A Two-step Genetic Algorithm for Mapping Task Graphs to a Network on Chip Architecture

Tang Lei Shashi Kumar
Department of Electronic/Computer Engineering,
School of Engineering, Jönköping University, Jönköping, Sweden
tang_lei_@sohu.com Shashi.Kumar@ing.hj.se

Abstract

Network on Chip (NoC) is a new paradigm for designing core based System on Chip which supports high degree of reusability and is scalable. In this paper we describe an efficient two-step genetic algorithm that has been used to build a tool for mapping an application, described by a parameterized task graph, on to a NoC architecture with a two dimensional mesh of switches as a communication backbone. The computational resources in NoC consists of a set of heterogenous IP cores. Our algorithm finds a mapping of the vertices of the task graph to available cores so that the overall execution time of the task graph is minimized. We have developed a NoC architecture specific communication delay model to estimate the execution time. Our algorithm is able to handle large task graphs and provide near optimal mapping in a few minutes on a PC platform. Our tool also provides facilities for specifying NoC architecture, generation and viewing synthetic task graphs and viewing the progress of the genetic algorithm as it converges to a solution.

1. Introduction

It is predicted that Moore's law about developments in semiconductor technology will continue to be valid for at least another ten years. In a few years, it will become possible for designers to integrate more than fifty microprocessors or different type IP cores on a single chip. It is a challenge to the VLSI CAD community to provide methodologies and tools using this large silicon capacity. Even bigger challenge is to achieve design of such complex systems on chip (SoCs) in a short design period and do it correct the first time. Reuse of cores and structured interconnection architectures will be the keys features of any successful future design methodology. Network on Chip (NoC) is a new paradigm for designing such future SoCs [1,2,3]. In the NoC paradigm a router-based network is used for packet switched on-chip communication among cores [4,5,6]. A typical NoC architecture will provide a scalable communication infrastructure for interconnecting cores. Since the communication infrastructure as well as the cores from one design can be easily reused for a new product, NoC provides maximum possibility for reusability. NoC also supports Globally Asynchronous Locally Synchronous (GALS) style of implementing physically large chips. Each core could be implemented as a separate clock domain and different cores could communicate among themselves using asynchronous communication through switches.

NoC provides a structured solution to interface synthesis problem in IP cores based SoC design and has the following advantages over the current approaches:

- The interconnection problem is greatly simplified therefore the designer can concentrate more on functionality and performance.

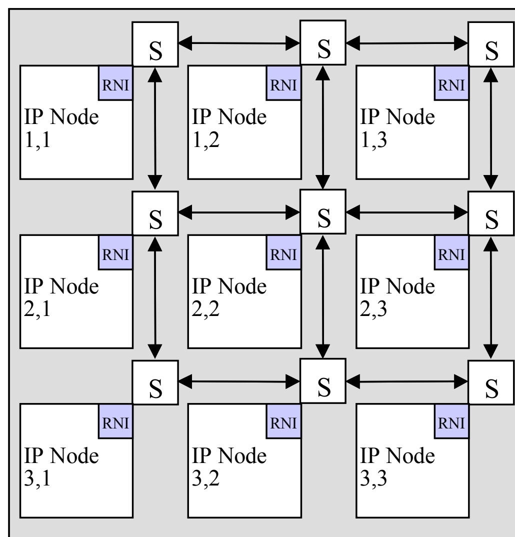

Figure1. **Structure of a 3*3 NoC [6]**

- Since the network can be mapped on the silicon surface in a controlled manner, the problem and errors due to long wires and global wires can be better managed. Some of NoC architectures recommend physical level-architectural level design integration [6]. In this proposal the switches are laid

out as a regular two-dimensional mesh. The IP cores are placed in the slots made by the switches.

- Most of the NoC architectures propose layered communication protocols for communication among cores. These are, generally, adapted from OSI reference model. This layered approach can help managing sequencing and synchronization problems.

The major disadvantage of the NoC approach is possible increase in the delay in communication. Figure 1 shows architecture and layout of a typical chip designed using NoC paradigm. Various resources (IP cores) are connected to the network of switches using a Resource Network Interface (RNI) [6]. Many EDA tools will be required for designing a system based on NoC approach [7]. An application or a set of applications for which a new SoC is required will be specified in some sort of a Task Graph (TG). An important tool will map an application specified as a TG onto a given or a fixed NoC chip.

In this paper, we describe algorithms and a tool to map a TG on a NoC chip with a mesh topology and populated with a known set of IP cores. Rest of the paper is organized as follows. In the next section, we discuss a typical design methodology and list steps required to design a SoC based on a specific NoC architecture. In section 3, we describe a mathematical model for NoC delay. Based on this model, in section 4 we present our two-step genetic algorithm for vertex mapping to IP cores in NoC. In section 5, we briefly describe a vertex mapping tool based on our algorithm. In section 6, we discuss some experiments and results using our tool.

2. Overview of a NoC design methodology

The following steps are required to implement a new application specified as a TG on a NoC architecture.

1. Decide the size of NoC communication backbone
2. Decide the communication protocols
3. Develop a set of basic communication services for resources to communicate among each other.
4. Allocate the IP cores to various available resource slots
5. Find a binding of each vertex of TG to a specific resource node in NoC architecture
6. Program/configure each resource according to the allocated set of tasks using a set of communication services.

In this paper, we concentrate on step 5. We assume that the topology, size, and resources of NoC have already been fixed.

Various options are available for mapping a TG corresponding to an application, or corresponding to a set of concurrent applications, on a NoC architecture in which communication backbone and resources (IP cores) have already been fixed. We assume that tasks are atomic and cannot be broken into smaller tasks. A mapping is called **static** if the resource on which it is going to be executed is decided before its execution and is not changed thereafter. A mapping is called **dynamic** if the placement of task could be changed during execution of the application. Obviously, a dynamic mapping could lead to higher performance and/or could help the system to have other nice properties like fault tolerance and lower power consumption. **Dynamic** mapping also leads to extra overheads, is more complex and difficult to test.

In this paper, we deal with static mapping problem. In concrete terms we discuss design and implementation of a TG mapping algorithm that maps any TG on a NoC architecture similar to described in Figure 1 with the objective function to minimize overall execution time. The inputs of the algorithm includes:

1) A fixed $X \times Y$ node NoC backbone, where every node is a fixed IP resource chosen from a library of IP cores with totally b types of IPs ($1 \le b \le X \times Y$)
2) A TG with m vertices' and n edges. Every vertex V_i executes a function f_i from a finite set of functions: $F = \{f_i : 0 \le i < m\}$; every edge E_j has its data-width: w_j ($0 \le j < n$), which is the amount of data transferring between the two vertices connected by this edge when a source vertex executes the corresponding function.

We further make the following assumptions about the architecture/TG:

- Each IP core can usually achieve many different vertex functions from the set F with different performances, i.e., with different execution delays. We say the execution time of a function f_i that cannot be executed by IP_j is ∞. It is also assumed that for each vertex function, there exists at least one IP core in NoC that is able to achieve it.
- Each IP core IP_k has an input data width I_k and output data width O_k.. This implies that at a physical level the core IP_k can receive data in units of I_k bits or send data in units of O_k bits.
- It is assumed that a NoC will have only a few types of IP cores. Based on this assumption we define $Group_k$ ($0 \le k < b$) as follow:

$$Group_k = \{\text{the subclass of all those nodes with type: } IP_k\} \quad (0 \le k < b) \qquad ①$$

The objective function for the algorithm is to find a mapping and a schedule for every TG vertex on a NoC node such that the overall execution time of the TG is minimized.

3. NoC delay modeling

As mentioned above, execution time of the TG is the

most important parameter of NoC to be optimized. In this section, we present a mathematical model for estimating delays in the NoC architecture.

3.1. System delay

The system delay of a given TG is the delay of the critical path in its execution and can be represented as follows:

$$T_{system} = T_{critical-path} = \sum_{V_i \in critical-path} Tv_i + \sum_{E_j \in critical-path} Te_j \quad ②$$

Where Tv_i is the delay of vertex V_i that belongs to TG's critical-path, Te_j is the delay of edge E_j that belongs to TG's critical-path. The vertex delay corresponds to the execution time of the function corresponding to the vertex. Edge delay corresponds to the time to communicate results of source vertex to the destination vertex corresponding to the directed edge. Since there are two factors contributing to the delay in a TG, a transformation is performed on the original "vertex based" TG to produce a new "vertex & edge based" enhanced TG, which is more convenient for delay calculations.

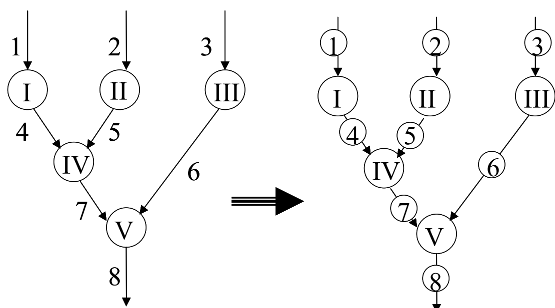

Figure 2. **A TG transformation**

Figure 2 shows an enhanced TG corresponding to an ordinary TG.

3.2. Edge delay

An edge of TG represents transmission of data from one vertex to another. Two vertices involved in communication may get mapped on to different NoC nodes. Since the communication in NoC is packet based, the delay of edge will depend on the vertex to node mapping, size of data and the network traffic situation at that time. During execution of the TG on NoC, it is possible that many edges are active concurrently and are conflicting in their use of communication resources like wires and switches. This traffic situation not only increase the communication delay of conflicting edges, but also delay the starting time of vertices and edges which have not yet been executed.

Because of the non-deterministic and concurrent nature of network traffic, it is difficult to make an exact model. The problem becomes even harder since different IP cores have different execution times when executing the same vertex, two different vertex mapping plans to nodes will induce quite different network traffic leading to different performances. This implies that we will need to adjust parameters of our model for every new mapping. In this paper we use a coarse traffic model that is not only fast to execute but is also accurate.

To get such a simple model, it is assumed that:

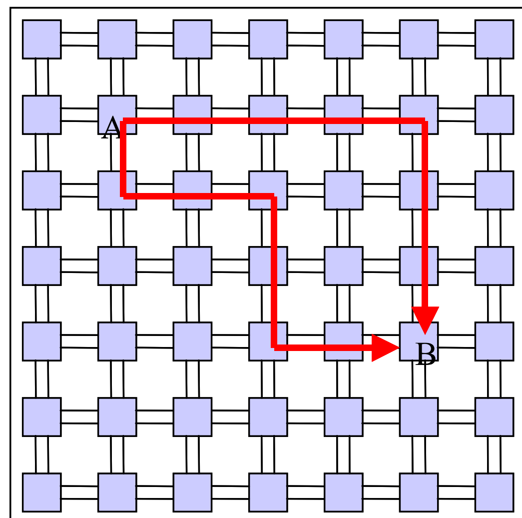

Figure 3. **Shortest paths from A to B on NoC**

1) Data always take the shortest distance in network. There often exist more than one such paths for a data going from node (x_1,y_1) to (x_2, y_2) in NoC mesh. The distance is estimated as:

$$(|x_1 - x_2| + |y_1 - y_2|) \quad ③$$

Figure 3 shows two different paths between nodes A and B with the same total length.

2) We assume that at any time a link in NoC network is only used by one TG edge. It is reasonable because:
   - Dynamic routing algorithms allow use of idle links at any time. A link will be idle for reassignment after finishing transmission of one packet.
   - A NoC switch provides high connectivity. In a mesh based network, every node can maximally receives 4 inputs, or generates 4 outputs simultaneously without requiring link sharing.
   - There exist many shortest distance paths between two nodes in a NoC network. More the distance between two nodes there will be larger number of such paths.

With the above assumptions, according to formula ③, transmission delay of an edge E_i can be simply estimated

as:

$$Te_i = k_e \cdot w_i \cdot (|x_1 - x_2| + |y_1 - y_2|)$$

Where k_e is a constant related to the properties of links and switches.

We can improve the above estimate by adding the time required to packetize the data at output nodes and depacketize at input nodes.

$$Te_i = w_i \cdot [k_O / O_{x1,y1} + k_e \cdot (|x_1 - x_2| + |y_1 - y_2|) + k_I / I_{x2,y2}] \quad ④$$

Where k_O and k_I are also constants related to switch speeds. Other parameters in formula ④ have already been defined in section 2.

Please note that usually, different vertex mapping plans to nodes will induce quiet different critical-paths in the same TG. That is the most significant difference between the performance of a TG on NoC design and the performance of aControl/Data Flow Graph on function units (+,–,×,/) based High-Level Synthesis design.

The reason of such difference is:

1) Every vertex can be executed by many different IPs with quite different performances.
2) Edge is also taken as a task. Every edge delay is affected seriously by the distance between two nodes that execute the functions of the vertices corresponding to the edge.

3.3. Average edge delay

Average Edge Delay is the average value of delay of all edges. It is a useful parameter and will be used in the algorithm later. Given the average values of data size, output and input port width (w_{ave} ,O_{ave},and I_{ave}), it is only affected by, what we call, Nodes Average Distance (NAD). NAD is defined as the average distance between two randomly selected nodes in NoC architecture. In a NoC with a X×Y mesh topology, it can be shown that:

$$NAD = \frac{X+Y}{3} \times \left(1 - \frac{1}{X \times Y}\right)$$

and the Average Edge Delay is given by:

$$Te_{ave} = w_{ave} \cdot [k_O / O_{ave} + k_e \cdot NAD + k_I / I_{ave}] \quad ⑤$$

4. Vertex mapping algorithm

Like other algorithms in the area of design automation, the algorithm of mapping a TG to a NoC architecture is a hard problem. Our attempt is to try to develop an algorithm that can give near optimal results within reasonable time, or, an algorithm with the best trade-off between result quality and computation time. Genetic algorithms have shown the potential to achieve this dual goal quite well [8,9]. We have developed a two-step genetic algorithm for our vertex mapping problem.

4.1. Basic idea

The problem is solved in two steps. In first step, we assume that all the edge delays are a constant and equal to Average Edge Delay. Based on this simplification, we decide the type of IP core for each vertex of the TG so that coarse edge delay based system delay is minimized. In the second step, using the actual edge delay based on our network traffic model, we try to further minimizing the system delay by finding an optimal binding of each TG vertex to a specific node of NoC selected from correlated IP Group (see "Group" definition in formula ①). We developed two genetic algorithms one for each of the above steps.

4.2. Patitioning: coarse edge delay based 1st genetic algorithm

Following steps are used to decide the IP core type for each vertex.

1) Generate a random population of valid solutions. Here each individual member of the population is a chromosome, which represents a concrete design plan of vertex mapping on IPs. A chromosome is constructed as follows:
   A) Every chromosome is a series of genes.
   B) Every gene represents a vertex in "vertex based" TG. Its concrete value is the IP number chosen for execution of this vertex.
2) Calculate the fitness function of every member as the reciprocal of its system delay (see formula ②). Here the Average Edge Delay formula ⑤ is used as delay of every edge.
3) Generate a better population from the old population. We use single-point-crossover method and mutation method to generate the next new population ensuring that the best individual of current population is saved for the next new population [8].
4) Go to step2 to generate a newer population repeatedly till a fixed number of loop iterations.
5) Choose some best individuals from the last population as the seeds for the 2nd algorithm.

4.3. Embedding: fine edge delay based 2nd genetic algorithm

The best seeds got from the result of the first algorithm will be used as the inputs of the 2nd genetic algorithm. Each of these seeds is actually one of the best coarse frames which tells that, in order to arrive to the best system performance, which vertex should be executed by which type of IP core group. So after choosing one seed

as a coarse frame, another genetic algorithm will be used to further decide which specific NoC node of the chosen IP type should be chosen for execution of the vertex to get the shortest edge delay.

This genetic algorithm is similar to the first algorithm except the following three points:

1) Every chromosome represents a concrete design plan of vertex mapping to NoC nodes. Every gene represents a vertex in "vertex based" TG. Its concrete value is the node number chosen from the fixed IP group decided in 1st algorithm.
2) Fitness function is also set as the reciprocal of system delay. But here the Exact Edge delay from formula ④ is used.
3) We chose only the best individual of the last population as the result.

In the above genetic algorithm, one best node allocation result is generated using one seed. Suppose that there are g seeds, then g different sub-genetic algorithms will be run in parallel. So, fine edge delay based 2nd Genetic Algorithm is actually composed of g smaller sub-genetic-algorithms. From the g locally best results derived from g different coarse frames only the best one of them will be chosen as the final solution.

4.4. Lower bound for the coarse delay

To test the effectiveness of the 1st Genetic Algorithm, a lower bound can be used assuming that every delay is equal to the average edge delay and every vertex is executed by the fastest suitable IP core.

4.5. Asynchronous ALAP and ASAP

Traditional ASAP and ALAP algorithms for graph scheduling assume that all vertices have the same delay and every edge has no delay. They can only be used for scheduling in High Level Synthesis for synchronous systems. But, these cannot be used in NoC design method, which is actually an asynchronous system. We have enhanced these algorithms in order to schedule TGs for NoC.

We use "vertex & edge based" TG for Asynchronous ALAP and Asynchronous ASAP scheduling methods where every delay of vertex and edge is exactly calculated and taken into consideration of vertex scheduling. Using Asynchronous ALAP and Asynchronous ASAP scheduling results will ensure that every vertex start time is between its ASAP and ALAP time. The critical path and the system delay (execution time) are also obtained.

4.6. Complexity analysis and validity of approach

4.6.1. Why two step approach? It is possible to solve vertex mapping problem in one step by using the Exact Edge Delay instead of Average Edge Delay when calculating the fitness function in the 1st Genetic Algorithm. But this algorithm will be extremely slow since in this algorithm every vertex gene must choose a value from the range of all nodes in NoC.

Suppose on average the number of nodes of each IP core type is a and there are b IP core types, then the total node amount is $X \times Y = a \times b$. Since the length of chromosome is m (m is the vertex amount defined in section 2), the search space of this genetic algorithm will be: $(a \times b)^m$, while our two-step Genetic Algorithm's search space will be of the size:

$$b^m + g \times a^m = b^m + g \times \frac{(X \times Y)^m}{b^m} \qquad ⑥$$

Since $(a \times b)^m >> b^m + g \times a^m$, the one-step Genetic Algorithm will either use too much time to run or may not get good results in reasonable time.

Formula ⑥ also tells that, when g is not very large, the shortest computation time with the same good result will be arrived when:

$$b \approx \sqrt{X \times Y} \qquad ⑦$$

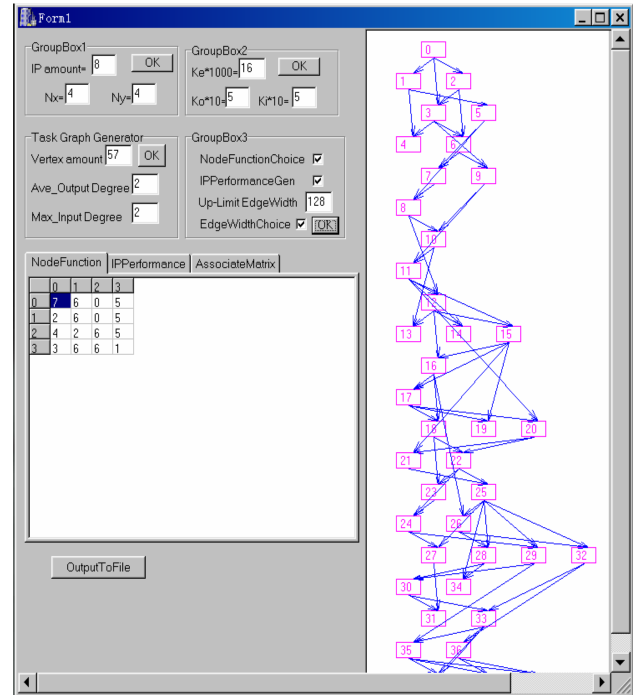

Figure 4. **Input data generation tool**

4.6.2. Validity. In the Two-Step Genetic Algorithm, we deal with vertex delay during in the 1st algorithm and deal with edge delay during the 2nd algorithm. This method should be under the assumption that the vertex delay is larger and more important than the edge delay. We have empirical evidence that with this assumption we can get very close to optimal results in reasonable computation

time. More concretely, in the cases when total edge delay is less than 1/4 of the total system critical-path delay (or when the average vertex delay is at least 3 times as the average edge delay) two-step Genetic Algorithm seems to give very good results. Larger edge delays can be permitted by using more seeds from 1st algorithm to 2nd algorithm.

5. Implementation of mapping tool

We have implemented two tools related to TG mapping to NoC architecture on Pentium III under Window 98 system. They are:

1. Input Data Generation Tool: This tool generates input for testing our mapping algorithm.
2. Task Graph Mapping Tool: This tool implements the two-step genetic algorithm. The tool has facility to specify parameters for genetic algorithm and report useful statistics and results.

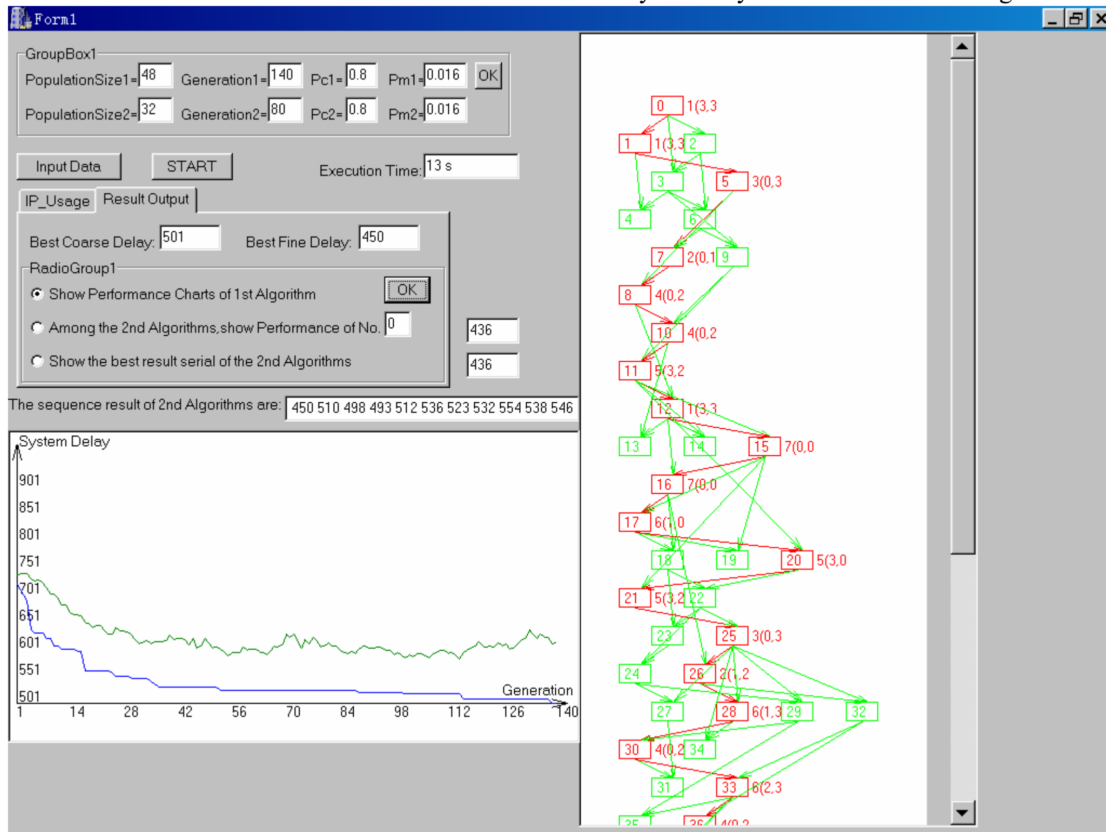

Figure 5. **Two-step genetic algorithm tool**

5.1. Input data generation tool

TGFF [10] is often used for system level design research, but no proper data generation tool has been applied for research of NoC based vertex mapping. In order to provide sufficient data for testing our idea, we developed a convenient specific tool by ourselves.

Figure 4 shows the user interface of Input Data Generation tool. As shown in the figure, a user can specify NoC size (N_x and N_y) and number of types of IP cores (IP amount) in Group Box1. Values of communication parameters (k_e, k_o and k_i) can be specified in Group Box2. Parameters related to TG, like number of vertices (up to 255), average input and output degrees of vertices can be specified in Task Graph Generator box3. The tool will generate a random graph within these constraints and display it in a separate window on the right. After TG generation, on choosing “Node-Function-Choice”, every node function in NoC will be randomly chosen from one of the IP cores available above. On choosing “IP-Performance-Gen”, delay of every IP core when executing each vertex of TG could also be randomly chosen. On choosing “Edge-Width-Choice”, the data transmission size between every pair of linked vertices via their common edge could be randomly chosen from 32 bits to the Up-limit Edge Width amount which you can set freely up to 1k bits.

At last, Output-To-File button will save all of the data available above to a TXT file for test.

5.2. Task graph mapping tool

Figure 5 shows the user interface of Two-Step Genetic Algorithm tool. Population sizes, total generation amounts, crossover probabilities, and mutation probabilities for the 1st genetic algorithm and the 2nd genetic algorithm can all be set freely in GroupBox1.

Input Data button will receive all the data in the special TXT file generated by the Input Data Generation tool. Then, the "START" button will run Two-Step Genetic Algorithm.

The best design plan with the shortest delay will be obtained after running. The shortest coarse delay from the 1st algorithm and the shortest fine delay got from the 2nd algorithm are all shown on the panel as can be seen in Figure 5. Total computation time is also shown on the right for efficiency measurement of the algorithm. Lower Bound is also shown there (for this case it is "436").

On the Radio-Group, one can display results after 1st algorithm or after the 2nd algorithm by appropriate choice. For example, if "Show Performance Chart of 1st Algorithm" is chosen, the working of the 1st genetic algorithm will be shown as curves on the chart at the bottom of the window. In this window the upper curve shows the individual average delay (execution time) of every generation; while the lower curve shows the best individual's execution time of the every generation. In the next section we will describe our experiments and results using this tool.

6. Experiments and results

We have carried out various experiments to test the efficiency of our vertex mapping algorithm and the developed tool. Table 1 shows some typical results. Columns in this table represent various experiments. Rows are divided into two categories. The top eight rows specify various input parameters and bottom five rows show the results. The input data of the first experiment is taken as standard. For the other experiments, only changes from the first experiments are listed, i.e., blank entry means the same value as in the first experiment.

We make the following observations while referring to Table 1.

1) In the 1st experiment, the shortest coarse delay equals to its Lower Bound, so it must be the best result. (IN fact, we have checked that the generated mapping plan is different from the Lower Bound mapping plan.)
2) The results demonstrate that algorithm works well for TG size from 17 vertices to 163 vertices. The computation varies from several seconds to a few minutes. Computation time increases with size of the vertex graph, which can be observed by noting computation times of experiment 1 and 7.
3) As expected, the computation time of the first experiment is smaller than the experiment 2 because number of IP types in experiment 1 is closer to optimal (according to formula ⑦, the shortest computation time will be achieved when $b \approx \sqrt{X \times Y}$).
4) In the 4th experiment, average input and output degrees are doubled. Computation time is twice as the 1st experiment. It is because that when genetic algorithm tries to count every individual's critical-path delay, the time of searching critical-path will be doubled when vertex's average input and output amount is doubled.
5) Shown as 8th experiment, gap between the best coarse delay and the best fine delay is much larger than the 1st experiment. The reason is that the average edge delay doubles due to Top-Edge-Width's doubling. (Comparison of the

Table 1. **Experimental results**

	Experiment No.	1	2	3	4	5	6	7	8	9	10	11	12
Input	IP-No	5	15	15						15	15	15	15
	Nx*Ny	4*4								8*6	8*6	8*6	8*6
	Vertex-Number	56						17			163	163	163
	Vertex Output Degree	2			4	4	4						
	Vertex Input Degree	2			4	4	4						
	Top-Edge-Width	128							256				256
	Population Size	32				64						64	64
	Generation Number	120		250			250					250	250
Result	Computation Time (s)	6	11	23	11	19	21	3	6	11	99	424	537
	Best Coarse Delay	552	411	346	613	581	586	175	690	510	1905	1805	2283
	Lower-Bound	552	290	290	563	563	563	175	670	405	1045	1045	1539
	Best Fine Delay	485	369	323	569	536	542	162	536	408	1690	1614	1781
	Critical-Path Consistency	Top 4	Top 1	Top 3	Top 8	Top 1	Top 1	Top 4	Top 5	Top 1	Top 1	Top 1	Top 1

12th to the 11th experiment shows the same.)

6) Comparison between the 9th and 2nd experiment shows that the gap between the best coarse delay and the best fine delay is larger. The reason for this is that the average edge delay is larger due to large size of NoC.
7) In all the experiments, only very few of the top local best designs among all the 16 local best designs (here *g* is chosen as 16) can share the same critical path. So usually different vertex mapping plans to nodes will induce different critical-path in a same TG.

7. Conclusion

NoC research is still in its infancy. In the coming years, we will see a lot of research effort to develop tools for SoC design using NoC approach. Our paper is an effort in this direction. In this paper we have presented a two-step genetic algorithm to map a TG to a specific NoC architecture with heterogeneous set of IP cores as resources. The key contributions of the paper are as follows:

- A simple but reasonable delay model is presented to estimate delay of messages from one NoC node to another node.
- An enhanced TG is proposed to model both task computation time and task communication time.
- The idea of Asynchronous ALAP and Asynchronous ASAP scheduling is proposed for scheduling vertices of TG on nodes of NoC and accurately finding the critical-path.
- An efficient two-step genetic algorithm is described which produce near optimal vertex mapping plan for NoC minimizing overall TG's execution time.
- A software tool set is being developed for research in NoC approach based SoC design. The paper describes successful development of two tools in this direction. The first tool is useful for generating all necessary input data for testing including parameterized TGs. The second tool implements the two-step genetic algorithm of mapping TG on a fixed NoC architecture.

The ideas described in this paper can be easily extended for other NoC architecture. There is a lot of scope to improve the algorithm. One of the improvements will be to remove the assumption that the execution time of the TG vertices is much larger than that of the communication delay due the edges. There is also scope to make our delay model even more accurate by including some parameters related to NoC switch design (such as buffer size, routing algorithms etc.).

8. References

[1] Luca Benini, and Giovanni De Micheli, "Network on Chips: A New SoC Paradigm", *IEEE Computer*, January 2002, pp. 70-78.

[2] Axel Jantsch, and Hannu Tenhunen (Editors), *Networks on Chip*, Kluwer Academic Publishers, Boston, USA, January 2003.

[3] D. Wingard, "Micronetwork Based Integration for SoCs", *Proceedings of the 38th Design Automation Conference*, ACM/IEEE, Las Vegas, Nevada, USA, June 2001, pp. 673-677.

[4] William J. Dally, and Brian Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks", *Proceedings of the 38th Design Automation Conference*, ACM/IEEE, Las Vegas, Nevada, USA, June 2001, pp. 684-689.

[5] Edwin Rijpkema, Kees Goossens, and Paul Wielage, "A Router Architecture for Networks on Silicon", *Proceedings of Progress 2001, 2nd Workshop on Embedded Systems*, October, 2001.

[6] Shashi Kumar et. al., "A Network on Chip Architecture and Design Methodology", *IEEE Computer Society Annual Symposium on VLSI*, Pittsburgh, Pennsylvania, USA, April 2002, pp. 117-124.

[7] Juha-Pekka Soininen et. al, "Extending Platform Based Designs to Network on Chip Systems", *Proceedings of the 16th International Conference on VLSI Design 2003*, IEEE, New Delhi, India, January 2003, pp. 401-408.

[8] John H. Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, Cambridge, MA, second edition, 1992.

[9] D.Saha, R.S.Mitra, and A.Basu, "Hardware Software Partitioning using Genetic Algorithm", *10th International Conference on VLSI Design*, IEEE, Hyderabad, Inidia, 1997, pp. 155-160.

[10] Robert P. Dick, David L. Rhodes, Wayne Wolf, "TGFF: Task Graphs for Free", *Proceedings of the 6th International Workshop on Hardware/software Co-design*, Seattle, Washington, USA, March 1998, pp.97-101.