An Enhanced Evolutionary Approach To Spatial Partitioning For Reconfigurable Environments
Pratibha P, Siva Nageswara Rao Borra, A Muthukaruppan, S. Suresh,
V. Kamakoti
kama@iitm.ernet.in
Digital Circuits and VLSI Laboratory,
Department of Computer Science and Engineering,
IIT-Madras, Chennai-600 036, India.
http://vlsi.cs.iitm.ernet.in

ABSTRACT
This paper introduces a novel parallel evolutionary methodology making use of ANN for solving the spatial partitioning problem for Multi-FPGA (Field Programmable Gate Arrays) architectures. The algorithm takes as input a HDL (Hardware Description Language) model of the application along with user specified constraints and automatically generates a task graph $G$; partitions $G$ based on the user specified constraints and maps the blocks of the partitions onto the different FPGAs in the given Multi-FPGA architecture, all in a single-shot. The proposed algorithm was successfully employed to spatially partition a reasonably big cryptographic application that involved a 1024-bit modular exponentiation and to map the same onto a network of nine ACEX1K based Altera EPIK30QC208-1 FPGAs. The suggested parallel evolutionary algorithm for the partitioning step was implemented on a 6-node SGI Origin-2000 platform using the Message Passing Interface (MPI) standard. The results obtained by executing the same are extremely encouraging, especially for larger task graphs.

Keywords: FPGA, Spatial Partitioning, Evolutionary Algorithms, Message Passing Interface, Parallel Computing, ANN, Pattern mapping, Feed forward networks.

1. INTRODUCTION
Functional specifications of semiconductor products change frequently in compliance with market requirements and evolving standards. This necessitates a hardware environment that can be programmed dynamically. Reconfigurable systems provide a viable solution to this problem. But, such reconfigurable systems demand expensive and tedious initialization phases that in turn entail concepts like temporal [1] and spatial partitioning [3]. A given application that is too large to fit into the reconfigurable system, all at once, is passed through the temporal partitioning phase, which ensures a break up of the application into temporal segments that maximize on resource utilization and minimize on execution time. Each of the temporal segments is then spatially partitioned in order to establish a mapping of the tasks and the memory segments, within a specific temporal segment onto the available FPGAs, in a manner that enhances the efficiency of the implemented logic.

An evolutionary approach to solve the spatial partitioning problem is presented in [3]. The concepts presented in [2], though not directly related to the theme of this paper, provide significant insight on the utilization of primitive genetic algorithms to solve the problem of hardware software partitioning and hardware design space exploration.

1.1 Previous work
The genetic algorithms are very effective in giving a very high quality results with reduction in run time, and help the solutions to get out of local maxima, particularly in spatial partitioning problems where multiple constraints are involved, and is reflected in experimental verifications. The genetic search procedure was developed by John Holland [4] in 1975 and since then has been used successfully to provide a solution for the various combinatorial problems in VLSI design automation [5,6]. Most of these approximations for partitioning use a difficult encoding and do not preserve the structure and are being revived. The recent paper published in this area is by Jose Ignacio Hidalgo [7]. The GA algorithm employed here [7] does not capture all the constraints for e.g. memory constraints, speed of execution, routing constraints etc. The CPU time required for getting desirable results is large. The partitioning done in [7] is at net list (cell) level. We consider the partitioning at task graph (behavioral) level. Iyad Quaiss [3] also considers the partitioning at behavioral level. The improvement of the results of our sequential algorithm over run time compared to [3] is shown in Table 5.1. This is the first attempt in employing parallel genetic algorithms using ANN for spatial Partitioning. The results are extremely encouraging as shown in Table 5.1, especially for larger task graphs.

1.2 Contributions of this paper
The algorithm suggested in this paper, takes as input a HDL model of the application along with user specified constraints and automatically generates a task graph $G$; partitions $G$ based on the user specified constraints and maps the blocks of the partitions onto the different FPGAs in the given Multi-FPGA architecture, all in a single-shot. The crux of our contribution is a genetic approach for the
spatial partitioning problem for FPGAs that is employed during the course of the partitioning step. The methodology proposed in the paper would blend in and complement temporal partitioning techniques as proposed in [3], as a result of which we cut down a substantial proportion of the time before the logic is mapped on to the FPGAs.

One of the most notable feature of this paper is that the spatial partitioning problem not only maps different tasks in a temporal segment to the multiple FPGAs, but also routes the pin to pin inter-FPGA connections between the multi-FPGAs such that, some critical parameters of the routing are taken care of. These parameters include time-critical nets, routing congestion on interconnection devices like FPlD and the overall inter-FPGA signal propagation delay.

2. Overview of the proposed methodology

The proposed methodology follows a three-phased approach, as shown in Figure 2.1 to meet the desired objectives. The first phase deals with the task graph generation, the second phase solves the problem of spatial partitioning using an evolutionary approach and the third phase will do the required mapping onto the given FPGAs. The third phase requires a trivial iterative algorithm. The complete flow is as shown in Figure 2.1.

3. SPATIAL PARTITIONING

The target architecture onto which the application is to be mapped is assumed to be a network of N FPGAs, each possessing a local memory, and all of them having access to a global shared memory [3]. Throughout this paper, let N refer to the total number of FPGAs available on the target architecture. During the process of spatial partitioning, we are required to assign a particular FPGA from the set of N FPGAs to every task that is a part of a particular temporal segment. Apart from that, the spatial partitioning phase also addresses the problem of mapping logical memory segments to local or shared memory. The complexity of the problem arises when we attempt to do the above in a manner that satisfies the various constraints associated with spatial partitioning [1].

Area Constraint: The logic of the different tasks mapped on to the N FPGAs dictates the amount of area required on each FPGA. As the available area for each FPGA is fixed, this imposes a constraint on the logic that could be realized.

Memory Constraint: The memory segments of the tasks of a spatial partition should not exceed the memory resources available.

Speed Constraint: The speed with which the implemented circuit works is of utmost importance. We are required to use certain heuristics to determine an implementation with good speed of execution. Though a solution may be acceptable, the speed of execution should be of an acceptable order; else it is not practically feasible.

Connection Constraint: The Multi-FPGA system would be required to establish a connection between specified FPGAs within it, and also to the external world. We should have sufficient pins to make the required connections, else there would arise a necessity to form new spatial partitions.

3.1 PHASE I: THE TASK GRAPH GENERATION

The first phase involves splitting a given application, specified in a HDL format, into smaller modules that can fit into the FPGAs taking care of all the interactions between the modules resulting from splitting a bigger module. As the application is generated from a temporal partitioning phase, it is possible to simulate the given application on the target architecture.

Definition: Given a set of n HDL modules $M=(m_1, m_2, \ldots, m_n)$ and k distinct pairs of edges $E=(e_1, e_2, \ldots, e_k)$ between modules such that each $e_i$ is an ordered pair of vertices $\langle m_i, m_j \rangle$, then the circuit can be represented as a graph $G=(M, E)$ which is called a task graph. A graph $G'=(M', E')$ is a sub graph of $G=(M, E)$ if and only if $M' \subseteq M$, $E' \subseteq E$.

A given module can be fit into a given single FPGA as long as the logic blocks and the number of pins required by the module does not surpass the maximum number of the available logic blocks and pins in the FPGA. The algorithm for the construction of the task graph is as follows:

Function TaskGraph (HDL description $M$)
begin
If $M$ fits in a single FPGA
then return
else begin
  • Split $M$ into smaller modules ($m_i$).
m₁,...,mₙ)
- Add nodes m₁, m₂,...,mₙ to task graph
- Update the task graph to capture the relationships
- For each mᵢ call TaskGraph(mᵢ)

End Function

The task array generated by this phase is given as an input to the genetic algorithm in the next phase.

3.2 PHASE II: THE GENETIC APPROACH

The second phase takes in's input from the first phase. The input is in the form of a task graph. This phase provides an optimal mapping of the tasks on to the given FPGA architecture using an evolutionary strategy.

The crux of any evolutionary approach lies in the "survival of the fittest" policy. A sequential (uniprocessor based) evolutionary approach for solving the problem of spatial partitioning has been presented in this paper. This approach involves generating a random population and enhancing the evolution of the same using the genetic (mating and mutation) operators. The different "chromosomes" are graded using a fitness function at the end of each generation. The subsequent selection of the best individuals from the population ensures that the quality of the global population drastically increases after each of the iterations. Initially we present the sequential algorithm. Subsequently, we extend the same to include parallelism in section 3.3. Finally, we incorporate the concept of Artificial Neural Networks (ANN) in section 4 for effective and fast evolution of the chromosomes.

3.2.1 ENCODING

The encoding used is identical to the one utilized in [1]. Two arrays are maintained, a task array Task and a memory array Memory. The length of Task is equal to the number of tasks in the task graph, t, while the length of Memory is equal to the number of memory segments, m. For 1 ≤ i ≤ t, the variable Task[i], ranging from 1 to N, represents the FPGA number to which task i is assigned to. Similarly for 1 ≤ i ≤ m, the variable Memory[i], ranging from 1 to N, represents the memory bindings. Memory[i]=0 implies that the memory segment i is mapped to the shared memory. The Task and Memory arrays together constitute a chromosome.

3.2.2 INITIAL POPULATION

The task arrays for all chromosomes in the initial population are set to random legal values. Then based on task assignments, for each chromosome, we map the logical memory segments to local/global physical memories. If the majority of the tasks, which access a given logical memory segment M, are assigned to a FPGA f, then M is mapped onto the local memory present in f. Through a lot of experimentation, we decided to start with an initial population of size N.

3.2.3 MATING

The following genetic operators are used in the mating phase of the proposed genetic algorithm.

Area Constraint Operator (ACO): This operator takes in two parents, Parent1 and Parent2, and produces a total of two offspring that are, with a high probability, better off as far as area constraints are concerned. Note that ACO only acts on the Task arrays of the parents and not on the Memory arrays. The ACO has been framed in a manner that retains all the information regarding the tasks up till the first violation of the area constraint. The parents are then crossed over in an attempt to reorganize the tasks in a manner that would progressively reduce the number of area conflicts, as the population evolves.

Function ACO (Parent P1, Parent P2)
- Obtain the index, i₁, of the first task in the Task array of P₁, that causes an area constraint violation. If such a conflict does not exist, then randomly choose an index.
- Obtain the index, i₂, of the first task in the Task array of P₂, that causes an area constraint violation. If such a conflict does not exist, then randomly choose an index.
- Obtain Child₁ by crossing over the two parents at i₁. Obtain Child₂ by crossing over the two parents at i₂.

End Function

Pin Constraint Operator (PCO): The PCO attempts to optimize on the delay caused by the pin connections and hence reduce the communication time of the circuit when it is up and running. The reasoning behind the PCO is to minimize the inter-FPGA communication. The time required for a signal to propagate from one FPGA to another is much more than the time taken for intra-FPGA information transmission. Hence, in the PCO, the task that uses the greatest number of pins is chosen, since that task might have been placed in the wrong FPGA. By moving the task to another FPGA, one might be able to drastically cut down on the number of pins used, and hence the total time required for the signals to propagate.

Function PCO (Parent P1, Parent P2)
- Obtain the index, i₁, of the task that utilizes the maximum number of pins in P₁. Break ties arbitrarily.
- Obtain the index, i₂, of the task that utilizes the maximum number of pins in P₂. Break ties arbitrarily.
- Obtain Child₁ by crossing over the two parents at i₁. Obtain Child₂ by crossing over the two parents at i₂.

End Function

Memory Constraint Operator (MCO): MCO requires two chromosomes, Parent₁ and Parent₂, and produces two
offspring that are hopefully better off as far as memory constraints are concerned. Since the amount of memory available is of a finite quantity, one has to ensure the judicious allocation of the memory resources. MCO initially checks for memory conflicts, and if they do not exist, it then checks for the memory segment that is most accessed by tasks in other FPGAs. Such a segment, if placed in another FPGA, would reduce the number of accesses to it from other FPGAs, as a consequence of which, the speed with which the circuit executes would be made more efficient.

**Function MCO (Parent P1, Parent P2)**
- Obtain the index, i1, of the first memory segment in the Memory array of P1 that causes a memory constraint violation. If no violation exists, then make i1 indicate the memory segment that is accessed the most by tasks in other FPGAs.
- Obtain the index, i2, of the first memory segment in the Memory array of P2 that causes a memory constraint violation. If no violation exists, then make i2 indicate the memory segment that is accessed the most by tasks in other FPGAs.
- Obtain Child1 by crossing over the two parents at i1. Obtain Child2 by crossing over the two parents at i2.

End Function

**Default Crossover Operator (DCO):** Most chromosomes in the population would already be valid solutions, but they may not be close to the optimal, hence we utilize the DCO in an attempt to bring the solution as close as possible to the optimal solution. The DCO chooses random locations in the Task and Memory arrays and crossovers the two parents to obtain the required offspring.

**Function DCO (Parent P1, Parent P2)**
- Choose a random index, i1, in the Task array.
- Choose a random index, i2, in the Memory array.
- Crossover P1 and P2 at i1 in the Task array and at i2 in the Memory array.

End Function

For the process of mating, the best of the population is chosen along with a few random selections as well. The chosen chromosomes are paired up, and one of the above four operators are applied randomly. Note that applying ACO to a solution that does not violate the area constraint would not be possible, in such cases the algorithm simply returns and the DCO is used.

### 3.2.4 MUTATION

The mutation operator is an attempt to save characteristics that have been lost over several generations. This operator randomly chooses a value from the Task and the Memory array and changes it to another legal value. A probability of mutation is associated with each chromosome in the population, which is basically inversely proportional to its fitness value. We choose $N_{mutation}$ number of chromosomes during every iteration for the mutation process.

### 3.2.5 FITNESS FUNCTION

The fitness function has been divided into five distinct sections. For each section we obtain a fitness value in the range [0..10], where 0 indicates a bad solution, and 10 indicates an excellent solution. We weigh the fitness values and scale them up appropriately to a value in the range [0..10]. The five sections are as follows:

**Area Fitness Value (AFV):** Obtain the number of FPGAs, C, in which the area required by the logic exceeds the area provided by the FPGA. AFV is determined by computing $((N-C)/N)*10$. The higher the value, the better the chromosome is with respect to the area constraint.

**Pin Connection Fitness Value (PFV):** Obtain the total number of FPGAs, E, in which the number of pins available is less than the total number of pins required by the logic and memory mapped on to it. PFV is determined by computing $((N-E)/N)*10$.

**Memory Fitness Value (MFV):** Obtain the number of local memory and shared memory resource violations, V. MFV is determined by computing $((N-V)/N)*10$. The equation has been framed in a manner such that a high value indicates a better solution with respect to the memory constraints.

**Speed Fitness Value (SFV):** The speed with which the implemented circuit operates is greatly dependent on the location of a task $t$, with respect to the variables it interacts with. The speed would be greatly enhanced if most of the variables used by $t$ were mapped onto the local memory of the FPGA in which $t$ is mapped. Critical variable interactions are given importance. The fitness value is a function of the proximity between the FPGAs mapping a task $t$ and those storing the variables that interact with $t$. Closer are the variables to $t$, the higher is the fitness value. This fitness function depends on the architecture of the multi-FPGA system.

**Routing Fitness Value (RFV):** The routing fitness value tries to minimize on various routing factors such as the total length of wires used to route, and the total number of segments used for routing. It ensures that the spatial partition is such that it is possible to find a routing on the target architecture. The routing mentioned here is the routing between the various FPGAs on the target architecture (inter-FPGA routing). Since the spatial partitioner, chooses only a solution which has a good possible routing, one can be sure that, the actual routing obtained would be very effective and efficient. This tight

---

1711
coupling between the routing and the spatial partitioning phase adds great strength to our approach.

The final fitness value is determined by weighting the above five fitness values in the required proportions and scaling the final value to a range of [0..10]. Such an approach of computing the fitness value takes into account all the factors associated with the most important one being given higher priority.

3.3 MIGRATION

The operators and functions considered so far are concerned only with the processor that executes them. The parallel genetic approach involves executing multiple copies of the above-mentioned sequential algorithm on multiple processors. Parallelism is incorporated in the algorithm by transferring the best solutions of each of the genetic algorithms across all other processors. Such a communication among the processes is established in this phase. In essence the best \( N_{\text{migration}} \) number of chromosomes from each process were chosen and broadcast to the rest of the processes. Such a broadcast is done at the end of every mating phase. A process, when it receives chromosomes from a foreign process, incorporates them into the local population and proceeds with execution. This not only enhances the performance of the algorithm due to a better exploration of the solution space but also leads to the development of a mutually beneficial environment.

3.4 THE FINAL GENETIC ALGORITHM

The sequential genetic algorithm, which provides a solution to the spatial partitioning problem, is provided below.

**SpatialGeneticSequential (Problem Specification)**

Create initial population

while (solution with the required fitness value has not been found) do

- Choose Parents for mating process
- Pair up Parents Randomly and Carry Out Mating
- Add Offspring to population
- Carry out Mutation
- Carry out Migration

return (the chromosome \( C \) with maximum fitness value)

End Function

The third phase is a simple mapping tool which requires an iterative algorithm and is hence not explicitly discussed here.

4. INCORPORATION OF ANN

The learning nature of ANN can be utilized to control the various parameters associated with the genetic algorithm. We have incorporated the neural networks into our genetic algorithm to dynamically control the rate of mating, mutation and migration. This ensures optimal utilization of resources and thereby cutting down the running time of our execution.

4.1 ANN FOR CONTROLLING MATING RATE

Whenever a process is caught in local maxima we should try to reduce the rate of mating because, an increased amount of mating would be futile as the results will always be around the local maxima only. *Hence it would be best if we just decrease the rate of mating, and increase the rate of mutation in an attempt to drive the process away from the local maxima.* We found through a lot of experimentation that if a process hovers about the same best fitness value for around ten iterations, then we can safely conclude that the process is stuck in local maxima. Our problem primarily belongs to one of the common problems of neural sciences, namely the "Pattern mapping problem".

We use a feedforward neural network and train it using a *supervised learning* strategy. The network has 40 input nodes, two layers of 90 and 35 hidden units respectively and 5 output nodes. The networks are fully connected allowing arbitrary temporal and spatial windows. The numbers of units have been determined experimentally as demonstrated. The output nodes put together, basically would give us the binary value of an integer, \( x \), in the range 0 to 31. This indicates the fraction of the population that would be utilized for mating. At any state the input vector to the ANN, \( a_m \), can be calculated using the function \( \text{Calculate} (\text{Input}) \). The algorithm ensures that the value \( a_m \) correctly captures the variations in the fitness values for the previous ten iterations.

**Calculate\( \text{Input} () \)**

Find the best fitness value in the previous 10 iterations and arrange them in a non-decreasing order in an array Fitness [10].

- Integer counter = 0
- Integer Inputval [10]
- Inputval [1] = 1;
- For counter = 2 to 10 do
  - if ( Fitness[counter] > Fitness[counter-1] )
    - Inputval[counter] = Inputval[counter-1]+1;
  - Else
    - Inputval [counter] = Inputval [counter-1];
- The input vector, \( a_m \), is computed by converting the elements of the Inputval array into a 4-bit binary representation and concatenating them all in order

Return \( a_m \)

End

We use the "Generalized delta rule" or the *Backpropagation learning* to obtain an optimum set of weights for the network (training). These rules basically rely upon a gradient descent along an error function. In our case, we define the error function as the squared difference between the actual output obtained from the network and the desired output (desired output is the output given in the
input-output pair). The crux of the whole problem depends upon training the network well enough to perform the pattern-mapping task properly. We feed in various combinations of ten fitness values and based on the variations among them, we assign the desired output for each such input pattern.

The training data is obtained from experimentation using Benchmark circuits. At every stage for different mutation and mating rates, we performed circuit simulations and decided the optimal values for that particular sequence of fitness functions. By enormous experimentation, we obtained the training data. The error function is used to ensure whether the ANNs are well trained to meet the expectations. We essentially form an input and map it onto the desired output. In case of mating rate, depending on the inputs we classify the desired output into four classes, namely excellent, good, average and bad. The values of the desired output corresponding to them are the 5-bit binary representation of the integers 4, 8, 16 and 32 respectively. These values are obtained experimentally. We feed in the generated combination and compare the value provided by the ANN network, with the output vector of the neural network, while \( b_i'(m) \) denotes the \( i \)th bit of the desired 5-bit output vector.

### 4.2 ANN FOR CONTROLLING THE MUTATION RATE

The ANN for calculating the mutation rate is same as the previous one used to compute the mating rate. The only difference to be observed is while training the network. The training patterns, which are similar to the only difference to be observed is while training the network. The error function is similar to the one used for mating rate, depending on the inputs we classify the desired output into four classes, namely excellent, good, average and bad. The values of the desired output corresponding to them are the 5-bit binary representation of the integers 4, 8, 16 and 32 respectively. These values are obtained experimentally. We feed in the generated combination and compare the value provided by the ANN network, with the desired output generated by us. We apply the error function given to the desired output and the obtained output as follows:

\[
E(m) = \frac{1}{2} \cdot \sum b_i(m) - b_i'(m)^2
\]

where \( k \) ranges from one to the number of output units, in our case five, and \( b_i(m) \) denotes the \( i \)th bit of the 5-bit output vector of the neural network, while \( b_i'(m) \) denotes the \( i \)th bit of the desired 5-bit output vector.

### 4.3 ANN TO COMPUTE THE MIGRATION RATE

In case of parallel execution of the genetic algorithm one of the key issues to be taken care of, is the number of chromosomes migrating from one process to another. A stable migration rate might lead to migration of unnecessary chromosomes across the processes and thereby causing wastage of resources. We therefore define an ANN, which intelligently finds the migration rate depending on the state of the current population. The ANN essentially strives to capture the fact that, the number of chromosomes migrating to a process from other processes should be high and the number of chromosomes migrating out from this process should be low, if the process starts going towards local maxima. We keep two local neural networks in each of the processors which compute the in and out migration rates of the processes. The two ANNs are described in the following section.

The neural network for finding the number of chromosomes migrating into a process should increase the intake when the network detects the process to be tending towards local maxima. On the other hand, the neural network for finding the number of chromosomes migrating out of a process should be such that it should greatly limit the broadcasting of the individuals of a process tending towards local maxima. The reason being that the introduction of these individuals into other populations may trigger them to proceed towards local maxima. The ANN is exactly similar to the one that is used to compute the mating rate of a process. It is important to note that such an implementation requires very carefully designed inter-process communication software that deals with the synchronization issues in a lucid manner. These two networks are similar in nature to the networks in Sections 4.1 and 4.2.

### 5. EXPERIMENTAL RESULTS

The proposed genetic approach with ANN was simulated on a 6-node SGI system. An implementation of modular exponentiation based architecture proposed by Thomas Blum and C.Paar was chosen to illustrate the efficiency of spatial partitioning algorithm proposed in this paper. The application has large utilization in the area of cryptography. It is a resource efficient architecture and suitable for implementation in FPGAs. The total number of logic blocks required by the circuit is 14365, which is certainly more than the maximum number of logic cells present in a single ACEX1K FPGA. Given the Verilog description of the architecture, the proposed algorithm mapped the same onto a network of nine FPGAs of the ACEX1K type of Altera (Number EPlK30QC208-I). Each ACEX1K FPGA has 1728 logic blocks and 147 I/O pins.

Each module had sub-instantiations of various other modules in it for e.g. the module systolic array itself instantiates 129 adder units. The application needs more logic blocks, memory and pins for implementation than which can be supported by one FPGA. To overcome the limitation, we first split the modules into sub-modules such that each instantiation in the parent module becomes a separate module by taking the interactions between them into consideration. Finally, we got around 151 modules. These modules are fed to the spatial partitioning algorithm. The spatial partitioning algorithm combined various modules together optimally and provided a mapping of the application onto the FPGAs. In the given problem the spatial partitioning algorithm returned an optimal mapping over nine FPGAs. The run-times were similar to the results presented in table 5.1.

In order to test the parallel genetic approach on its own, a random test generator was implemented that provided us with an effective test generation methodology. The generator generates random task graphs and assigns a
random number of functional units and also a random value for memory usage for each of the generated tasks. However special care was taken that the parameters were in coherence with the actual values encountered while simulating any practical model. The results of such experimentation are presented in table 5.1.

In order to conduct the simulation in an effective manner, we had to decide upon the most important parameter for the ANN. That is basically the number of previous iterations to enable the network to use in order to enable it to make suitable decisions. Using the graph(5.3) we compared the fitness values of the solution for 10 tasks with different number of input units of the ANN. Note that if the input value is such that it essentially captures the best fitness values for the last 10 iterations, then we would require a total of 40 bits, or 40 input units, since each fitness value requires 4 bits. NI denotes the number of best fitness values of the previous iterations that our ANN takes as input. Hence if we wanted to consider the last 10 best fitness values before making a decision, then NI would take on a value of 10. It can be observed that for a value of NI=9 or 10 we obtain high fitness values. The reason for such a behavior is attributed to the fact that NI=9 or 10 strikes a correct balance, in the network, between the rate of mutation and mating.

We had to fix the initial population of our genetic algorithm as it directly affects the solution space exploration and thereby the runtimes. An initial population in the range N to 2N, where N is the number of tasks in the task graph, assisted in attaining an optimal balance between the runtimes and fitness values (graph 5.1 and 5.2). By performing similar experiments to fix the rate of mating, we obtained the rate of mating in the range of Ni4 to Ni8 as ideal. The reason for the anomalous behavior in the graph 5.4 and 5.5 is attributed to the fact that the rate of mating affects the runtime and the fitness values significantly. We also had to fix the rate of mutation of the algorithm. The rate of mutation is a vital parameter as it helps the process to get out of the local maxima. However a higher rate of mutation may also affect the good solutions. We found that a rate of mutation of N/8 is ideal for our algorithm. The rate of migration that plays a major role in determining the quality of the solutions, and the time consumed for their generation, was fixed at a value of N/8, which was determined by experiments (graph 5.6 and 5.7) of similar type as for rate of mating.

6. CONCLUSION

In this paper we have presented a completely automatic parallel spatial partitioner, which takes a HDL description of a big application and maps onto a given Multi-FPGA architecture. The crux of our approach is a spatial partitioner, which uses a parallel evolutionary approach to solve the problem. The suggested approach was successfully employed to spatially partition a reasonably big cryptographic application that involved a 1024-bit modular exponentiation and required more than 14365 logic blocks. Given the Verilog description of the architecture, the algorithm mapped the same onto a network of nine FPGAs of the ACEX1K type of Altera (Number EP1K30QC208-1). Each ACEX1K FPGA has 1728 logic blocks and 147 I/O pins. The genetic spatial partitioner was also experimented on random task graphs. The results are compared in Table 5.1 with the corresponding execution times of the genetic approach presented in [3]. The comparison shows that our program takes one-third of the time that is taken by the approach in [3], even for task graphs with small number (for e.g., 20) of tasks. Our approach becomes faster with more number of tasks, implying its scalability with increasing number of tasks. The fitness function and the genetic operators employed by the spatial partitioner also considers optimization of the inter-FPGA routing on a Multi-FPGA architecture, while arriving at a spatial partition.

Future work in this field may be oriented in enhancing the crossover operators and the fitness functions. We are currently working in controlling the evolution of the genetic approach using artificial neural networks to further reduce the time complexity of our algorithm.
REFERENCES


