Transformation of Sequential Program to KPN - An Overview

Danish Ather
Sr. Lecturer
Department Of Computer Applications
Teerhanker Mahaveer University

Raghuraj Singh
Professor,
Computer Science & Engineering Department,
H.B.T.I., Kanpur

Vinodani Katiyar
Professor
SRCET
Computer Science & Engineering Department, Lucknow

ABSTRACT
This paper describes a general transformation theory in transforming a sequential C application to Kahn Process Network. It briefly describes in detail the two major transformation steps namely task partitioning and channel generation. We also discuss the previous approaches which automate the transformation from sequential model to parallel model and compare these with our approach.

Keywords
Kahn Process Network, Matlab, Partition Analysis, Channel Placement Analysis, Optimizations, loop parallelization, loop outlining, Unparsing

1. INTRODUCTION
Transformation converts an application specified in sequential model of computation (in imperative language such as C, Matlab and so on) to parallel model of computation (Kahn Process Network). The transformation can be manual or automatic. Transformation involves identification of section of code, which can be transformed as tasks of KPN. A task defines the minimal computation unit that is mapped to a processing element. Once transformation into tasks is done, appropriate channels are introduced between the tasks. Figure 2shows a transformed KPN for the sequential program of Figure 1. Function "main" is transformed into task "Task main", function "x" is transformed into task "Task x" and function "y" is transformed into task "Task y". The global "variable" global in sequential C source is now local to all tasks and therefore channels are introduced between tasks. As global variable "global" is written in "main" and read in function "x" and "y", channels are added from "main" to "x" and "main" to "y" in final KPN. Channels are also introduced for function parameters "m" and "n" in the generated KPN. Two major steps required in transforming a sequential application to a Kahn Process Network are task partitioning and channel generation. Irrespective of the type of source language in which sequential application is written, these two steps must be performed to realize the required KPN.

Below we give an overview of these two major steps:

- Function based Partitioning
- Loop based Partitioning

```
intint_global                 void x( int a )          void y( int b )
main()                        -                        -
-                             intlocal_x               intlocal_y
for(i=1;i<=100;i++)           ……..……                  ……l
-                             ……..……                  ocal_y=sub5()*
int_global=sub();             local_x = sub4(          int_global
a=sub1();                     ) * int_global           ……
b= sub2();                    ……..                     ……
x(a);                         ……                       ……
y(b);                         ……..                     ……
}                             }                        }
}
```

Figure 1 A Sequential C program

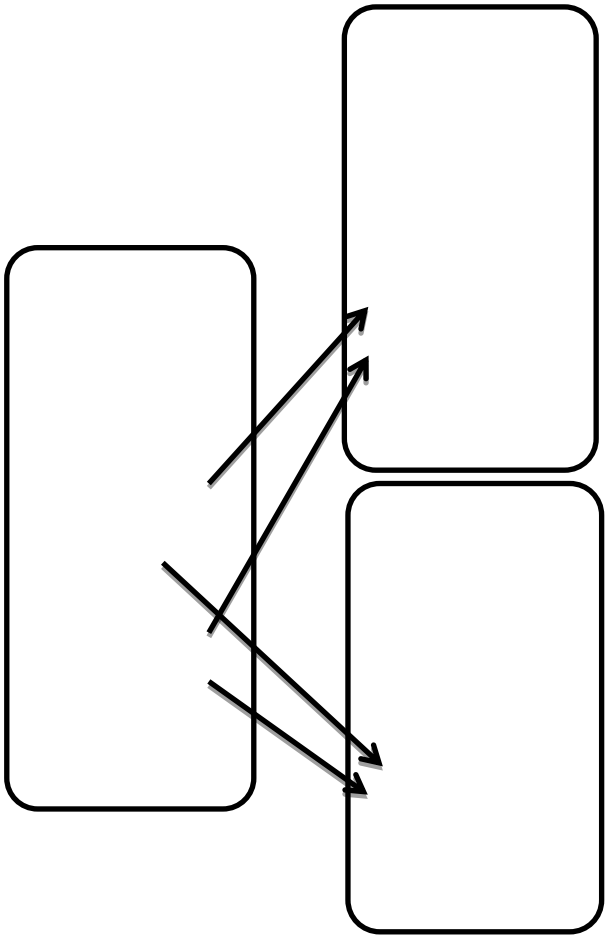

Figure 2A transformed KPN for Figure 1

2. FUNCTION BASED PARTITIONING

Partitions the source application at function boundaries into tasks. It means that each function or a set of functions in the input application can constitute a single task. Function based partitioning achieves function parallelism from the program. Figure 3 shows a layout of a program where functions partitioned into tasks are bounded in a box. Functions "main", "functionG" and "functionF" belong to Task 1. Functions "functionA" , "functionB" , "functionC" and "functionD" belong to Task 2 and function "functionE" belongs to Task 3.

3. LOOP BASED PARTITIONING

Partition loop into separate tasks by dividing the iteration space of the loop into set of iteration spaces where each iteration space is a separate task. This type of partitioning achieves data parallelism from the program. Figure 1.4 shows a loop partitioned into 4 tasks. Loop for i =1 to 100 is partitioned into 4 sub loops where each loop iteration is one fourth of original loop. Each of the 4 sub loops can transfer and receive data through communication channel.

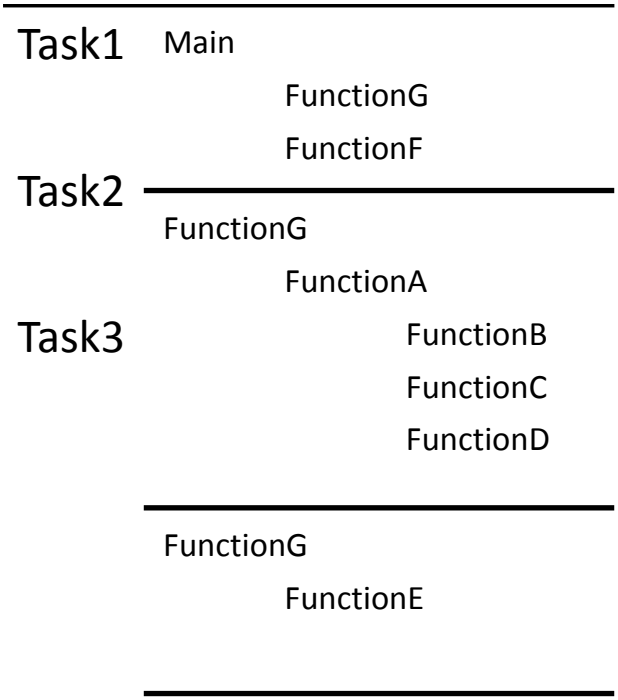

Figure 3 Function Based Task Partitioning

4. CHANNEL GENERATION

Once the input application is partitioned into tasks, appropriate communication channels are added between the tasks. A communication channel is FIFO channels that are connected to exactly two processes through read and write ports. One process acts as the producer and writes data elements to the channel and other process acts as the receiver and receives data tokens.

Variable/data needs to be identified for communication between separate tasks. Also the direction of channel needs to be determined based on flow of data from one task to another. Channels are generated as a result of existence of global shared data, function call parameters and updation of array in loop parallelization.

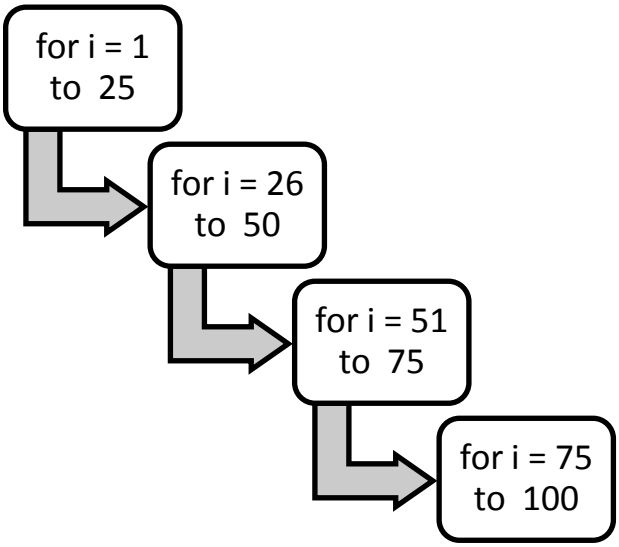

Figure 4 Loop Based Task Partitioning for i=1 to 100

5. "EFFECTIVENESS" OF TRANSFORMATION

A transformation is "effective" if it is correct and it leads to an efficient KPN. Correctness and efficiency are the two most important factors that need to be taken care of in transformation. The resulting KPN functionally match the original program and also achieve higher throughput when implemented on a multiprocessor platform.

5.1 Correctness of Transformation

Following are the cases where the resulting KPN is incorrect:

- The output generated by executing KPN does not match with the output generated by executing its sequential counterpart. An incorrect output can occur because of passing of incorrect values from one task to another at different points in program execution.
- Whenever there is a read of a data token by a task that is written by another task, a channel is added between those tasks. Addition of a channel can lead to incorrect output by KPN if channel is not added at the point of read and last write of that data token. In such a case reading task will not get new updated value of the data token.
- A task gets blocked pre-maturely (without giving the required output) as a result of reading a token from an empty channel (this is a property of KPN

that if the channel has no token then read from an empty channel blocks the task at that point where read is being done).

- A task gets blocked pre-maturely as a result of full channel (if a channel is full then write action cannot write more tokens into the channel which will block the task).
- KPN runs into an infinite execution without getting terminated. This case occurs when no task gets blocked or there is no exit condition for the KPN.

5.2 Efficiency of KPN

The resulting KPN is efficient if it is not sequential. As an example We take Figure which is a partial KPN generated after task partitioning but without channels generation. Figure also displays the layout of sequential program. Function "main" calls "functionF" and then "functionG". Function "functionG" calls "functionA" and then "functionE". Function "functionA" calls in turn calls "functionB" , "functionD" and "functionC" . Suppose there are two global shared variables "G1" and "G2". "G1" is updated in function "main" and read in function "functionB". "G2" is updated in function "functionD" and read in function "functionF". As "main" and "functionB" belong to different task a channel will be added for shared variable "G1" and as "functionD" and "functionF" are in different tasks a channel will be added for shared variable "G2" .Channels can be added in two different ways as shown in Figure and Figure 1.6 which are the two KPNs for the same case and are functionally correct. In Figure let the write of "G1" be positioned in "functionG" and read for "G1" be positioned in "functionA" before calls to "functionB", "functionC" and "functionD". Also let the write of "G2" be positioned in "functionA" after calls to "functionB" , "functionC" and "functionD" and read of "G2" be positioned in "functionG" after write statement of "G1". The KPN of Figure is inefficient KPN and is almost sequential. As Task 2 remain blocked until control reaches "functionG" in Task 1 and thereafter Task 1 remain blocked until control reaches back to "functionA" in Task 2. In Figure 5 let write of "G1" be positioned in "main" and read for "G1" be positioned in "functionB". Also let write of "G2" be positioned in function "functionD" and read be positioned in function "functionF". So in this case channel access statements are added just before and after the updation and read of global variable. This generated KPN is efficient as duration for which tasks gets blocked is considerably less.

5.3 Overall Flow in Existing Approaches

In this section we discuss the general flow in transforming sequential C programs to Kahn Process Networks. As C is widely used as a language of choice for writing programs and most of the code for streaming and signal processing programs is also specified in C, We concentrate on transforming applications written in C to KPN. Most of the previous researches like Sprint and Harmonic have chosen applications written in C as the source applications on which transformations are done. Figure 7 shows the flow chart for transforming a sequential C application to a Kahn Process Network and gives a brief description of each action in the flow process. We briefly describe each action given in the flow chart to show why it is used as a part of transformation.

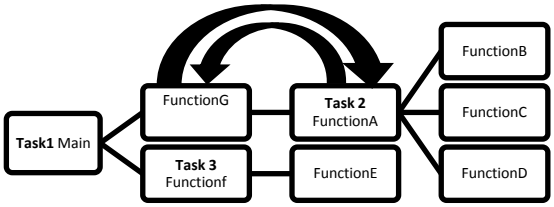

Figure 5A Complete Sequential KPN

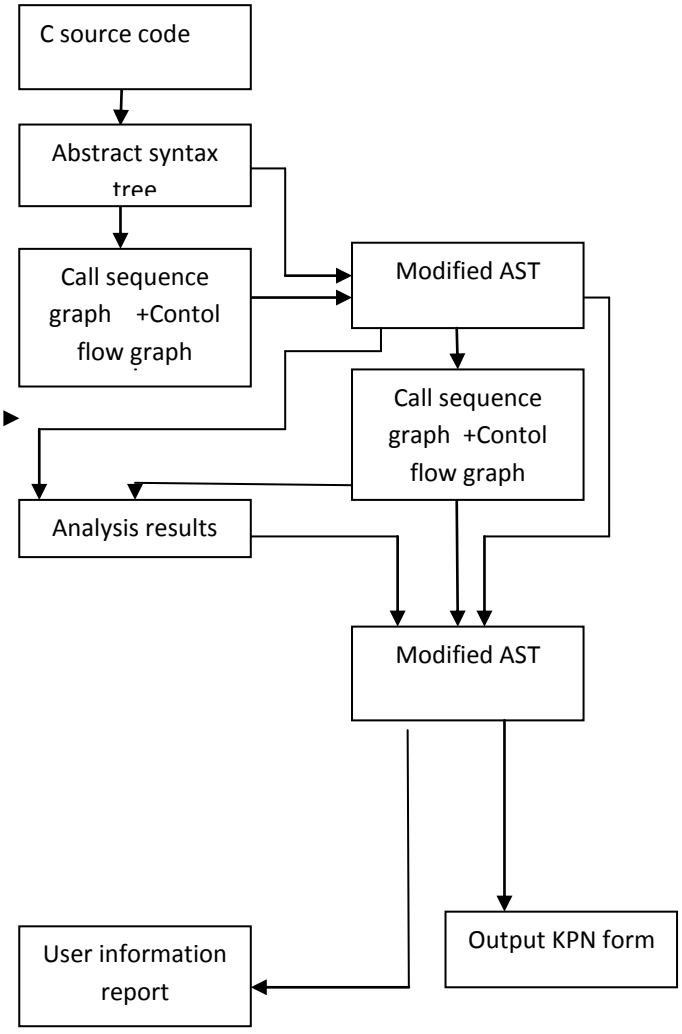

Figure 7Transformation of sequential C application to KPN

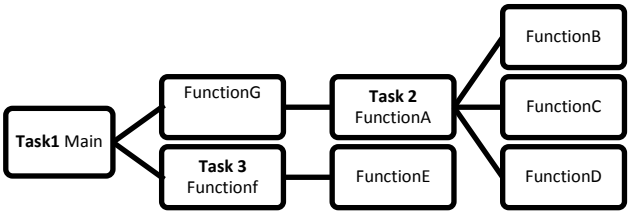

Figure 6 An efficient KPN

5.4 Parsing

Parsing action generates an abstract syntax tree (AST) data structure of the input C application's source code. AST is the tree representation of the abstract syntactic structure of the source code. Each node of the tree denotes a construct occurring in the source code. Any modification to the source code implies modifying the attributes of the specific node of the AST and then un-parsing the AST to generate the modified source code. AST generation is an important step in source code analysis as it makes source code available in the form of tree like data structure which is easier to modify based on program semantics.

5.5 Control Flow Traversal

Control flow traversal takes the AST data as input and traverses the AST in the pattern similar to the flow of control of the program when it is executed. For a C program, control flow traversal starts from "main" function and on encountering a function call, gets its associated function definition and continues traversal from that definition recursively. Control flow traversal is used to generate a call sequence graph and control flow graph of the whole program . In a call sequence graph vertices denote functions. An edge between two vertices u and v denotes a call from u to v in the program. In a control flow graph the basic blocks of the program denote vertices and edges denote control transfers.

5.6 Task Partitioning

Task partitioning divides the program into a set of tasks where each task is a process in a KPN. Each task has a set of instructions that execute sequentially on a processing element. Task Partitioning can be done at function boundaries where each function or a set of functions is made a task or partitioning can be done by transforming loop into tasks by parallelizing them. In Compaan [2] task partitioning is done by dividing loops into tasks. In Sprint [12] and Harmonic [13] task partitioning is done at function boundaries. Task partitioning can be automatic or user directed where user annotates the beginning of the function with a directive to indicate that this function needs to be made a task or part of the task. This user directive strategy for task partitioning is used in [12]. In Compaan [2] task partitioning is automatic where each loop is partitioned into more than one task. To partition loops into task Compaan uses integer linear programming approach to calculate dependence between iterations of a loop.

As task partitioning modifies the AST of the program, control flow traversal is performed on this modified AST to generate a fresh call graph and control flow graph.

5.7 Intermediate Analysis

Intermediate analysis action mainly includes two types of analysis:

• Pointer analysis

• Global shared data analysis

Such analysis is required for insertion of appropriate communication channels for concurrent tasks generated during task partitioning step. Communication channels are added between tasks if they read/update any shared global data in the program or there are variable in the parameters of the parent function of the task. As a pointer may point to a shared variable in a program, it is necessary to get accurate points to information for all pointer variables in a program. The global shared data analysis gets a list of all global shared variables in a program so that appropriate channels can be added between tasks sharing a particular global variable.

5.8 Channel Generation

The transformation tools automatically detect the need for a communication channel and insert them between tasks generated in previous step. The tasks interact with the channels through interface functions read and write. These interface functions separate computation in the tasks from communication in the channels thus interleaving computation and communication. Two classes of communications channels are used, one for communicating scalar data and other for communicating vector data. For communicating a scalar data the read and write interface have the channel information and variable name as the parameters. For communicating a vector data the read and write interface have the channel information, variable name and number of tokens to be passed as the parameters.

In case of Sprint [12] channels are generated for variables that are shared between tasks and variables that occur in the parameter of a function that separate the task boundary. In case of Compaan [2] channels are generated for variables inside a loop as a result of loop being divided into tasks. Channel generation for a C program takes program AST, call graph, control flow graph and intermediate analysis results as input information. Based on this information this steps identifies the variables and their types between which channels will be inserted. Once variable identification is done appropriate scalar or vector channel is inserted.

5.9 Generate User Information

Among all existing approaches discusses here, user information report is generated by only Sprint [12]. Such user information is reported when the transformation tool is not able to perform partitioning or channel generation because of the violation of some constraint. For example Sprint requires that there is only one entry and one exit statement for a given task. If this condition is violated a user report is generated so that user can manually modify the source code.

5.10 Unparse

The unparse step simply performs unparsing of the AST to generate the modified source code.

6. OVERALL FLOW IN SUGGESTED APPROACH

In this section we describe the flow involved in suggested approach for transformation of a sequential application to KPN model. suggested methodology is based on the flow described in Figure 7 and extends it at different stages of the flow. Figure 8 shows the transformation flow we have implemented in our approach which is the extension of the basic flow described in previous section.

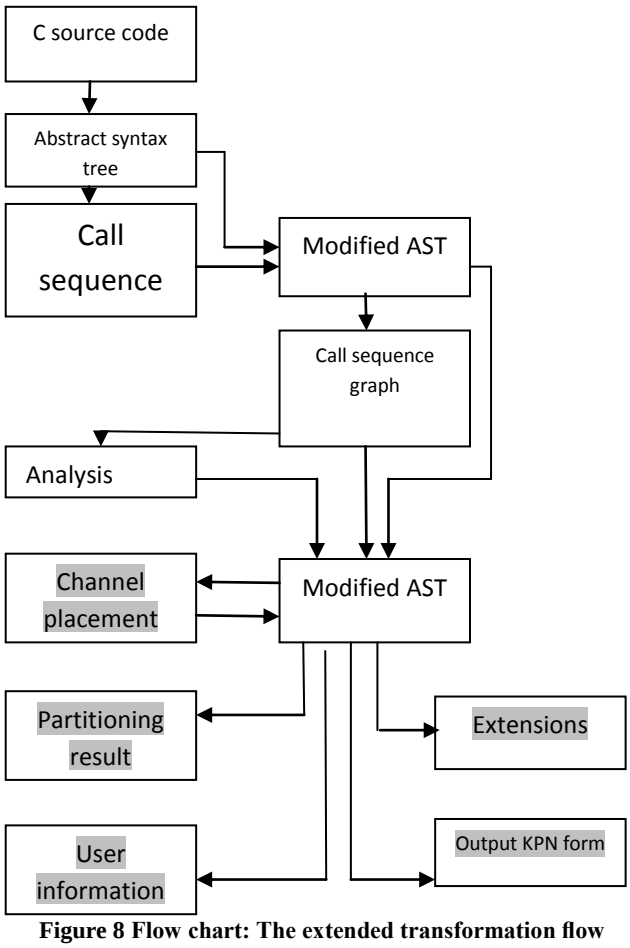

Figure 8 Flow chart: The extended transformation flow for Figure 7

The boxes shown in gray color in Figure 8 are the parts that we have extended in the general flow of Figure 7. Some boxes which are white signify that this generated data exists in both original flow and the modified flow in our approach but our approach generates some additional information that is not present in the existing ones. As shown in the Figure 7we have extended the flow by implementing the following:

- Partition Analysis
- Channel Placement Analysis
- Optimizations:Includes function inlining, loop parallelization and loop outlining
- Generate user information: Generation of graphical output of Tasks and channels of resulting KPN
- Unparsing: Generation of .c and .h file for each task of KPN

6.1 Partitioning analysis

The **partitioning analysis** gives feedback to the user on the quality of task partitioning as good partitions can result in higher throughput of the program. The partitioning analysis outputs if the partitioning is good or bad. A partitioning is good if tasks can run in parallel. A partitioning is termed as bad if it may result in sequential execution of some of the tasks of the KPN i.e. task i executes after task j has completed its execution . Such partitioning is reported to the user so that user can modify the partitioning candidates so that inter task execution is not sequential.

6.2 Channel placement analysis

Channel placement analysis modifies the placement of read and writes channel interface statements during the channel generation stage. Such analysis is necessary to make resulting KPN functionally correct as well as efficient. As our approach can take any arbitrary C program as input, some cases may occur during channel generation stage which may make resulting KPN incorrect. Channel placement analysis is required to prevent such cases where read and write are moved to a different location in the source code. The details of the cases and their solution are discussed in next paper. Apart from correctness, channel analysis can improve KPN efficient by moving the channel access statements. Such improvement can occur as a result of reduction in waiting time of a task for the data by providing the required data much before in the execution sequence.

6.3 Optimizations

The **optimizations** include function inlining, loop optimization and loop outlining. Function inlining is useful to reduce the context switch time between function calls in a task by placing the function body in place of its call in some other function. Function inlining implemented in our approach works within a task and not across tasks. Loop parallelization divides loops into tasks by dividing the iteration space of a loop into different tasks where each sub iteration space can run concurrently. Like functions, loops are the computation intensive parts of the program and need to be partitioned into tasks. Loop parallelization was implemented in our approach to provide data parallelism along with function parallelism.

6.4 Generate user information

To provide information to the user for task and channels generated from the tool we have provided a graphical output of tasks and channels in **generate user information.** This output provides sufficient information to the user regarding the parts of the program that were made tasks and also provide information regarding the variables for which channels were added and also the location of read and write channel interface statements. Such graphical information can also help user to identify the locations where partitioning can be modified so as to make final KPN more efficient. Such output is not included in any of the existing tools.

The **Unparsing** action generates a .c and .h file for each task. Such kind of output is useful for testing the KPN as well as implementing the KPN on multiple processor platform as each .c and .h file acts as a single unit that can be converted to a form to be run by KPN scheduler. As an example, to test the generated KPN from our tool on YAPI [6] which is a c++ library to run programs transformed as KPN , We just need to modify each .c file into a class . Such kind of output is not generated in any of the existing approaches.

7. CONCLUSION

suggestedmethodology implements the basic overall flow but extends it to provide certain benefits which are not present in existing approaches. Here we compare characteristics of our approach with existing ones in order to mark the improvements provided with our approach. We have differentiated suggested approach from previous approaches on following basis:

- *Directly transform sequential C source*: This describes if manual modification is required to the input source to be accepted by the target tool or can it be directly accepted as-it-is without any manual modification. For example, in Compaan the sequential C source first needs to be converted to MATLAB source manually.
- *Perform partitioning analysis*: Analysis of task partitioning to suggest if partitioning was good or bad. Partitioning is good if it does not result in sequential execution of tasks.
- *Channel placement analysis*: Modify placement of read and write channel interface statements to improve correctness and efficiency of resulting KPN.
- *Uses KPN computational model*: Describes if the transformed application is a Kahn process Network model of computation.
- *Provide data parallelism*: Describes if the tool handles loops by transforming a loop into some K tasks.
- *Provide functional parallelism*: Describes if the tool handles functions by distributing functions across different tasks.
- *Code inlining*: Refers to replacing function call with the body of the associated function definition. Code inlining is used to reduce context switch time between functionsby reducing the number of functions in source.
- *Generate task and channel report*: Describes if the tool generates some kind of descriptive output in terms of task and channels to give some understanding to the user regarding the generated final KPN.

Parameters	Compaan	Sprint	Harmonic	SuggestedTool
Directly transform sequential source	No	Yes	Yes	Yes
Perform Code profiling	No	No	No	Yes
Perform Partitioning analysis	No	No	No	Yes
Channel Placement Analysis	No	No	No	Yes
Use KPN computational model	Yes	Yes	No	Yes
Provide functional parallelism	No	Yes	Yes	Yes
Code Inlining	No	No	No	Yes
Generates task and channel report	No	No	No	Yes

Table 1 Table of Comparison

8. REFERENCES

[1] Bharath, N., and Nandy, S. A runtime mechanism for detection of artificial deadlocks in process networks. In Circuits and Systems, 2004. MWSCAS '04. The 2004 47th Midwest Symposium on (25-28 2004), vol. 2, pp. II-437 - II-440 vol.2.

[2] B.Kienhauis, E., and E.F.Deprettere. Compaan : Driving processnetworks from matlab for embedded signal processing architecture. In proceedings of Eighth International workshop CODES (2000).

[3] Buss, M., Edwards, S., Yao, B., and Waddington, D. Pointeranalysis for source-to-source transformations. In Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on (302005), pp. 139 - 148.

[4] C.Liao, D.J.Quinlan, J., and T.Panas. Extending automatic parallelization to optimize high leel abstraction for multicore. In Proceedingsof 5th international workshop on openMP IWOMP (2009), pp. 28-41.

[5] Dave, B., Lakshminarayana, G., and Jha, N. Cosyn: Hardware-software co-synthesis of heterogeneous distributed embedded systems.Very Large Scale

Integration (VLSI) Systems, IEEE Transactions on 7,1 (Mar 1999), 92 -104.Bibliography 107

[6] De Kock, E., W.J.M.Smith, P. v. d. W., Brunel, J., W.M.Kruijtzer, P.Lieverse, K., and G.Essink.Yapi: Application modelling for signal processing systems. In Proceedings of the 37thAnnual Design Automation Conference (2000), pp. 402-405.

[7] Dulloo, J., and Marquet, P. Design of a real-time scheduler forkahn process networks" on multiprocessor systems. Rapport LIFL 2003-2006 (September 2003).

[8] Edward.A.Lee, and Thomas.M.Parks. Dataow process networks.In Proceedings Of IEEE (May 1995), vol. Vol 83, pp. 773-801.

[9] Geilen, M., and Basten, T. Requirements on the execution ofkahn process networks. In ESOP'03: Proceedings of the 12th European conference on Programming (Berlin, Heidelberg, 2003), Springer-Verlag,pp. 319-334.

[10] Graphviz. Graph vizualization library. http://http://www.graphviz.org/.

[11] Haid, W., Schor, L., Huang, K., Bacivarov, I., and Thiele, L.Eficient execution of kahn process networks on multi-processor systemsusingprotothreads and windowed fifos. In Embedded Systems for RealTime Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on (15-16 2009), pp. 35 -44.

[12] J. Cockx, K.Donolf, B., and R.Stahi. SPRINT:A tool to generateconcurrent transaction-level models from sequential code. In EURASIPJournal on Applied Signal Processing (January 2007), vol. 1, p. 213.Bibliography 108

[13] W.Luk, J.G.F coutinho, T., Y.M.Lam, W., and K.W.Susanto,O.Liu, W. Harmonic: A high level compilation toolchain for hetrogeneous systems.

[14] In IEEE international SOC conference (Sept 9-11 2009).