# GSA: Scheduling and Allocation using Genetic Algorithm

Shahid Ali, Sadiq M. Sait, Muhammed S.T. Benten
King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia

## Abstract

*This paper describes a unique approach to scheduling and allocation problem in high-level synthesis using genetic algorithm (GA). This approach is different from a previous attempt using GA [1] in many respects. Our contributions include: a new chromosomal representation for scheduling and two subproblems of allocation; and two novel crossover operators to generate legal schedules. The approach has been tested on various benchmarks and results are compared with those obtained by other approaches such as simulated evolution, tabu search, HAL, SALSA II, STAR, etc.*

## 1 Introduction

Operation *scheduling* and hardware *allocation* are the two most important phases in the synthesis of circuits from behavioral descriptions. Scheduling and allocation are closely interrelated, but are usually dealt separately because of the complexity involved. Optimizing them separately gives suboptimal results because the possibility that the best designs (in terms of overall cost) may require suboptimum schedules and/or allocations is not considered. Several optimization techniques can be used for this purpose such as *simulated annealing* [2] and *integer programming* [3].

**Genetic algorithm** (GA) is another promising global optimization technique [4]. It works by emulating the natural process of evolution as a means of progressing toward the optimum. The algorithm starts with a *population* which consists of several solutions to the optimization problem. A member of population is called an *individual*. A *fitness* value is associated with each individual. Each solution in the population or an individual is encoded as a string of symbols. These symbols are known as *genes* and the solution string is called a *chromosome*. The values taken by genes are called *alleles*. Several pair of individuals (*parents*) in the population *mate* to produce *offsprings* by applying the genetic operator *crossover*. Selection of parents is done by repeated use of a *choice* function. A number of individuals and offsprings are passed to a new *generation* such that the number of individuals in the new population is the same as old population. A *selection* function determines which strings form the population in the next generation. Each surviving string undergoes *mutation* and *inversion* with a specified probability.

This paper describes a unique approach to scheduling and allocation using GA for data-oriented control/data flow graphs (CDFGs). The remainder of this paper is organized as follows: Genetic scheduling and allocation (GSA) is described in Sections 2 to 9. Section 10 presents the results of scheduling and allocation on various benchmarks and Section 11 concludes the paper and discusses some future work.

## 2 Cost function

In order to formulate scheduling and allocation as an optimization problem, a suitable cost function is required. The optimization technique will then attempt to optimize the value of this function. Since we want to optimize scheduling and allocation tasks jointly we need to incorporate both time related and hardware related terms in our cost function. The cost function $C$ that will be optimized by the genetic algorithm is given and explained below:

$$\begin{aligned} C \; = \; & W_{cs} * N_{cs} + W_{reg} * N_{reg} + W_{bus} * N_{bus} + \\ & W_{fu} * N_{fu} + W_{ic} * N_{ic} \end{aligned} \quad (1)$$

where, $W$ is the weight assigned and $N$ is the number. The subsripts $cs$, $reg$, $bus$, $fu$, and $ic$ corresponds to control steps, registers, buses, functional units and interconnections.

The algorithm starts with a specified upper bound on the number of control steps. During the optimization process the operations are assigned to control steps and functional units. Each functional unit has two inputs labeled as 1 and 2. Besides assignment of operations to control steps and functional units, variables are assigned to functional unit inputs. Constants are always assigned to same input as it helps in optimizing the number of interconnections. The number of registers and buses are optimized. Allocation of variables to registers and data transfers to buses is not actually made. The number of registers and buses as given by the final solution are optimal for the given schedule. Only a compromised estimation of the interconnection cost is used.

## 3 Chromosome

Genetic algorithms work on the coding of the problem rather than on the actual problem. This coding is known as chromosomal representation. Devising a good coding is particularly necessary for better design space exploration by the genetic algorithm. A given high-level specification of the description of the circuit is compiled using *lex* and *yacc* Unix utilities. A CDFG is then obtained from the compiled version. Any schedule should satisfy the precedence constraints implied by CDFG.

Since we want to combine scheduling and allocation into one optimization problem, the coding has to reflect this. This can be done only to a certain extent as finding an encoding for all the parameters is nearly impossible as there are too many constraints. The coding that is adopted is shown in Figure 1. Each gene has three values - control step number, functional unit number, and the number of the functional unit input to which the left variable of the operation is assigned. The first row in the figure gives the operation number to which the above three values correspond. This coding will be manipulated by the genetic operators. It is necessary to see why this coding is good enough to optimize scheduling and allocation tasks. With this representation the three subproblems are solved completely, namely, control step assignment, functional unit assignment, and functional unit input assignment. Given this information, the exact number of registers and buses can be found, whereas only a fair estimation of interconnection cost can be obtained. The chromosome in [1] has operation number in depth-first order and alleles corresponding to mobility (see below) values that are filled constructively. Special genes at the end of chromosome give the number of each type of functional unit.

| Operation Number | 1 | 2 | 3 | ... | ... | 8 | 9 |
|---|---|---|---|---|---|---|---|
| Control Step | 4 | 2 | 3 | ... | ... | 2 | 1 |
| Function Unit | 1 | 3 | 1 | ... | ... | 2 | 3 |
| FU input | 1 | 1 | 2 | ... | ... | 2 | 1 |

Figure 1: Chromosome.

## 4 Initial population

Good initial population is necessary for proper functioning of genetic algorithm reported in this research. Genetic algorithms work by adopting good structures from the population to generate better individuals. Therefore, initial population should be as diverse as possible. In this implementation the members of the initial population are created by using following four scheduling schemes: (1) As Soon As Possible (ASAP) scheduling, (2) As Late As Possible (ALAP) scheduling, (3) Mobility-down variation of ASAP, and (4) Mobility-up variation of ALAP.

ASAP scheduling assigns the operations in the earliest possible control steps, whereas ALAP scheduling assigns the operations in the latest possible control steps. For a given limit on control steps, mobility of each operation, which is the difference between ASAP and ALAP control step values, is calculated. The term mobility-down scheduling as used here means that operations are scheduled in ASAP manner within their mobility range taking care of the precedence constraints. Note that in a CDFG, mobility is used from upper nodes to lower nodes. If we reverse this sequence, we will get mobility-up scheduling. Functional units for each control step are assigned sequentially and randomly perturbed in the end. Assignment of left variable to functional unit input is done randomly.

## 5 Choice function

The first step to get new generation is to select parents on which genetic operators are to be applied. The selection of parents is an important step which affects the population in the new generation. Selection of fittest parents leads to premature convergence. Thus an appropriate choice function is required. This depends on how the fitness of a member of the population is calculated.

### 5.1 Fitness calculation

Genetic algorithm works naturally on the maximization problem whereas our cost function has to minimized. Thus the cost minimization problem is converted to a fitness maximization problem as follows. The maximum cost $C_{max}$ in the entire population is determined and each cost $c_i$ is subtracted from this value to get the fitness $f_i$ of individual $i$. Fitness scaling is used to avoid premature convergence. One method is *linear scaling* [4]. Linear scaling runs into problems in later runs of the genetic algorithm when most of the fitness values are close to each other and some lethal members have very low fitness values. This leads to negative fitness values. To avoid this situation *sigma* ($\sigma$) *truncation* was proposed [4]. All the fitness values are preprocessed to calculate modified fitness values $f_i'$ as follows:

$$f_i' = f_i - (f_{avg} - C_{mult} \times \sigma) \tag{2}$$

where $\sigma$ is the standard deviation of the population and $C_{mult}$ is the multiplying constant between 1 and 3. The negative values ($f_i' < 0$) are arbitrarily set to zero. After this truncation, linear scaling can proceed without the danger of negative results.

### 5.2 Sample space

Based on the scaled fitness value a probability is calculated for each individual. This is multiplied by the size of the population $n$ to get expected number of times an individual should be selected ($e_i$) as parent:

$$e_i = (f_i' / \sum_{i=1}^{n} f_i') \times n \tag{3}$$

A *sample space* is defined based on $e_i$ values. It consists of an array of records with two fields - a member identification number field and a probability field. For example if $e_j = 2.6$, then individual $j$ will receive three slots $(j, 1.0)$, $(j, 1.0)$, and $(j, 0.6)$ in the sample space. Assume that there are total of $m$ slots in the sample space. To select a parent a random number is generated between 1 and $m$ and the individual corresponding to that slot is selected as parent with the probability of that slot. This process is repeated until a parent is selected. According to this scheme fitter individual will get more slots in the sample space and have high chance of being selected. Note that the scheme still maintains diversity in the population because the selection is random over the sample space.

## 6 Crossover

The nodes in CDFG have precedence constraints that should not be violated when the crossover operator is applied. In [1], a simple two point crossover followed by a modified ASAP scheduling was proposed. This technique can produce schedules which are longer than the specified control step limit and is thus believed to take longer to find good schedules. Note that scheduling is performed each time the crossover is applied. We opted to have a crossover that will always give valid schedule rather than a crossover where scheduling has to be done separately. Given the coding as described in a previous section, it is a difficult proposition that is to be resolved. If we fix the order of nodes in the chromosome, a simple one or two point crossover will result in an invalid offspring chromosome. Following schemes are developed to generate valid offspring.

### 6.1 Alternating crossover

The term alternating crossover as used here means that given the same order of genes in both parents, we take genes from the two parents in the alternating sequence such that whenever there is a violation of precedence constraint we take the gene from the other parent but maintain the alternating sequence. It is found that if we put the genes in the reverse depth-first order such that successors are always on the left hand side of their predecessors, we can use the alternating crossover to generate valid offspring. It works because whenever we take a node that is to be scheduled all of its successors are already scheduled and thus we can check for any violations.

A working example of the alternating crossover is shown in Figure 2. Figure 2(a) shows the two selected parents ($p_1$ and $p_2$) for crossover and Figure 2(b) shows the resulting offspring ($os$) with genes labeled with the parent tag from which it is taken. It can be seen that there are no scheduling violations in this example. An example which results in such a violation is shown in Figure 3. Figure 3(a) shows two parents. As indicated in 3(b), during crossover we take alternating genes from each parent. At one point we can not take gene from parent 1 so this gene is taken from parent 2 but the alternating sequence is maintained and the next gene is also taken from parent 2.

### 6.2 Order crossover

It is found that alternating crossover is not able to adopt good structures from the parents. The main reason for this is that it works bottom up and things become constrained for upper operations. Thus the chances of mixing the genes becomes less. For this reason we started looking for a better crossover operator. Let us remove the restriction on the order of the genes in the chromosome. A simple *order crossover* works as follows. A cross point is randomly generated and genes on left side of one parent are copied to offspring in those positions. The other parent is scanned from left to right and these genes are stored in the remaining positions of the offspring in that order (Figure 4). This ensures that no genes are duplicated or missed.

Using this simple order crossover will of course give invalid schedules. The technique we adopted to avoid

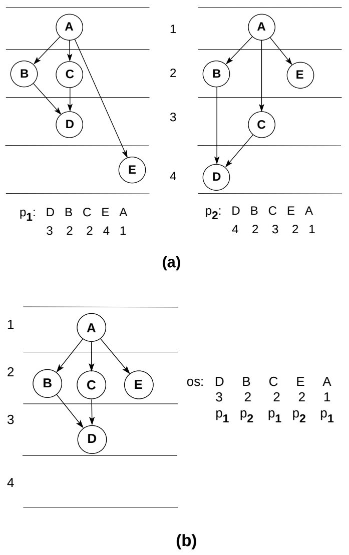

Figure 2: Alternating crossover example with no scheduling violations: (a) Parents; (b) Offspring.

invalid schedules is as follows. The cross point is randomly generated and left genes of one parent are copied to the offspring. This determines the schedule for some operations. Given schedule for some operations in CDFG, the ASAP schedule for the remaining operations can be determined. Those genes from the other parent which do not violate the precedence constraints are copied to the offspring and those which do violate are taken from the first parent. The ASAP values are used to check any violations. An example of this is shown in Figure 5. The cross point is between the third and fourth gene of parent $p_1$. The left three genes (A, C, F) from parent $p_1$ are copied to the offspring and the ASAP schedule for the remaining genes as induced by genes (A, C, F) is determined. Since none of the remaining genes from the other parent violate the precedence constraints they are copied without any trouble. This crossover is able to group together good structures in an offspring which is passed from generation to generation.

## 7 Functional unit violation

Functional unit and functional unit input assignments are also taken from the same parent. One can easily notice that sometimes this will result in concurrent assignment of the same functional unit to two or more operations in the same control step. One way to resolve this situation is to include a violation term in the cost function of Equation 1. Thus the cost function will then become:

$$\begin{aligned} C &= W_{cs} \times N_{cs} + W_{reg} \times N_{reg} + W_{bus} \times N_{bus} + \\ &\quad W_{fu} \times N_{fu} + W_{ic} \times N_{ic} + W_{viol} \times N_{viol} \qquad (4) \end{aligned}$$

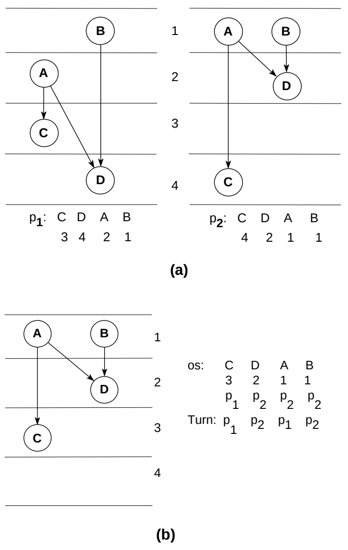

Figure 3: Alternating crossover example with scheduling violations: (a) Parents; (b) Offspring.

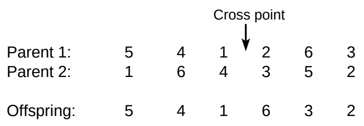

Figure 4: Simple order crossover.

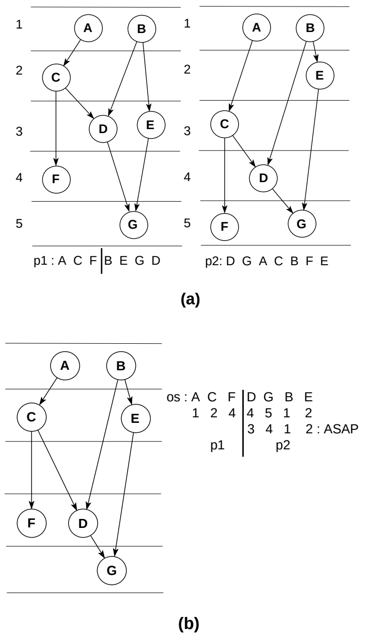

Figure 5: Example of order crossover tailored for scheduling: (a) Parents; (b) Offspring.

where, *viol* refers to violation. The other way around is to reassign the functional units for violating operations only. The advantage that one can think of for the first scheme is that one would expect the functional unit assignment to improve genetically. But if there are too many violations then it will undermine any genetic improvement. This is indeed the case as found by experiments. Thus the second scheme looks practical and is used in our current implementation.

## 8 Mutation

Three types of mutation operators are used in the present implementation. **Control step mutation** is the most important type of mutation. An operation is selected randomly. An attempt is made to either move it up or down. The direction is generated randomly. If it does not result in any violation, its control step value is changed. Control step mutation has very far reaching effects. It can produce better schedule and reduce the number of functional units, buses and registers. The second type of mutation is **functional unit assignment mutation**. An operation is selected randomly and a new functional unit number is generated. If this one is not used by any other operation in that control step then the functional unit assignment of the operation is changed to this one, otherwise another mutation attempt is made. In both cases mutation attempts are made a limited number of times. The last type of mutation is **functional unit input mutation**. An operation is selected randomly and if it is a commutative operation then the assignment of its left variable to the functional unit input is changed. The last two types of mutation help in reducing the number of interconnections.

## 9 Selection

Crossover is applied on the population with a specified rate. After the application of crossover is complete, we get an increased population consisting of parents and offsprings. We opted to have a fixed population size. Thus the next step is to transfer some of the individuals among parents and offsprings to the next generation. This is done by a selection function based on fitness value. We create another sample space in the same manner as discussed in Section 2 for the increased population. Thus the selection function is the same as the choice function. This is applied as many times as population size to get the new population. It is found that good results can be obtained if this scheme is combined with one or more of the following schemes: (1) Always selecting the best individual in the population, (2) Selecting a specified quantity of the best individuals, and (3) Selecting some specified quantity randomly. These schemes help in improving the search and maintaining the diversity in the population, which is necessary for search space exploration, and avoids premature convergence to the local optimum.

## 10 Results

Genetic scheduling and allocation (GSA) is tested on various benchmarks. Table 1 shows the results for differential equation benchmark. Table 2 shows the results for more complicated fifth order elliptic wave filter (EWF) benchmark. STAR system uses parallel data transfers, so that the bus comparison with this system is of little significance. The results shown for $17_{LU}$ control steps are for $L$oop $U$nfolding. Table 3 shows the results obtained for discrete cosine transform (DCT) benchmark. The results are compared with scheduling and allocation using tabu search (TSA) [5], simulated evolution (SE) [6], HAL system [7], SALSA II [8], STAR system [9], EMUCS system [10] and CATREE system [11]. Comparisons are given for number of control steps (CS), adders (+), multipliers (*), functional units capable of performing additon and subtraction (+/-), registers (Reg), and multiplexers or buses (Mx). The $p$ in (*) column stands for pipelined multiplier. It is assumed that addition takes one whereas multiplication takes two control steps.

## 11 Conclusions

Genetic algorithm is a promising optimization technique. This paper has presented its application to scheduling and allocation in high-level synthesis. Results obtained on data-oriented CDFGs are comparable to those obtained by other systems. Two new crossover operators are presented which can find application in many other areas. Future work will focus on designing a complete data path synthesis system

| System | CS | ALU | * | Reg | Mx |
|---|---|---|---|---|---|
| GSA | 8 | 1 | 1p | 5 | 4 |
| TSA | 8 | 1 | 1p | 5 | 4 |
| HAL | 8 | 1 | 1p | 5 | 4 |
| SE | 8 | 1 | 1p | 5 | 5 |

Table 1: Differential Equation Results.

supporting chaining of operations using GA.

**ACKNOWLEDGMENT**

The authors wish to acknowledge the King Fahd University of Petroleum and Minerals for support of this work.

| System | CS | + | * | Reg | Mx |
|---|---|---|---|---|---|
| GSA | 17 | 3 | 3 | 11 | 10 |
| TSA | 17 | 3 | 3 | 11 | 10 |
| SE | 17 | 3 | 3 | 11 | 11 |
| HAL | 17 | 3 | 3 | - | - |
| SALSA II | 17 | 3 | 3 | - | - |
| GSA | 17 | 3 | 2p | 11 | 10 |
| TSA | 17 | 3 | 2p | 11 | 10 |
| SE | 17 | 3 | 2p | 11 | 12 |
| HAL | 17 | 3 | 2p | - | - |
| CATREE | 17 | 3 | 2p | 12 | - |
| SALSA II | 17 | 3 | 2p | - | - |
| GSA | $17_{LU}$ | 2 | 1p | 10 | 8 |
| TSA | $17_{LU}$ | 2 | 1p | 10 | 8 |
| STAR | $17_{LU}$ | 2 | 1p | 11 | 5* |
| GSA | 18 | 2 | 2 | 11 | 8 |
| TSA | 18 | 2 | 2 | 10 | 8 |
| SE | 18 | 2 | 2 | 10 | 9 |
| SALSA II | 18 | 3 | 2 | - | - |
| GSA | 19 | 2 | 2 | 10 | 7 |
| TSA | 19 | 2 | 2 | 10 | 7 |
| SE | 19 | 2 | 2 | 10 | 11 |
| HAL | 19 | 2 | 2 | 12 | - |
| EMUCS | 19 | 2 | 2 | 12 | 12 |
| GSA | 19 | 2 | 1p | 11 | 8 |
| TSA | 19 | 2 | 1p | 10 | 7 |
| SE | 19 | 2 | 1p | 11 | 9 |
| HAL | 19 | 2 | 1p | 12 | 6 |
| STAR | 19 | 2 | 1p | 11 | 4* |
| SALSA II | 19 | 2 | 1p | - | - |

Table 2: EWF Results.

| System | CS | +/- | * | Reg | Mx |
|---|---|---|---|---|---|
| GSA | 7 | 6 | 8 | 19 | 18 |
| TSA | 7 | 6 | 8 | 19 | 18 |
| SALSA II | 7 | 6 | 8 | - | - |
| GSA | 7 | 8 | 4p | 21 | 21 |
| TSA | 7 | 6 | 5p | 19 | 17 |
| SALSA II | 7 | 8 | 4p | - | - |
| GSA | 8 | 5 | 6 | 15 | 17 |
| TSA | 8 | 5 | 6 | 15 | 16 |
| SALSA II | 8 | 5 | 6 | - | - |
| GSA | 8 | 5 | 4p | 17 | 15 |
| TSA | 8 | 5 | 4p | 16 | 16 |
| SALSA II | 8 | 5 | 4p | - | - |
| GSA | 9 | 4 | 6 | 15 | 14 |
| TSA | 9 | 4 | 6 | 14 | 15 |
| SALSA II | 9 | 4 | 6 | - | - |
| GSA | 9 | 4 | 3p | 13 | 14 |
| TSA | 9 | 4 | 3p | 13 | 14 |
| SALSA II | 9 | 4 | 3p | - | - |

Table 3: DCT Results.

## References

[1] N. Wehn, M. Glesner, and M. Held. A novel scheduling/allocation approach for datapath synthesis based on genetic paradigms. In *IFIP Working Conference on Logic and Architecture Synthesis, Paris*, pages 47–56, 1990.

[2] Srinivas Devadas and A. Richard Newton. Algorithms for hardware allocation in data path synthesis. *IEEE Transactions on Computer-Aided Design*, 8(7):768–781, 1989.

[3] M. Balakrishnan and P. Marwedel. Integrated scheduling and binding: A synthesis approach for design space exploration. In *Proceedings of the 26th Design Automation Conference*, pages 68–74, June 1989.

[4] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Inc., 1989.

[5] Shahid Ali. Scheduling and allocation in high-level synthesis using genetic algorithm. Master's thesis, King Fahd University of Petroleum and Minerals, Department of Information & Computer Science, Dhahran 31261, Saudi Arabia, June 1994.

[6] Tai A. Ly and Jack T. Mowchenko. Applying simulated evolution to high level synthesis. *IEEE Transactions on Computer-Aided Design*, 12(3):389–409, March 1993.

[7] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design*, 8(6):661–679, June 1989.

[8] Michael R. Rhinehart and John Nestor. SALSA II: A fast transformational scheduler for high-level synthesis. In *1993 IEEE International Symposium on Circuits and Systems*, pages 1678–1681, 1993.

[9] Fur-Shing Tsai and Yu-Chin Hsu. STAR: An automatic data path allocator. *IEEE Transactions on Computer-Aided Design*, 11(9):1053–1064, September 1992.

[10] C. Y. Hitchcock and D. E. Thomas. A method of automatic data path synthesis. In *Proceedings of the 20th Design Automation Conference*, pages 484–489, June 1983.

[11] C. H. Gebotys and M. I. Elmasry. A VLSI methodology with testability constraints. In *Proceedings of the 1987 Canadian Conference on VLSI(Winnipeg)*, October 1987.