POLITECNICO DI TORINO
Repository ISTITUZIONALE

Asynchronous scheduling/binding using a genetic approach

Original
Asynchronous scheduling/binding using a genetic approach / Blunno, I., Lazarescu, M.T.. - ELETTRONICO. - (2002). (MIPRO 2002 Opatija, Croatia May 2002).

Availability:
This version is available at: 11583/2507490 since:

Publisher:

Published
DOI:

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

16 October 2026

Asynchronous Scheduling/Binding Using a Genetic Approach

Ivan Blunno
Politecnico di Torino
Torino, Italy
blunno@polito.it

Mihai Lazarescu
Cadence Design Systems
Milan, Italy
mihail@cadence.com

Abstract

In this paper a new approach to operation scheduling and binding in asynchronous High Level Synthesis (HLS) is presented. We developed a *genetic algorithm* and integrated it inside Pipefitter [5], an existing tool for the automated synthesis of asynchronous circuit. A Control Data Flow Graph (CDFG), derived by Pipefitter from an HDL specification, is the input format for our algorithm. The designer can steer the search of an optimized solution either in the direction of minimum area or in the direction of maximum throughput. In the final solution each operation in the CDFG will be assigned to an operative unit (static binding), while the execution sequence will be determined run-time by the control unit (dynamic scheduling) in order to improve performances. This solution is then returned to Pipefitter that will complete the synthesis process down to the layout level.

1 Introduction

The transition from "System-on-Board" to "System-on-Chip" (SoC) approach can be considered a key element in the last few years microelectronic design trend. Basic devices like microcontrollers, DSP's, memories, FPGA's that once were placed on the same board can now be fitted onto a single die. However, the capability of reducing the transistor size thereby reaching a higher density of components on the same chip is two fold; smaller and faster devices can be designed and fit on the same die allowing to reach very high clock frequency and level of integration, but, at the same time, problems like Electro Magnetic Interference, interfacing and clock distribution are becoming more difficult to solve. Asynchronous systems seem to be better suited than synchronous ones for helping the issues mentioned above for a number of reasons:

- Operations are performed on a distributed time range, avoiding the simultaneous switching of logic gate and hence reducing current and voltage glitches on the power supply which are responsible for high frequencies Electro Magnetic Emissions.
- Asynchronous systems naturally adapt their speed and performance to the environment in which they are working and interfacing to each other can be easily done without the design of specific units dedicated to this purpose. This properties makes also the reuse of devices easier (e.g., reusable IP cores).
- Asynchronous devices are synchronized using local handshakes instead of a global clock signal, tackling the issue of distributing a low-skew clock signal to a large number of memory elements (i.e., flip-flops).

Despite the many advantages presented above, in the past years, asynchronous circuits have just been taken in consideration for very few niche applications. Asynchronous design is indeed made much harder than synchronous one due to the presence of hazards. In the last decade, however, the interest for the asynchronous world has significantly increased yielding to the development of some asynchronous Electronic Design Automation (EDA) tools. The algorithm presented in this paper is part of one of this tools. Section 2 is meant to give a general overview on existing synchronous and asynchronous HLS approaches. Our approach will be described in detail in sections 3 and 4. In section 5 the results of the implementation of our algorithm will be showed through an example. The influence of some parameters on the algorithm is shown in section 6. Finally, section 7 concludes the paper discussing some possible future improvements.

2 Asynchronous High Level Synthesis

High Level Synthesis is the design process where a behavioral description is mapped onto a register transfer level (RTL) representation that implements the specified behavior [1]. Three main tasks can be identified as part of HLS: allocation, scheduling, and binding. In this paper we focus our attention only on the automation of scheduling and binding tasks. In particular, only operative units (OU) and multiplexer binding is optimized, while each variable is synthesized as a separate register.

HLS basically consists in deciding which physical unit is responsible for each logical operation (binding) and at which time each operation has to be performed (scheduling). If the designer is looking into an unconstrained implementation for the circuit the two tasks can be considered separately from each other. For instance it could be possible to allocate an OU for each logical operation and execute them all sequentially. However, in most applications a small area occupation and a fast execution time are requested. In fact HLS aims at finding a trade off between two conflicting requirements:

- Using the minimum number of resources in order to reduce the area of the implemented circuit (resource-constrained scheduling/binding).
- Execute as many operations as possible concurrently in order to improve the performance of the implemented circuit (time-constrained scheduling/binding).

An approach that takes in consideration both the constraints is referred to as time- and resource-constrained scheduling/binding.

The existing synchronous scheduling/binding algorithms are based on the division of the time into control steps [4]. The use of a global clock signal guarantees all the control steps to have the same length. On this basis some algorithms, such

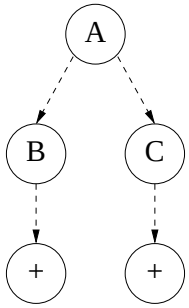


Figure 1: Fragment of CDFG

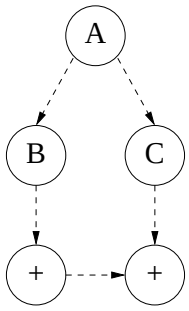


Figure 2: Fragment of modified CDFG

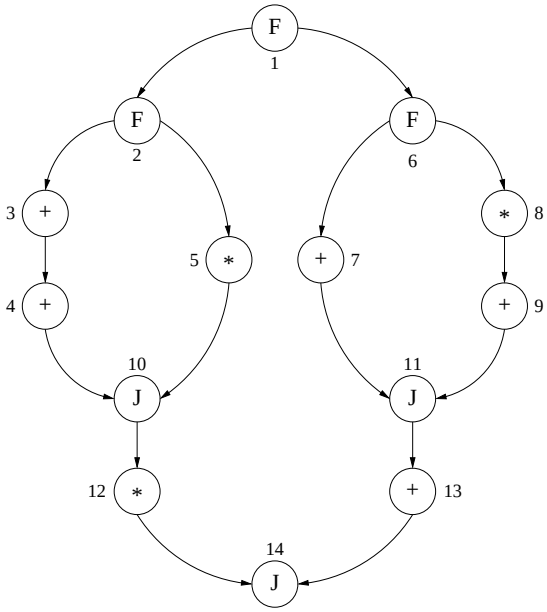


Figure 3: Example of CFG

as ASAP/ALAP scheduling, list scheduling, and force-directed scheduling have been developed.

Asynchronous operations, unfortunately, don't have a fixed execution time and therefore it is not possible to use approaches similar to those used in the synchronous case. Each operation can take any amount of time between a minimum and a maximum value. This characteristic makes asynchronous scheduling and binding much more complicated and explains why in the past only few attempts were made to solve this problem. A number of algorithms have been developed all based on the idea of traversing the CDFG as it was a timed Petri net [2, 3]. Time slots are associated with each operation and a static order for their execution is determined. The vagueness of this estimation however can bring to very inefficient solutions. For this reason our approach was based on a completely different idea: dynamically modifying the order of execution by means of arbitration.

Let's consider, for example, the fragment of Control Flow Graph (CFG) shown in figure 1. After operation A has been completed, two branches can be executed simultaneously. That means that operations B and C will start at the same time. The sum operations present on each branch can start at any time inside a minimum and maximum time range, depending on the end time of operations B and C. The duration of each operation can also be identified by a minimum and a maximum value. We can finally define two time slots identifying when the operations can occur: $S_1 = (t_{1_{min}}, t_{1_{max}})$ and $S_2 = (t_{2_{min}}, t_{2_{max}})$. If we have only one OU able to perform the sum operation, we have to decide which of the two sums has to be performed first. If the two slots are not overlapping, no conflict can occur between the two operations, hence no action has to be taken. Otherwise we have to decide which operation has to be performed first. In order to do this, we have to add a control edge in the CFG going from the first operation to the second one. We choose the order of execution trying to optimize the average case.

In figure 2 is shown the case in which the left sum will always be performed before the right one (i.e., operation B is in general faster than operation C). However, once an order of execution has been chosen, even when operation C terminates before operation B (and hence the right sum could start) we will have to wait for operation B to end and therefore for the left sum to be performed. In our approach, a non fixed order of execution is implemented by the controller. When either the B or C operation completes, the following sum will be enabled to execute. The other sum will wait until the first one has been completed. This kind of approach requires the use of arbiters and therefore a larger area occupation but, in general, it leads to much more efficient solutions.

3 The dynamic scheduling approach

A generic CFG is a set of *nodes* and *edges*. Each node represents an operation, while each edge represents a sequential relation between operations. Each operation must be performed by a physical operative unit, while each operative unit can perform more than one operation, but only one at a time. The designer has to specify how many resources are available and list which operations they can perform. These specifications are usually referred to as *resource allocation*.

The example shown in figure 3 can help us to explain how our approach works.

Four kinds of nodes are shown in this CFG: F nodes, representing *fork* operations, and J nodes, representing *join* operations are control nodes, while the nodes labeled + and * are sum and multiplication nodes.

Let's assume that we allocate two physical resources: one able to perform only sums, the other able to perform only multiplications. Thus, nodes 3, 4, 7, 9, and 13 will be assigned to the first OU unit, while nodes 5, 8, and 12 will be assigned to the second one. The problem of managing the conflicts between the operations that could compete for the same resource is solved using arbiters. When the algorithm establishes that two or more operations are assigned to the same OU and may be concurrent, an arbiter is generated. This arbiter will dynamically schedule the requests that will come to the OU. In this case, a five-input arbiter should be used for the adder and a three-input arbiter for the multiplier (i.e., one input for each node).

The same example can be complicated further if two OU's are able to perform a sum. In this case, we have more than one possible solution to the problem. We could, for example, choose the one found before, where only one OU and a five-inputs arbiter was used. Another possible solution could be

to use two adders: one for operations 3 and 4 and the other for operations 7, 9, and 13. In this case, we wouldn't need an arbiter for the first OU (since the two operations are executed one after the other) while we would need a two-inputs arbiter for the second one, where operations 7 and 9 could try to access the adder at the same time.

Finding the optimal solution for such a problem is a matter of choosing whether it is better to have one adder and one five-inputs arbiter or two adders and one two-inputs arbiter. In order to do this a cost function must be determined that provides the algorithm with a criteria to evaluate each solution.

It must be also taken in consideration that the resources shared by more than one operation could have to be provided with input multiplexers. Swapping the two operators (whenever possible) can help remove some multiplexers and reduce the total area for the circuit. For example, if we assign the two operations $Y = A + B$ and $Y = C + A$ to the same adder, a two-inputs multiplexer would be needed on each input. Swapping either the operands of the first sum or those of the second one would save one multiplexer (for both operations the register A would be connected to the same input of the adder).

4 A formal approach to the algorithm

Two nodes can be in conflict when they are on concurrent branches. In order to identify all possible conflicts between operations without traversing the graph every time the binding is changed, each node is labeled with all the fork nodes that precedes it and are still not closed by a join node. Therefore, a fork-label L_N of node N will be a list of couples (F_k, B_k); F_k is the fork node on whose branch the node N is executed, while B_k is the actual branch on which N is executed. Two nodes are conflicting when all of the following three conditions are met:

- They have one or more fork nodes in common in their fork-labels.
- The two nodes are not on the same branch.
- The two operations represented by the two nodes have been bound to the same operative unit.

In the example of figure 3, the node N_3 has the fork-label $L_{N_3} = \{(F_1, 1), (F_2, 1)\}$, while the node N_7 has the fork-label $L_{N_7} = \{(F_1, 2), (F_6, 1)\}$. If these two nodes are bound to the same resource, they are conflicting since in their fork-label the first element refers the same fork node but with a different branch value.

The labeling operation is performed only once at the beginning, since it depends only on the topology of the CFG and not on the binding choices performed by the algorithm.

A solution for the binding problem consists in assigning each node which performs an operation (i.e., non control nodes) to a physical resource and in deciding whether to swap the operators for that operation or not. We can define a *binding element* as a couple of variables, one representing the resource to which the node is assigned and the other to define if the input must be swapped for that operation: $B_j = \{R_j, W_j\}$. The swapping variable W_j can be assigned value 0 or 1 (swapped or not-swapped). Such a solution can be represented by a vector $V = \{B_{N_1}, B_{N_2}, ..., B_{N_l}\}$, where l is the total number of operation nodes.

As the number of nodes and resources increases, the number of solutions can became very large and exploring them all next to impossible. For example, a CFG with 15 nodes, each of which can be assigned to 3 possible resources (with 2 possible values for the swapping variable) has $(2 \cdot 3)^{15} \simeq 4.7 \cdot 10^{11}$ possible solutions! In these situations it is not possible to use traditional linear programming algorithms [6]. Self-adaptive algorithms (e.g., genetic, neuro-fuzzy, simulated annealing, etc.) on the other hand, are a possible way to tackle this problem.

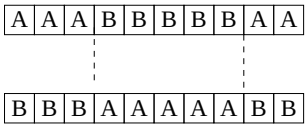


Figure 4: Crossover scheme

Genetic algorithms (GA) mimic the natural evolution process of a population of chromosomes, where those which are fit for the "environment" survive and generate new ones, while the others are deleted. The key aspect of this class of algorithms is the choice of a good representation for both the solutions of the problem and a good fitness function to evaluate them.

In our approach, the binding elements associated with each node play the role of *genes*, while a vector of genes (i.e., a solution) play the role of a *chromosome*. A set of chromosomes will be referred to as *population*. The environment which applies a sort of natural selection on chromosomes is played by the CFG itself in the form of the fork-labels introduced above.

The genetic algorithm can be summarized as follows:

1. **New population generation**. The initial population is generated randomly. A larger population increases the probability to find the optimal solution, but the computational effort increases, too. A similar observation can be made about the number of iterations of the process. How these parameters influence the efficiency of the algorithm will be discussed in section 6.
2. **Population evaluation**. The population is evaluated by estimating the number of resources, multiplexers and arbiters used. Each of them must be associated with a cost. A higher cost for OU's will result in a smaller circuit area, since solutions with fewer OU's will be preferred by the algorithm. On the other hand, higher cost for arbiters will result in higher circuit throughput, because the algorithm will favor solutions with more OU's and fewer arbiters (i.e., fewer conflicts). The choice of costs is therefore a means for the designer to direct the algorithm toward either a small area or a high throughput solution.
3. **Population sorting**. The chromosomes in the population are then sorted out. The worst ones are discarded and replaced by new ones generated by mating the best ones.
4. **Chromosomes mating**. The scheme used to mate chromosomes is the typical two-points crossover scheme shown in figure 4, where two indexes are randomly chosen and all the genes between them are exchanged.
5. **Chromosomes mutation**. In order to apply some random variations to the population, some small changes are carried out over chromosomes. This process can help the algorithm to avoid getting stuck around local minimums. The probability which characterizes this process is another parameter that will be discussed in section 6.

5 A simple example: an arithmetic unit

In this example, we will show the results of the use of our tool on a simple arithmetic unit, whose CFG is shown in figure 5. The genetic algorithm has been run on this specification 3 times with different costs and allocations:

Run 1. Two adders and two multipliers have been provided for the first run, and the cost of arbiters has been set to 0. As a result all the sums have been bound on one adder and all the multiplications on one multiplier. Two arbiters have

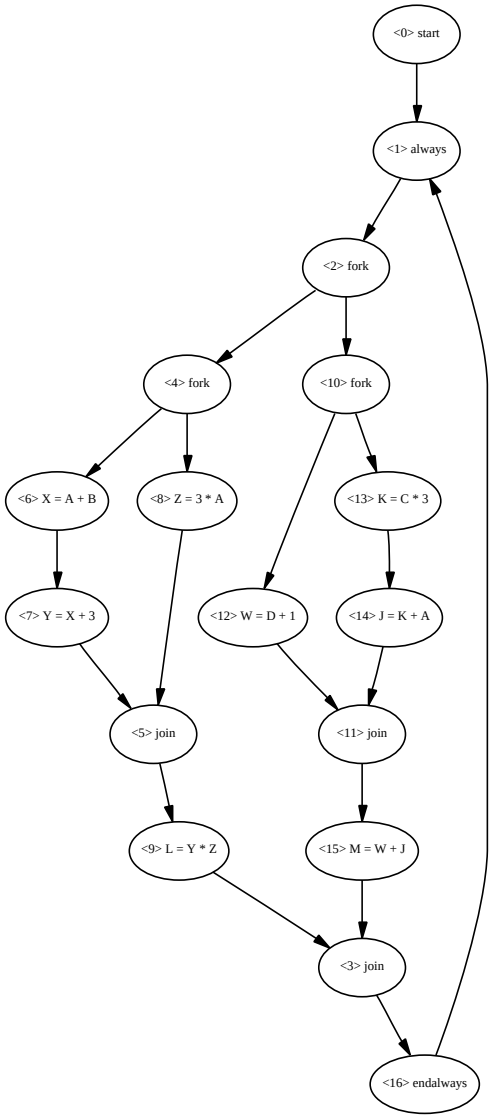


Figure 5: Arithmetic unit CFG

been specified: a five-inputs arbiter for the adder and a three-inputs one for the multiplier. The only interesting result is the swapping of the input variables for operations 13 and 14 in order to reduce input multiplexers area.

Run 2. For the second run, the same number of functional units have been provided as the first run. In this case, however, their cost has been set to 0, while the cost of the arbiters have been set to a greater value. The algorithm found a solution where both multipliers have been used in order to avoid conflicts (no arbiter was needed) and two adders have been used in order to minimize the number of conflicts. An arbiter was still necessary because of the conflict between operations 12 and 14.

Run 3. For the third run the same costs have been used for arbiters and functional units as in the previous run. One more adder has been allocated. A solution without conflicts and therefore without arbiters has been found by the algorithm.

All the scheduling/binding processes have been run using 100 chromosomes and 100 iterations, with a mutation probability of 5%. Each run took less than a second to complete on a 800MHz CPU. Table 1 summarizes the results for the example described in this section.

	Available		Used		
Run	**ADD**	**MUL**	**ADD**	**MUL**	**Conflicts**
1	2	2	1	1	7
2	2	2	2	2	2
3	3	2	3	2	0

Table 1: Results for examples of section 5

6 Quality considerations

The problem of finding the optimal allocation, scheduling, and binding for an asynchronous circuit is of class NP complete. The algorithms that explore the whole solution space run into serious efficiency limitations when attempting to solve problems of practical size. The use of heuristics emerged as an efficient mean to limit the computation load and improve the overall algorithm efficiency.

In this work we used genetic algorithms. Like many other heuristics, these algorithms are not guaranteed to reach the best solution. They have an incremental approach instead, attempting to improve the solution quality every new iteration. Moreover, using relatively few hardware resources, the genetic algorithms are able to achieve high quality solutions even with a coarse description of what the optimum is (e.g., they can converge even using just a criterion to discriminate any two valid solutions, without quantifying their individual quality).

The convergence of the genetic algorithms depends on many factors, such as: the representation chosen for the physical problem, the population size, the quality function, the algorithms used for searching the solution space (typically mutation and crossover), etc. Tweaking all these parameters by hand often prove to be time consuming and a heuristic work by itself [7].

However, without exploring these parameters, we cannot know if the algorithm converged on a local optimum, far from the overall best, nor even if the convergence speed (i.e., the use the algorithm makes of the hardware resources) is good [8, 9].

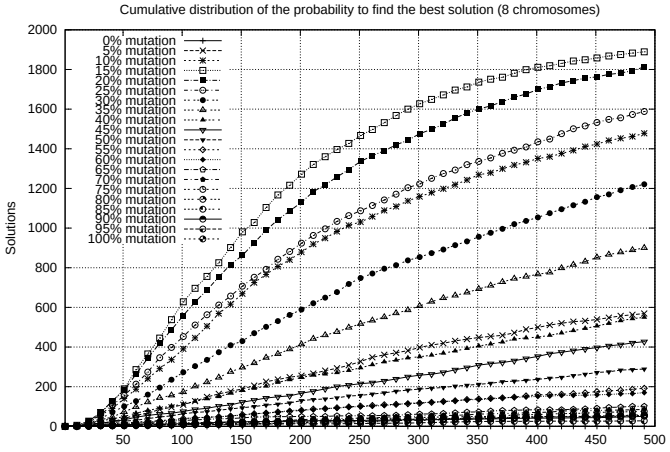
In the sequel we will present some experimental results regarding the influence of the variation of the genetic algorithm-specific parameters over the convergence and the probability to find the best solution. The goal of this exploration is to obtain a fully adaptive algorithm, able to autonomously tune its parameters on the class of problem to solve.

In our experiments, the same problem was solved for 2000 times (full scale on the Y axis), using a random starting point and 500 generations (full scale of the X axis). The sweep parameters were the mutation probability (0-100%) and the population size (4-1024 chromosomes). The best possible solution for the problem was known, in order to be able to evaluate the quality of the algorithms.

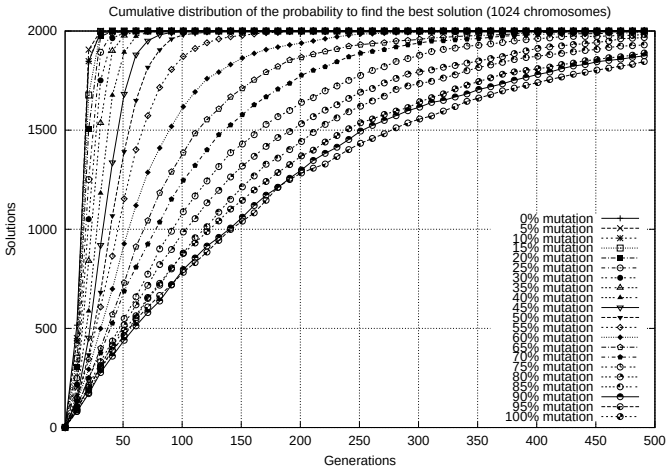
In figure 6 are reproduced the results for two characteristic cases. In these graphs, each point $P(x, y)$ measures how many runs needed less than or at most x generations to find the best solution. These graphs can also be seen as the cumulative distribution of the probability density to find the best solution.

In figure 6 (a), a very thin population with respect to problem size was used. Conceptually, this population is not able to maintain enough diversity to ensure a good exploration of the solution space, thus is prone to be trapped in local optimums. We can see that it needs a good influx of variations from outside (about 15% mutation ratio) to be able to perform enough solution space exploration to find the overall best solution.

On the other side, figure 6 (b) shows that a large population with respect to problem size is very likely to have intrinsically enough diversity for finding the best solution using a very few



(a)



(b)

Figure 6: Efficiency of the genetic algorithms with respect to the population size and probability of mutation (2D representation)

generations and low (if any) external diversity (mutation ratios very close to 0%).

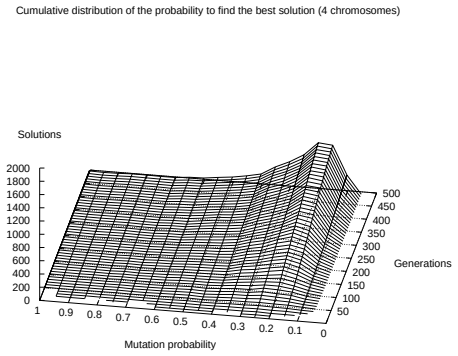
In both cases, as we would expect, a high mutation ratio (close to 100%) is perceived as a random factor, which can overwhelm the quality function feedback and evenly distribute the chances to find the best solution with respect to the number of generations. In figure 6 this can be seen as an almost straight line of constant slope.

The optimum of the genetic algorithm parameters should seek to minimize two negative effects:

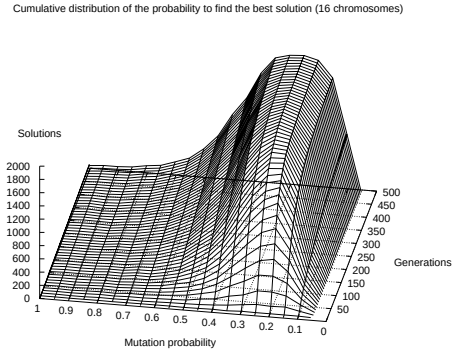
- the mutation probability should be chosen such way as to bring enough diversity to avoid local optimums on one hand, but also avoid disturbing the selection based on the quality function feedback;
- the population can drain out too many computation resources if oversize, while it may get easily trapped into local optimums if too thin.

In figure 7 are presented the same results using 3D graphs. This makes very easy to observe the impact the population size and the mutation probability have on the quality of the genetic algorithm.

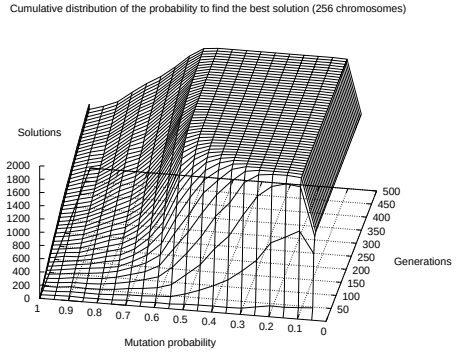
Figure 7 (a) uses a very thin population, of only 4 chromosomes. The lack of intrinsic diversity makes almost impossible



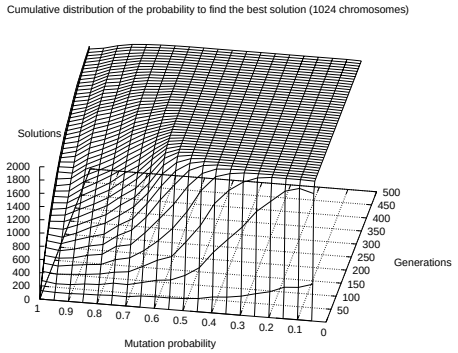
(a)



(b)



(c)



(d)

Figure 7: Efficiency of the genetic algorithms with respect to the population size and probability of mutation (3D representation)

to find the best solution, even after many generations, in absence of external variations (0% mutation probability). The best this thin population can do is for around 15% mutation probability, while for higher ratios the selection feedback from the quality function is cluttered by too much randomness.

Figure 7 (b) shows how a larger population (of 16 chromosomes) is capable to make good use of external variations (mutation ratios of 25-30%) to accelerate the search for the optimum solution. On this population size we can still see that there is not enough intrinsic diversity to find the best solution in absence of mutations, as well as the negative impact of too much randomness induced by very high mutation ratios.

Higher population sizes (256 chromosomes in figure 7 (c) and 1024 chromosomes in figure 7 (d)) exhibit both enough intrinsic diversity to find the best solution in absence of mutations, as well as better resilience to external random influxes for higher mutation ratios. However, large populations mean higher use of computational resources and a trade-off should be found.

7 Conclusions and future work

Asynchronous circuit allocation, scheduling, and binding is a very complex problem. In this paper, an effective method based on genetic algorithms for scheduling and binding was presented.

The algorithm can be directed to optimize the circuit area or the throughput. The hazards are avoided by automatic insertion of arbiters whenever necessary and the number of input multiplexers for shared resources is minimized as well.

Moreover, experimental results that illustrate the influence of main parameters on the genetic algorithm convergence are presented. These open the way to automatic parameter tuning at run-time, greatly improving the efficiency and quality of the algorithm.

References

[1] D. D. Gajski, Loganath, and Ramachandran, "Introduction to High-Level Synthesis", in IEEE Design & Test of Computers, Vol. 11, No. 4, Oct-Dec 1994, pp. 45-54.

[2] R. M. Badia and J. Cortadella, "High-Level Synthesis of Asynchronous Systems: Scheduling and Process Synchronization", European Design Automation Conference, Feb 1993, pp. 70-74.

[3] J. Cortadella, R. M. Badia, E. Pastor, and A. Pardo, "Achilles: A High-Level Synthesis System for Asynchronous Circuits", 6^{th} Workshop on High-Level Synthesis, 1992.

[4] R. A. Walker and S. Chaudhuri, "High Level Synthesis: Introduction to the Scheduling Problem", IEEE Design & Test of Computers, Vol. 12, Issue 2, summer 1995, pp. 60-69.

[5] I. Blunno and L. Lavagno, "Automated synthesis of micro-pipelines form Verilog HDL ", IEEE 6^{th} Symposium on Advanced Research on Asynchronous Circuits and Systems, April 2000, pp. 84-92

[6] K. H. Borgwardt, "The simplex method: a probabilistic analysis", Springer-Verlag, 1987.

[7] A. E. Eiben, R. Hinterding, and Z. Michalewicz. "Parameter control in evolutionary algorithms", IEEE Transactions on Evolutionary Computation, 3(2):124–141, 1999.

[8] S. F. S. Vincent, A. Cicirello, "Modeling GA Performance for Control Parameter Optimization", Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), Morgan Kaufmann, 10-12, pp. 235-242, 2000.

[9] T. Krink, B. H. Mayoh, and Z. Michalewicz, "A PATCHWORK Model for Evolutionary Algorithms with Structured and Variable Size Populations", Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, 13-17, Morgan Kaufmann, pp. 1321-1328, 1999.

[10] L. Davis, "Handbook of genetic algorithms", VNR, 1991.