RPNG: A Tool for Random Process Network Generation

Basant Kumar Dwivedi
Calypto Design Systems (I) Pvt. Ltd.
4^{th} Floor, Som Datt Tower
K-2, Sector 18, NOIDA-201301, India
basant@calypto.com

Harsh Dhand, M. Balakrishnan and Anshul Kumar
Dept. of Computer Science &Engineering
Indian Institute of Technology Delhi
New Delhi-110016, India
{harshdhand, mbala, anshul}@cse.iitd.ernet.in

Abstract

In this paper, we present a user controllable pseudo random process network generator (RPNG). It generates random process networks which can be used as test cases for tools related to application specific multiprocessor architectures. RPNG generates database of computation and communication attributes of process networks for various processors and communication resources. These attributes are controlled by user specified parameters. RPNG also generates code for the process network ensuring that these networks are deadlock free. Generated code can be used as a workload either on a simulator or on an actual platform to study the underlying architecture. Another advantage of RPNG is that it enables one to reproduce results of other researchers.

1 Introduction

Many streaming media applications such as MPEG2, MPEG4, H264 etc. can be represented as Kahn Process Networks (KPN) [9, 4]. In KPN nodes represent processes and arcs represent first in first out (FIFO) communication between processes. This representation of the application allows one to expose functional and data parallelism available in the application quite well. Figure 1 shows KPN model of MPEG-2 video decoding [12] application as an example.

Effectiveness of KPN representation for streaming media application is well established in literature [4, 13, 8, 3, 15]. Unlike TGFF [5] and STG [16] which generate pseudo random directed acyclic task graphs for scheduling problems, there is no effort to the best of our knowledge towards having a set of benchmarks in KPN form which enables research in application specific multiprocessor synthesis for process networks. Objective of RPNG is to fill this gap.

Rest of the paper is organized as follows. In Section 2, we discuss a tool such as RPNG for generating process networks randomly is required and highlight our main contributions. Typical structure of a process network is discussed in Section 3. Procedure for generating topology of the process network is discussed in Section 4. Section 5 describes in detail how we insert computation and communication within a process and generate code and Section 6 describes generation of the database for application specific multiprocessor architecture synthesis problem instance. In Section 7, we describe some experimental results followed by conclusions in Section 8.

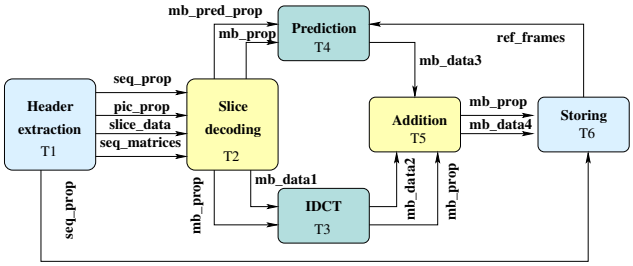


Figure 1. MPEG-2 video decoder

2 Motivation for RPNG

To synthesize the architecture for applications starting from the process network representation, a designer can take one of two possible approaches. The first approach relies on extracting periodic directed acyclic graph (DAG) out of the application and performing synthesis by static scheduling, resource allocation and binding. We refer to this approach as *static scheduling based approach*. This approach has extensively been studied in the literature from cost [2, 1] as well as power optimization [11, 14] point of view. DAG of the application can be extracted by decomposing and unrolling inner loops as discussed in [1, 10]. For example, Figure 2 shows part of the DAG obtained after decomposing the process network of MPEG2 decoder shown in Figure 1 at the granularity of a picture. As pointed out in

[1], we also note that sub-processes (such as T1_1 in Figure 2) derived from the same process, should be mapped to the same resource due to strong internal communication (variable sharing etc.). This constraint has been termed as *type consistency constraint*.

The second approach tries to synthesize the architecture by exploiting the fact that *type consistency constraint* anyway needs to be respected, Hence, it uses higher level characteristics of the process network without decomposing it. In this, resource allocation and binding is performed and presence of a dynamic scheduler is assumed to take care of run-time scheduling. We refer to this method as *partitioning based approach* [13, 3, 7, 6].

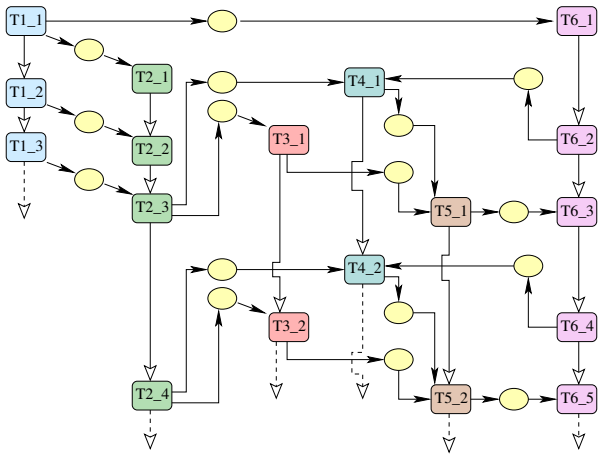


Figure 2. Unrolled MPEG-2 video decoder

Let us consider again the MPEG-2 video decoder shown in Figure 1 to compare the two synthesis approaches discussed above. Processes *IDCT, Prediction, Addition* and *Storing* operate at macroblock level. Figure 2 shows part of the decomposed process network in the form of DAG and Figure 3 shows corresponding pseudo code. If this decoder performs decoding of pictures of size 512×512, then there will be around $(512 \times 512)/(64 \times 6) = 682$ macroblocks assuming 4:2:0 chroma format. Now we observe that if we try to derive DAG by unrolling inner loops of Figure 3, there will be more than $682 \times 4 = 2728$ computation sub-processes and $682 \times 6 = 4092$ communication sub-processes. Hence, total sub-processes (n) will be more than 6820.

If we choose algorithm proposed in [1] as a representative of *static scheduling based approach*, then this algorithm can solve the above problem in $n.Q.K^m$ time. Here n is the total number of sub-processes, Q is the bound on critical path delay, K is the maximum number of mappings for any sub-process and m is a number such that $0 \leq m \leq r \leq n$. Here r is the number of processes in the original application (6 in this example). It can be easily seen from Figure

```
do_mpeg2_picture_decoding {
    header extraction;
    for each macroblock {
        do variable length decoding; do IDCT; do prediction;
        add IDCT and prediction results;
        store macroblock in the picture buffer;
    }
}
```

Figure 3. Computation within MPEG-2 video decoder

2 that there will be at least 682 sub-processes on the critical path. Hence critical path delay Q will be $C_1 \times 682$ where C_1 is some constant ≥ 1. Assuming $m = r/2$ and $K = 4$, time taken T_1 by Chang's algorithm proposed in [1] will be:

$$T_1 = 6820 \times C_1 \times 682 \times 4^3 = C_1 \times 640 \times 682^2$$
$$\Rightarrow T_1 > C_1 \times (640)^3$$

On the other hand, if we follow *partitioning based approach* by using an algorithm such as one proposed in [6], it takes only $T_2 = (N_Q)^4$ running time. Here N_Q is the number of queues in the original process network (15 in the above example). Clearly T_1 is much larger than T_2. Now it can be noticed that *partitioning based approach* to synthesize the architecture for process networks is quite meaningful due to the following advantages.

- Partitioning based synthesis algorithms run faster because of much smaller problem size.
- It enables the designer to explore a larger design space.

TGFF [5] and STG [16] generate pseudo random directed acyclic task graphs for synthesis problems targeted for *scheduling based approaches*. On the other hand, there is no such effort to the best of our knowledge towards having a set of benchmarks in process network form as input to *partitioning based approach* for architecture synthesis. Essentially, these process networks should serve the following objectives:

- creating many problem instances for synthesis of architectures for process networks enabling research in this domain
- producing benchmarks as workload for multiprocessor simulator or a real platform to study the underlying architecture
- enabling researchers to reproduce results of others

Objective of RPNG is to support the above objectives for process networks. RPNG accepts a set of parameters such as upper bounds on number of processes, on number of input and output channels of a particular process, amount of

nesting of loops in a process etc. and generates the process network randomly alongwith the database. Generated database is nothing but a test case for architecture synthesis based on process network. This database essentially consists of computation and communication characteristics of the process network on processors and communication resources. RPNG also generates source code of the process network which is compatible with our thread library. The code generation phase of RPNG makes sure that process network is deadlock free. Using RPNG, a large number of random process networks can be generated quickly which allows one to effectively evaluate architectural synthesis algorithms and reproduce results of other researchers.

3 Structure of a Process Network

All the processes of the process network can be categorized into three classes: primary input (PI), intermediate and primary output (PO) processes. PI processes are the one which read data tokens from external world and initiate data flow inside the process network. Intermediate processes communicate with other processes only through internal channels, whereas PO processes act as sink and write data tokens to the external world. PI, intermediate and PO processes together define the order in which data flows in the process network. A sample process network is shown in Figure 4 where various PI and PO processes can be identified. In Section 5, we describe in detail how this classification of processes helps to generate deadlock free code for the process network.

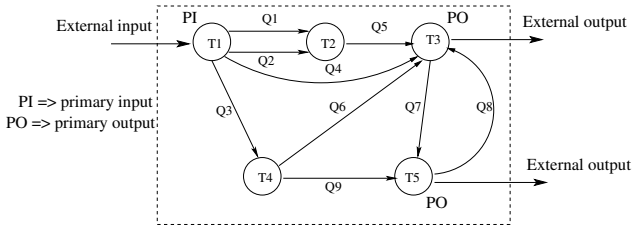


Figure 4. Example process network

As discussed above, PI processes are responsible for initiating data processing and communication. Intermediate processes further perform additional processing and data eventually reaches PO processes. It means that data flow inside the process network is from PI processes to PO processes which essentially implies that there is always some path from PI to PO processes. This property which we term as *path property* must be ensured while adding new processes and channels during process network generation.

A process in the process network of a streaming application is nothing but a nested loop in which outer one is an infinite loop due to streaming nature of the process. Inside each loop computation and communication are interleaved. Figure 5 shows a sample process. *ComputeStatement* is some processing on internal data of the process. These are the statements which introduce finite computation delays. *CommmunicateStatement* is a communication on some FIFO channel of the process network. This communication is either reading a token from an input channel (READ statement) or writing a token into an output channel (WRITE statement). *ComputeStatement*, *CommmunicateStatement* and loops *LoopStatement* can appear in any order in a generic process.

```
LoopStatement {
    ComputeStatement;   CommmunicateStatement;
    LoopStatement {
        ComputeStatement;   CommmunicateStatement;
        ...  }
    ...  }
```

Figure 5. Structure of a process

Channels of the process network are FIFO in nature. In KPN, writes into the channels are non-blocking which could potentially lead to unbounded memory requirements. However, in practice, size of channel is limited. Hence, while generating the process network, we also specify randomly maximum number of tokens which can reside in a channel. This makes RPNG generated process network WRITE blocking as well. The code generation phase of RPNG takes this into consideration as well and makes sure that it does not introduce deadlock.

4 Process Network Generation

RPNG uses Algorithm 1 to generate a standalone process network. The basic ideas behind this algorithm are as follows.

- By construction, ensure *path property* such that each process contributes towards dataflow from some PI to some PO process.
- Allow a channel to be present between any two processes. This leads to generation of a cyclic multi-graph structure of the process network as discussed earlier.
- Insert statements during code generation in a manner such that at any point in time, at least one process is ready to read/write data to/from a channel. This makes sure that the generated process network is deadlock free.

Based on the above ideas, RPNG works in three phases: creating process network topology, inserting computation and communication statements in the processes which helps

in generating code and creating database for the synthesis problem instance. In Algorithm 1, steps 1-5, 6 and 7 correspond to these three phases respectively. The first phase is discussed in this Section and rest of the phases are described in Sections 5 and 6 respectively.

Algorithm 1 createPN

1: *initPN*: initialize the process network.
2: **repeat**
3: *addProcess*: insert a new process alongwith associated channels.
4: *addChannel*: select *doFanout* (connecting output channels of some process) or *doFanin* (connecting input channels of some process) with equal probability.
5: **until** upper bound on number of processes present in the process network (ub_num_processes) is reached
6: *generateCode*: insert READ and WRITE communication statements in processes and dump source code of the process network.
7: *createDatabase*: generate computation and communication characteristics of the process network onto architectural resources based on the user specified attributes.

Various phases of process network generation are controlled by a set of parameters. Parameters which control the topology of the process network and code generation are as follows. Other parameters and attributes are discussed in detail in Section 6.

- **num_pi:** Number of PI processes which act as sources in the process network.
- **ub_num_po:** Upper bound on number of PO processes. These are the sink processes. In the final process network, there will be at least one PO process.
- **num_processes:** Number of processes present in the process network. This must be $\geq$ (num_pi + ub_num_po).
- **ub_num_in_chns:** Upper bound on the number of input channels of a process.
- **ub_num_out_chns:** Upper bound on the number of output channels of a process.
- **ub_nesting_level:** Upper bound on the number of nested loops within any process.

4.1 Initialization of process network

In step 1 (*initPN*) of Algorithm 1, we instantiate num_pi PI processes and ub_num_po PO processes. At this stage, it is required that there is at least one path from a PI to some PO process. To create this path, we choose an unconnected PI (say P_i) and another process from the set of connected PI and all PO processes (say P_j is chosen) randomly with uniform probability. At this point a new channel is instantiated with reader as P_j and writer as P_i. Thus we establish paths from each PI to some PO process.

ub_num_processes	num_pi, ub_num_po	ub_num_in_chns, ub_num_out_chns
6	2, 2	3, 3

Table 1. Parameters for process network graph generation

Figure 6(a) shows the structure of a process network after *initPN* for input parameters specified in Table 1. In Figure 6, *PI_i* is i^{th} PI process, *PO_j* is j^{th} PO process, *IntP_k* is k^{th} intermediate process and *C_l* is l^{th} channel considered so far. Since initial path from a PI process to some PO process could be either direct or through some other PI process, some of PO processes might remain unconnected. *PO_1* is one such PO process in Figure 6(a). If there is any such process, we remove it from the process list. It can be easily observed that there will be at least one PO process which is connected to some PI process after *initPN* step.

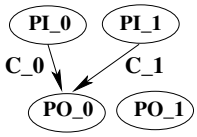
4.2 Addition of a new process

In step *addProcess* of Algorithm 1, to add a new process, one of the channels is chosen out of all the channels with uniform probability. Now if a new node (P3) is inserted between two already existing nodes say P1 and P2, then the channel P1→P2 is removed and two additional channels are added P1→P3 and P3→P2. Since P1 is reachable from some PI (due to initialization), P3 will also be reachable because of channel P1→P3. Similarly, since there exists a path from P2 to PO, a path will also exist from P3 to PO because of the channel P3→P2. Thus while adding new processes, *path property* remains valid.

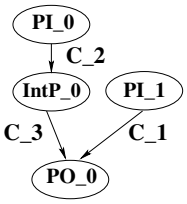
Figure 6(d) shows an instance of *addProcess*. Channel *C_3* is selected with uniform probability out of all the channels present in the process network and broken into two parts. *C_3* is removed and two new channels (*C_6* and *C_7*) alongwith a new intermediate process *IntP_1* are introduced.

4.3 Addition of a new channel

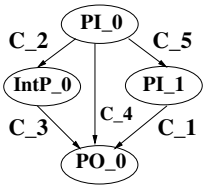
Though a channel is removed and two new channels alongwith a new process are added in *addProcess* step, more channels are added mainly in step 4 (*addChannel*) of Algorithm 1. *doFanout* and *doFanin* have certain similarity



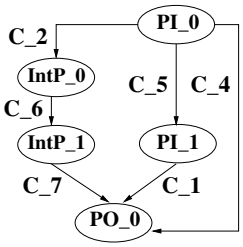
(a) After initialization



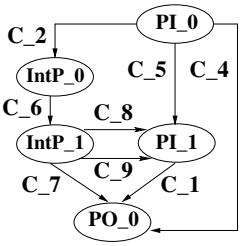
(b) After 1^{st} addProcess



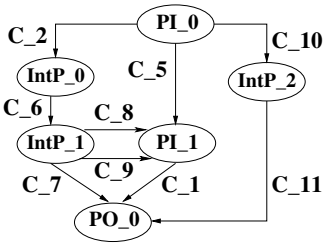
(c) After 1^{st} addChannel



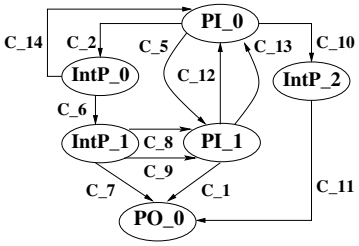
(d) After 2^{nd} addProcess



(e) After 2^{nd} addChannel



(f) After 3^{rd} addProcess



(g) After 3^{rd} addChannel

Figure 6. Various stages of process network graph generation

with *fanout* and *fanin* steps of TGFF [5] respectively. Unlike TGFF where *fanout* and *fanin* steps result in addition of new nodes also, we don't add new processes in *addChannel*, but only establish connections between unutilized input and output ports of processes. The reason for not adding new processes in this step lies in the structure of process network which could be a cyclic multi-graph. Our approach leads to introduction of back edges, whereas TGFF approach always generates directed acyclic graph (DAG).

Algorithm 2 describes *doFanout* step of *addChannel*. Figures 6(c) and 6(e) show addition of new channels after *doFanout*. Table 2 shows fanout probabilities corresponding to step 6 of Algorithm 2. To select a process such that its unused output channels can be connected, a random number with uniform probability is generated in the range $[0, 1]$. In this case, this number was 1 and hence as per Table 2, process IntP_1 was selected and channels C_8 and C_9 were added.

Similar to *doFanout*, Algorithm 3 describes *doFanin* step of *addChannel*. Table 3 shows fanin probabilities as computed in steps 2-5 of Algorithm 3. Now we need to select a process which will get its unused input channels connected (step 6 of Algorithm 3). Again a random number was generated in the range $[0, 1]$ with uniform probability. At this particular step, 0.3333 was found. This led to selection of process PI_0 and channels C_12, C_13 and C_14 were added in the network of Figure 6(f). Resultant process network is shown in Figure 6(g).

Algorithm 2 doFanout

1: Compute total_available_output_links at all the processes
2: **for all** processes **do**
3: Compute available_output_links at this process
4: Compute fanout probability of the process which is available_output_links/total_available_output_links
5: **end for**
6: Select source process (src_process) out of all the processes based on their fanout probabilities.
7: Prepare the list of the processes other than src_process such that they have some unutilized input candidate channels.
8: If there are not enough input links, then whatever is available should be connected. Otherwise, destination process should be chosen with a uniform probability distribution.

	PI_1	PI_0	PO_0	IntP_0	IntP_1
Unused out links	2	0	3	2	2
Fanout probability	0.222	0	0.333	0.222	0.222

Table 2. Fanout probabilities for Figure 6(d)

Algorithm 3 doFanin

1: Compute total_available_input_links at all the processes
2: **for all** processes **do**
3: Compute available_input_links at this process
4: Compute fanin probability of the process which is available_input_links/total_available_input_links
5: **end for**
6: Select destination process (dest_process) out of all the processes based on their fanin probabilities.
7: Prepare the list of the processes other than dest_process such that they have some unutilized output candidate channels.
8: If there are not enough output links, then whatever is available should be connected. Otherwise, source process should be chosen with a uniform probability distribution.

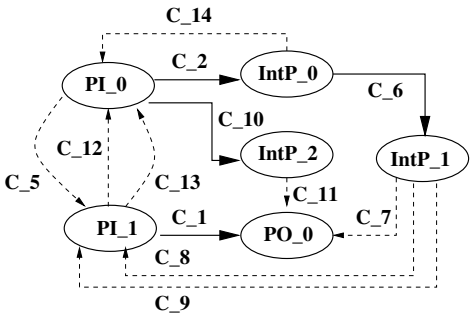
	PI_1	PI_0	PO_0	IntP_0	IntP_1	IntP_2
Unused in links	0	3	0	2	2	2
Fanout probability	0	0.333	0	0.222	0.222	0.222

Table 3. Fanin probabilities for Figure 6(f)

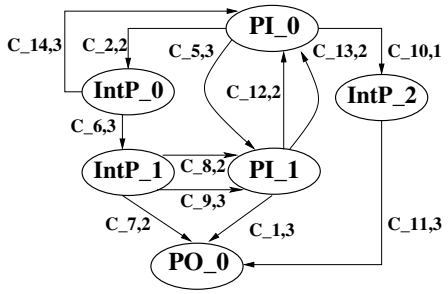
5 Code generation

The order of READ and WRITE communication statements in each process is important as it would determine whether the process network generated will be deadlock free or not. A process network is deadlock free if at no point in time during its execution, all the processes block, either on READ or on WRITE communication statements. A process would block on a READ if the channel from which it is to read is empty and it would block on a WRITE if corresponding output channel is full. For a deadlock free execution, we have to ensure that at any time there exists at least one process which can read/write from/to some channel.

In RPNG, we have introduced the notion of process flow level which primarily comes from the observation that PI processes are the ones which initiate data flow inside the process network and rest of the processes aid this data flow in some order. Process flow levels are assigned in *assignProcessFlowLevels* step of Algorithm 4. This step is a simple breadth first search with the PI processes being assigned the highest flow level and all the processes to which they write as level 1 lower than flow level of PI and so on. Processes once assigned are not considered again. Figure 7(a), shows how the process flow levels are assigned for the process network of Figure 7(b). Initially PI processes PI_0 and PI_1 are assigned flow level 2. Now reader processes of output channels of PI processes are IntP_0, IntP_2 and PO_0. So these processes are assigned flow level 1. Similarly reader process of output channels of processes at flow level 1 which have not been assigned flow level is only IntP_1. Hence this process is assigned flow level 0.



(a) Processes flow levels



(b) Channel nesting levels

Figure 7. Nesting and flow levels in example process network

READ and WRITE communication statements must be inserted inside the processes by taking care of their flow levels. Since PI processes (being at the highest flow level) start data flow, any PI process must first write tokens into output channels. Any read from the input channel at a PI process should come later. Considering this sequence of communication statements in PI processes, the following sequence (we call this flow constraints) of READs and WRITEs needs to be ensured while inserting communication statements inside any other processes for deadlock free execution.

1. Read from higher flow level process.
2. Write to process which is at the same flow level or lower flow level.
3. Write to higher flow level process.
4. Read from a process which is at the lower flow level or same flow level.

Flow constraints 1 and 2 state that each process reads data from higher flow level, does some computation and writes to a process at a lower flow level. These two constraints would have sufficed for the case of directed acyclic graphs. However, processes may also write to a higher flow level process due to cycles present in the process network. So these WRITEs must occur before the process reads from a lower flow level process. This ordering is followed for all the processes at every insertion of READ or WRITE statement. Hence the property that at least one process remains unblocked is never violated.

As discussed in Section 3, a process is basically a nested loop with computation and communication taking place in these loops in an interleaved manner. Hence we associate a loop nesting level to each channel randomly in the range 1 to **ub_nesting_level** in step 6 (*assignNestingLevelToChannels*) of Algorithm 4. Nesting levels of channels of Figure 6(g) are shown in Figure 7(b) next to each channel's name when **ub_nesting_level** was 3.

Algorithm 4 generateCode

1: *assignNestingLevelToChannels*: randomly associate loop nesting level to each channel.
2: *assignProcessFlowLevels*: assign data flow level to each process.
3: **for all** processes **do**
4: *insertLoops*: insert empty nested loops in the process depending on maximum nesting level of its channels.
5: *insertChannelWrites*: insert WRITE communication statements in the process.
6: *insertChannelReads*: insert READ communication statements in the process.
7: **end for**
8: *printCode*: dump source code of the process network.

Insertion of various statements in processes starts with inserting loops in step *insertLoops* of Algorithm 4. We find out maximum loop nesting of a process by looking at nesting levels of its input and output channels and insert that many loops. For example, maximum loop nesting in the process PI_0 of Figure 7(b) is 3. We insert 3 nested loops for this process as shown in Figure 8. In RPNG, each *LoopStatement* is deterministic i.e. its lower and upper bounds are known. We also impose the restriction that bounds of a loop at any nesting level must be same across all processes to make sure that there is neither underflow nor overflow of tokens inside any channel.

After inserting empty loops in processes, we insert WRITE communication statements. While inserting these statements, flow constraints 2 and 3 discussed above are taken care of. Figure 8 shows WRITE statements after *insertChannelWrites* step of Algorithm 4.

Insertion of READ statements in the processes takes

```
LoopStatement {
    WRITE into channel C_10;
    LoopStatement {
        WRITE into channel C_2;
        LoopStatement {
            WRITE into channel C_5;
        }
    }
}
```

Figure 8. WRITE statements in process PI_0

place in *insertChannelReads* step of Algorithm 4. Here we need to make sure that insertions of READs are as per flow constraints 1 and 4 discussed earlier. So at any nesting level, READs from higher level processes are inserted before any WRITEs and READs from lower level processes are appended after all the WRITEs. Then we insert *ComputeStatement*s. To give a feel of how final structure of processes look like, Figures 9 and 10 show statements of two processes at different flow levels communicating with each other for two processes of the process network shown in Figure 7(b).

```
LoopStatement {
    ComputeStatement;   WRITE into channel C_10;
    LoopStatement {
        ComputeStatement;   WRITE into channel C_2;
        LoopStatement {
            ComputeStatement;   WRITE into channel C_5;
            ComputeStatement;   READ from channel C_14;
        }
        ComputeStatement;   READ from channel C_12;
        ComputeStatement;   READ from channel C_13;
    }
}
```

Figure 9. Statements in process PI_0

```
LoopStatement {
    LoopStatement {
        ComputeStatement;   READ from channel C_2;
        LoopStatement {
            ComputeStatement;   WRITE into channel C_6;
            ComputeStatement;   WRITE into channel C_14;
        }
    }
}
```

Figure 10. Statements in process IntP_0

We discussed above that in RPNG, each *LoopStatement* is deterministic and bounds of these statements are same at each nesting level across all processes. This is essential to make sure that reader processes consume the same number of tokens which writer processes have produced. Though loops are deterministic, RPNG allows to have different loop bounds at different nesting levels.

Finally, step *printCode* of Algorithm 4 dumps code of the whole process network in C language compatible with our thread library. Our thread library is also written in C and internally it uses *pthreads* for parallelism. If generated

code is used as a workload on a simulator or on an actual platform, then underlying threading mechanism *pthreads* should be replaced accordingly.

It can be noticed that a *ComputeStatement* is the one which imposes processing delays. In Section 6, we discuss input parameters in the form of a set of attributes which decide computation and communication structure of the process network. Out of these attributes, **stmt_processor_attribute** relate *ComputeStatement* statements of processes to processors for corresponding mapping. Examples of these attributes are *n_cyc_avg* and *n_cyc_var*, which are the average and variance for the number of cycles taken by some *ComputeStatement* on a processor respectively. By making use of these parameters, we assign average and variance for number of cycles taken by each *ComputeStatement* on a particular processor. Depending on the final architecture and application mapping, this information can be used by the run time schedulers to provide data dependent behavior for generated process network.

6 Creation of Database

RPNG allows a user to specify a number of attributes. These attributes are associated with the processes, channels and architectural resources and are used to generate computation and communication characteristics of the process network on various processors and communication resources. RPNG accepts the following types of attributes.

- **channel_attribute:** Attributes associated with each channel.
- **processor_attributes:** Attributes specific to a processor.
- **memory_attributes:** Attributes specific to a memory.
- **switch_attributes:** Attributes specific to a switch. In RPNG, a switch can be a cross-bar switch or a bus.
- **process_processor_attributes:** Attributes which relate processes to processors for corresponding mapping.
- **stmt_processor_attributes:** Attributes which relate *ComputeStatement* statements of processes to processors for corresponding mapping.

Similar to TGFF, we specify two parameters *av* and *mul* for each attribute apart from its name. We use the same equation given in TGFF to generate attribute value.

$$attrib = av + jitter(mul.rand, var) \quad (1)$$

where *rand* is a random value between -1 and 1, *var* is an input parameter in the range $[0,1]$ and $jitter(x,y)$ returns a random value in the range $[x.(1-y),\ x.(1+y)]$.

There are very few applications available in the process networks form. Due to this reason, exact probability distribution for each attribute could not be obtained. That is why we had to rely on approach similar to TGFF to randomly generate values of different attributes. However, structure of RPNG is very general and one can easily refine RPNG to apply exact probability distributions for various attributes.

As an example, Figure 11 shows some of the attributes for the problem instance of application specific multiprocessor synthesis problem for the process networks. Here channel_attribute *thr* which is throughput (number of tokens/second on the channel), *channel_size* which is maximum number of tokens in a channel and *token_size* which is the size of a token in any channel must be specified. All other channel attributes are optional. Here *thr* is nothing but the performance constraint on the application. Similarly stmt_processor_attributes *n_cyc_avg* and *n_cyc_var*, which are the average and variance for the number of cycles taken by some *ComputeStatement* on a processor, are required and other stmt_processor_attributes are optional. Apart from these, other attributes shown in Figure 11 should always be specified.

```
num_processors                3
num_memories                  3
num_switches                  3
channel_attribute             channel_size 4 2
channel_attribute             token_size 1000 100
channel_attribute             thr 500 100
processor_attributes          freq 500 50
processor_attributes          contxt 200 0
processor_attributes          cost_proc 100000 50000
processor_attributes          cost_base_lm 20 5
memory_attributes             bwmm 10 2
memory_attributes             cost_base_sm 15 4
stmt_processor_attributes     n_cyc_avg 200 50
stmt_processor_attributes     n_cyc_var 30 10
```

Figure 11. Sample attributes for database

Now for input parameters of Table 1 and attributes specified in Figure 11, partial database of 0^{th} processor is shown in Figure 12. In this figure, *pr_cyc_avg* is the average number of cycles taken by a process for its single iteration on the corresponding processor. This was computed by accumulating delays of various statements present in the process. This partial database shows part of the problem instance of application specific multiprocessor synthesis problem for the process networks. Thus we see that RPNG is capable of handling an arbitrary number of attributes and generating problem instances and test cases.

```
@Processor 0 {
# freq  contxt  cost_proc   cost_base_lm
 448.35 200.00  55029.19    14.77
#-----------------------------------------
# process_id  pr_cyc_avg
 0  2060.000000
 1  1883.000000
 2  1530.000000
 3  1745.000000
 4  1540.000000
 5  753.000000
}
```

Figure 12. Partial database

7 Experimental Results

We performed a number of experiments using RPNG in which we applied process network generated by RPNG as input to algorithm presented in [6]. This algorithm takes application in the form of process network and synthesizes multiprocessor architectures such as shown in Figure 13. Table 4 shows some of the experimental results. The third row in this table gives time taken to reach the solution and last two rows give number of processors and shared memories synthesized for some of the problem instances. Whenever, more than one shared memory modules were instantiated, we obtained the interconnection network similar to the architecture of Figure 13. We did these experiments on a workstation with Intel XEON [17] CPU running at 2.20GHz. It can be observed that the solutions for all these process networks took less than one second. It can be noticed that RPNG allowed one to conduct a large number of application specific multiprocessor synthesis experiments by quickly producing a number of test cases.

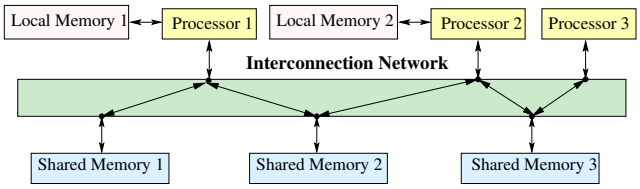


Figure 13. Target architecture template

Number of processes	10	20	30
Number of queues	19	57	60
Time taken in Sol.	<1 sec	<1 sec	<1 sec
Number of instantiated processors	3	5	9
Number of shared memories	1	8	8

Table 4. Experiments using RPNG

8 Conclusions

We presented a tool for randomly generating process networks which can be used for generating test cases for synthesis of application specific multiprocessor architectures. Apart from generating computation and communication characteristics of the process network in the database, RPNG also produces a "source" code which executes without deadlock. This code can be used as a workload for simulation purposes to study the underlying architecture.

RPNG is highly parametrized and capable of handling a variety of computation and communication attributes. Using RPNG, a large number of process networks can be generated quickly which in turn allows one to comprehensively test the algorithms and also reproduce test cases of other researchers to compare the results. We have used RPNG extensively for performing experimentations on our framework for synthesis of application specific multiprocessor architectures for applications represented as process networks [6]. RPNG has been implemented in C++ and it is available on request.

9 Acknowledgments

We are very thankful to Naval Research Board, Govt. Of India for sponsoring this work under the project titled "Srijan - A Methodology for Synthesis of ASIP based Multiprocessor SoCs".

References

[1] J.-M. Chang and M. Pedram. Codex-dp: Co-design of Communicating Systems Using Dynamic Programming. *IEEE Trans. on CAD*, 19(7):732–744, July 2000.

[2] B. P. Dave, G. Lakshminarayana, and N. K. Jha. COSYN: Hardware Software Cosynthesis of Heterogeneous Distributed Embedded Systems. *IEEE Trans. on VLSI Systems*, 7(1):92–104, Mar. 1999.

[3] E. A. de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. International Symposium on System Synthesis (ISSS-2002)*, pages 68–73, Oct. 2002.

[4] E. A. de Kock et al. YAPI:Application Modeling for Signal Processing Systems. In *Proc. Design Automation Conference (DAC-2000)*, pages 402–405, June 2000.

[5] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task Graph Generation for Free. In *Proc. 6th International Workshop on Hardware/Software Codesign (CODES - 1998)*, pages 97–101, 1998.

[6] B. K. Dwivedi, A. Kumar, and M. Balakrishnan. Automatic Synthesis of System on Chip Multiprocessor Architectures for Process Networks. In *Proc. Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2004), Stockholm, Sweden*, pages 60–65, Sept. 2004.

[7] B. K. Dwivedi, A. Kumar, and M. Balakrishnan. Synthesis of Application Specific Multiprocessor Architectures for Process Networks. In *Proc. 17th International Conference on VLSI Design, Mumbai, India*, pages 780–783, Jan. 2004.

[8] O. P. Gangwal, A. Nieuwland, and P. Lippens. A Scalable and Flexible Data Synchronization Scheme for Embedded HW-SW Shared-Memory Systems. In *Proc. Int. Symposium on System Synthesis (ISSS-2001)*, pages 1–6, Oct. 2001.

[9] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. IFIP Congress 74*. North Holland Publishing Co, 1974.

[10] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4):406–471, Dec. 1999.

[11] J. Luo and N. K. Jha. Low Power Distributed Embedded Systems: Dynamic Voltage Scaling and Synthesis. In *Proc. International Conference on High Performance Computing (HiPC-2002)*, pages 679–692, Dec. 2002.

[12] MPEG. *Information technology - Generic coding of moving pictures and associated audio information: Video*. ISO/IEC 13818-2, 1996.

[13] A. D. Pimentel, L. O. Hertzberger, P. Lieverse, P. van der Wolf, and E. F. Deprettere. Exploring Embedded Systems Architectures with Artemis. *IEEE Computer*, (11):57–63, Nov. 2001.

[14] D. Shin and J. Kim. Power-Aware Scheduling of Conditional Task Graphs in Real-Time Multiprocessor Systems. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED-2003)*, pages 408–413, Aug. 2003.

[15] T. Stefanov and E. Deprettere. Deriving Process Networks from Weakly Dynamic Applications in System-Level Design. In *Proc. International Conference on HW/SW Codesign and System Synthesis (CODES+ISSS - 2003)*, pages 90–96, Oct. 2003.

[16] STG. http://www.kasahara.elec.waseda.ac.jp/schedule/.

[17] XEON. http://www.intel.com/products/server/processors.