A Process Splitting Transformation For Kahn Process Networks

Sjoerd Meijer, Bart Kienhuis
Leiden Institute of Advanced Computer Science (LIACS)
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

{smeijer,kienhuis}@liacs.nl

Alex Turjan, Erwin de Kock
NXP Semiconductors
High Tech Campus 46
5656 AE Eindhoven
The Netherlands

{alex.turjan,erwin.de.kock}@nxp.com

Abstract

In this paper we present a process splitting transformation for Kahn process networks. Running applications written in this parallel program specification on a multiprocessor architecture does not guarantee that the runtime requirements are met. Therefore, it may be necessary to further analyze and optimize Kahn process networks. In this paper, we will present a four-step transformation that results in a functionally equivalent process network, but with a changed and optimized network structure. The class of networks that can be handled is not restricted to static networks. The novelty of this approach is that it can also handle processes with dynamic program statements. We will illustrate the transformation prototyped in GCC for a JPEG decoder, showing a 21% performance improvements.

1. INTRODUCTION

Multi-core or multi-processor architectures are being introduced more and more to meet the dramatic increase in compute power. Examples are the IBM Cell processor [7] and the Wasabi architecture [11] currently being developed within Philips Research. The availability of these architectures is the first step in meeting the performance demands. The next step and challenge is to fully take advantage of these architectures; applications that were running in a single thread before, must be carefully partitioned and mapped onto the architecture. Kahn Process Networks (KPN) are very suitable for systematic mapping onto multiprocessor architectures [10]. A Kahn Process Network is a model of computation [5] that allows multiple parallel processes to communicate over unbounded first-in-first-out queues or FIFOs without following a global schedule. In [2], De Kock has shown that by changing the network structure of a KPN, the total execution time of an application can be improved. To achieve this, he uses a *process splitting transformation* that selects a process from the original Kahn process network and creates a number of copies of it such that the computational workload is distributed over these copies. Performing the splitting transformation changes the structure of a Kahn process network, while the functionality remains the same. This means that after the creation of the split-up processes, the network communication must be adequately extended and adapted. If we apply the transformation, then the network is changed as depicted in Figure 1, to which we will refer as the *producer-transformer-consumer* example.

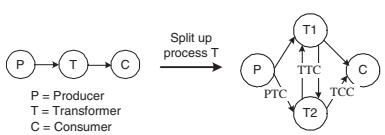

Figure 1: Splitting Process T

In the communication between a producer and consumer pair, a Data Dependence function (DD-function) can be defined. It retrieves for an iteration point at the consumer side, the iteration point of the producer side where a token has been produced. For static affine nested loop programs (SANLP), the Compaan compiler [6] can determine the DD-function as depicted in Figure 2. For each

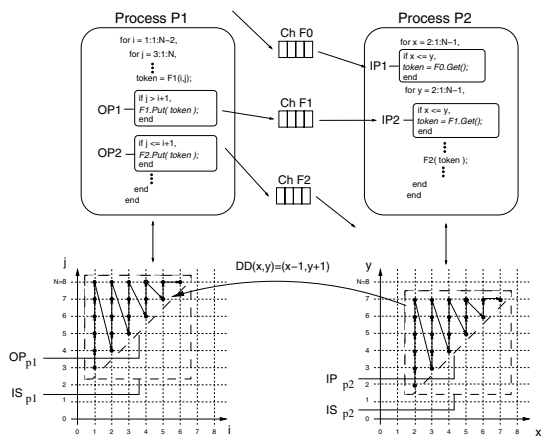

Figure 2: Producer/Consumer Communication

process P_i there is an iteration space IS_{P_i} which is made of integer points describing the repetitive loop structure of the process. An iteration space has a number of Input Ports (IP), where an input port IP_{P_i} represents a subset of the process iteration space: $IP_{P_i} \subset IS_{P_i}$. Similarly, an Output Port (OP) is defined as $OP_{P_i} \subset IS_{P_i}$ Within this context, the DD-function represents the dependency between the iterations from the input ports and output ports and is defined as $DD_{F_k} : IP_{p2} \rightarrow OP_{p1}$, where F_k is a FIFO channel and $p1, p2$ processes. Since communication channel F_k is a FIFO channel, it turns out that the DD_{F_k} is bijective, and we can also define the inverse data dependency function: $DD^{-1}_{F_k} : OP_{p1} \rightarrow IP_{p2}$. The data dependence function together with a partitioning function will guide the producer to send the tokens to the correct consumer after splitting. Splitting process $P2$ of Figure 2 means that we assign the iteration points of the iteration space over the two new processes. To illustrate this, two common cases have been depicted in Figure 3, where in A) the inner loop iterator is used for splitting, and in B) the outer loop iterator is used. The iteration points of the gray boxes are assigned to one process and the non-gray points to a second process. Once the iteration space has been split up, it is important to know to which of the new processes an iteration point belongs. Therefore, we define the partitioning function $p : IS \rightarrow Z$. In combination with the splitting factor s, it associates to each iteration point of the original process iteration space IS a number.

978-3-9810801-2-4/DATE07 © 2007 EDAA

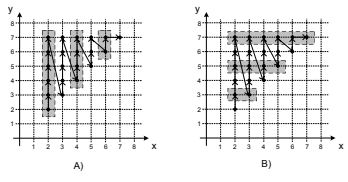

Figure 3: Splitting the process iteration space

This number represents a split-up process to which a particular iteration point belongs after the splitting transformation. Based on the partitioning function the splitting is modeled as follows:

$$\text{If } p(x)\%s = j, \text{ then after splitting } x \in P^j, \tag{1}$$

where $0 \leq j \leq n-1$ represent the process number and $\%$ denotes the modulo operator. Having defined both the DD-function and partitioning function, we can now formally define how the producer calculates the process number to which a token should be send to. If we split-up a consumer process C of a P/C pair, we get two new processes C_1 and C_2 and distinguish the following two cases:

$$\text{if } p\left(DD^{-1}(x)\right) \% \; s = 0 \text{ send data to process } C_1, \tag{2}$$

$$\text{else if } p\left(DD^{-1}(x)\right) \% \; s = 1 \text{ send to process } C_2. \tag{3}$$

This models the splitting for static affine nested loop programs (SANLP). The question is whether this this approach can also be applied in a given KPN with possible dynamic statements. Figure 4

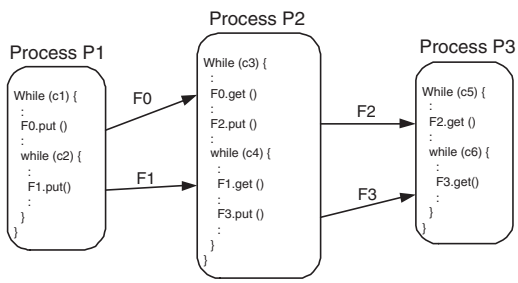

Figure 4: Simplified KPN example

shows a representative example of a dynamic streaming application as the processes use a `while` loop and conditions not necessarily known at compile time. If we split-up process $P2$ following the strategy discussed before, we obtain a network as depicted in Figure 5. If the DD-function can be determined at compile time, the

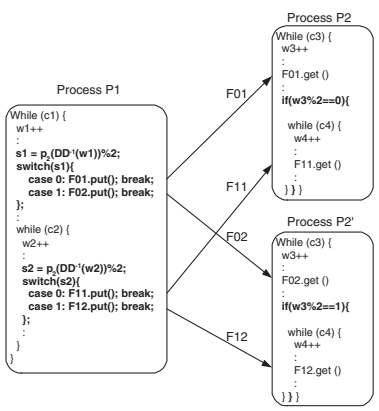

Figure 5: Split up Consumer of a P/C pair

splitting can be modeled as depicted in 5, but we will see that this cannot always be done.

2. PROBLEM DEFINITION

In the example of Figure 5, the DD-function cannot be determined at compile time. The network of Figure 4 is implemented as nested loops with conditions $c1$, $c2$, etc., which are not known at compile time. An example of a dynamic condition is `a(i) > N`, where the value of array `a(i)` is read from an incoming FIFO channel. If dynamic conditions are involved, the DD-function cannot be determined and the splitting transformation cannot be applied. However, we want a solution for applying the splitting transformation without having to determine the DD-function. As a result of the process splitting transformation, the DD^{-1} is needed in process $P1$ of Figure 5. In case of static code we can determine DD^{-1}, but the question is what about dynamic code?

3. RELATED WORK

The transformation presented in this paper aims for a performance improvement by adjusting the network structure of a KPN. Kahn process networks [5] is a model of communication used to specify the task level parallelization within applications. The process splitting transformation will also heavily rely on data flow analysis, since it must guarantee the correctness of the transformation. The process splitting transformation is inspired by the work of de Kock in [2], where task unrolling is mentioned as a possible KPN optimization. In [10] and [14] it is shown that source-code transformations in the process network model can lead to great performance improvements. The process splitting transformation, which distributes the computation of a single process over multiple processes, is closely related to the transformation of a `do` loop into a `dopar` parallel loop as described in [13]. Rijpkema et al [9] and Turjan et al [12] showed that a KPN can be automatically derived from static affine nested-loop programs by the Compaan compiler. In the synthesization of process networks, data flow analysis for scalars and arrays as described by Feautrier [4] [**?**] is crucial.

4. SOLUTION

In this section we present the solution approach for the process splitting transformation that takes as input a dynamic application specified as a Kahn process network. With respect to the communication as depicted in Figure 1, we distinguish three problem areas which must be taken into account to generate a valid network. These three problem areas are discussed below:

- Producer-Transformer Communication (PTC). The producer (denoted by P in Figure 1) had only one output FIFO before the transformation. After the transformation there are two and control must be implemented internally to the producer to send the token to the appropriate FIFO channel.
- Transformer-Transformer Communication (TTC). The transformer process must communicate data between the split-up processes ($T1$ and $T2$) if there are any loop-carried data dependences.
- Transformer-Consumer Communication (TCC). Similar to the control of token production in PTC, consumer process C must read from the correct input FIFO; initially there was only one, where there are two after the transformation.

Furthermore, we assume that there is a blocking read mechanism in a KPN, and that for each FIFO channel the number of written tokens is equal to the number of tokens read from the FIFO channel.

In other words, all channels are point-to-point, and every token that goes in, must come out. We observe (see Figure 4) that for each `put` statement to a FIFO channel at the producer side, there is a corresponding `get` statement at the consumer side at exactly the same loop-nest level. Based on the property that the loop iterators at the producer and consumer side are equal for a given FIFO channel and corresponding `put` and `get` statement, we conclude that the DD-function is not needed for the process splitting transformation. For processes $P1$, $P2$ and FIFO $F0$ of Figure 4 and 5, we perform the tests of Algorithm 1 to check whether the process splitting transformation can be applied.

Algorithm 1 Substitution of DD-function

Require: Process $P1$, $P2$, and FIFO channel $F0$
Ensure:
  if $loopnest(P1.F0) = loopnest(P2.F0)$ **then**
    if $notGuarded(F0.put)$ && $notGuarded(F0.get)$ **then**
      $insertCounters\ (P1)$;
      $insertCounters\ (P2)$;
      $insertSwitchStat\ (P1.F0)$;
    end if
  end if

In function `insertSwitchStat`, formulas 2, 3 and the usage of the DD-function are not not needed anymore, but more detailed information about this procedure is discussed in section 4.3. To summarize, the transformation can be applied under the following conditions: 1) a `put` and corresponding `get` primitive must take place at the same loopnest level and cannot be guarded by an if-statement, and 2) all tokens produced must be consumed. Taking the problem areas defined in section 2 into account, we introduce a four-step process splitting approach: 1) partitioning of the computation of the split-up process over the newly created processes, followed by the adjustments of the communication: 2) PTC, 3) TTC and 4) TCC. But before applying the transformation, the following parameters must be determined:

1. Determine the most computational expensive processes (in number of cycles). This can be done by profiling the application, or by annotating the source-code with pragmas which can trigger the compiler to do the transformation.
2. Based on the information gathered in the first step, determine how many times a process has to be split up, which we call the splitting factor or s in short.
3. Partitioning of the iteration space. Depending on the splitting factor, the iteration space has to be partitioned over a number of subprocesses.
4. Loop-nest level at which the splitting takes place. In case of nested for-loops, the question is whether to split at the inner or outer loop-nest level.

From the algorithm parameters mentioned above, different choices lead to different performance of a network. For example, choosing a particular partitioning function or loop-nest level could make a difference. Once the algorithm parameters have been determined, the actual problems of producer-transformer and transformer-consumer communication need to be addressed. For both token production and consumption we can define a static and dynamic solution. Solutions for the *Producer-Transformer* communication are, 1) the producer filters the tokens (static solution), or 2) the producer sends all tokens to all subprocesses (dynamic solution). Solutions for the *Transformer-Consumer* communication are, 1) the consumer knows by it self when to switch (static solution), or 2) each producer sends a signal to the consumer when to switch reading data from a different FIFO (dynamic solution).

Having defined the problems and the corresponding possible solutions, we will explain each of the four steps of the process splitting transformation in detail in the following sections, the first one being the partitioning of the iteration space, and then the three areas where the token communication must be adapted.

4.1 Partitioning

In the examples discussed so far in the introduction, the partitioning was based on loop counters and a modulo condition. While this is one possibility to split up the iteration space, there are other computationally less expensive solutions. For example, in case of while-loops a simple finite state machine can be used and for for-loops the starting value and stride can be adjusted.

4.2 Transformer-Transformer Communication

Closely related to partitioning, is the transformer-transformer communication (TTC) step of the transformation process. TTC occurs when data must be communicated due to data dependences between statements assigned to different processes. This is closely related to partitioning since different partitioning functions can assign data dependent statements to the same process or not. A case where TTC must be implemented to guarantee correct behavior of the network, where, for example, the even iterations are assigned to one process and the odd to another, is the following: `for(int i=1; i<10; i++) { a[i] += a[i-1]; }` Note that the assignment statement results in a loop carried data dependency: iteration i consumes data produced at a previous iteration $i-1$. In our approach we can detect whether there are any loop-carried dependences, but do not split if this is the case (see also section 6). We will leave process splitting that results in transformer-transformer communication (TTC) for future research.

4.3 Producer-Transformer Communication

In this section we will define a static and dynamic method for the producer-transformer communication. If we consider the producer-consumer pair (P/C pair) as depicted in Figure 6, we see that the producer has two options with regards to the number of tokens sent to the consumer. It either sends the tokens to all split up consumers or it selects the correct consumer, which we will call the dynamic and static solution respectively.

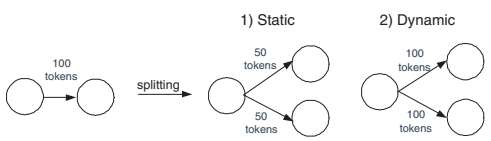

Figure 6: Static vs. Dynamic Solution

Of both the dynamic and static solution, the dynamic solution is the most general in the sense that the producer simply sends all tokens to all split-up processes. This is attractive to do, because in this way we only need to have control at the consumer side. Disadvantage of this approach is the substantial increase in communication. The static solution however, is an improvement of the dynamic solution because the tokens are filtered at the producer side. In this way

we do not have the communication overhead as we have in the dynamic solution. Therefore, we will discuss only the static approach and apply it to the network already given in Figure 4. If we split-up process $P2$ of this network, we obtain the KPN as depicted in Figure 7. Also note that counters $w1, .., w6$ have been introduced which will be used for splitting up the iteration space. We copied

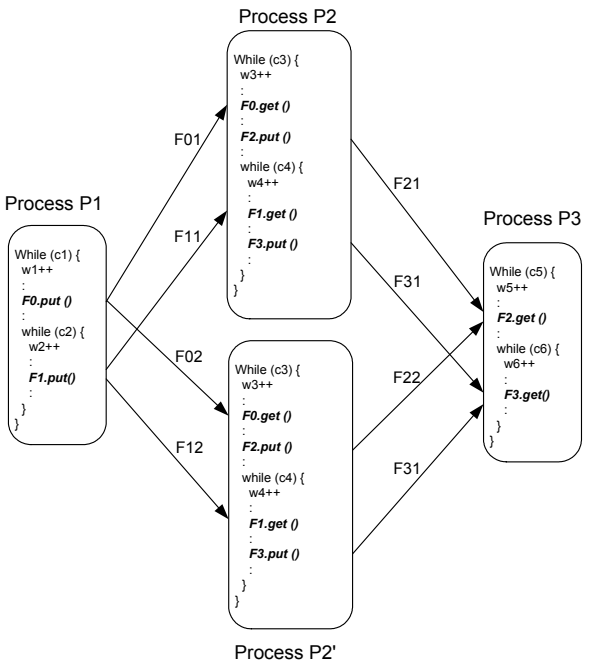

Figure 7: Incorrect KPN after splitting process P2

process $P2$ to a new process $P2'$ and added extra FIFOs. However, this KPN is functionally incorrect and changes have to be made in the three problem areas to come to a functionally correct KPN. This has been indicated by the statements in bold and in a step-by-step approach explained in this section, the changes made to this KPN will be explained in order to come to a functionally correct KPN. We see that in Figure 7, FIFO channel $F0$ becomes $F01$ and $F02$, and FIFO channel $F1$ becomes $F11$ and $F12$. Therefore, statements $F0.put()$ and $F1.put()$ of the original process are not valid any more and control must be implemented to send a token to either one of the new FIFO channels. Using the partitioning and data dependence function, the token production would be modeled as $p(DD^{-1}(w1, w2))$. The producer calculates at which iteration point the token is going to be consumed and to which one of the processes $P2$ or $P2'$ a token must be send to. But if we cannot determine the DD-function, we use the observation that the loop counters are equal at the same loop-nest level a token is communicated. In this way, a simple mapping between the iteration points of the consumer and producer is established. This means that in the example of Figure 7, that $p(DD^{-1}(w1, w2)) = w4$, and since $w2 = w4$ we can use $w2\%2 == 0$ and $w2\%2 == 1$ at the producer side to send the tokens to processes $P2$ or $P2'$ respectively. This is depicted in Figure 8.

4.4 Transformer-Consumer Communication

To complete the transformation of the *producer-transformer-consumer* example, the communication between the processes transformer and consumer must be restored. In the original network, the consumer reads from one input FIFO. This changes by applying the splitting transformation. Now the consumer needs to have control

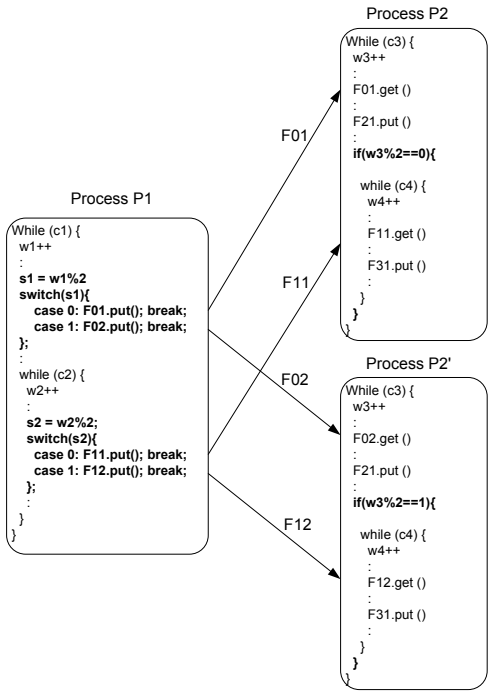

Figure 8: Static approach for the producer-transformer pair

to read from correct input FIFO. This control is obtained similar to the solution approach of the producer-transformer. Instead of using the DD^{-1}, we use the DD and partitioning functions to determine where an input token is produced. If x is an iteration point from process C that consumes data produced by process T then:

$$\text{if } P_l(DD(x)) \,\%\, s == 0 \text{ get data from process } T1, \quad (4)$$

$$\text{else if } P_l(DD(x)) \,\%\, s == 1 \text{ get data from } T2. \quad (5)$$

But since the DD^{-1} cannot be determined, we follow the same solution approach as presented in the solution for producer-transformer communication, and read the tokens based on the loop counters and the modulo condition. We omit the figure for this P/C pair, since it's almost identical to Figure 8, with the difference that the control is implemented at the consumer side.

5. MULTIPLE SPLITTINGS

So far, we discussed the splitting of one process only. This can be extended to multiple process splitting (used in the case studies). Splitting multiple, possibly neighboring, processes requires the introduction of so called *copy nodes*, because they map two incoming channels to one outgoing channel. This allow us to follow exactly the same solution approach as discussed before. Without going into details, the intuitive idea is depicted in Figure 9. These

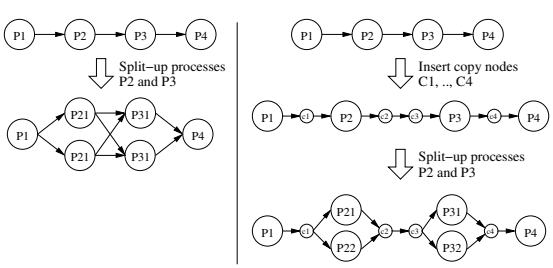

Figure 9: Two step approach for multiple splitting

copy nodes (denoted by $c1, .., c4$ in Figure 9) only read and write tokens; there is no computation involved. Constructing these copy nodes is simple to perform, because they have the same structure as the process that produces the data. Each process will be surrounded by a pair of copy nodes such that the network structure does not get too complicated and our four-step transformation can be applied.

6. IMPLEMENTATION

To prototype the new process splitting transformation in a compiler, we used GCC 4.1 and its data flow analysis to implement a data dependence graph (DDG) for the process(es) we are interested in. If we decide to split a process, we check the DDG for the legality of the transformation; we do not split if the data flow analysis indicates the existence of a loop-carried data dependency. In pseudo-code, the procedure is implemented as follows:

Algorithm 2 Process Splitting

Require: A function f

Ensure: Copy the class structure to which function f belongs to and adjust the structure of it.

```
DDG ddg ⇐ createDDG ();
LOOP loop ⇐ findSplitPragma (f);
if loop then
  bool hasLCD ⇐ hasLoopCarriedDeps (ddg, loop);
  if hasLCD = FALSE then
    function f' ⇐ copyFunction (f);
    int split_factor ⇐ 2;
    insertModuloCond (f, split_factor);
    insertModuloCond (f', split_factor);
    class c ⇐ findClassDefinition (f);
    adjustNetworkStructure (c, f, f');
  end if
end if
```

When the dataflow analysis permits the splitting of a process, we copy the original function to a new one and insert the modulo condition. But this is only the start of the transformation. We expect our applications to be specified as C++ applications, and use the YAPI [3, 8] threading library to implement the processes of a KPN as a C++ class. Therefore, in order to copy/modify a process, we need to reconstruct the C++ class in the SSA Intermediate Representation (IR) of GCC. This is not trivial to do, because all C++ classes are lowered to `structs` in the SSA IR. Basically, the whole notion of Kahn process networks, FIFOs, network structure, etc., has to be introduced into GCC. For these reasons, we have semi-automated the process splitting transformation. When GCC tells the transformation can be applied without introducing interprocess or transformer-transformer communication, we make the final source-code transformations by hand. This means that we have not implemented the function `adjustNetworkStructure` of Algorithm 2 yet.

7. JPEG CASE STUDY AND RESULTS

We illustrate the process splitting transformation in this section based on the JPEG decoder application and determine the most computationally expensive process first (see Figure 10).

We see that there is one process, the *raster* process, that determines for a great part the total execution time. It exceeds the average number of cycles needed for computation compared to the other problems. The horizontal axis displays the processes of the JPEG decoding application and the vertical axis the number of cycles a process needs to finish. We want to distribute the computation of the raster processes over two processes using the process splitting transformation.

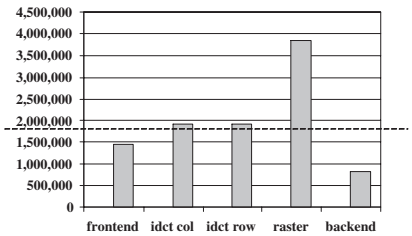

Figure 10: Profile of the JPEG decoder application

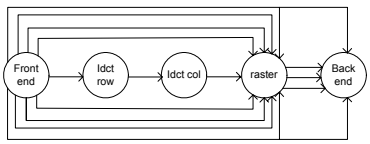

Figure 11: JPEG decoder application specified as a KPN

Figure 11 illustrates the original and unmodified Kahn process network. Based on the profile information, we decide to split the raster process such that we obtain the network as depicted in Figure 12. Note that in this example some of the FIFO channels have been left out for the sake of clarity.

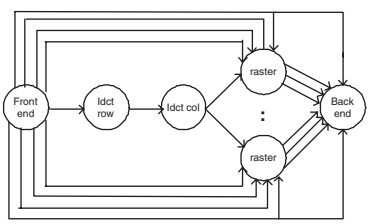

Figure 12: JPEG application with the raster process split-up

As indicated by the dotted line in Figure 12, one is free to choose the splitting factor. We have carried out a case study in which we split up the computation of one process over two new processes only. We ran the JPEG decoder on a simulator for the Wasabi multiprocessor architecture that is currently being developed within Philips Research. The simulator counts the number of cycles for the eCos Real-Time Operating System (RTOS) [1] on which we ran the JPEG application. By using the YAPI threading library, the processes are implemented as threads for which the operating system allocates available resources. Before discussing the results of the application and split-up processes, we investigate the performance of the unmodified KPN, and the KPN with copy nodes inserted. Table 1 shows the execution time of these two KPNs in million of cycles (all the other numbers represent running time in millions of cycles as well).

When we map the unmodified JPEG application on two processors, we see that it scales compared to the application running on one processor. Remarkable is the observation that the execution times go up on three and four processors. While further scaling could not be expected, an increase in cycles is a surprising and undesired

[1] http://ecos.sourceware.org/

# CPUs	Original	Copy nodes
1	18.62	22.36
2	10.48	12.67
3	**10.11**	11.61
4	18.32	16.69
5	20.77	28.04

Table 1: Execution time of original KPN and the KPN with four copy nodes inserted

result. We see that by adding more processors the parallelization gain is killed by the extra communication the processors introduce. Another observation is the substantial overhead the copy nodes introduce; for three processors, the KPN with the copy nodes need 1.5 millions cycles more to finish. Table 2 shows the result of the KPN where different processes have been split-up. The second column illustrates the results of the network where processes idctcol and idctrow have been split-up. The same applies to the remaining columns; they denote processes that have been split-up into two processes. The two values in bold in Tables 1 and 2 illustrate the

# CPUs	row+col	row+col+raster	col+raster	raster
1	24.38	24.62	23.63	20.23
2	14.00	14.33	13.46	11.37
3	12.76	11.44	10.57	8.77
4	17.31	11.07	9.41	**7.95**
5	28.09	12.68	9.76	8.76

Table 2: Execution times of KPN with different processes split-up

minimum of the original and the network where the raster process has been split-up, which are 10.11 and 7.95 respectively. It shows that splitting up the *raster* process is the most profitable transformation we could do on the JPEG application: we see an improvement of 21.36%. Another interesting observation is that only splitting up the raster process is beneficial. While this was already indicated by profiling the application (see Figure 10), it is remarkable that in all the other cases, including the one where *raster* and *idctcol* both have been split-up, the results are not that good or get worse. This is caused by the operating system used and the threading library.

8. CONCLUSION

With the introduction of multi-core or multi-processor architectures, programming and exploiting the available resources becomes more and more challenging. Therefore, we assume that our applications are specified as Kahn process networks, which is a model of computation where multiple processes can run in parallel and communicate over unbounded FIFO channels. Still additional KPN transformations are required to meet the desired performance requirements. We have presented a process splitting transformation and showed that a 21% performance improvement can be obtained by reducing the total execution time of the JPEG decoder application. We have prototyped the transformation in GCC. Given the results, this research is continued as a NEVA MEDEA+ European funded project. The transformation presented are currently being implemented in the CoSy compiler [1] developed by ACE Associated Computer Experts.

9. REFERENCES

[1] Martin Alt, Uwe Asmann, and Hans van Someren. Cosy compiler phase embedding with the cosy compiler model. In *Computational Complexity*, pages 278–293, 1994.

[2] E. A. de Kock. Multiprocessor mapping of process networks: a JPEG decoding case study. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 68–73, New York, NY, USA, 2002. ACM Press.

[3] E.A. de Kock et al. YAPI: Application modeling for signal processing systems. In *Proc. 37th Design Automation Conf. (DAC 2000)*, pages 402–405, New York, USA, 2000. ACM Press.

[4] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[5] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[6] Bart Kienhuis, Edwin Rijpkema, and Ed 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.

[7] D. Pham et al. The design and implementation of a first-generation cell processor. In *In ISSCC Digest of Technical Papers*, pages p. 184–5, 2005.

[8] http://y-api.sourceforge.net/.

[9] Edwin Rijpkema. Modeling Task Level Parallelism in Piece-wise Regular Programs, 2002. PhD thesis, Leiden University, The Netherlands.

[10] Todor Stefanov, Bart Kienhuis, and Ed 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 Press.

[11] Paul Stravers and Jan Hoogerbrugge. Homogeneous multiprocessing and the future of silicon design paradigms. In *In Prce. International Symposium on VLSI Technology, Systems, and Applications (VLSI-TSA 2001)*, April 2001.

[12] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Translating affine nested-loop programs to process networks. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 220–229, New York, NY, USA, 2004. ACM Press.

[13] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[14] Claudiu Zissulescu, Bart Kienhuis, and Ed F. Deprettere. Increasing pipelined ip core utilization in process networks using exploration. In *FPL*, pages 690–699, 2004.