An adaptive genetic algorithm for dynamically reconfigurable modules allocation

Vincenzo Rana, Chiara Sandionigi, Marco Santambrogio and Donatella Sciuto

chiara.sandionigi@dresd.org,
{rana, santambr, sciuto}@elet.polimi.it

Politecnico di Milano - Dipartimento di Elettronica e Informazione
Via Ponzio 34/5 - 20133 Milano, Italy

Abstract. This paper aims at defining an adaptive genetic algorithm tailored for the allocation of dynamically reconfigurable modules. This algorithm can be tuned at run-time with a set of parameters to best characterize different architectural scenarios (i.e., single device or multi-FPGAs characterized by several kinds of communication infrastructures) and to adapt the performance of the algorithm itself to the scenario in which it has to operate.

The proposed approach has been validated on a large set of meaningful combinations of parameters (i.e. changing the mutation or the crossover probability), in order to demonstrate the possibility of performing either a fast or an accurate allocation phase.

1 Introduction

Nowadays, thanks to reconfigurable devices (such as FPGAs), it is possible to dynamically tailor the hardware to a specific application, in order to dramatically improve its performance. One of the most suitable approaches in the development of reconfigurable systems is the module-based approach (see [1]), in which the original application is partitioned into several functions, each one of them implemented as a single module. These modules, thus, can be either dynamically loaded into the system or removed from the system, in order to change its overall functionality. The most recent Xilinx design flow, the Early Access Partial Reconfiguration (EAPR) flow, is based on the same approach, as described in [2].

One of the most interesting challenges in such a scenario is the allocation of requested modules in the free space of the reprogrammable device. The allocation phase has to take into account the fragmentation of the device in order to keep the maximum set of contiguous free slots, able to contain bigger modules. On the other hand, this phase has to be executed in a very short time, since it is not desirable to further increase the overhead due to the reconfiguration processes.

The approach presented in [5] trades the execution time for quality of placement, introducing a placement algorithm that is a hybrid solution of the best-fit

and first-fit algorithm. Another feasible solution to this problem is represented by the adaptive genetic algorithm proposed in this paper. This algorithm can be tuned for different scenarios of dynamic reconfiguration. In fact, since it can be executed with a different combination of parameters, it can perform the allocation task either in a very short time or in a very accurate way, as shown by the presented experimental results.

This paper deals with the application of an adaptive genetic algorithm to the allocation of dynamically reconfigurable modules, introducing a very flexible approach to perform the allocation phase. In particular, the next section presents the scenario in which the genetic algorithm can be applied. Section 4 introduces the genetic algorithm on which the adaptive genetic algorithm presented in this paper is based. Section 5 describes the details of the adaptive genetic algorithm and all the parameters that it is possible to tune in order to achieve different levels of performance. Section 6 presents the experimental results that prove the effectiveness of the proposed approach. Finally, conclusions are drawn in Section 7.

2 Module based reconfiguration approach

As previously hinted, one of the more widely used approaches to reconfiguration is the module based approach, that has been proposed by Xilinx in [1]. This approach consist of splitting the reconfigurable device into two different parts:

- a static part, and
- a reconfigurable part.

The reconfigurable part has to be furthermore partitioned in a set of reconfigurable slots, as shown in Figure 1 (2). Both the size of the static part and the number of reconfigurable slots (that strictly depends on reconfigurable modules size) can be tuned in order to adapt the system to the particular design.

In order to change the functionality of the implemented system, it is possible to develop a set of reconfigurable modules, as shown in Figure 1 (3). These modules can be of different size, but they have to span the whole height of the device in order to be compliant with Xilinx Virtex 2 and Xilinx Virtex 2 Pro reconfigurable devices (while the newest Xilinx Virtex IV and Xilinx Virtex V devices also support rectangular modules of any size).

Each reconfigurable module can be dynamically placed on one or more reconfigurable slots (depending on its size), as shown in Figure 1 (4), where modules A, B and C have been configured on the reconfigurable part of the device.

When a module ends the computation, it can be unused for a unknown time interval (such as modules B and C in Figure 1 (5)); in this case, it is possible to remove the module from the system in order to free the resources occupied by the module itself. Another solution, presented in Figure 1 (6), consists of keeping the module configured on the device, implementing thus a sort of module cache. The latter solution occupies a larger amount of reconfigurable resources, but it

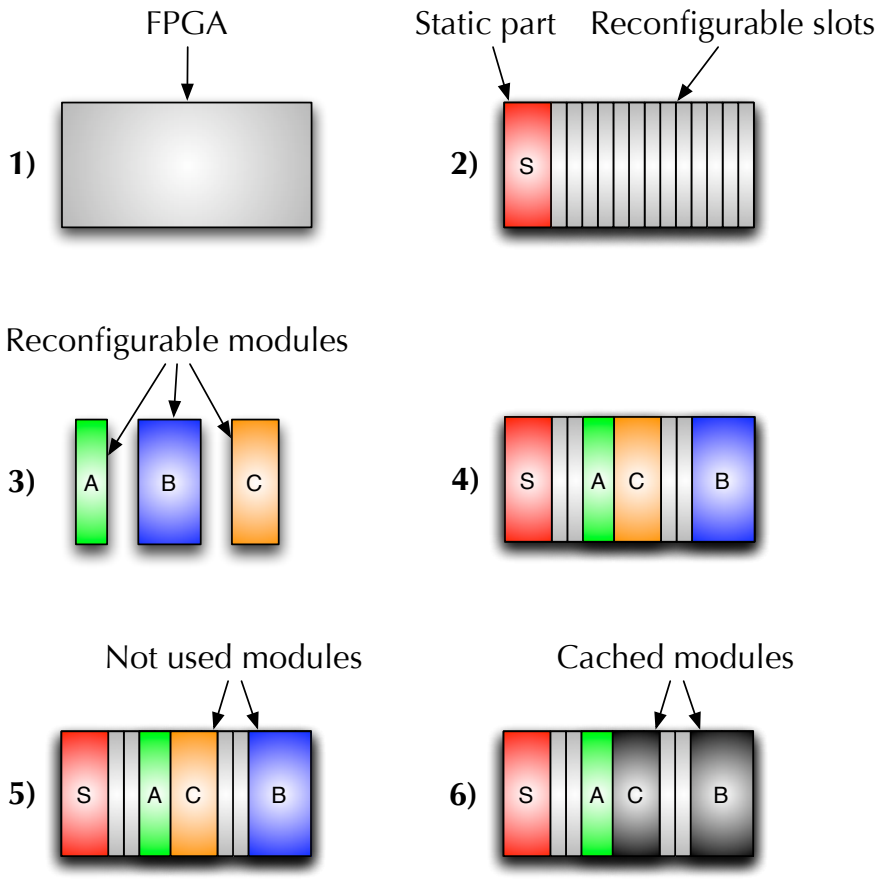

Fig. 1. Module based reconfiguration approach

makes it possible to avoid a reconfiguration (avoiding thus the reconfiguration time overhead) when a module that is cached is required. For instance, referring to Figure 1 (6), if either module B or module C is required, there is not the need to perform a reconfiguration, since they are both already configured in the system; this means that they can be used at any time without requiring any additional setup time.

3 Reconfiguration scenarios

One of the most general platforms on which a configurable or reconfigurable system can be developed is a multi-FPGA scenario where the reconfigurable resources are distributed on several interconnected FPGAs. In such a scenario it is common to have a master FPGA able to reconfigure, partially or totally,

other slave FPGAs. These slave FPGAs can be divided into several slots that can be filled with IP-Cores (or modules) by the master FPGA.

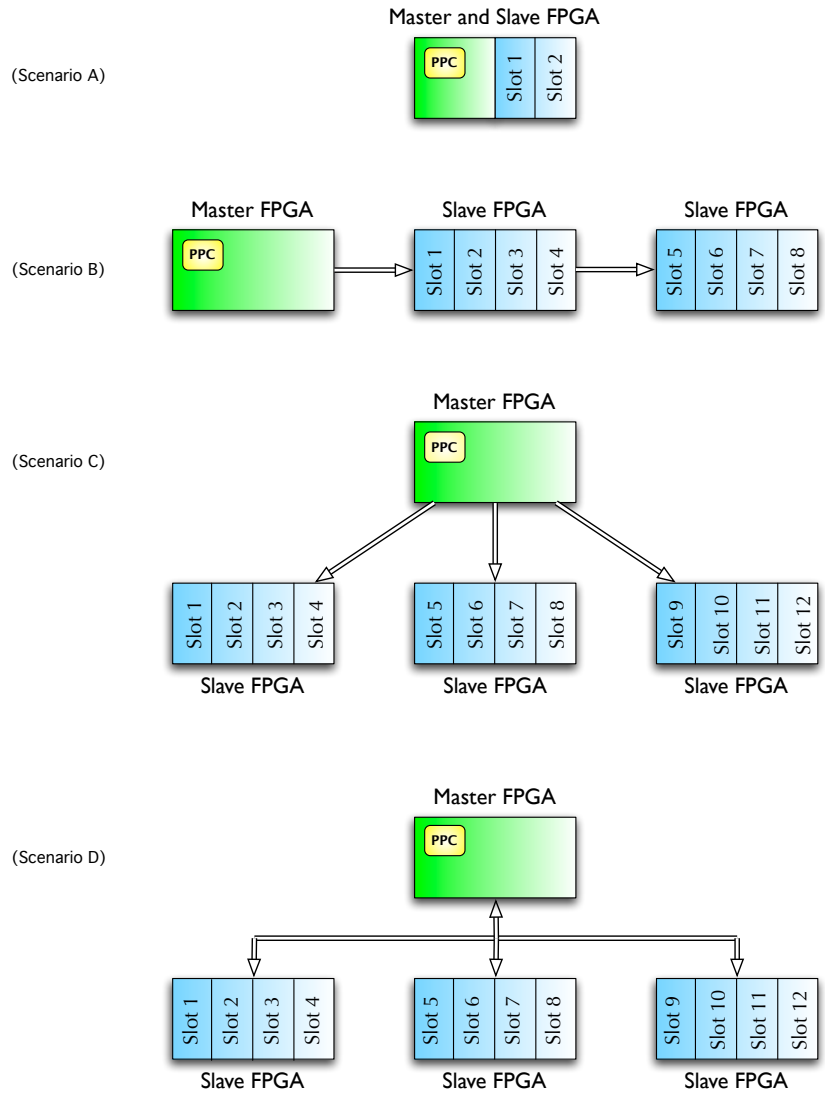

Fig. 2. Multi-FPGA scenarios

Figure 2 presents a collection of different scenarios. In all these scenarios, each master FPGA is characterized by the presence of an embedded PowerPC processor, on which the Operating System runs, in addition to the static hard-

ware components such as a memory controller, general purpose inputs/outputs, and a reconfiguration manager.

The slave FPGAs, instead, hold the reconfigurable resources used to dynamically load hardware modules into the system. These resources are used according to a 1D-placement with a granularity of four CLB (Configurable Logic Block) columns [6]. This means that dynamic modules always use the full height of the FPGA, while their width is a multiple of four CLB columns, even if this scenario can be easily extended to the 2D scenario realized using Xilinx Virtex-4 [3] and Virtex-5 FPGAs [4].

In the first scenario, called Scenario A in Figure 2, there is one FPGA that is used both as a master and as a slave FPGA. An example of such a scenario can be found in [8]. This FPGA is logically divided into two different parts:

- a **fixed part**, that is the part of the FPGA that contains the PowerPC processor and that acts as a single master FPGA;
- a **reconfigurable part**, that is handled as a single slave FPGA, even if the number of slots that it is possible to configure is smaller.

On the other hand, in all the remaining scenarios each FPGA of the system acts either as a master or as a slave FPGA, without logical internal divisions.

The differences between these scenarios reside in the different ways in which the communication infrastructure is implemented. The second scenario (Scenario B) presents a chain communication in which the master FPGA can communicate with just one slave FPGA, and each slave FPGA can communicate just with the following one, for instance by using a communication module in the last slot.

Scenario C and Scenario D, instead, represent a point to point connection and a bus-based connection, respectively. In both these scenarios the master FPGA is able to communicate directly with each slave FPGA. [7] presents an architecture that can be represented using Scenario D.

Even if the presented scenarios differ in the logical partitioning of master and slave FPGAs sets and in their communication infrastructures, they can be reduced to the same class of platforms from the software point of view. For this reason they can be handled by the same software solution, as described in the following.

4 The genetic algorithm

A first version of the genetic algorithm that can be used for the allocation of dynamically reconfigurable modules has been first proposed in [9]. This approach proposes the encoding of a single chromosome (that has to contain the information about the solution that it represents) as a pair of arrays, the *Slots* and the *Modules* arrays:

- The *Slots* array consists of a collection of genes, which contain the information on which module is configured on each slot of the reprogrammable

device. In particular, each gene directly corresponds to a single slot of a slave FPGA. Since on a device of n slots it is possible to configure not more than n modules (this is possible only when each configured module requires just one slot), the alleles of these genes are represented by a number between *0* and *n-1*. The numbers contained in the *Slots* array correspond to the position of a gene in the second array.

– The *Modules* array consists of a set of genes that represent hardware IP-Cores. The following numbers represent the coding of the alleles for this kind of gene:
  - 0: this number means that the module is not configured on the reprogrammable device, since it has not been placed yet or it has already been deleted from the system;
  - 1: this number indicates that the module has been already configured on the FPGA and it is still running, so at this time it cannot be directly unloaded from the system;
  - -2: a module characterized by this number is a cached IP-Core. In other words it is a module that has already been placed on the reprogrammable device but it is not currently used, thus it is possible to unload it to overwrite its slots with the configuration of a more useful IP-Core.

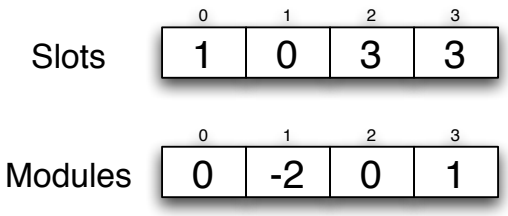

Fig. 3. Genetic algorithm chromosome

The example shown in Figure 3 represents a status of the system in which the second module (module 1) is configured on the first slot of the FPGA (slot 0) and the fourth module (module 3) is placed on the third and on the fourth slot (slot 2 and slot 3), while the second slot (slot 1) is free (since the first module, module 0, is not configured).

The *Slots* array gives further information, indicating that the second module (module 1) is cached, while the fourth module (module 3) is still running. This means that the largest module that is possible to configure starting from this status is a module that requires two slots, since it can be configureb on the first two slots of the FPGA (slot 0 and slot 1), by unloading the second module (module 1) that is currently cached.

After the choice of the proper coding for chromosomes, genes and alleles, it has to be defined a suitable fitness function. Main objective of the allocation

manager is to handle the configurable space of the reprogrammable device to avoid both a waste of slots and the refusing of the configuration of an IP-Core, that happens when there is no place where it is possible to configure it. This means that it is desirable to keep the free slots all together, without breaking them in a lot of smaller separate set of free slots, since a large collection of contiguous slots allows to configure also bigger modules.

For this reason the fitness function has been defined as a number that increases of a small quantity for each free slot. This quantity starts from a default value, but it gets bigger when a free slot is followed by another free slot. On the opposite, when a free slot is followed by a slot containing a cached or a running module, the gain comes back to the default value. Moreover, to prefer solutions with a large number of cached modules, that are useful to speed up the reconfiguration process, a fixed reward is introduced for each cached IP-Core of the solution.

Figure 4 shows an example of the evaluation of the fitness function of three given chromosomes, with a default gain of 2 points, increased of 1 point for each contiguous free slot, and a fixed reward of 1 point for each cached module. The three chromosomes are very similar, but the seventh module (module 6) is placed in a different position in each solution. In the first example (A), the seventh module is located at the end of the FPGA, in the second example (B) it is configured to break the set of the last four free slots, while in the third example (C) it has been placed in the most suitable location, that is the second slot (slot 1). Even if the number of configured IP-Cores, the number of cached modules and the total number of free slots are the same for all the solutions, the first one presents two sets of free slots (whose sizes are 1 and 3 slots, respectively) with a fitness of 13, the second one 4 sets (with sizes of 1, 2 and 1 slots, respectively) with a fitness of 11, while the third one is a single set of 4 slots with a fitness of 16. Obviously the last solution is the most suitable, since it is the only one that allows the configuration of a new module that requires 4 contiguous slots, in fact it presents the largest fitness value within the class of the presented solutions.

The proposed genetic algorithm is performed each time a set of new modules has to be configured on the reprogrammable devices of the system. If each module can be placed in n positions, an exhaustive search with a set of m IP-Cores requires n^m evaluations of feasible solutions. With a genetic algorithm it is possible to considerably decrease the time required by the allocation process, since it works on a smaller set of solutions, trying to modify them to reach a good sub-optimum solution in a reasonable time.

In particular, the first step of the algorithm is the creation of an initial set of randomly generated chromosomes. Then, after the fitness evaluation, a subset of chromosomes is chosen to create a new population. These chromosomes are called parents of the offspring, that is formed through the crossover process.

The crossover task is performed by randomly choosing two parents. The new chromosome is generated by keeping the genes of the first part of the first parent, while the other genes are directly taken from the second parent. During this phase it is possible to introduce, with a random probability, a mutation.

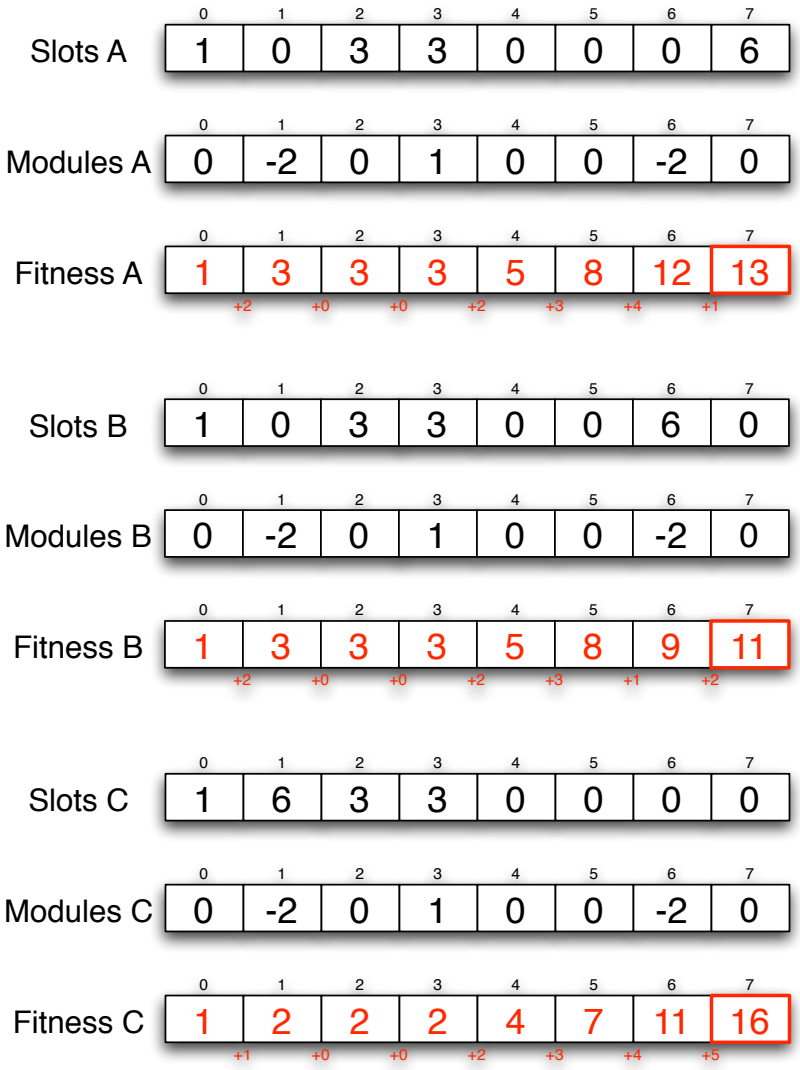

Fig. 4. Fitness evaluation examples

This is defined as a change in the partial solutions found by the parents, which means that the location inherited by the parents can be randomly modified, to prevent that all solutions in the population fall into a local optimum.

5 Adaptive genetic algorithm

The genetic algorithm described in Section 4 has been extended with a set of configurable parameters that make the algorithm dynamically adaptive with respect to the platform scenario where it has to work. These parameters provide the possibility of choosing either a fast or a very accurate allocation phase, depending on the timing performance and on the space constraints.

The parameters that can be tuned to tailor the solution onto a specific scenario are:

- **initial population size**, that is the initial size of the randomly generated population, as described in Section 5.1;
- **selection size**, the number of chromosomes that are chosen to create the new population, described in Section 5.2;
- **maximum number of rounds**, introduced in Section 5.3, that is the maximum number of generations that can be performed before stopping the execution of the algorithm;
- **minimum fitness**, described in Section 5.4, that is the fitness threshold;
- **crossover probability**, that is the probability of performing a crossover of two parents in order to generate a new offspring (otherwise the first parent is not modified), as presented in Section 5.5;
- **neutral mutation probability**, described in Section 5.6, that is the probability of performing a neutral mutation on the new chromosome;
- **positive mutation probability**, described in Section 5.7, that is the probability of performing a positive mutation on the new chromosome;
- **negative mutation probability**, described in Section 5.8, that is the probability of performing a negative mutation on the new chromosome.

Each parameter can be tuned in order to achieve the desired performance, both in terms of time and in terms of refused modules.

It is possible, in fact, that a particular scenario requires a fast allocation phase, for instance when the module that has to be deployed has to be available in a very short time. In this case it is possible to run the genetic algorithm with a set of parameters that provides a feasible position for the module in a fast way. The execution of the algorithm with this set of parameters affects the performance of the algorithm itself and increases the fragmentation of the reconfigurable device, but this negative effect can be kept under control by choosing the most suitable set of parameters, as shown in Section 6.

On the other hand, when a module is requested in advance with respect to its real utilization time (for instance when pre-fetching is performed), it is possible to execute the genetic algorithm with a set of parameters that allows the search for a solution that minimizes the fragmentation of the reprogrammable device. To achieve this result, it is necessary to know the right set of parameters that are able to reduce the average number of refused modules during the whole life of the system.

For these reasons, each parameter has been tested with a large set of significant values, as described in the following sections.

5.1 Initial population size

Each time a module is requested, the genetic algorithm has to create an initial population that consists of randomly generated individuals. Each one of these individuals has to satisfy all the constraints, since it has to represent a feasible solution. The single chromosome within the population will change its characteristics, but the total number of chromosomes will not change, since the population size is fixed to the value of the size of the initial population. The initial population size, then, will affect the whole execution of the genetic algorithm, since it represents the size of the population on which each operation (such as crossover and mutations) will be performed. The genetic algorithm has been tested with three different values, that are 10, 50 and 100 chromosomes.

5.2 Selection size

When the fitness of each chromosome of the population is evaluated in order to choose the chromosomes that will act as parents (that are, in other words, the chromosomes with the maximum fitness value) during the generation of the new population, it is possible to select a set of these chromosomes that will be kept, without any changes, in the next generation. The selection size is hence the number of chromosomes that will be kept without any changes, while the difference between the initial population size and the selection size represents the number of chromosomes that have to be created during the offspring generation phase. The selection size depends on the initial population size: for this reason the values of the selection size has been chosen as 1/4, 1/2 and 3/4 of the initial population size, that represent three different situations, in which few, half or a lot is preserved from the previous generation.

5.3 Maximum number of rounds

The generation (that consists of the evaluation of the fitness, in the selection of the most suitable solutions and in the generation of the children) has to be performed either for the maximum number of rounds or until the minimum fitness is reached. In the first case, in which the minimum fitness is never reached, the value that represents the maximum number of rounds has to be chosen keeping into account that a big value requires a large execution time, while a small value can lead to a solution that is not optimal and that increases the fragmentation of the reconfigurable device. In particular, in our experiments, we used for this parameter the following values: 10, 20 and 50.

5.4 Minimum fitness

The minimum fitness represents the threshold that has to be exceeded in order to accept a chromosome as a final solution. This parameter is very important since it allows an early-stop of the algorithm when a good solution has been found. Obviously, with a small minimum fitness value, the final solution will

not be optimized, while a big value of this parameter will probably bring the algorithm to execute for the maximum number of rounds, as described in Section 5.3. The minimum fitness is hence a measurement of the goodness of the desired soution. The goodness index will be explained more in details in Sextion 6. For our experiments we used three different values: 100, 1000 and 2000.

5.5 Crossover probability

The crossover task is performed by randomly choosing two parents within the set of the selected chromosomes, as introduced in Section 5.2. Each new chromosome is generated by keeping the genes of the first part of the first parent, while the other genes are directly taken from the second parent. When the crossover is not performed, the new chromosome is equal to one of the two parents, chosen randomly. In both cases, children always represent valid solutions for the given problem. The crossover parameter is hence responsible for the generation of an offspring that mixes the good characteristics of the most suitable solutions of the previous generation, in the hope to determine a better one. In our experiments we tested this probability with the following values: 25%, 50% and 75%.

5.6 Neutral mutation probability

Each time a new chromosome is generated it is possible to perform a neutral mutation by modifying the position of the requested module within the reconfigurable device (for this reason it has been called neutral mutation, since it preserves the status of the modules configured on the reconfigurable device). This mutation allows the generation of a new solution that was not present in the initial population, so it is an index of the difference between the solutions achieved by a population and the following one. The new location of the requested module has to be a feasible position, since each chromosome has always to represent a feasible solution. As with the other probabilities, we tested this parameter with the following values: 25%, 50% and 75%.

5.7 Positive mutation probability

With a positive mutation it is possible to free space on the reprogrammable device by deleting a module that was previously kept in cache. This mutation allows the increase of the number of positions where the requested module can be placed (as described in Section 6) without any penalization. The slots occupied by the deleted module are marked in a special way, since they have to be recognized at the end of the algorithm, when slots that have been deleted but that are not used by the requested module can be simply reintroduced without introducing any overhead and increasing the goodness of the final solution. Also this probability has been tested with the following values: 25%, 50% and 75%.

5.8 Negative mutation probability

A negative mutation, in which a module that has been removed from the cache will be reintroduced in the cache, can be introduced to increase the goodness of the solution at run-time. This kind of modules, in fact, can be reintroduced in the cache in order to avoid the placement of the requested module, without any penalization, in a location that will lead to delete a cached module. In our experiments, we used the following values for this probability: 25%, 50% and 75%.

6 Experimental results

Each combination of the values of the parameters presented in Section 5 has been tested in order to achieve the performance characterization of all the possible sets of parameters.

The base scenario on which these tests have been performed consists of a reconfigurable device that has been divided in fifty reconfigurable slots. Furthermore, the size of the single module that can be deployed on the system ranges from one to three slots.

Table 1. Parameters values

Parameter	First value	Second value	Third value
Initial population size (IPS)	10	50	100
Selection size	$\frac{1}{4} * IPS$	$\frac{1}{2} * IPS$	$\frac{3}{4} * IPS$
Maximum number of rounds	10	20	50
Minimum fitness	100	1000	2000
Crossover probability	25	50	75
Neutral mutation probability	25	50	75
Positive mutation probability	25	50	75
Negative mutation probability	25	50	75

Table 1 presents all the possible values for each parameter. Since there are eight parameters and each parameter presents three different values, it is nec-

essary to perform $3^8 = 6561$ experiments in order to evaluate all the possible combinations of the parameters' values.

For each combination of parameters an experiment has been performed that consists of the following steps:

- fifty tests, consisting of fifty module requests each, have been performed. In particular, each test performs the following tasks:
  - a random module request is given as input to the genetic algorithm;
  - the result (success/fail) of this process and the time required for its execution are stored to calculate the fitness of the current solution;
  - randomly a module is deleted from the reconfigurable device (in order to avoid the saturation of the device itself);
- at the end of each test the status of the reprogrammable device has been reset and the average results of the simulations (number of refused modules, cash index and timing performance) have been updated.

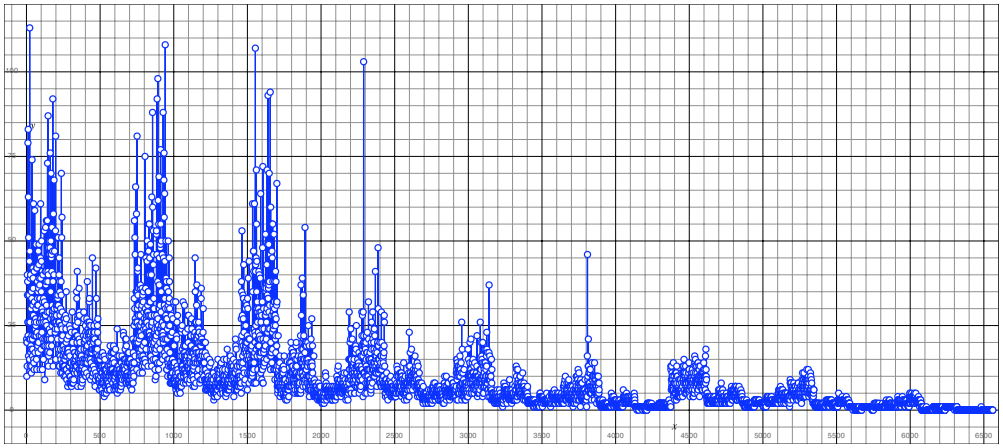

Fig. 5. Goodness index for all the solutions

Figure 5 shows the average goodness index (that represent the fitness of a given solution) for each combination of parameters (the test flow previously described has been performed two times in order to avoid erroneous results). The goodness has been evaluated as follows:

$$Goodness = \frac{CI}{ET * RM}$$

where:

- CI is the Cache Index: this index is inversely proportional to the fragmentation of the reprogrammable device ($CI = \frac{1}{Fragmentation}$);

– *ET* is the Elapsed Time: it represents the time necessary to perform a whole experiment, that consists of 2500 module requests;
– *RM* is the number of Refused Modules. In other words, this index represents the number of modules that have not been placed during the execution of the algorithm.

Table 2. Top four experimental results

Combination number	**22**	**942**	**1554**	**2289**
Initial population size	10	10	10	10
Selection size	2	7	5	5
Maximum number of rounds	10	20	50	10
Minimum fitness	100	100	100	1000
Crossover probability	25	50	25	25
Neutral mutation probability	75	75	50	75
Positive mutation probability	50	50	50	25
Negative mutation probability	25	75	75	75
Number of refused modules	250	230	175	262
Elapsed time (s)	0.5465	0.544	0.6885	0.5295
Cash index	15450	13540	12977	14399
Goodnes index	113	108	107	104

Table 2 chesh index and inversely proportional to both the number of refused modules and the elapsed time. It is also possible to tune this goodness function in order to give more importance to the first two components (for instance, by using this function for the goodness index, $Goodness = \frac{CI^2}{ET*RM^2}$,the result will be a solution optimized in terms of the number of refused modules) or to the last one (for instance, by using the following function, $Goodness = \frac{CI}{ET^2*RM}$, the result will be a solution optimized with respect to timing performance).

Table 3 presents two combinations of parameters that lead to a very small number of refused modules (both combinations have achieved less then 200 refused modules). In both these combinations the maximum number of rounds has been set to 50 and in the second one the initial population size has been set to 50 too.

Table 3. Refused modules optimization

Combination number	**1554**	**1891**
Initial population size	10	50
Selection size	5	37
Maximum number of rounds	50	50
Minimum fitness	100	100
Crossover probability	25	50
Neutral mutation probability	50	25
Positive mutation probability	50	25
Negative mutation probability	75	25
Number of refused modules	175	180
Elapsed time (s)	0.6885	1.792
Cash index	12977	17381
Goodnes index	107	54

Table 5 shows the top three combinations that are able to perform the allocation of a requested module in a very short time. By using these combinations, in fact, it is possible to accomplish a single module request in less than 0.2 milliseconds, since 2500 modules requests require less than 0.5 seconds. All the combinations presented in Table 5 are characterized by an initial population size of 10, by a selection size of 7, by a maximum number of 10 and by a minimum fitness of 100.

Table 4. Timing optimization

Combination number	**166**	**169**	**179**
Initial population size	10	10	10
Selection size	7	7	7
Maximum number of rounds	10	10	10
Minimum fitness	100	100	100
Crossover probability	25	25	25
Neutral mutation probability	25	25	50
Positive mutation probability	50	75	75
Negative mutation probability	25	25	50
Number of refused modules	522	546	335
Elapsed time (s)	0.473	0.476	0.477
Cash index	12579	11854	14770
Goodnes index	51	46	92

Finally, Figure 6 shows a comparison between timing performance of the genetic algorithm and an exhaustive approach. The experiment has been performed on a reconfigurable module 8 columns wide. Communication overhead (that is needed in order both to perform a module request and to know where the module has been placed) is around 1 ms. Since communication with the Reconfigurator Manager occurs two times, the total communication overhead is around 2 ms. These values, that can be found in Table 5, are the same for both the genetic algorithm and the exhaustive approach.

On the other hand, the Reconfigurator Manager overhead strictly depends on the algorithm that has been chosen. With an exhaustive approach, around 3 ms are needed in order to perform the allocation phase, while the genetic algorithm is able to decrease this value to 1 ms (introducing a negligible worsening in the quality of the output, as shown by the previous experimental results). Thus, the adoption of a Reconfigurator Manager based on the proposed genetic algorithm

provides the possibility to achieve a total speedup of ~ 1.5, since the genetic algorithm makes it possible for the Reconfiguration Manager to perform the allocation phase around 3 times faster.

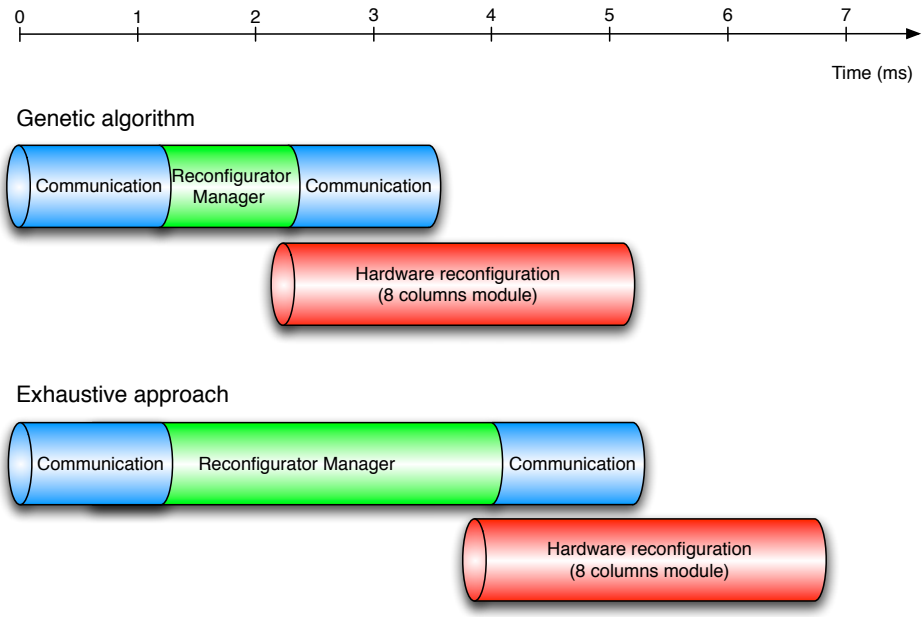

Fig. 6. Timing performance comparison

7 Conclusions

Figure 5 proves that the goodness index (Y-axis), evaluated for all the possible combinations of the parameters (X-axis), is a cyclic function and that it is significantly affected by the changes in the parameters value.

Furthermore, results presented in Section 6 have shown how it is possible to perform an allocation of a requested module with a different combination of parameters in order to achieve different optimizations. It is possible either to minimize the number of refused modules or to reduce the time required for the computation. It is also possible, finally, to use a combination of parameters that optimizes the goodness index; this makes it possible to achieve an optimal compromise between the three presented metrics.

The genetic algorithm presented in Section 4 and extended as described in Section 5 has been proved to be an effective solution for dynamically reconfigurable modules allocation.

Table 5. Timing performance comparison

Algorithm	Exhaustive	Genetic
Hardware reconfiguration (8 columns module)	$\sim 3\ ms$	$\sim 3\ ms$
Communication overhead (request)	$\sim 1\ ms$	$\sim 1\ ms$
Communication overhead (response)	$\sim 1\ ms$	$\sim 1\ ms$
Reconfiguration Manager overhead	$\sim 1\ ms$	$\sim 3\ ms$
Reconfiguration Manager speedup	~ 3	1
Total time	$\sim 5\ ms$	$\sim 7\ ms$
Total speedup	$\sim 1.5\ ms$	1
Total reconfiguration overhead	$\sim 2\ ms$	$\sim 4\ ms$
Total reconfiguration overhead w.r.t. hardware reconfiguration time	$\sim$66 %	$\sim$133 %

References

1. Xilinx Inc., Two Flows of Partial Reconfiguration: Module Based or Dif- ference Based, Tech. Report XAPP290, Xilinx Inc., November 2003.
2. Xilinx Inc., Early Access Partial Reconfiguration User Guide, Tech. Report UG208, Xilinx Inc., March 2006.
3. Xilinx Inc., Virtex-4 User Guide, Tech. Report UG070, Xilinx Inc., April 2007.
4. Xilinx Inc., Virtex-5 User Guide, Tech. Report UG190, Xilinx Inc., February 2007.
5. K. Bazargan, R. Kastner, M. Sarrafzadeh *Fast template placement for reconfigurable computing systems*, IEEE design and test - Special issue on reconfigurable computing, pages 68-83, Volume 17, Issue 1, January 2000.
6. H. Kalte, M. Porrmann, U. Ruckert *System-programmable-on-chip approach enabling online fine-grained 1D-placement*, IPDPS'04, Workshop 3, page 141.
7. H. Kalte, M. Porrmann, U. Ruckert *A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs*, Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC), 2002.
8. Alberto Donato and Fabrizio Ferrandi and Marco D. Santambrogio and Donatella Sciuto *Coperating system support for dynamically reconfigurable SoC architectures.*, IEEE-SoCC, 2005.
9. Vincenzo Rana, Chiara Sandionigi and Marco Domenico Santambrogio, *A genetic algorithm based solution for dynamically reconfigurable modules allocation*, Southern Conference on Programmable Logic, pages 183-186, 2007.