Chapter 10
Application Mapping in Network-on-Chip Using Evolutionary Multi-objective Optimization*

Abstract. Network-on-chip (NoC) are considered the next generation of communication infrastructure, which will be omnipresent in most of industry, office and personal electronic systems. In the platform-based methodology, an application is implemented by a set of collaborating intellectual properties (IPs) blocks. In this chapter, we use multi-objective evolutionary optimization to address the problem of mapping topologically pre-selected sets IPs, which constitute the set of optimal solutions that were found for the IP assignment problem, on the tiles of a mesh-based NoC. The IP mapping optimization is driven by the area occupied, execution time and power consumption.

10.1 Introduction

As the integration rate of semiconductors increases, more complex cores for *system-on-chip* (SoC) are launched. A simple SoC is formed by homogeneous or heterogeneous independent components while a complex SoC is formed by interconnected heterogeneous components. The interconnection and communication of these components form a *network-on-chip* (NoC). A NoC is similar to a general network but with limited resources, area and power. Each component of a NoC is designed as an *intellectual property* (IP) block. An IP block can be of general or special purpose such as processors, memories and DSPs [4].

Normally, a NoC is designed to run a specific application. This application, usually, consists of a limited number of tasks that are implemented by a set of IP blocks. Different applications may have a similar, or even the same, set of tasks. An IP block can implement more than a single task of the application. For instance, a processor IP block can execute many tasks as a general processor does but a multiplier IP block for floating point numbers can only multiply floating point numbers. The number of IP blocks designers, as well as the number of available IP blocks, is growing up fast.

In order to yield an efficient NoC-based design for a given application, it is necessary to choose the adequate minimal set of IP blocks. With the increase of IP

* This chapter was developed in collaboration with Marcus Vinícius Carvalho da Silva.

N. Nedjah and L. de Macedo Mourelle, *Hardware for Soft Computing and Soft Computing for Hardware*, Studies in Computational Intelligence 529,
DOI: 10.1007/978-3-319-03110-1_10,

blocks available, this task is becoming harder and harder. Besides IP blocks carefully assignment, it is also necessary to map the blocks onto the NoC available infra-structure, which consists of a set of *cores* communicating through *switches*. A bad mapping can degrade the NoC performance. Different optimization criteria can be pursued depending on how much information details is available about the application and IP blocks.

Usually, the application is viewed as a graph of tasks called *task graph* (TG). The IP blocks features can be obtained from their companion documentation. The IP assignment and IP mapping are key research problems for efficient NoC-based designs. These two problems are *NP*-hard problems and can be solved using multi-objective optimizations.

In this chapter, we propose a multi-objective evolutionary-based decision support system to help NoC designers. For this purpose, we propose a structured representation of the TG and an IP repository that will feed data into the system. We use the data available in the Embedded Systems Synthesis benchmarks Suite (E3S) [2] as our IP repository. The E3S is a collection of TGs, representing real applications based on embedded processors from the Embedded Microprocessor Benchmark Consortium (EEMBC). It was developed to be used in system-level allocation, assignment, and scheduling research. We used the NSGA-II, which is an efficient multiobjective algorithm that uses Pareto dominance as a selection criterion [1]. The algorithm was modified according to some prescribed NoC design constraints.

The rest of the chapter is organized as follows: First, in Section 10.2, we present briefly some related research work. Then, in Section 10.3, we introduce an overview of NoC structure. Subsequently, in Section 10.4, we describe a structured TG and IP repository model based on the E3S data. After that, in Section 10.5.1, we introduce the mapping problem in NoC-based environments. Then, in Section 10.5, we sketch the NSGA-II algorithm used in this work, individual representations and objective functions for the optimization stage. Later, in Section 10.7, we show some experimental result yield. Last but not least, in Section 10.8, we draw some conclusions and outline new directions for future work.

10.2 Related Work

The problems of mapping IP blocks into a NoC physical structure have been addressed in some previous studies. Some of these works did not take into account of the multi-objective nature of these problems and adopted a single objective optimization approach. Hu and Marculescu [4] proposed a branch and bound algorithm which automatically maps IPs/cores into a mesh based NoC architecture that minimizes the total amount of consumed power by minimizing the total communication among the used cores. Lei and Kumar [7] proposed a two step genetic algorithm for mapping the TG into a mesh based NoC architecture that minimizes the execution time. In the first step, they assumed that all communication delays are the same and selected IP blocks based on the computation delay imposed by the IPs only. In the second step, they used real communication delays.

Murali and De Micheli [8] addressed the problem under the bandwidth constraint with the aim of minimizing communication delay by exploiting the possibility of splitting traffic among various paths. Zhou et al. [10] proposed a multi-objective exploration approach, treating the mapping problem as a two conflicting objective optimization problem that attempts to minimize the average number of hops and achieve a thermal balance. Jena and Sharma [5] addressed the problem of topological mapping of IPs/cores into a mesh-based NoC in two systematic steps using the NSGA-II [1]. The main objective was to obtain a solution that minimizes the energy consumption due to both computational and communicational activities and also minimizes the link bandwidth requirement under some prescribed performance constraints.

10.3 NoC Internal Structure

A NoC platform consisting of architecture and design methodology, which scales from a few dozens to several hundreds or even thousands of resources [6]. As mentioned before, a resource may be a processor core, DSP core, an FPGA block, a dedicated hardware block, mixed signal block, memory block of any kind such as RAM, ROM or CAM or even a combination of these blocks.

A NoC consists of set of *resources* (R) and *switches* (S). Resources and switches are connected by *links*. The pair (R,S) forms a *tile*. The simplest way to connect the available resources and switches is arranging them as a mesh so these are able to communicate with each other by sending messages via an available path. A switch is able to buffer and route messages between resources. Each switch is connected to up to four other neighboring switches through input and output channels. While a channel is sending data another channel can buffer incoming data. Fig. 10.1 shows the architecture of a mesh-based NoC where each resource contains one or more IP blocks (RNI for resource network interface, D for DSP, M for memory, C for cache, P for processor, FP for floating-point unit and Re for reconfigurable block). Besides the mesh topology, there are more complex topologies like *torus*, *hypercube*, 3-*stage clos* and *butterfly*. Note that every resource in the NoC must have an unique identifier and is connected to the network via a switch. It communicates with the switch through the available RNI. Thus, any set of IP blocks can be plugged into the network if its footprint fits into an available resource and if this resource is equipped with an adequate RNI.

10.4 Task Graph and IP Repository Models

In order to formulate the IP mapping problem, it is necessary to introduce a formal definition of an application first. An application can be viewed as a set of tasks that can be executed sequentially or in parallel. It can be represented by a directed graph of tasks, called *task graph*. A *Task Graph* (TG) $G = G(T,D)$ is a directed graph where each node represents a computational module in the application referred to as task $a_i \in T$. Each directed arc $d_{i,j} \in D$, between tasks a_i and a_j, characterizes either

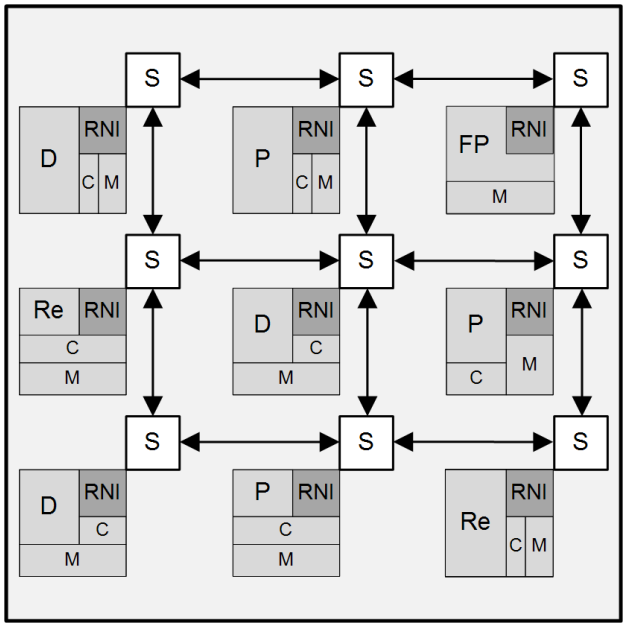


Fig. 10.1 Mesh-based NoC with 9 resources

data or control dependencies. Each task a_i is annotated with relevant information, such as a unique identifier and type of processing element (PE) in the network. Each $d_{i,j}$ is associated with a value $V(d_{i,j})$, which represents the volume of bits exchanged during the communication between tasks a_i and a_j. Once the IP assignment has been completed, each task is associated with an IP identifier. The result of this stage is a graph of IPs representing the PEs responsible of executing the application.

An *Application Characterization Graph* (APG) $G = G(C,A)$ is a directed graph, where each vertex $c_i \in C$ represents a selected IP/core and each directed arc $a_{i,j}$ characterizes the communication process from core c_i to core c_j. Each $a_{i,j}$ can be tagged with IP/application specific information, such as communication rate, communication bandwidth or a weight representing communication cost. A TG is based on application features only while an APG is based on application and IP features, providing us with a much more realistic representation of the an application in runtime on a NoC. In order to be able to bind application and IP features, at least one common feature is required in both of the IP and TG models.

The E3S (0.9) Benchmark Suite [2] contains the characteristics of 17 embedded processors. These processors are characterized by the measured execution times of 47 different type of tasks, power consumption derived from processor data sheets, and additional information, such as die size, price, clock frequency and power consumption during idle state. In addition, E3S contains task graphs of common tasks in auto-industry, networking, telecommunication and office automation. Each one

of the nodes of these task graphs is associated with a task type. A task type is a processor instruction or a set of instructions, e.g., FFT, inverse FFT, floating point operation, OSPF/Dijkstra [3], etc. If a given processor is able to execute a given type of instruction, so that processor is a candidate to receive a resource in the NoC structure and would be responsible for the execution of one or more tasks.

Here, we represent TGs using XML code. A TG is divided in three major elements: *taskGraph*, *nodes* and *edges*. Each *node* has two main attributes: an unique identifier (*id*) and a task type (*type*), chosen among the 47 different types of tasks present in the E3S. Each *edge* has four main attributes: an unique identifier (*id*), the *id* of its source node (*src*), the *id* of its target node (*tgt*) and an attribute representing the communication cost imposed (*cost*).

The IP repository is divided into two major elements: the *repository* and the *ips* elements. The *repository* is the IP repository itself. Recall that the repository contains different non general purpose embedded processors and each processor implements up to 47 different types of operations. Not all 47 different types of operations are available in all processors. Each type of operation available in each processor is represented by an *ip* element. Each *ip* is identified by its attribute *id*, which is unique, and by other attributes such as *taskType*, *taskName*, *taskPower*, *taskTime*, *processorID*, *processorName*, *processorWidth*, *processorHeight*, *processorClock*, *processorIdlePower* and *cost*. The common element in TG and IP repository representations is the *type* attribute. Therefore, this element will be used to bind an *ip* to a *node*. The repository contains IPs for digital signal processing, matrix operations, text processing and image manipulation.

These simplified and well-structured representations are easily intelligible, improve information processing and can be universally shared among different NoC design tools.

10.5 Multi-objective Evolution

Optimization problems with *concurrent* and *collaborative* objectives are called Multi-objective Optimization Problems (MOPs). Objectives o_1 and o_2 are said to be collaborative if the optimization of o_1 leads implicitly to the optimization of o_2 while these would be said to be concurrent if the optimization of o_1 leads to the deterioration of o_2. In such problems, all *collaborative* objectives should be grouped and a single objective among those should be used in the optimization process, which achieves also the optimization of all the collaborative objectives in the group. However, concurrent objectives need all to be considered in the process. The best solution for a MOP is the solution with the adequate trade-off between all objectives.

10.5.1 The IP Mapping Problem

The platform-based design methodology for SoC encourages the reuse of components to increase reusability and to reduce the time-to-market of new designs. The designer of NoC-based systems faces two main problems: selecting the adequate

set of IPs that optimize the execution of a given application and finding the best physical mapping of these IPs into the NoC structure.

The main objective of the IP assignment stage is to select, from the IP repository, a set of IPs that minimize the NoC consumption of power, area occupied and execution time. At this stage, no information about physical allocation of IPs is available so optimization must be done based on TG and IP information only. So, the result of this step is the set of IPs that maximizes the NoC performance. The TG is then annotated and an APG is produced, wherein each node has an IP associated with it.

Given an application, described by its APG, the problem that we are concerned with in this chapter is to determine how to topologically map the selected IPs onto the network, such that the objectives of interest are optimized. Some of these objectives are: latency requirements, power consumption of communication, total area occupied and thermal behavior. At this stage, a more accurate execution time can be calculated taking into account of the distance between resources and the number of switches and links crossed by a data package along a path. The result of this process should be an optimal allocation of the one of the presecribed IP asssignments, selected in an earlier stage, to execute the application, described by the TG, on the NoC structure.

The search space for a "good" IP mapping for a given application is defined by the possible combinations of IP/tile available in the NoC structure. Assuming that the mesh-based NoC structure has $N \times N$ titles and there are at most N^2 IPs to map, we have a domain size of $N^2!$. Among the huge number of solutions, it is possible to find many equally good solutions. In huge non-continuous search space, deterministic approaches do not deal very well with MOPs. The domination concept introduced by Pareto [9] is necessary to classify solutions. In order to deal with such a big search space and trade-offs offered by different solutions in a reasonable time, a multi-objective evolutionary approach is adopted.

10.5.2 EMO Algorithm

The core of the proposed tool offers the utilization of the well-known and well-tested MOEA: NSGA-II [1]. It adopt the domination concept with a ranking schema for solution classification. The ranking process separates solutions in *Pareto fronts* where each front corresponds to a given rank. Solutions from rank *one*, which is the *Pareto-optimal* front) are equally good and better than any other solution from Pareto fronts of higher ranks.

NSGA-II features a fast and elitist ranking process that minimizes computational complexity and provides a good spread of solutions. The elitist process consists in joining parents and offspring populations and diversity is achieved using the *crowded-comparison operator* [1].

The basic work flow of the algorithm starts with a random population of individuals, where each individual represents a solution. Each individual is associated

with a rank. The selection operator is applied to select the parents. The parents pass through crossover and mutation operators to generate an offspring. A new population is created and the process is repeated until the stop criterion is satisfied.

10.5.3 Representation and Genetic Operators

The individual representation is shown in Fig. 10.2–(a). The tile indicates information on the physical location on which a gene is mapped. On a $N \times N$ regular mesh, the tiles are numbered successively from top-left to bottom-right, row by row. The row of the i^{th} tile is given by $\lceil i/N \rceil$, and the corresponding column by i mod N.

The crossover and mutation operators were adapted to the fact that the set of selected IPs can not be changed as we have to adhere to the set of prescribed IP assignments. For this purpose, we propose a crossover operator that acts like a shift register, shifting around a random crossover point and so generating a new solution, but with the same set of IPs. This behavior does not contrast with the biological inspiration of evolutionary algorithms, observing that certain species can reproduce through parthenogenesis, a process in which only one individual is necessary to generate an offspring.

The mutation operator performs an *inner swap mutation*, where each gene receives a random mutation probability, which is compared against the system mutation probability. The genes with mutation probability higher than the system's are swapped with another random gene of the same individual, instead of selecting a random IP from the repository. This way, it is possible to explore the allocation space preserving any optimization done in the IP assignment stage. The crossover and mutation strategies adopted in the IP mapping stage are represented in Fig. 10.2–(b) and Fig. 10.2–(c), respectively.

10.6 Objective Functions

During the evolutionary process, the fitness of the individuals with respect to each one of the selected objectives (i.e. *area*, *time*, and *power*) must be efficiently computed. After a through analysis of all possible design characteristics, we decided that the adequate trade-off can be achieved using only minimization functions of objectives *area*, *execution time* and *power consumption*.

10.6.1 Area

In order to compute the area required by a given mapping, it is necessary to know the area needed for the selected processors and that required by the used links and switches. As a processor can be responsible for more than one task, each APG node must be visited in order to check the processor identification in the appropriate XML element. Grouping the nodes with the same *processorID* attribute allows us to implement this verification. The total number of links and switches can be obtained

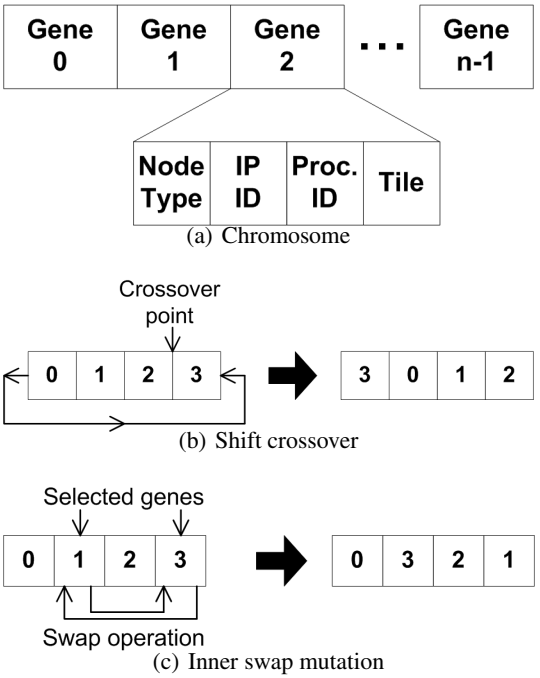


Fig. 10.2 Chromosome and application of the proposed shift crossover and inner swap mutation

through the consideration of all communication paths between exploited tiles. Note that a given IP mapping may not use all the available tiles, links and switches. Also, observe that a portion of a path may be re-used in several communication paths.

In this work, we adopted a fixed route strategy wherein data emanating from tile i is sent first horizontally to the left or right side of the corresponding switch, depending on the target tile position, say j, with respect to i in the NoC mesh, until it reachs the column of tile j, then, it is sent up or down, also depending on the position of tile j with respect to tile i until it reaches the row of the target tile. Each communication path between tiles is stored in the routing table. The number of links in the aforementioned route can be computed as described in Equation 10.1. This is also represents the distance between tiles i and j and called the *Manhattan distance* [7].

$$nLinks(i,j) = |\lceil i/N \rceil - \lceil j/N \rceil| + |i \bmod N - j \bmod N| \tag{10.1}$$

In the purpose of computing efficiently the area required by all used links and switches, an APG can be associated with a so-called *routing table* whose entries describe appropriately the links and switches necessary to reach a tile from another. The number of hops between tiles along a given path leads to the number of links between those tiles, and incrementing that number by 1 yields the number of traversed

switches. The area is computed summing up the areas required by the implementation of all distinct processors, switches and links.

Equation 10.2 describes the computation involved to obtain the total area for the implementation a given IP mapping M, wherein function $Proc(.)$ provides the set of distinct processors used in APG_M and $area_p$ is the required area for processor p, function $Links(.)$ gives the number of distinct links used in APG_M, A_l is the area of any given link and A_s is the area of any given switch.

$$Area(M) = \sum_{p \in Proc(APG_M)} area_p + (A_l + A_s) \times Links(APG_M) + A_s \qquad (10.2)$$

10.6.2 Execution Time

To compute the execution time of a given mapping, we consider the execution time of each task of the critical path, their schedule and the additional time due to data transportation through links and switches along the communication path. The critical path can be found visiting all nodes of all possible paths in the task graph and recording the largest execution time of the so-called critical path. The execution time of each task is defined by the *taskTime* attribute in TG. Links and switches can be counted using the routing table. We identified three situations that can degrade the implementation performance, increasing the execution time of the application:

1. *Parallel tasks mapped into the same tile*: A TG can be viewed as a sequence of horizontal levels, wherein tasks of the same level may be executed in parallel, allowing for a reduction of the overall execution time. When parallel tasks are assigned in the same processor, which also means that these occupy the same tile of the NoC, they cannot be executed in parallel.
2. *Parallel tasks with partially shared communication path*: When a task in a tile must send data to supposedly parallel tasks in different tiles through the same *initial* link, data to both tiles cannot be sent at the same time.
3. *Parallel tasks with common target using the same communication path*: When several tasks need to send data to a common target task, one or more shared links along the partially shared path would be needed simultaneously. The data from both tasks must then be pipelined and so will not arrive at the same time to the target task.

Equation 10.3 is computed using a recursive function that implements a depth-first search, wherein function $Paths(.)$ provides all possible paths of a given APG and $t_0(a)$ is the required time for task a. After finding the including the total execution time of the tasks that are traversed by the critical path, the time of parallel tasks executed in the same processor need to be accumulated too. This is done by function *SameProcSameLevel*(.). The delay due to data pipelining for tasks on the same level is added by *SameSourceCommonPath*(.). Last but not least, the delay due to pipelining data that are emanating at the same time from several distinct tasks yet for the same target task is accounted for by function *DiffSrcSameTgt*(.).

$$Time(M) = \max_{r \in Paths(APG_M)} \left(\sum_{a \in r} t_0(a) + \sum_{i \in \{1,2,3\}} t_i(r) \right) \tag{10.3}$$

Function t_1– *SameProcSameLevel*(.) compares tasks of a given same level that are implemented by the same processor and returns the additional delay introduced in the execution of those tasks. Algorithm 10.1 shows how function *SameProcLevel*(.), that uses information from path *r*, application task graph and its corresponding characterization graph to compute the delay in question.

Algorithm 10.1. SameProcSameLevel(r) – t_1

```
1: time := 0
2: for all a ∈ r do
3:    for all n ∈ T do
4:       if T.level(a) = TG.level(n) then
5:          if APG.processor(a) = APG.processor(n) then
6:             time := time + n.taskTime
7:          end if
8:       end if
9:    end for
10: end for
11: return time
```

Function t_2– *SameSourceCommonPath*(.) computes the additional time due to parallel tasks that have data dependencies on tasks mapped in the same source tile and yet these share a common initial link in the communication path. Algorithm 10.2 shows the details of the delay computation using information from path *r*, application task graph and its corresponding characterization graph. In that algorithm $TG.targets(a)$ yields the list of all possible target tasks of task *a*, $APG.initPath(src, tgt)$ returns the initial link of the communication path between tiles *src* and tgt and *penalty* represents a time duration needed to data to cross safely from one switch to one of its neighbors. This penalty is added every time the initial link is shared.

Function t_3– *DiffSrcSameTgt(.)* computes the additional time due to the fact that parallel tasks producing data for the same target task need to use simultaneously at least a common link along the communication path. Algorithm 10.3 shows the details of the delay computation using information from path *r*, application task graph and its corresponding characterization graph. In that algorithm, $APG.Path(src, tgt)$ is the ordered list of all links crossed from task *src* to task tgt and *penalty* has the same semantic as in the Algorithm 10.2.

10.6.3 Power Consumption

The total power consumption of an application NoC-based implementation consists of the power consumption of the processors while processing the computation

Algorithm 10.2. SameSrcCommonPath(r) – t_2

```
1: penalty := 0
2: for all a ∈ r do
3:    if TG.targets(a) > 1 then
4:       for all n ∈ TG.targets(a) do
5:          for all n' ∈ TG.targets(a) | n' ≠ n do
6:             w = APG.initPath(a,n);
7:             w' = APG.initPath(a,n');
8:             if w = w' then
9:                penalty := penalty + 1
10:            end if
11:         end for
12:      end for
13:   end if
14: end for
15: return penalty
```

Algorithm 10.3. DiffSrcSameTgt(r) – t_3

```
1: penalty := 0
2: for all a ∈ r do
3:    for all a' ∈ r | a' ≠ t do
4:       if TG.level(a) = TG.level(a') then
5:          for all n ∈ TG.targets(a) do
6:             for all n' ∈ TG.targets(a') do
7:                if n = n' then
8:                   w := APG.Path(a,n);
9:                   w' := APG.Path(a',n');
10:                  for i = 0 to min(w.length, w'.length) do
11:                     if w(i) = w'(i) then
12:                        penalty := penalty + 1
13:                     end if
14:                  end for
15:               end if
16:            end for
17:         end for
18:      end if
19:   end for
20: end for
21: return penalty
```

performed by each IP and that due to the data transportation between the tiles. The former can be computed summing up attribute *taskPower* of all nodes of the APG and the latter is the power consumption due to communication between the application tasks through links and switches. The power consumption due to the computational activity is simply obtained summing up atribute *taskPower* of all nodes in the APG and is as described in Equation 10.4.

$$Power_p(M) = \sum_{a \in APG_M} power_a \tag{10.4}$$

An energy model for one bit consumption is used to compute the total energy consumption for the whole communication involved during the execution of an application on the NoC platform. The bit energy (E_{bit}), energy consumed when a data of one bit is transported from one tile to any of its neighboring tiles, can be obtained as in Equation 10.5:

$$E_{bit} = E_{S_{bit}} + E_{L_{bit}} \tag{10.5}$$

wherein $E_{S_{bit}}$ and $E_{L_{bit}}$ represent the energy consumed by the switch and link tying the two neighboring tiles, respectively [4].

The total power consumption of sending one bit of data from tile i to tile j can be calculated considering the number of switches and links the bit passes through on its way along the path, as shown in Equation 10.6.

$$E_{bit}^{i,j} = nLinks(i,j) \times E_{L_{bit}} + (nLinks(i,j) + 1) \times E_{S_{bit}} \tag{10.6}$$

wherein function $nLinks(.)$ provides the number of traversed links (and switches too) considering the routing strategy used in this work and described earlier in this section. The function is is defined in Equation 10.1.

Recall that the application TG gives the communication volume ($V(a,a')$) in terms of number of bits sent from the task a to task a' passing through a direct arc $d_{a,a'}$. Assuming that the tasks a and a' have been mapped onto tiles i and j respectively, the communication volume of bits between tiles i and j is then $V(i,j) = V(d_{t,t'})$. The communication between tiles i and j may consist of a single link $l_{i,j}$ or by a sequence of $m > 1$ links $l_{i,x_0}, l_{x_0,x_1}, l_{x_1,x_2}, \ldots, l_{x_{m-1},j}$.

The total network communication power consumption for a given mapping M is given in Equation 10.7, wherein $Targets_a$ provides all tasks that have a direct dependency on data resulted from task a and $Tile_a$ yields the tile number into which task a is mapped.

$$Power_c(M) = \sum_{\substack{a \in APG_M, \\ \forall a' \in Targets_a}} V(d_{a,a'}) \times E_{bit}^{Tile_a, Tile_{a'}} \tag{10.7}$$

10.7 Results

First of all, the implementation of the algorithm was validated using mathematical known MOPs and the results were compared with the original results that were obtained by Deb to validate NSGA-II [1]. The simulation converged to the true Pareto-front. For NoC optimization, only the individual representation and the objective functions were changed, keeping the ranking, selection, crossover and mutation operators unchanged. Different TGs generated with TGFF [2] and from E3S, with sequential and parallel tasks, were used.

Many simulations were performed to find out the setting up of the parameters used in NSGA-II for solving the IP mapping problem. The results of these simulation allowed us to set the population size to 600, mutation probability to 0.01, crossover probability to 0.8 and tournament size to 50 and run the algorithm of 100 generations. The application, represented as a TG in Fig. 10.3, was generated with TGFF [2]. Note that this TG presents four levels of parallelism.

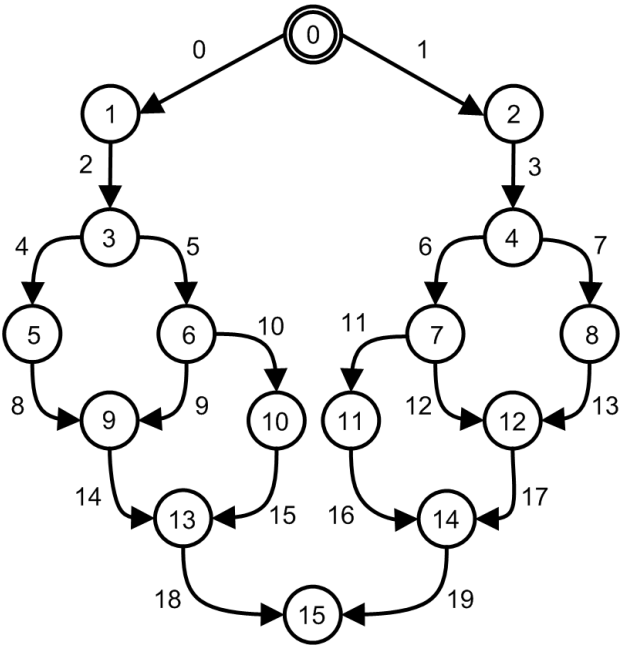
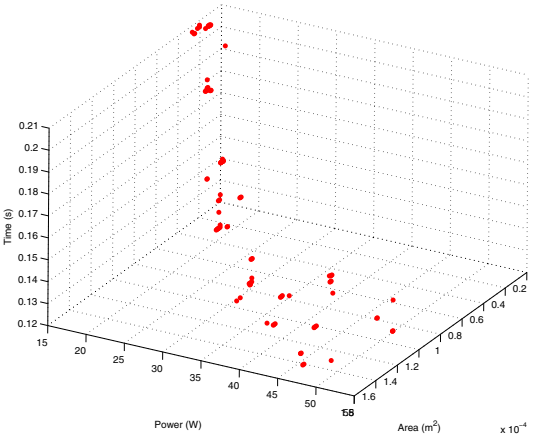


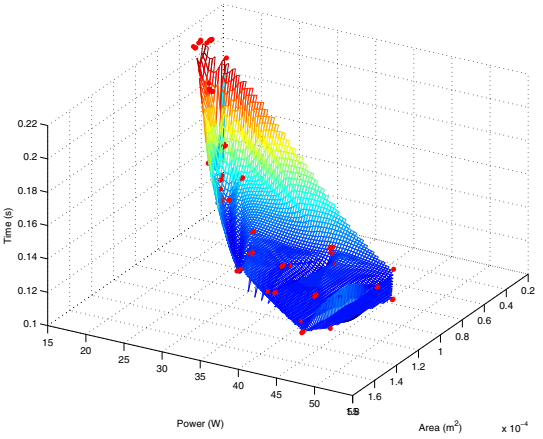
Fig. 10.3 Task graph of 5 levels of parallelism

Analyzing the results obtained from the first simulations, we observed that in order to achieve the best trade-off, the system allocated many tasks for the same processor, which reduces area and execution time but generates *hot spots* [10]. A hot spot is an area of high activity within a silicon chip. Hot spots can damage a silicon chip and increases power consumption because of Avalanche Effect. In order to avoid the formation of hot spots, a *maximum tasks per processor* constraint was imposed in the evolutionary process. This parameter is decided by the NoC designer based on some extra physical characteristics. We adopted a maximum of 2 tasks per processor. Figure 10.4–(a) shows the Pareto-front discrete points. Figure 10.4–(b) shows the Pareto-front formed by the Pareto-optimal solutions. Note that many solutions have very close objectives values. The IP assignment of the TG represented in Fig. 10.3 was able to discover 97 distinct optimal IP assignments. From those 97 distinct of IP assignments, 142 optimal mappings were generated.

Fig. 10.5–(a) represents the $time \times area$ trade-off, Fig. 10.5–(b) depicts the $power \times time$ trade-off and Fig. 10.5–(c) plots the $power \times area$ trade-off. As we



(a) Pareto-optimal solutions



(b) Pareto-front

Fig. 10.4 Pareto-optimal solutions and Pareto-front of the 142 optimal IP mappings obtained for the task graph of Fig. 10.3

can observe, comparing the dots against the line of interpolation, the trade-off between time and area and between power and time is not so linear as the trade-off between power and area. Fig. 10.5–(a) shows that solutions that require more area tend to spend less execution time because of the better distribution of the tasks allowing for more parallelism to occur. Fig. 10.5–(b) shows that solutions that spend less time of execution tend to consume more power because of IP's features, such as higher clock frequency, and physical effects like intensive inner-electrons activity.

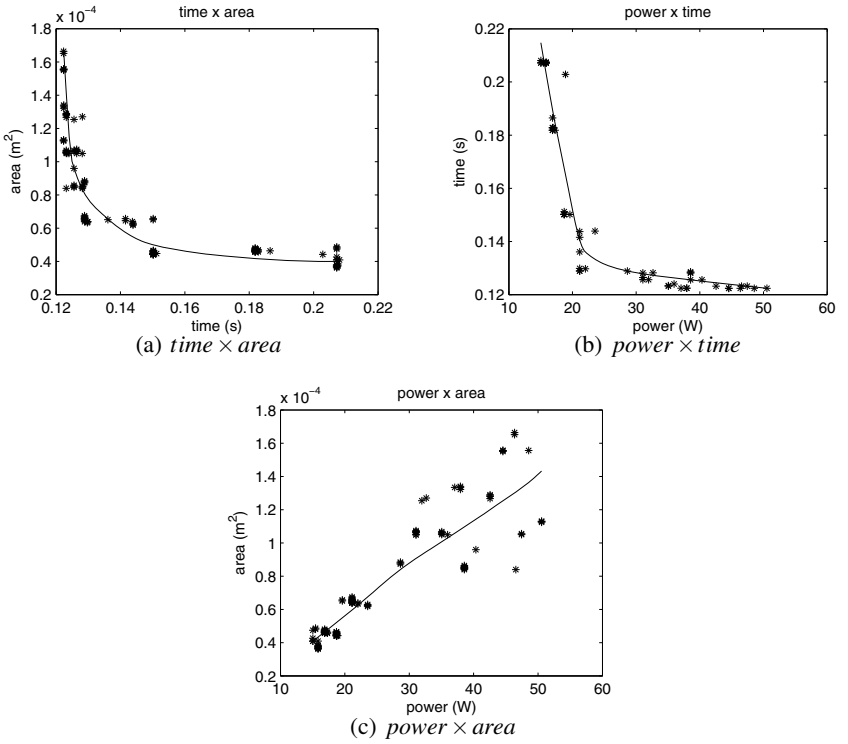


Fig. 10.5 Trade-offs representation of the 142 IP mappings for the task graph of Fig. 10.3

Fig. 10.5–(c) shows a linear relation between power consumption and area. Those values and units are based on E3S Benchmark Suite [2].

For a TG of 16 tasks, a 4×4 mesh-based NoC is the maximal physical structure necessary to accommodate the corresponding application. The obtained solutions showed that no solution used more than ten resources to map all tasks. The unused 6 tiles may denote a waste of hardware resources, which consequently lead to the conclusion that either the geometry of the NoC is not suitable for this application or the mesh-based NoC is not the ideal topology for its implementation.

As a specific mapping example, we detail one of the solutions, which seems to be a moderate solution with respect to every considered objectives. Table 10.1 specifies the processors used in the solution. We can observe that all parallel tasks were allocated in the distinct processors, which reduces execution time. The number of processors were minimized based on the optimization of the objectives of interest and this minimization was controlled by the maximum tasks per processor constraint to avoid hot spots [10]. The processors were allocated in such way to avoid delay of communication due to links and switches disputed by more than one resource at the same time.

Table 10.1 Processors of an illustrative solution of the mapping problem

TG Node	0	1	2	3	4	5	6	7
Proc ID	32	32	15	13	17	0	6	17
IP ID	942	937	458	378	490	43	240	480
Tile	0	0	4	5	10	6	1	10
TG Node	8	9	10	11	12	13	14	15
Proc ID	30	6	13	0	30	15	23	23
IP ID	855	216	379	13	862	456	724	719
Tile	9	1	5	6	9	4	8	8

10.8 Summary

In this chapter, we propose a decision support system based on MOEA to help NoC designers allocate a prescribed set of IPs into a NoC physical structure. The use of NSGA-II, which is one of the most efficient such an algorithm for 100 vezes allowed us to consolidate the obtained results. Structured and intelligible representations of a NoC, a TG and of a repository of IPs were used and these can be easily extended to different NoC applications. Despite of the fact that we have adopted E3S Benchmark Suite [2] as our repository of IPs, any other repository could be used and modeled using XML, making this tool compatible with different repositories. The proposed *shift crossover* and *inner swap mutation* genetic operators can be used in any optimization problem where no lost of data from a individual is accepted. Future work can be two-fold: adopting a dynamic topology strategy to attempt to evolve the most adequate topology for a given application and exploring the use of different objectives based on different repositories.

References

1. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE-EC 6, 182–197 (2002)
2. Dick, R.P., Rhodes, D.L., Wolf, W.: TGFF: Task Graphs For Free. In: Proceedings of the 6th International Workshop on Hardware/Software Co-design, pp. 97–101. IEEE Computer Society, Seattle (1998)
3. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
4. Hu, J., Marculescu, R.: Energy-aware mapping for tile-based NoC architectures under performance constraints. In: ASPDAC: Proceedings of the 2003 Conference on Asia South Pacific Design Automation, pp. 233–239. ACM, New York (2003)
5. Jena, R.K., Sharma, G.K.: A multi-objective evolutionary algorithm based optimization model for network-on-chip synthesis. In: ITNG, pp. 977–982. IEEE Computer Society (2007)
6. Kumar, S., Jantsch, A., Millberg, M., Öberg, J., Soininen, J.-P., Forsell, M., Tiensyrjä, K., Hemani, A.: A network on chip architecture and design methodology. In: ISVLSI, pp. 117–124. IEEE Computer Society (2002)

7. Lei, T., Kumar, S.: A two-step genetic algorithm for mapping task graphs to a network on chip architecture. In: DSD, pp. 180–189. IEEE Computer Society (2003)
8. Murali, S., Micheli, G.D.: Bandwidth-constrained mapping of cores onto NoC architectures. In: DATE, pp. 896–903. IEEE Computer Society (2004)
9. Pareto, V.: Cours D'Economie Politique. F. Rouge, Lausanne (1896)
10. Zhou, W., Zhang, Y., Mao, Z.: Pareto based multi-objective mapping IP cores onto NoC architectures. In: APCCAS, pp. 331–334. IEEE (2006)