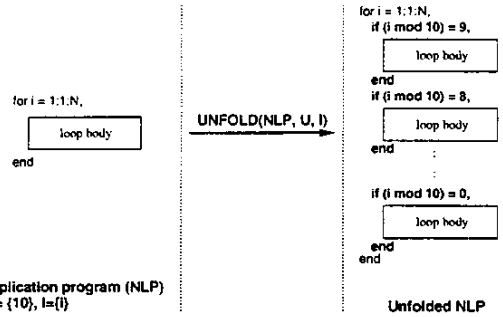

Figure 5: Simple example illustrating the UNFOLD() transformation shown in Figure 4.

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.

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.

3.3 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)* as 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 integral matrix and b is an integral vector;
- **STEP2** - Use the *skewing matrix* M to transform P as follows:
  $A.M^{-1}.M.I \geq b \Longrightarrow A'.I' \geq b$.
  where $A' = A.M^{-1}$ and $I' = M.I$;
- **STEP3** - Use the Fourier-Motzkin (FM) procedure [1] 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 $A.I \geq b$. Next, we use the skewing matrix M to

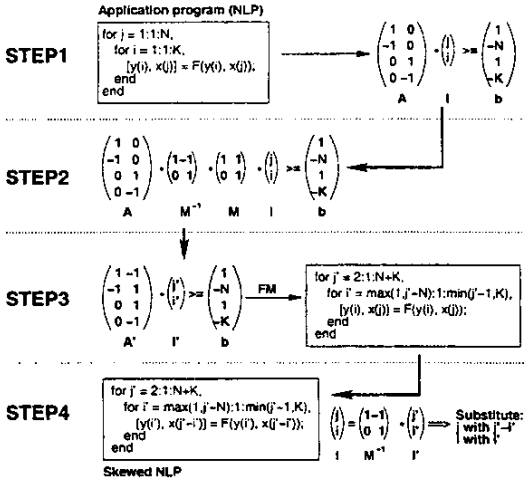

Figure 6: Simple example illustrating the four steps in the SKEW(NLP,M) procedure.

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.

4. COMPILER

In this section, we briefly describe our aggressive parallel compiler COMPAAN which exploits the result of the transformations presented in Section 3. COMPAAN (Compilation of Matlab to Process Networks) [7] is a method and tool set (MATPARSER, DGPARSER, PANDA) for transforming affine nested loop programs (NLP) [2] written in Matlab into a Kahn Process Network (KPN) specification.

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 [5]. 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*. Also, MATPARSER can handle non-linear operators like Max, Min, Ceil, Floor, Mod and Div. Therefore, it can handle the result of the skewing and unfolding transformations presented in Section 3. Next, a tool called DGPARSER [2] converts the SAC description into a *Polyhedral Reduced Dependence Graph* (PRDG) [7] description. The PRDG is a compact graphical representation of the SAC using parameterized polyhedral embeddings of the atomic functions. Finally, the PANDA tool [7] uses the PRDG description in order to generate the Kahn Process Network description and the individual

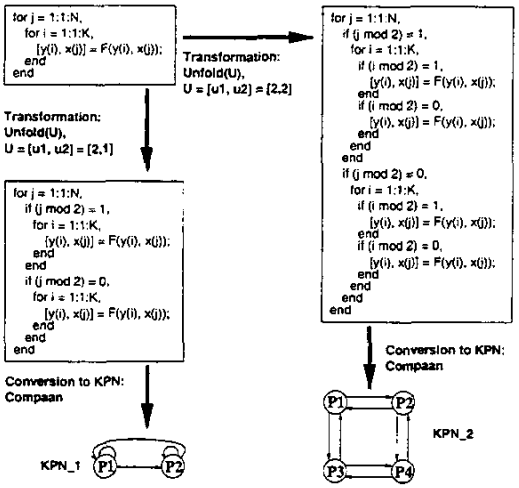

Figure 7: An example of generating two possible Kahn Process Networks from a single application using the *unfolding* transformation and the COMPAAN tool.

processes.

5. EXAMPLES

In this section, we demonstrate the use of our algorithmic transformations in combination with the COMPAAN tool set. 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 7. It is a 2-deep affine nested loop program written in Matlab. In Figure 7 first we apply the unfolding transformation on our application and then we use COMPAAN to convert the transformed code into a KPN description. 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 7- KPN_1 and KPN_2).

In Figure 8, we show another example in which we use the same application as in Figure 7. 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 in combination. KPN_4 in Figure 8 is derived by applying first the skewing transformation with $M = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ and then the unfolding transformation with $U = [2, 1]$.

6. EXPERIMENTS AND RESULTS

In this section, we present some of the experiments we have done in order to evaluate and show the usefulness of the algorithmic transformation techniques presented in this paper. We built a Y-chart environment extended with the *Application Transformation Layer* as shown in Figure 2. As an input application for the transformation layer we used the QR-decomposition algorithm [12] because it

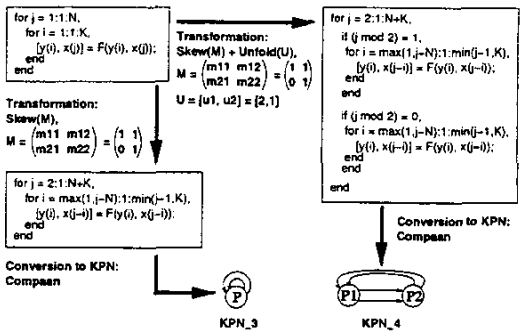

Figure 8: An example of generating two possible Kahn Process Networks from a single application using the *skewing* and *unfolding* transformations and the COMPAAN tool.

is common computational intensive task in many signal processing applications like Digital Beamforming, Adaptive Digital Filtering etc. The algorithm was written in Matlab. The application transformation layer applies the *Unfolding* and *Skewing* transformations on the QR algorithm and generates alternative application instances - Process Networks - as synthesizable VHDL. We mapped these instances onto a Xilinx XCV1000E FPGA device which was the architecture template for our experiments. The mapping was done by a synthesizer and place-and-route tools provided by Xilinx. The performance analysis was done using the timing analysis and simulation tools from the Xilinx Foundation® package.

Figure 9 shows the estimated total execution time for three application instances of the QR-decomposition algorithm. These instances were derived automatically by applying the transformation techniques presented in Section 3. The results show that the effect of

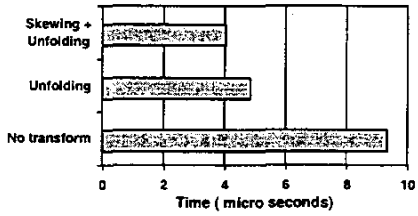

Figure 9: Execution time of the QR algorithm transformed by using the *unfolding* and *skewing* transformations. The unfolding factor is 3 and the size of the input data matrix is 10 by 6.

applying our transformations is that we can generate alternative application instances with different performance when mapping them onto an architecture template (in our case an FPGA). It can be seen from Figure 9 that the unfolding and skewing transformations improve significantly the performance.

Figure 10 shows the results obtained from the exploration of the performance of ten application instances of the QR algorithm derived by applying only the unfolding transformation with unfolding factors from 1 to 10. Again, the results show that the performance can be significantly improved. In this experiment we also measured how much time it takes to obtain the results presented in Figure 10. The time taken for these ten experiments to be processed

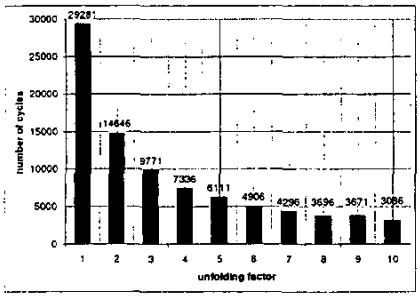

Figure 10: Exploration of the performance of the QR algorithm unfolded by factors from 1 to 10. The size of the input data matrix is 48 by 16.

automatically from Matlab to a hardware mapping onto an FPGA and VHDL simulation was within 8 hours. Table 1 shows the processing times for some of the experiments in more details. The second row *"Transform+Compile"* shows the processing times for our tools MATTRANSFORM and COMPAAN– step 1 and step 2 in Figure 2. The row *"Mapping+Simulation"* gives the time needed to express the Process Networks in terms of a synthesizable VHDL code, to map this VHDL code on an FPGA and finally to obtain performance numbers from VHDL simulation – step 3 in Figure 2.

Table 1: Processing Times (hh:mm:ss).

	Unfold 2	*Unfold 5*	*Unfold 10*
Transform+Compile	00:00:08	00:00:18	00:00:29
Mapping+Simulation	00:22:54	01:24:44	04:47:30
Total	00:23:02	01:25:02	04:47:59

The last row of Table 1 suggests that an extensive design space exploration of alternative application instances can be done in a relatively short amount of time. Moreover, the accuracy of the results obtained during the exploration is within 5%, because we did very detailed VHDL cycle accurate simulation. The results given in the second row of Table 1 show that the application transformation layer presented in Section 2 generates very fast alternative application instances from a given application. The time to do this is only a few seconds, whereas the time to map the instances onto an FPGA and simulate them varies form minutes to hours - see row 3 of Table 1. However, there is a potential to improve the mapping and simulation time (row 3 of Table 1) by using some system-level design space exploration tools like SPADE [9] and ORAS [6]. Preliminary results indicate that the mapping and simulation time can be reduced to a few minutes instead of several hours obtaining performance numbers with reasonable accuracy.

7. RELATED WORK

The *Unfolding* and *Skewing* transformations presented in this paper 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 [10]. However, there are some important differences. First, we use our transformations for generating a set of Kahn Process Networks corresponding to an application (nested loop program) thereby generating alternative application instances. Using the *Unfolding* transformation to generate Process Networks we do *reverse* partitioning compared to [13]. We start by putting all computational workload in one process and by unfolding we partition the workload over more processes. Second, we developed procedures to do these 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. Third, our transformations aim at exposing and exploiting the task-level parallelism available in an application, whereas the transformations in [10] aim at exploiting the fine-grain instruction-level parallelism.

8. CONCLUSIONS

In this paper, we presented algorithmic transformation techniques for deriving a set of application instances (Kahn Process Networks) corresponding to an application. These techniques support a system designer in exploring alternative instances of an application mapped onto an architecture template. We have implemented our techniques in the tools MATTRANSFORM and COMPAAN which means that the process of deriving alternative instances is fully automated for applications described as affine nested loop programs. Therefore, the presented techniques help a system designer to speedup significantly the process of exploring alternative application instances in system level design. Our experiments and results show that an extensive design space exploration of alternative application instances can be done in a relatively short amount of time with accuracy of the results within 5%.

9. REFERENCES

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. ACM SIGPLAN'91*, pages 39–50, June 1991.

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

[3] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

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

[5] B. Kienhuis. MatParser: An array dataflow analysis compiler. Technical report, University of California at Berkeley, 2000. UCB/ERL M00/9.

[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] B. Kienhuis, E. Rijpkema, and E. F. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proc. 8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, CA, USA, May 3-5 2000.

[8] P. Lieverse, T. Stefanov, P. van der Wolf, and E. Deprettere. System Level Design with SPADE: an M-JPEG Case Study. In *Proc. Int. Conference on Computer Aided Design (ICCAD'01)*, pages 31–38, San Jose CA, USA, Nov. 4-8 2001.

[9] P. Lieverse, P. van der Wolf, K. Vissers, and E. Deprettere. A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems. *Int. Journal of VLSI Signal Processing for Signal, Image and Video Technology*, 29(3):197–207, 2001.

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

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

[12] J. Proakis, C. Rader, F. Ling, C. Nikias, M. Moonen, and I. Proudler. *Algorithms for Statistical Signal Processing*. Prentice Hall, Inc., 2002.

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