Automatic Synthesis of System on Chip Multiprocessor Architectures for Process Networks

Basant Kumar Dwivedi, Anshul Kumar and M. Balakrishnan
Dept. of Computer Science and Engineering
Indian Institute of Technology Delhi, New Delhi, India
{basant, anshul, mbala}@cse.iitd.ernet.in

ABSTRACT

In this paper, we present an approach for automatic synthesis of System on Chip (SoC) multiprocessor architectures for applications expressed as process networks. Our approach is targeted towards design space exploration (DSE) and thus the speed of synthesis is of critical interest. The focus here is on the problem of resource allocation and binding with a view to optimize cost under performance constraints. Our approach exploits adjacency relation of processes and uses a dynamic programming based algorithm to synthesize the architecture including interconnection network. We have done a number of experiments on real as well as randomly generated process networks. The results have been compared with an optimal MILP formulation. They conclusively show that this approach is fast as well as effective and can be employed for DSE.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Interconnection architectures*

General Terms

Design, Performance

Keywords

Application Specific Multiprocessors, Kahn Process Networks, Partitioning

1. INTRODUCTION

Typically streaming applications are modeled as either process networks [1, 2, 3] or periodic task graphs in the form of directed acyclic graph (DAG). Processes in a process network communicate through FIFO queues. Unlike task graphs, in process networks, computation and communication are interleaved and parallelism present in the application is fully exposed.

Synthesis of optimal application specific multiprocessor architectures for DAG based periodic task graphs with real time processing requirements has been extensively studied in literature [4, 5, 6, 7] from cost as well as power optimization point of view. However, synthesis for process networks is not widely explored. The approaches for DAG based task graphs rely on static scheduling and solve the synthesis problem which essentially consists of architectural resource allocation, binding of application components to architecture, and scheduling. This approach can also be used for process networks as well by unrolling inner loops of processes and decomposing it in the form of a periodic DAG based task graph [8, 5]. Such an approach might lead to a very large size graph for most of the real life applications making the approach impractical.

We further note that even if a process is decomposed into sub processes, they need to be mapped onto the same resource [5]. This requirement arises from the observation that sub-processes derived from the same process are tightly coupled in terms of sharing of variables etc. It suggests that the process network need not be further decomposed and its higher level characteristics can be used to synthesize the architecture for the given process network(s).

As pointed out above, the problem of automatic synthesis of optimal architecture for process networks has not been adequately addressed in the literature. Piementel et al. [9] propose a co-simulation based approach for design space exploration of architectures for Kahn Process Networks (KPNs) [1]. In this methodology, architecture and application mapping is specified manually and co-simulation is used to do performance analysis. A three stage process network refinement based approach has been proposed in [10] for heterogeneous multiprocessor mapping of process networks. In this approach, process network refinements are done manually to reduce communication overhead and expose more data parallelism. Here, underlying architecture is a fixed bus based architecture which is specified manually and a dynamic scheduler is used to take care of scheduling of processes. Dynamic scheduling problem of process network has also been addressed in [11, 12] on the given architectural resources.

There has been some work in the direction of defining suitable communication architecture also for streaming applications. Leijten et al. [13] have proposed PROPHID multiprocessor architecture. This architecture essentially consists of a general purpose processor for control oriented processes, a number of application domain specific (ADS) processors and FIFO buffers for communications. Allocation of interconnection network (IN) bandwidth to various data streams has been addressed in [14]. In this work, architectural resource allocation and binding of application onto architecture is assumed to be pre-specified. PROPHID architecture does not exploit communication properties of application effectively to reduce interconnection network cost.

In this work, we fill the gaps present in automatic synthesis of optimal architectures for process networks and propose a framework for the same. Architectural resource allocation is performed

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'04, September 8–10, 2004, Stockholm, Sweden.
Copyright 2004 ACM 1-58113-937-3/04/0009 ...$5.00.

in which resources (processors, memories etc.) are chosen from a component library. Alongwith resource allocation, we perform binding of various processes of the process network onto processors and queues onto memories. Another novel feature of our work is synthesis of interconnection network by exploiting communication pattern present within the application process network.

The outline of rest of the paper is as follows. Section 2 gives details of system model. Section 3 formulates the synthesis problem addressed in this paper and provides an intuition into our approach. Section 4 gives details of synthesis algorithm. In Section 5 experimental results are given followed by conclusions in Section 6.

2. SYSTEM MODEL

2.1 Application Model

In the previous Section, we said that converting the process network in the form of a DAG by unrolling inner loops might result in a very large task graph. Let us consider MPEG-2 video decoder [15]. Figure 1 shows its process network annotated with queue parameters. Here processes *IDCT, Prediction, Addition* and *Storing* operate at macroblock level. Figure 2 shows corresponding pseudo code. If this decoder performs decoding of frames 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 2, there will be more than $682 \times 4 = 2728$ sub-processes communicating with each other. For applications which use MPEG2 such as video-conferencing, number of sub-processes will be even larger. Hence, the process network need not be decomposed and its higher level characteristics should be exploited to synthesize the architecture.

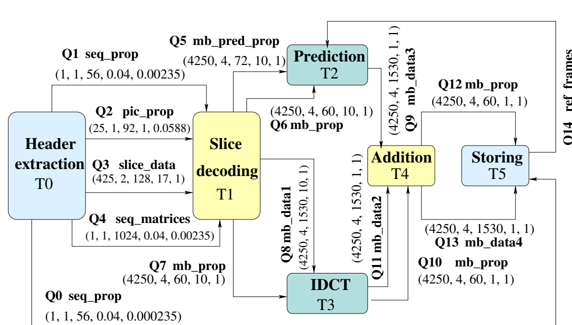

Figure 1: Annotated MPEG-2 video decoder process network

```
do_mpeg2_frame_decoding {
    header extraction;
    for each macroblock {
        do variable length decoding; do IDCT; do prediction;
        add IDCT and prediction results;
        store macroblock in the frame buffer; } }
```

Figure 2: Computation within MPEG-2 video decoder

As shown in Figure 1, the application is specified in the form of a process network. Processes communicate with each other using FIFO queues and computation and communication within any process is interleaved. All the processes are assumed to be periodic. In each iteration of their invocation, they produce or consume certain number of tokens on queues connected to them. We further assume that size of a token being communicated on any queue always remains same and size of each queue (maximum number of tokens within it) is known a priori. We note that once size of each queue is specified, writes on any queue also becomes blocking. We further assume that a dynamic scheduler will take care of the run time scheduling of mapped process network.

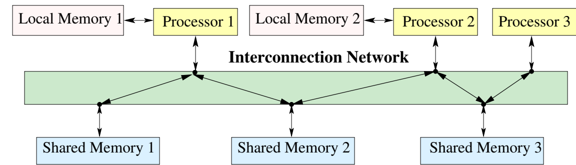

Figure 3: Target architecture template

The real time constraint is specified in terms of how often data tokens needs to be produced/consumed on/from queues. We represent this as throughput constraints on queues which is nothing but the number of tokens to be produced/consumed per second on it. In turn, these throughput requirements on the queues impose processing requirements on corresponding reader and writer processes. This appears in the form that a process must make a certain number of iterations per second. This is derived from the throughput constraints on the queues. It is equal to the maximum of throughput constraint of the queue divided by number of tokens produced or consumed on that queue by the process under consideration. This maximum is taken over all the queues connected to the process.

2.2 Architectural Component Library

Our synthesis algorithm, as described in Sections 4, tries to minimize cost and improve resource utilization. This results in architectures of the type shown in Figure 3, consisting of a set of processors alongwith their local memories, shared memories and interconnection network. Our architectural component library contains modules corresponding to each of these. A processor could be a non programmable unit like ASIC or a programmable unit such as a RISC or DSP processor. A processor has a set of attributes. These are: cost of the processor, frequency, number of cycles taken by the processor during a context switch and the number of cycles taken by a process when mapped onto the processor for one iteration without memory conflicts and context switch overheads.

Associated with each processor, there is a local memory containing queues which have their both reader and writer processes mapped onto associated processor. On the other hand, shared memory modules are accessed by multiple processors. We characterize these memories in terms of their base cost and bandwidth. The base cost is the equivalent hardware cost of allocating a single word in corresponding memory. Here we assume that there is no limit on the size of any memory.

The interconnection network of the synthesized architecture is a partial cross-bar as shown in Figure 3. Typically these switches are implemented using multiplexers having cost which depends on its size (number of sources). Hence, to take care of interconnection cost, the component library also defines cost of each link.

3. THE SYNTHESIS PROBLEM

The synthesis problem can be stated as follows:

Given the application in the form of a KPN and a component library, synthesis of the architecture consists of allocation of processors, local and shared memories, binding of processes to processors and queues to local or shared memories and allocation of communication components such as buses to minimize total cost.

Figure 4 shows an instance of a synthesized architecture. In this example, the application KPN is composed of 3 processes and 3 queues. The synthesized architecture consists of 2 processors, 1 local memory and 1 shared memory. *Queue 1* is mapped to the local memory of *Processor 1* because reader and writer processes of *Queue 1* are mapped here. On the other hand *Queue 2* and *Queue 3* are mapped to the shared memory as their reader and writer processes are mapped to two different processors.

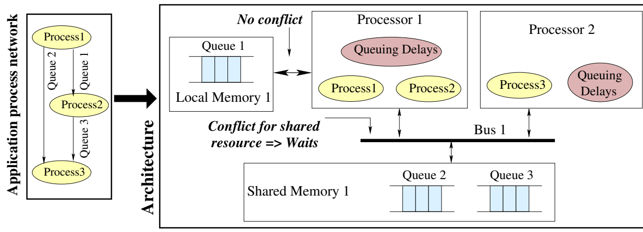

Figure 4: Synthesis example

The **objective function** of our synthesis problem is to minimize hardware cost. In the mapping stage, it essentially consists of cost of processors used, local memory modules, shared memory modules and interconnection cost. Now the synthesis problem needs to be solved under a number of constraints which are described next.

3.1 Mapping Constraints

These are the constraints which define mapping of process network and architecture instance.

1. A process can be mapped to only one processor.
2. A queue is mapped onto a local memory only when its reader and writer processes are mapped to the same processor.
3. A queue is mapped to either local memory of a processor or shared memory module.
4. A processor PR_k will communicate with a shared memory module SM_l when some reader or writer of a queue Q_j is mapped onto PR_k and queue itself is mapped onto SM_l.

3.2 Performance Constraints

3.2.1 Bandwidth Constraint at Shared Memories

Arrival rate of read or write requests at a shared memory module SM_l, depends on communication rates and sizes of data transfer during each request. This request is at some queue of the process network mapped onto SM_l. Bandwidth $bwmm_l$ of this shared memory module should be larger than arrival rate. If QSM_l is the set of queues mapped to shared memory SM_l, thr_j is the throughput constraint for queue Q_j and sz_j is token size for it, then

$$\sum_{Q_j \in QSM_l} thr_j \times sz_j \leq bwmm_l \tag{1}$$

3.2.2 Capacity Constraint at Processors

Referring to Figure 4, we note that there are three possible states of an instantiated processor. Either processor is executing some process, or it is switching context from one process to other or it is waiting due to conflicts while accessing shared memories. A processor offers number of time units equal to its frequency in one second. We call it processor's capacity. Now, total number of cycles required in various states of the processor in one second must not exceed its capacity. This is what we check while verifying whether performance constraints at a particular processor is satisfied or not.

Context Switch Overheads

When multiple processes are mapped onto a single processor, there will be context switch overheads. A process gets blocked while reading a token from an empty queue or during writing a token into queue which is full. We notice that a writer process gets blocked only when it has written at least len_j (maximum number of tokens in Q_j) tokens on Q_j. Similarly, a reader process gets blocked only when it has read at least same number of tokens from the queue. Here, we assume that a process can be blocked only when it is communicating. So, if the set of queues which have either their reader or writer mapped on an instantiated processor IPR_p is QPR_p, then number of context switches per second ($ncontxt_p$) on IPR_p becomes $\sum_{Q_j \in QPR_p} \frac{thr_j}{len_j}$.

Queuing Delays

The mutual interference will appear during accesses on two queues only when they are mapped to the same shared memory module and respective readers and writers are mapped to different processors. We want to find out equivalent waiting time on the processor. Consider synthesis example of Figure 4. If somehow, we get the queuing delay processor 1 will face when it makes a read/write access on Queue 2, we can compute the total overhead per second. To compute it, queuing delays are multiplied by the throughput requirements of Queue 2.

In [16], we proposed an approach to estimate the queuing delays faced by individual processors while accessing some queue on per access basis. We used this method and pre-computed all possible kinds of queuing delays (read-read, write-write, read-write and write-read). We do this for all the processor pairs present in the component library and each pair of queues. Here write-read refers to the case when processor 1 is making a write access to one of the queues and processor 2 is making a read access as in Figure 4. Similarly other delays can also be interpreted.

Now, if two queues Q_r and Q_s are mapped to the same shared memory instance and their readers are mapped to different processor instances IPR_p and IPR_q with types PR_p and PR_q respectively, then equivalent waiting delay at IPR_p will be:

$$data_rrcon_p = \sum_{Q_r}\sum_{Q_s} rrcon[r][s][p][q] \times thr_r \tag{2}$$

where $rrcon[r][s][p][q]$ is read-read precomputed queuing delay as described above. Similar equations can also be written for write-write ($data_wwcon_p$) cases. In this paper, we have considered shared memory modules with one write and one read ports. However, Equation 2 can be easily refined for arbitrary number of ports.

Capacity Constraint

As we discussed earlier, a processor offers number of time units equal to its clock frequency (cycles per second). This must accommodate computation requirements of processes mapped, context switch overheads and waiting time due to data communication queuing delays. If PPR_k is the set of processes mapped to processor instance IPR_p with type PR_k, then the following equation must be satisfied at each processor for the mapping instance to be feasible.

$$\begin{aligned} \forall p:\ & \textstyle\sum_{i \in PPR_p}(ncy_{ik} \times iter_i) + ncycontxt_k \times ncontxt_p + \\ & + (data_rrcon_p + data_wwcon_p)\ \leq freq_k \end{aligned} \tag{3}$$

where $ncycontxt_k$ is the number of cycles taken by processor type PR_k in a context switch. The left hand side of above Equation is composed of total computation requirements of processes mapped, total context switch overheads, and queuing delays respectively.

4. SYNTHESIS ALGORITHM

We propose a heuristic based solution for our synthesis problem. We construct the solution by employing a fast dynamic programming based algorithm. Our approach not only allows it to be used in a design space exploration loop, but also provides a good initial solution for an iterative refinement phase.

We note that the following considerations help us to minimize the cost of synthesized architecture.

- Mapping a process onto a low cost processor.
- Mapping densely connected processes to the same processor will result in putting more queues in the local memory. This in turn helps to reduce interconnection network (IN) cost.
- Mapping queues, where communication takes place more frequently, onto the local memories will help to reduce conflicts at shared memories. It will allow more queues to share the same shared memory resulting in lower IN cost.

One can make the observation that out of above considerations, last two become effective when two adjacent processes from the process network graph are mapped to the same processor. The condition is that these two processes should either be communicating over a number of queues or communicating more frequently. So, mapping such adjacent processes onto lower cost processors would result in a cost effective solution. This is the basis of Algorithm 1, where we first create a list of adjacent processes and then try to map them onto one processor wherever possible.

Algorithm 1 synthesize_architecture

1: Derive Communication Requirement Graph (CRG)
2: Create *adjacent_process_list* by performing weighted topological sort on CRG starting from its *root* vertex
3: Compute partial map $PM[m,n]$ for processes for each $m \rightarrow n$ such that $m \leq n$ in *adjacent_process_list*
4: Refine architecture using shared memory merging

4.1 Computation of Partial Map

In the first three steps of Algorithm 1, we incrementally build partial map for all the processes. We define the partial map (PM) for a set of processes. PM for this set essentially contains its mapping information i.e. binding information of processes of this set to processors and queues (having their reader and writer processes in this set) to memories. For example, Figure 7 shows the partial map for set of processes $[T_4, T_3, T_5, T_0]$ of Figure 1. We note that this set is mapped onto two processors. We term this as multiple processor mapping (MPM). Furthermore, partial map for $[T_4, T_3]$ contains a single processor and associated local memory. We term this as single processor mapping (SPM). Partial map of $[T_5, T_0]$ has similar structure. We also note that queues other than what has been shown in Figure 7 are still unassigned to any memory. That is why this mapping information is partial. In rest of Section, we will describe how adjacent processes (*adjacent_process_list*) are found and partial map is computed. From now onwards, we will use term partial map (PM) only for processes ordered in *adjacent_process_list*.

4.1.1 Derivation of CRG

To create a list of adjacent processes, we derive undirected CRG from the original process network in step 1 of Algorithm 1. CRG is derived by collapsing all the edges between two vertices in original process network into a single edge annotated with total communication between them. Hence, weight on an edge between two processes T_a and T_b in CRG becomes $\sum_{Q_j}(sz_j \times thr_j)$. Here, Q_j is any queue between these processes, sz_j is the size of a token being transferred over this queue and thr_j is its throughput constraint.

Figure 1 shows annotated MPEG-2 video decoder. Here 5-tuple next to each edge is the parameter values for size of a token being transferred, the maximum number of tokens allowed in queue, throughput constraint, number of tokens produced or consumed by the writer or reader process in its single iteration respectively. Corresponding CRG is shown in Figure 5.

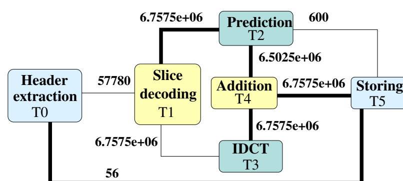

Figure 5: CRG of MPEG-2 video decoder

Next, the vertex, having maximum weight on one of its edges in CRG, is chosen as *root*. Weighted topological sort is performed on CRG starting from this vertex. In this phase, a new edge on the path is chosen which has the maximum weight and leads to another vertex which has not yet been visited. In Figure 5, it starts from T_1 and takes the path $T_1 \rightarrow T_2 \rightarrow T_4 \rightarrow T_3$. The whole path is deleted from CRG and new *root* is chosen if there is no path from the last node on this path to other vertices. For this example, sequence $\{T_1, T_2, T_4, T_3, T_5, T_0\}$ is the *adjacent_process_list*.

Processes of *adjacent_process_list* can be mapped onto different processors in a number of ways. Figure 6 shows two such possibilities for the process network of Figure 1. Here, root node is the *adjacent_process_list* obtained earlier. Next, we assume that there are three processors PR_1, PR_2 and PR_3 in the component library. If we cannot map the whole *adjacent_process_list* onto a single processor due to performance constraints, then we need to break this list into two and evaluate them separately. This can be done in a number of ways which correspond to various sub-trees in Figure 6. We see that in sub-tree 1, we can map process group $[T_1, T_2]$ onto processor PR_1, but process group $[T_4, T_3, T_5, T_0]$ couldn't be mapped onto a single processor. Hence, this list needs to be broken further. We stop exploring this sub-tree as soon as all the group of processes in it are mapped onto some processor under performance constraint. Similarly other sub-trees are also evaluated and the lowest cost sub-tree is selected as the solution.

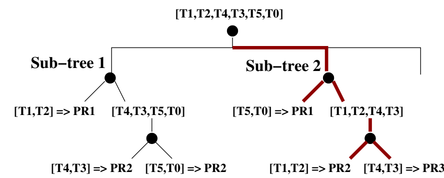

Figure 6: Example solution tree

We observe that the architecture can be synthesized by building partial maps recursively in a bottom up manner. This suggests that a dynamic programming based algorithm can be used for synthesis. In step 3 of Algorithm 1, partial map matrix PM for the solution is created using a similar algorithm. Here, partial map $PM[m,n]$ refers to the mapping information of processes in *adjacent_process_list* between indices m and n.

Now, there exists two possible solutions for the partial mapping of processes of *adjacent_process_list*.

4.1.2 Solution 1: single processor mapping

First we evaluate whether all processes of sequence $m \rightarrow n$ can be mapped onto the same processor from the component library. We choose the lowest cost processor PR_k which satisfies performance constraints. If there is no such processor, then cost of this solution is infinite. Otherwise, single processor partial mapping ($SPM[m,n,PR_k]$) cost becomes:

$$SPM[m,n,PR_k].cost = pr_cost_k + \sum_{Q_j} sz_j \times len_j \times cost_base_lm_k \tag{4}$$

where reader and writer processes of the queue Q_j are in group $m \rightarrow n$ itself. In Equation 4, the cost is composed of cost of the processor chosen (pr_cost_k) and cost of the local memory. Latter is equal to the equivalent hardware cost of mapping single word into the local memory ($cost_base_lm_k$) multiplied by the total local memory required ($\sum_{Q_j} sz_j \times len_j$). For example, if we are computing partial map $PM[2,3]$ ($[T_4, T_3]$) for *adjacent_process_list* of Figure 5 and these processes can be mapped onto a single processor, then queues Q_{10} and Q_{11} will be mapped to the local memory. Rest of the queues connected to this partial map will remain unassigned.

4.1.3 Solution 2: multiple processor mapping

Next, a pivot p in *adjacent_process_list* is moved from m to n. If reader and writer of queue Q_j are in groups $m \rightarrow p$ and $(p+1) \rightarrow n$ or vice versa, then cost of multiple processor partial map ($MPM[m,p,n]$) is composed of 4 components: cost of partial maps $PM[m,p]$ and $PM[p+1,n]$, cost of new shared memories instantiated and cost of new links introduced. To illustrate, let us consider *adjacent_process_list* of Figure 1 again. While combining partial maps $PM[2,3]$ ($[T_4,T_3]$) and $PM[4,5]$ ($[T_5,T_0]$), we instantiate new shared memories for queues Q_{12} and Q_{13} because partial maps communicate via these queues. As shown in Figure 7, two new links are also introduced for each queue. Hence, total cost of new partial map is computed as follows.

$$MPM[m,p,n].cost = PM[m,p].cost + PM[p+1,n].cost + \\ + \sum_{Q_j}(sz_j \times len_j \times cost_base_sm + 2 \times cost_in) \quad (5)$$

We choose the lowest cost solution out of all single processor mappings and multiple processor mappings. This can be done as given in Equation 6. If a single processor mapping $SPM(m,n,PR_k)$ is chosen, then a new instance of PR_k is created, otherwise solution around selected pivot is accepted as $PM[m,n]$. Once matrix PM is fully computed, $PM[0,|T|-1]$ (where $|T|$ is the number of processes) gives us the solution.

$$PM[m,n] = min_cost\ \{\ min_cost_{0\leq k<|PR|}SPM(\ m,n,PR_k\ ), \\ min_cost_{m<p<n}\ \{\ MPM(\ m,p,n\ )\}\ \ \} \quad (6)$$

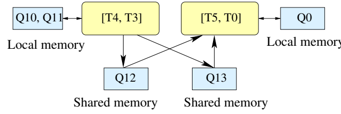

Figure 7: Example partial map

Table 1 shows partial map (PM) matrix for *adjacent_process_list* of Figure 5 assuming that there are three types of processors PR_1, PR_2 and PR_3. We note that partial map matrix is basically an upper triangular matrix. Each valid entry is a 2-tuple. First field of this tuple is the pivot as explained above and second field is processor type (-1 means that corresponding partial map requires multiple processors). We start from $PM[0,|T|-1]$ and proceed until we find valid processor type for each partial map in the path. In this example, it reduces to *Sub-tree 2* of Figure 6.

	0	**1**	**2**	**3**	**4**	**5**
0	0, PR_1	**1, PR$_2$**	1, -1	**1, -1**	2, -1	**3, -1**
1	NA	1, PR_1	1, -1	2, -1	2, -1	3, -1
2	NA	NA	2, PR_1	**3, PR$_3$**	2, -1	3, -1
3	NA	NA	NA	3, PR_1	4, -1	4, -1
4	NA	NA	NA	NA	4, PR_1	**5, PR$_1$**
5	NA	NA	NA	NA	NA	5, PR_1

Table 1: Partial map matrix

4.2 Merging of shared memories

Partial map computed in Equation 6, produces a solution in which a separate instance of shared memory module is created for each queue not mapped in any local memory. This results in a costlier interconnection network. Hence, in step 4 of Algorithm 1, we create new mappings by pairwise merging of shared memory instances in the given mapping under performance constraints. For example, Consider a merger of both shared memories shown in Figure 7. This merger can be done by putting all the queues mapped on these shared memories into one of them and deleting the other such that performance constraints are not violated. This will allow us to remove two links and simplify the interconnection network.

4.3 Algorithm Complexity

Total number of entries in partial map matrix PM are $\frac{|T|^2}{2}$. The single processor mapping (SPM in Equation 6) is evaluated for all the processors available in the component library ($|PR|$). Further, as per Equations 4 and 3, this evaluation will take $O(|T|+|Q|)$ steps. Here $|Q|$ is the total number of queues in the process network. So, complexity of the first term in Equation 6 is $O(|PR| \times (|T|+|Q|))$. Now, the number of pivots in Equation 6 are at most $|T|$. Further, evaluation of a solution at any pivot can be done in $O(|Q|)$ because in equation 5, at most $|Q|$ queues need to be checked. Hence, complexity of the second term in Equation 6, is $O(|T| \times |Q|)$. Therefore, the overall time complexity of computing partial map is $O(|T|^2 \times (|PR| \times (|T|+|Q|) + |T| \times |Q|))$. Since value of $|PR|$ is small, complexity of computing partial map is bounded by $O(|T|^3 \times |Q|)$.

Now, number of pairs at any stage of shared memory merging is no more than $|Q|^2$ and there cannot be more than $|Q|$ merges. Further, checking performance constraints as per Equation 3 is bounded by $O(|T|+|Q|)$. So time complexity of shared memory merging stage is $O((|T|+|Q|) \times |Q|^3)$. Hence, Algorithm 1 has complexity of $O(|T|^3 \times |Q| + |T| \times |Q|^3 + |Q|^4)$ which becomes $O(|Q|^4)$ as typically $|T| \leq |Q|$.

5. EXPERIMENTAL RESULTS

We performed two sets of experiments, first on real life application MPEG-2 decoder and then on a set of randomly generated process networks. We implemented a multithreaded library using which an application written in C can be modeled as KPN. We converted the sequential code of MPEG2 decoder into the KPN as shown in Figure 1. Process *Header extraction* finds out header information of the video sequence. Process *Slice decoding* performs variable length decoding at slices. Similarly functionality of processes *IDCT, Prediction, Addition* and *Storing* are obvious from their names. These four processes work on macroblocks.

First part of Table 2 gives other process parameters of MPEG-2 decoder KPN. These parameters were obtained by using the procedure described in [17] for ARM7TDMI [18] and LEON [19] processors. Second part of Table 2 show processor parameters. Each processor type (PR_k) has associated cost (pr_cost_k), context switch overhead on processor ($contxt_k$), its frequency ($freq_k$) and cost of mapping unit size of the queue onto local memory ($cost_base_lm_k$). Processors PR_0 and PR_1 are taken as different implementations of LEON. Similarly, PR_2 is an implementation of ARM7TDMI. Apart from this, presence of one type of memory unit was assumed with bandwidth of 5,000,000 bytes/second and base cost of 0.6 units. Link cost $cost_in$ was taken to be 5,000 units. Corresponding synthesis result is shown in Figure 8. We note that it is a heterogeneous architecture consisting of three instances of processor type PR_1 and single instance of PR_2. Moreover, one shared memory was found to be sufficient.

Process parameters						
Number of cycles taken by a process T_i per iteration on processor PR_k for $\forall k$						
k	ncy_{0k}	ncy_{1k}	ncy_{2k}	ncy_{3k}	ncy_{4k}	ncy_{5k}
0	666714	55969	32795	40087	32940	10036
1	666714	55969	32795	40087	32940	10036
2	978350	75550	44096	44087	46742	20449

Processor parameters				
PR_k	$contxt_k$	pr_cost_k	$freq_k$	$cost_base_lm_k$
PR_0	200	200000	150e+06	0.5
PR_1	200	400000	200e+06	0.5
PR_2	100	170000	133e+06	0.5

Table 2: Process and processor parameters

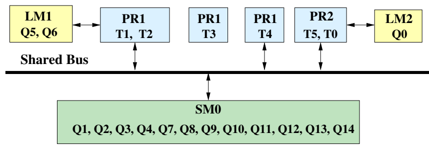

Figure 8: Synthesized architecture and mapping for MPEG2 video decoder

We also performed a number of experiments using a Random Process Network Generator (RPNG) [20]. Given the process network parameters, RPNG generates cyclic multigraphs as process networks alongwith its deadlock free code. Table 3 shows some of the results using RPNG. The third row in this table gives time taken to reach at 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 architecture of Figure 3. We did these experiments on a workstation having Intel XEON [21] CPU running at 2.20GHz. It can be observed that the solutions for all these process networks took less than one second. This makes our approach suitable to be used in a design space exploration framework.

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 3: Experiments using RPNG

Apart from doing experiments as explained above, we also compared quality of solution given by the approach presented in this paper against solution provided by the mixed integer liner programming (MILP) formulation of the same problem [16]. We did these experiments on process networks having number of processes < 8 and number of queues < 15. This is because for larger problem sizes, MILP solutions took unacceptable amount of time. We observed that for these cases, the solution was on an average $10-15\%$ poorer with the worst case being 20%.

6. CONCLUSIONS

In this paper, we presented an approach for automatic synthesis of multiprocessor SoC architectures for applications modeled as process networks. We make use of adjacency relationship of processes in the process network and evaluate different architecture alternatives using a dynamic programming based algorithm. Our approach is flexible and can be easily extended for synthesis for low energy or for multiobjective optimizations. We performed experiments on real benchmarks and also on a number of randomly generated process network. Results show that our approach is fast as well as effective and suitable for design space exploration.

7. REFERENCES

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

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

[3] Todor Stefanov and Ed Deprettere. Deriving Process Networks from Weakly Dynamic Applications in System-Level Design. In *Proc. Int. Conf. on HW/SW Codesign and System Synthesis (CODES+ISSS)*, October 2003.

[4] Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha. COSYN: Hardware Software Cosynthesis of Heterogeneous Distributed Embedded Systems. *IEEE Trans. on VLSI Systems*, 7(1):92–104, March 1999.

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

[6] Jiong Luo and Niraj K. Jha. Low Power Distributed Embedded Systems: Dynamic Voltage Scaling and Synthesis. In *Proc. International Conference on High Performance Computing (HiPC)*, December 2002.

[7] Dongkun Shin and Jihong Kim. Power-Aware Scheduling of Conditional Task Graphs in Real-Time Multiprocessor Systems. In *Proc. International Symposium on Low Power Electronics and Design (ISLPED)*, August 2003.

[8] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.

[9] Andy D. Pimentel, Louis O. Hertzberger, Paul Lieverse, Pieter van der Wolf, and Ed F. Deprettere. Exploring embedded systems architectures with artemis. *IEEE Computer*, (11):57–63, November 2001.

[10] E. A. de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proc. Int. Symposium on System Synthesis (ISSS)*, October 2002.

[11] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, EECS Dept. Berkeley, 1995.

[12] Twan Basten and Jan Hoogerbrugge. Efficient Execution of Process Networks. In *Communicating Process Architectures*, 2001.

[13] J. A. Leijten, J. L. van Meerbergen, A. A. Timmer, and J. A. G. Jess. PROPHID: A Heterogeneous Multi-Processor Architecture for Multimedia. In *Proc. International Conference on Computer Design (ICCD)*, 1997.

[14] J. Leijten, J. van Meerbergen, A. Timmer, and J. Jess. Stream communication between real-time tasks in a high-performance multiprocessor. In *Proc. Design, Automation and Test in Europe (DATE)*, 1998.

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

[16] Basant K. Dwivedi, Anshul Kumar, and M. Balakrishnan. Synthesis of Application Specific Multiprocessor Architectures for Process Networks . Technical report, Dept. of Computer Science & Engg., Indian Institute of Technology Delhi, 2003.

[17] Manoj Kumar Jain, M. Balakrishnan, and Anshul Kumar. An Efficient Technique for Exploring Register File Size in ASIP Synthesis. In *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, October 2002.

[18] http://www.arm.com.

[19] http://www.gaisler.com/leon.html.

[20] Basant Kumar Dwivedi, Harsh Dhand, M. Balakrishnan, and Anshul Kumar. RPNG: A Tool for Random Process Network Generation. Technical report, Dept. of Computer Science & Engg., Indian Institute of Technology Delhi, 2004.

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