Genetic Algorithms for High-Level Synthesis in VLSI Design

C Mandal[†] **P P Chakrabarti**[‡]

{chitta, ppchak}@cse.iitkgp.ernet.in

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
WB 721302, INDIA

Abstract— **VLSI design involves a number of steps such as system-level design, high-level synthesis (HLS), logic design, test generation and physical design. All these steps involve combinatorial optimizations that are NP-complete. Genetic algorithms (GA) have been used to solve many problems in VLSI design. HLS is the crucial step where the architecture of the system is decided upon.**

We have worked on several problems relating to high-level synthesis, and developed GAs for them. In this paper we describe our GAs for the following three problems and describe some general methods that we have used in these GAs to enhance their operation.

- **Minimum node deletion (MND).**
- **Allocation and binding for data path synthesis.**
- **Scheduling, allocation and binding for the synthesis of structured architectures.**

All of the above problems are NP-complete. We have used the following techniques to enhance the operation of the GA:

- **Population control to enforce diversity within a relatively small population size.**
- **Solution completion using approximate algorithms to generate superior valid solutions.**
- **Selection control to reduce crossover between incompatible members.**

These GAs have been tested on the usual benchmarks and the results have been found to be acceptably good. The enhancing techniques we describe here are of a general nature and may be used with other GAs to produce better results.

Keywords—**VLSI Design, High-Level Synthesis, Data path Design, Structured Architecture, Genetic Algorithm**

I. Introduction

In this paper we describe GAs to solve three problems that arise in high-level synthesis. These problems are of increasing complexity and are: the minimum node deletion problem, allocation and binding for high-level synthesis and the synthesis of structured architectures.

We first briefly highlight the important steps in high-level synthesis. The aim of high-level synthesis is to enable the designer to start designing at a higher level of abstraction. A digital system typically consists of a number of storage devices, buses, functional units to perform logical and arithmetic operations, and interface ports. The storage devices could be individual registers or small memories of one or two ports. A regular single port memory has an address port and a data port. A dual port memory has two sets of address and data ports, and permits two concurrent accesses to its memory locations. Such a system is usually driven by a clock and its operation is typically characterized by the events that take place in each clock cycle. Current design flows usually require the designer to determine each of these events, ie. in each time step, the data that must be put on each bus, the operations that must be performed and the registers where newly created results must be stored. This is a cumbersome process, and it is desirable to specify the design at a higher level of abstraction, such as a program written in a common high-level language (say C). Such a program may be translated into a set of directed acyclic graphs as the one shown in figure 1. Given such a high-level description and some design parameters, we would now like the register-transfer level (RTL) design to be automatically generated.

This would require principally the following steps to be performed: scheduling of operations into specific time steps, formation of functional units (FU) to execute operations, formation of a storage configuration to store values and interconnect configuration. The overall operation of the system is as follows: The operation scheduled in a particular time step will execute on a designated functional unit. The operands for the operation have to be fed using some of the buses to the functional unit. The result has to be transferred using a bus to a storage location designated to store the result. Thus, the scheduling problem is to determine which operations are to be scheduled in which time step. The allocation and binding problem is to determine the RTL data path components (storage elements, functional units) and their interconnections using interconnect elements. The port assignment problem comes in as part of the interconnect configuration if dual-port memories are used. All of these are computationally hard (NP-complete) problems. There are also no particularly good heuristic methods to solve these problems in an approximately optimal manner. In this context the genetic algorithm was applied to solve these problems. We now move on to introduce the specific problems dealt with in this paper.

The first problem of node deletion arises in the context of port assignment of dual port memories. It is necessary to assign accesses to the memory in each time step to one of the two ports. The goal is to make an assignment that leads to minimum interconnect. In the second problem

[†]C Mandal acknowledges Kingston University, UK for partial support of this work.

[‡]P P Chakrabarti acknowledges the Department of Science and Technology, Government of India for partial support of this work.

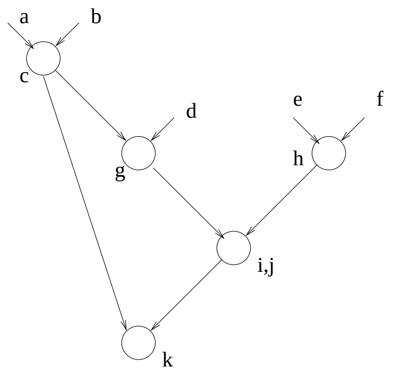


Fig. 1. A sample directed acyclic graph.

of allocation and binding we start with a register transfer level (RTL) design specification and wish to assign the operations to functional units, storage variables to memory and transfers to buses. The goal is to perform this assignment to minimize the overall cost of the functional units, registers and estimated interconnect cost. The last problem is to start with an unscheduled data flow specification and have a RTL specification and a structured data path to satisfy the RTL specification. The goal is to be able to have a RTL specification within a specified number of time steps and to have a minimum cost structured data path to satisfy it.

In all these three cases we have had to rely on enhanced crossover mechanisms. The crossover used for the minimum node deletion (MND) problem was the best in the sense that we were able to get a theoretical guarantee that it would generate an optimum solution with high probability. In case of MND the crossover alone was able to contribute heavily towards obtaining optimal solutions. The next two problems on data path synthesis are of a more intricate nature, where it would be easy for a more traditional crossover to generate mostly infeasible solutions. An overwhelming proportion of infeasible solutions places a heavy demand on computing resources to run the GA. We worked around this problem by using a crossover mechanism which would always generate a feasible solution after inheriting attributes from parent solutions. Another problem was the proliferation of copies of a slightly better solution in the population, thereby killing the diversity within the population. This problem was solved by using a replacement scheme that forcibly retains some solutions in the population to enforce a minimum degree of population diversity.

Our experience with these three GAs indicates that a combination of a robust crossover and a diversity sustaining replacement mechanism have helped us solve three combinatorial problems of increasing complexity relating to high-level synthesis, with satisfactory results.

The rest of the paper describes the problem formulation and the corresponding genetic algorithms for the above three problems, starting with MND and then allocation and binding and finally the structure architecture synthesis problem.

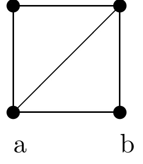


Fig. 2. A non-bipartite graph.

II. Minimum node deletion

On-chip dual port memories are increasingly being used as architectural elements. When multi-port memories are used it becomes necessary to carefully assign the accesses to its cells over its ports so as to minimize the cost of interconnecting the memory with other circuit elements. This is the origin of the port assignment (PA) problem. PA for dual ports memories is particularly useful as these are the most commonly used multi-port memories. Optimal PA for dual port memories directly maps on to the minimum node deletion problem (MND) as follows: Certain nodes (which may be a register, a port of an ALU, etc.) need to be directly connected to one or more ports of the dual port memory to be able to send or receive data from it. If there are two nodes that need to transfer data to or from the memory in the same time step then they must have connections to different ports of the memory. Otherwise, they may possibly be connected to the same port of the memory. This situation may be represented by putting an edge between nodes that transfer data to or from the memory in the same time step. We thus have a graph whose nodes are the nodes in the circuit that need to be connected to the memory and has edges as just described. Connection to a port of the memory may be represented by assigning a color to that node. Since there are only two ports in a dual port memory, we may use only two colors. We prefer to connect a node to just a single node of the memory to reduce interconnection costs. Thus we try to color the nodes of our graph with two colors. In some cases we shall succeed – when the graph is bipartite. It is easy (in polynomial time) to determine whether a graph is bipartite. If not then some of the nodes will have to be connected to both the ports of the memory. We may delete these nodes from the graph and check whether the rest of the graph is bipartite. To reduce our interconnection costs we try to get a bipartite graph by deleting a minimum number of nodes – hence the MND problem. We shall now discuss a GA for MND.

A graph is said to be bipartite if it can be coloured using just two colours. The node deletion problem is to determine, for a non-bipartite graph, the smallest set of vertices that need to be deleted to make the graph bipartite.

III. Algorithm for Minimum Node Deletion

A. Solution representation

Convention bit string representations sometimes mask the structure inherent in the solution. Davis [1] pointed out that employing non-bit-string solutions for specific optimization problems is advantageous. For MND we have found it convenient to represent the solution directly as three sets. The first two sets contain the vertices corresponding to each of the two colours. The third set contains vertices that could not be two coloured and which are chosen for deletion.

Example 1: For the graph of figure 2 one solution could be $< \{b,d\},\{c\},\{a\} >$. The last set of the tuple corresponds to the set of deleted vertices which are connected to both the ports. Another solution could be $< \{a\},\{c\},\{b,d\} >$. □

B. Fitness function

The fitness function is defined as

$$g = |set\ \ of\ \ deleted\ \ vertices| \tag{1}$$

Minimization of g is the objective.

C. Initial population generation

Each member of the initial population is a randomly generated valid solution. While generating a solution, each vertex of the graph is tested for possible membership in one of the two partially constructed colour classes. In case of a failure in inclusion to one colour class, the membership for the other colour class is checked. In case of a repeated failure the vertex is marked for deletion. The sequence in which the vertices are visited while constructing a solution is also random.

D. Reproductive plan

The parent selection policy, crossover and mutation operations constitute a reproductive plan. For a particular generation we have selected two parent solutions for crossover to generate each child solution, randomly without replacement from the current population. Only one offspring has been generated as a result of a single reproduction. The number of reproductions performed in one generation is determined by the crossover rate.

The crossover is performed as follows. The offspring solution first inherits a subset of a colour class from one of the parent solutions. During crossover, larger colour classes are chosen for inheritance with a higher probability, while smaller ones are selected with a lower probability. The inherited class is now augmented with uncoloured vertices according to the algorithm in figure 3. The augmentation is based on a graph colouring algorithm presented in [2]. The algorithm is successively applied to the two inherited colour classes. The augmentation algorithm is as follows. Let V be the set of vertices of the graph. Let the initial inherited colour class on which the algorithm is applied be X. $\Gamma(X)$ is defined as the subset of $V - X$, such that for each element of the subset there is an element in X to which it is connected by an edge. The effect of the steps (1) and (5) of the augmentation algorithm is to remove from Y all those vertices which have an edge with at least one vertex of X. The newly defined Y has the property that any of its vertices can be augmented to X. The process of augmentation continues till the set Y becomes empty.

```
1.  $Y = V - \Gamma(X)$
2.  while $(Y \neq \emptyset)$
3.  {   among all $y \in Y$ let $x$ have
        the minimum degree in $Y$
4.      $X = X \bigcup \{x\}$
5.      $Y = Y - (\{x\} \bigcup \Gamma(x))$
6.  }
```

Fig. 3. The augmentation algorithm

The vertex in Y that is to be selected is determined by the simple heuristic of step (3) of the algorithm.

After the first colour class of the offspring is formed, the second colour class is also formed similarly. The set from which the second colour class is inherited is as follows. Let P_1 and P_2 be the two parents. Let S_{11} and S_{12} be the two sets in P_1. Similarly, let S_{21} and S_{22} be the two sets in P_2. Suppose that the first colour class had been formed from S_{11} of P_1. Let

$$c_0 = \frac{|S_{11} \cap S_{21}|}{|S_{11} \cup S_{21}|} \qquad and \qquad c_1 = \frac{|S_{11} \cap S_{22}|}{|S_{11} \cup S_{22}|}.$$

The values c_0 and c_1, $0 \leq c_0, c_1 \leq 1$ represent the affinity of S_{11} with S_{21} and S_{22} of P_2, respectively. Let $S = S_{2(i+1)}$, such that $c_i \leq c_{1-i}, i \in \{0,1\}$. S is the colour class of P_2 which is less affine to S_{11}. Normally the second colour class of the offspring is inherited from S, otherwise inheritance is from S_{12}. The vertices that could not be included in the two colour classes of the offspring are placed in the third set for deletion. This method of construction ensures that each solution constructed is a valid solution.

During crossover, inheritance of only a part of the colour class may be considered equivalent to the process of mutation. The *mutation rate* instead of being kept fixed, is varied with the standard deviation of the fitness value of the candidate solutions. Should the fitness function values tend to become uniform the mutation rate goes up.

E. Replacement policy

In our implementation all the offspring generated in the current generation replace the maximum cost solutions in the current population. This corresponds to the survival of every new offspring generated for at least one generation. Any existing better solution found survives since the worst solutions are always replaced. This corresponds to an elitist policy.

F. Deceptability of the Crossover

It has been shown in [3] that if the crossover operation is free of type II deceptability, then the GA may be expected to lead to the optimal solution. The crossover would be free of type II deceptability, if on crossing two solutions with high fitness value, the resulting new solution also has a high fitness value [3]. The crossover used here has not been proved to be strictly free of type II deceptability, but it is likely to be so. We show this by probabilistic arguments.

First a probabilistic analysis of the augmentation algorithm is presented.

Let the two colour classes be B_1 and B_2 and let the third set be D. The analysis is applicable for random graphs satisfying the following:

1. $|B_1| = |B_2| = m$.
2. $|D| = k$, thus $|V| = 2m + k$.
3. By definition B_1 and B_2 are independent, (that is, there are no edges between any pair of elements of a particular set). An edge may be present between a member of B_1 and a member of B_2 with probability p. An edge may be present between a member of B_1 and a member of D with probability p. An edge may be present between any two members of D with probability p.
4. With probability $q = 1 - p$ the edge in question is absent.

Let $X \subset B_1$ and $|X| = r$.

Y, as computed in step (1) of the augmentation algorithm is $(B_1 - X) \bigcup (B_2 - \Gamma(X)) \bigcup (D - \Gamma(X))$.

It may be shown that $|B_2 - \Gamma(X)| \approx mq^r$.

Similarly it may also be shown that, $|D - \Gamma(X)| \approx kq^r$.

Let $d_Z(v)$ be the expected degree of $v \in Z \cap Y,\ Z, Y \subseteq V$, in Y.

Let $d_{B_1} = d_{B_1}(v) \approx mpq^r + kpq^r$. This is the expected degree of a vertex of $B_1 \cap Y$ in Y.

Let $d_{B_2} = d_{B_2}(v) \approx (m - r)p + kpq^r$. This is the expected degree of a vertex of $B_2 \cap Y$ in Y.

Let $d_D = d_D(v) \approx (m - r)p + mpq^r + (kq^r - 1)p$. This is the expected degree of a vertex of $D \cap Y$ in Y.

$$d_2 = d_D - d_{B_1} = (m - r)p - p = (m - (r + 1))p \qquad (2)$$

$$d_1 = d_{B_2} - d_{B_1} = p(m(1 - q^r) - r) \qquad (3)$$

It is desirable that $d_1 > 0$ and $d_2 > 0$, for this will ensure, with high probability (**whp** [2]), that if X turns out to be a subset of B_1 or B_2 then it will be augmented by members of B_1 or B_2, respectively. To satisfy $d_2 > 0$ it is necessary that $r < (m - 1)$. Thus when the subset is being augmented with the last element, the algorithm is not expected to guarantee the selection of the correct element. However, in the stochastic environment of the GA this does not pose a serious problem.

To satisfy $d_3 > 0$ it is necessary that

$$m \ > \ \frac{r}{1 - q^r} \qquad (4)$$

For $r = 1$, it is necessary that $m > \frac{1}{p}$. For somewhat large values of m and not too sparse graphs this will be satisfied. Also, for $r \ = \ m - 1$,

$$m \ - \ \frac{m - 1}{1 \ - \ q^{m-1}} = \frac{1 \ - \ mq^{m-1}}{1 \ - \ q^{m-1}}$$

Again for somewhat large values of m, this expression is positive. Now consider the function $\frac{x}{1 - q^x}$, $x > 0$.

$$\frac{d}{dx}\left(\frac{x}{1 - q^x}\right) = \frac{1 \ - \ q^x(1 \ - \ x\ln q)}{(1 - q^x)^2} =$$

no. of. nodes	edge prob.	deletion [a] by GA2	cost u.b.
20	0.082	0.000	0
20	0.107	0.000	0
20	0.154	0.903	1
20	0.250	3.483	4
20	0.400	6.903	7
20	0.500	8.903	9
20	0.650	10.70	11
30	0.082	0.000	0
30	0.107	1.000	1
30	0.154	3.677	4
30	0.250	8.677	9
30	0.400	13.61	14
30	0.500	16.80	17
30	0.650	19.70	20
40	0.082	1.709	2
40	0.107	3.806	4
40	0.154	8.516	9
40	0.250	15.35	16
40	0.400	22.29	23
40	0.500	25.48	26
40	0.650	28.77	29
60	0.082	7.451	8
60	0.107	12.38	13
60	0.154	20.38	21

[a] The deletion in each line has been reported as the average obtained by running GA2 on 30 individual random graphs with known upper bounds.

TABLE I
PERFORMANCE OF GA2 ON RANDOM GRAPHS WHERE AN UPPER BOUND ON THE NUMBER OF NODES TO BE DELETED IS KNOWN

$$\frac{1 \ - \ e(\ln(ez^x)/(ez^x))}{(1 - q^x)^2}, where \ \ z = 1/q.$$

Depending on the value of q the derivative may be negative for small values of x, for larger values of x it is positive and approaches 1. Thus, if (4) is satisfied for $r = 1$ and $r = m-1$, then it will be satisfied for all intermediate values of r. This, in general, will not be true for all members of the population. However, in the stochastic environment of GA it will be satisfied by at least a few members of the population.

It is reasonable to assume that solutions whose colour classes are *close* to the colour classes of an optimal solution will have relatively high fitness values. It has also been ensured that the augmentation algorithm will, *whp*, augment an inherited colour class which is a subset of an optimum colour class with the appropriate vertices. Thus, solutions with high fitness values, when combined through crossover should also result in solutions with high fitness values.

IV. Experimentation for Dual Port Memory PA

The GA for MND is referred to as GA2 here. The experimentation consists of two parts. The first part of the experimentation deals with the testing of GA2, while in the second part the quality of the estimate has been tested.

GA2 has been implemented in C in the UNIX environment on a SUN 3/280. It has been tested on graphs of both small and relatively large numbers of vertices. While testing GA2 on the smaller graphs, it has been possible to compare the results against the exact solutions. This testing has been done on random graphs of the type $G_{n,p}$, where n is the number of nodes in the graph, and each of the possible $\frac{n(n-1)}{2}$ edges is present with probability p. Twelve sets of random graphs of ten, twelve, fourteen and sixteen vertices with edge probabilities of 0.3, 0.5 and 0.7 were generated. The testing for each set was carried out on thirty graphs of that type. For these small graphs (*up to 16 vertices*), in each case the GA was able to obtain the optimal solution.

For the relatively larger graphs it was not feasible to find the exact solution for comparing the result obtained by GA2. Therefore, a different method of testing has been employed here. GA2 was now tested against random graphs which have been generated such that the upper bound on the number of nodes to be deleted is known. The method of constructing random graphs with a known upper bound on the number of nodes to be deleted has been explained in [4], [5]. The test results for graphs of 20, 30, 40 and 60 vertices and various edge probabilities have been presented in table I. In this table the first column shows the number of vertices, $|V|$, in the graph and the second column the edge probability p'. For each $< |V|, p' >$ combination of a row of the table thirty random graphs of that type were generated so that no more than the number of vertices specified in the last column, the upper bound, needs to be deleted to render the graph bipartite. The third column is the average number of vertices deleted by GA2. It will be observed that deletion of GA2 is very close to the upper bound, occasionally doing slightly better. A similar method of testing, for another graph problem (the graph 3-colourability problem), has been used in [6].

The above results indicate that an efficient genetic algorithm has been developed to solve the dual port memory PA as the minimum node deletion problem. We have also been able to obtain a theoretical guarantee that the GA will find an optimal solution.

V. Allocation and binding for data path synthesis

The basic input to data path synthesis (DPS) is a set of operations and their interdependencies. These are typically expressed as directed acyclic graphs of operations [7]. DPS involves scheduling of operations followed by allocation and binding. After scheduling has been done the resulting design is called a Register Transfer Level (RTL) specification, which indicates data transfers to and from registers and operations performed on the data, in each time step. The latter step of allocation and binding consists of several sub-tasks which include determining the mix of functional units, grouping variables and assigning these variable clusters to storage units, memory port assignment when multi-port memories are used in the design, mapping operations to the functional units and mapping transfers to buses, when buses are used. The problem treated here is concerned with the allocation and binding aspects of DPS. This is a computationally hard problem and many of its sub-problems are NP-complete [8]. These concerns have motivated us to develop a Genetic Algorithm (GA), called GABIND, for synthesizing optimized data paths from a given scheduled data flow graph. GABIND builds on previously developed successful heuristics, such as force [9], by incorporating them into the GA.

GABIND performs the following tasks: formation of functional units (FU) (see example 2), binding operations to FUs, binding transfers to buses, allocating storage, binding variables to storage units and allocating switches to interconnect FUs and memory units to the interconnecting buses. An important aim of developing GABIND was to be able to satisfy all transfers using a given number of buses and not relying on an unpredictable number of point-to-point interconnections. The output is an optimized data path which correctly implements the computation given to GABIND in the form of scheduled data flow graphs. The optimization is performed to jointly minimize the cost of the FUs, the storage units and the switches for interconnection used in the data path. Specifications for subsequent synthesis of the controller are also generated.

The rest of the discussion of allocation and binding is organized as follows. The architectural considerations used for the synthesis scheme are described in section VI. The GA to solve the problem is described in section VII. GABIND employs an algorithmic crossover, which is described in section VIII. The experimental results and conclusions are given in sections IX.

Example 2: We give a small example of a schedule and a possible data path (figure 4) to implement the schedule.

```
time |operations on      | operations on
step |functional unit 1  | functional unit 2
-----+-------------------+------------------
  1  |   x = dx +  x     |   v1 = dx *  x
  2  |  v0 =  u *  3     |   v6 =  u * dx
  3  |  v2 = v0 * v1     |   v3 =  y *  3
  4  |         x <  a    |    y =  y + v6
  5  |  v4 =  u - v2     |   v5 = v3 * dx
  6  |   u = v4 - v5     |
```

In this data path, two functional units (FU-1: `<+,-,<,*>`, FU-2: `<+,*>`) and five storage elements have been used. The storage elements are as follows: one single port memory of two cells to store `dx` and `a`, a single register to store the constant 3 – shown as a single port memory one cell and three dual port memories to store the variables indicated in the diagram. Five buses and required interconnection links from the devices to the buses are present. "Copper" contact between the buses and the links are indicated by filled circles, while switched contacts between them are indicated by hollow circles. □

VI. Underlying Architectural Considerations

The optimization performed by GABIND is based on the architectural considerations described in this section. GABIND takes as input a scheduled data flow graph

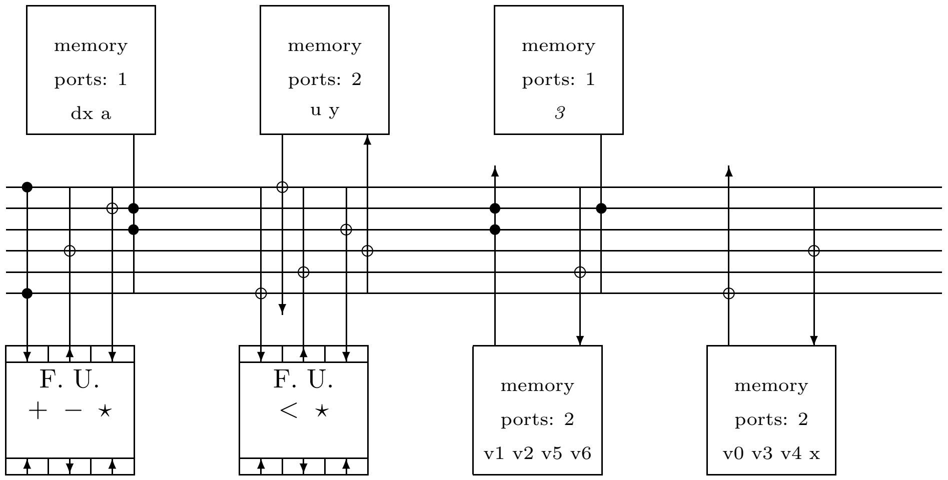


Fig. 4. Diffeq. data paths for two f.u.'s and six time steps.

(SDFG) of the operations. It accepts the *number of FUs* and *buses* as user specified design parameters. The former indicates the total number of sites where operations may be performed while the latter indicates the total number of paths for carrying data transfers. The minimum number of FU sites should equal the maximum number of operations that are scheduled to execute concurrently in the given SDFG. An additional FU site generally leads to an overhead in the interconnection and the control logic. The capability of an FU is determined by the set of operations of the SDFG that it needs to execute. Arithmetic pipelining, often used for multiplication in an FU is supported and has been used for some of the examples.

Storage is implemented using *multi-port memories and register files* in addition to individual registers. By placing several variables in a single unit the number of independent sources and sinks of data is reduced. The cost of some memory units will be known in advance. Cost of other memory units, having p-ports and n cells, is computed by GABIND using the formula: $c_m(n,p) = n(\alpha p + \beta) + \gamma p$, where α is the cost of the access logic per port per cell, β is the cost of each cell, and γ is the cost of the driver and other logic per port of the memory. In order to achieve low access time for a memory, the maximum number of cells that a memory can have is restricted to some predefined "small" number, input as a design parameter.

All components are connected to one or more buses. The connection may be switched or un-switched, ie. physical. Interconnecting buses are often major contributors to the routing area for data paths. The number of buses serves as an effective handle to control their proliferation. A sufficient number of buses need to be present to satisfy concurrent transfers between the FUs and the memories that are eventually formed. Multiple data transfers arising from a *common source* are identified for possible use in interconnect optimization. Such transfers may be routed through a common bus making better use of existing connections.

VII. The GA Based Solution

GABIND employs a genetic algorithm to perform optimizations and solve the problem. Our technique makes use of the GA as an efficient randomized search scheme for finding good solutions. It also differs from the usual GAs on several aspects, explained below. The motivating factor for taking this approach was to have a GA to solve the problem in reasonable time with a population of solutions of practical size. We found it fit to incorporate known good heuristics such as force to speed up the search process. Qualitative justifications for the design decisions for several aspects of the GA are given at appropriate places. The main features of the GA are as follows.

A. Design representation

A structured solution representation has been used. For each operation and each transfer there are fields indicating the FU or the bus to which it is bound, respectively. The binding of a variable indicates the memory and the number of ports that it has. Individual binding decisions of operations, transfers and storage are highly interdependent.

B. Crossover

This is the most important step in the GA. Application of a traditional recombinant crossover often results in offsprings that do not represent a feasible solution, thus wasting computation time. A randomized heuristic algorithmic crossover has been used to ensure that a crossover always results in a feasible solution. The role of the heuristic is to avoid generating extremely poor solutions. The randomization ensures that the application of the heuristic does not seriously arrest the search that takes place in course of the GA based optimization. This technique has been successfully applied in [10].

Three steps are involved in the crossover. First, it is determined which of all the attributes of both the parent solutions will be considered for inheritance. Then a tentative partial data path (TPDP) is formed by inheriting some of these attributes. Finally, the complete offspring solution is formed by completing the partial solution. Details of the crossover are given in section VIII.

C. Population control

It is important to ensure that diversity of the population is sustained throughout the run of the GA. This has been achieved as follows. First, a minimum number of solutions having the k^{th}, $k > 1$, best overall solution cost are retained. This policy is implemented for up to a fixed value of k. Second, the minimum number of distinct memory configurations in the population is maintained above a certain minimum number. This condition may not be satisfied at the beginning but once sufficient memory configurations have been produced, a certain number of solution groups having the same memory configuration are maintained. These memory configurations are tracked according to the memory configuration cost only. Third, a few solution groups with the same memory configuration as lower cost solutions are also maintained. The conscious decision to ensure memory diversity has been taken in view of the vast number of memory formations possible, as compared to FU formations.

D. Parent selection

Crossover is performed between two solutions taken from the population of solutions. A solution is selected only once during one generation to ensure maximum participation of solutions in the crossover. The selection policy gives preference to choosing parent solutions which are more fit. To choose better fit parents, a list of solutions whose cost is less than some threshold is maintained. The threshold is determined according of the distribution of the solution costs in the population. Solutions can be picked up from this list at random.

Due to the strong interdependence between binding decisions, it is likely that two good solutions will have highly incompatible solution attributes. A crossover between such a pair of solutions is very likely to produce an offspring of high cost or low fitness value. This was experimentally observed during development. It has been suggested in [3] that special precautions need to be taken to handle such a case, as an excessive amount of type II deceptability could undermine the GA for the particular problem. Therefore, a provision has been made to choose parents that are genetically less incompatible. Parents may be chosen such that they have identical memory configurations. During crossover the use of core attributes helps by reducing the incidence of "noisy attributes".

E. Other aspects

First an initial population of feasible solutions is created. Each solution is produced by randomly generating feasible binding and allocation decisions. The cost of each solution is computed and stored. The population control data structures are then created. Offspring solutions which are produced are integrated into the main population of solutions only after the current generation is completed. They replace an equal number of solutions from the current population. This is a flexible compromise between replacing the entire population, and replacing just one solution. The GA is started to run for a certain minimum number of generations. Every time there is an improvement it is run for at least another fixed number of iterations in the hope of another improvement. Finally, the data path corresponding to any one of the best solutions obtained is output.

VIII. Details of Crossover

Crossover is performed in five phases, described below. Actual allocations and bindings are made in the final phase.

A. Determining prominent solution attributes

The cost of a solution is sensitive to the bindings. An unfavorable binding could give rise to additional data path elements. For this reason a 0/1 gradation is performed for the operation and transfer bindings in the parent solutions. The aim of transfer binding gradation is to consider only the more frequently accessed component connections to each bus before proceeding with the inheritance. Similarly, the aim of operation binding gradation is to consider for inheritance only the more frequently used functionality of each FU. In the implementation the better bindings are marked *core* while the inferior ones are marked *non-core*.

Variable to memory bindings are graded in a continuous scale. For a particular memory the points accessing it are determined. The importance of each such point has been defined as the number of variables of the memory that are accessed by that point. The importance of a variable has been defined as the $\sum(importance\ of\ points\ that\ access\ the\ variable)$. The *spread* of a memory has been defined as the total number of points accessing the memory. The relative importance of a variable has been defined as:

$$\frac{(min.\ spread\ among\ all\ mems.)*(imp.\ of\ var.)}{\alpha_v(spread)*(max.\ imp.\ of\ var.\ in\ mem.)},\ \alpha_v \geq 1.$$

The above scheme is intended to distill out only some of the binding decisions which are likely to work together as good building blocks, while filtering out the noisy building blocks. This GA will still benefit from implicit parallelism, but less than the usual analytical value. We feel that for the usual analytic results to apply, the required population size would be too large to be useful.

B. Correspondence Between Data Path Elements

A matching between the data path components of the two parent solutions is used while performing inheritances. Affinity measures between components are computed based on similarity of bindings of operations, transfers and variables, with FUs, buses and memories, respectively. A greedy algorithm driven by edge weights is then used to match these.

C. Operation and Transfer Binding Inheritance Plan

A tentative plan of operation and transfer bindings to be inherited, time step by time step, is constructed. In each time step, either *core* operation or core transfer bindings are inherited first. The choice is made probabilistically. Next, associated core transfer or operation bindings, respectively, are attempted to be inherited. This is done by inspecting the buses or FUs one by one, respectively. If operation bindings are inherited first in a time step then core transfers connected with these operations are inherited provided the target bus is available. Similarly, the case of first inheriting transfer bindings is handled. The tentative binding inheritance plan implies a tentative allocation scheme for the data path to be constructed. The actual allocation and binding is explained later in this section.

D. Memory Formation

First, a blank memory configuration is formed by inheritance. A variable inherits the memory binding with a probability which is either the *register inheritance probability* parameter, or the *importance* of the variable in the memory, as defined earlier in this section. After inheritance is completed, in general, there will still be variables to be mapped to memories. These remaining variables are packed into the memories already constructed during inheritance. Those variables which could not be packed into these memories are packed into new memories. The choice of memories to be packed is governed by a simple heuristic. The heuristic is to choose the variable for which the number of unmapped variables that can still be packed into this memory without increasing the number of ports is maximum.

E. Final Generation of Actual Allocations and Bindings

The actual operation and transfer bindings are now made, time step by time step, in three phases: completing implied bindings, performing bindings by inheritance and completion of pending bindings. This also completely determines allocation of all data path components.

The first phase is trivial involving only bookkeeping steps. For the second phase, first the operations are processed and then the transfers are handled. For each operation binding in the inheritance plan if the corresponding FU is available, then the actual binding is set. If the FU does not already implement that type of operation then possibility of doing so is decreased. Similarly, transfer bindings are inherited but with some additional processing. While making a transfer binding if the existing links between and FUs, system ports and the memories with the buses suffice to support the transfer then the binding is directly made. If new links need to be introduced at both the source and the destination of the transfer then the inheritance is not made. If only one new link is needed then the inheritance is done probabilistically. Whenever a new link is introduced the data path is updated.

After the first two phases, in general, some operations and transfers will still remain unmapped. The operation and then the transfer bindings are made using a force directed completion algorithm, time step by time step. The decisions are made in a best first approach selecting the binding that leads to the least force. The force is computed in a way to encourage utilization of existing data path components, and discourage introduction of new components.

IX. Experimental Results for GABIND

GABIND has been tested on a Silicon Graphics Indigo (IRIS) workstation (R4000SC RISC CPU, 100Mhz (int.), 50Mhz (ext.)) with the standard benchmark examples of Facet [11], differential equation solver (Diffeq.) [9] and elliptic wave filter (EWF) [12]. The results have been tabulated along with those of some other well known systems in table II. The columns of the table indicate the technique, the number of multiplexer channels (#M), the number of links (#L), the number of storage cells (#C), the memory configuration, the FU configuration and the run time. A memory configuration of the form $< x, y >$, indicates y memories each having x ports. GABIND is able to synthesize the designs using only single or double port memories. A double port memory of one cell is equivalent to a register. The results indicate that the cost of FUs and total number of multiplexer channels are consistently kept low. Sometimes the storage requirements are marginally higher than competing systems. It may be noted that because of the high level of design, DPS techniques usually cannot be compared exactly. It was observed that the solution quality is not critically sensitive on the GA parameters. In general, the time taken by the algorithm depends on the total number of time steps used in the schedule and is proportional to it. A larger population size is required for designs involving higher number of FUs or time steps. The same GA parameters were used for all designs, although the optimal result is obtained for the smaller examples with a smaller population size.

Given a schedule of operations, GABIND is able to synthesize globally optimized data paths in terms of the cost of the functional units, multiplexing switches and storage elements. The synthesized data paths compare well with those produced by other contemporary systems. Operation pipelining and multi-cycling are supported. Storage implementation can accommodate individual registers, single or multi-port memories. GABIND relies on the genetic algorithm to perform optimization. For this GA we have developed a specific crossover based on a force directed completion algorithm. We have shown experimentally that the GA framework can be applied successfully for structured representations suitable for DPS.

X. Synthesis of structured architectures

This is similar to the earlier problem of allocation and binding, except that scheduling of operations and transfers are additional sub-problems, and the architectural constraints are much stronger. SAST (structured architecture synthesis tool) essentially takes as input, precedence constraints between operations represented as a partial order, and outputs a schedule of operations and transfers, and a

System name	#M	#L	#C	memory config.	FU config.	CPU time
Facet in 4 time steps, 3FUs						
Facet	11	—	8	—	—	—
Splicer	8	—	7	—	—	3s
HAL	6	13	5	—	—	—
Vital-NS	6	12	5	—	—	1.5s
GABIND	5	11	6	$< 2, 3 >$ $< 1, 2 >$	$\langle + \rangle$, $\langle + \mid \star \rangle$, $\langle - \& / \rangle$	28s
Diffeq. in 4 time steps						
Using single cycle multipliers and 5 FUs						
Splicer	11	—	6	—	—	—
HAL	10	25	5	—	—	40s
Vital-NS	12	22	5	—	—	3s
GABIND	8	18	5	$< 2, 5 >$ $< 1, 1 >$	$2\star$, +, -, <	38s
Using single cycle multipliers and 3 FUs						
GABIND	12	16	6	$< 2, 4 >$ $< 1, 2 >$	$\langle + \star \rangle$, $\langle \star \rangle$, $\langle +, -, < \rangle$	32s
Diffeq. in 8 time steps, 2FUs, 1 pipelined multiplier						
HAL	13	19	5	—	—	120s
Vital-NS	13	17	5	—	—	2.5s
GABIND	7	13	5	$< 2, 2 >$ $< 1, 2 >$	$\langle \star \rangle$, $\langle +, -, < \rangle$	24s
EWF in 17 time steps, pipelined multiplier						
HAL	31	—	12	—	$3+$, $2\star$	120s
SAM	31	50	12	—	$3+$, $2\star$	—
PSGA_Syn	—	—	10	—	$3+$, $2\star$	10s
Vital-NS	32	50	11	—	$3+$, $2\star$	110s
STAR	26	—	11	—	$2+$, $1\star$	—
GABIND	29	29	13	$< 2, 5 >$ $< 1, 1 >$	$2+$, $1\star$	210s
EWF in 18 time steps, pipelined multiplier						
HAL	34	—	12	—	$3+$, $1\star$	240s
SAM	30	40	12	—	$3+$, $1\star$	—
PSGA_Syn	—	—	10	—	$3+$, $1\star$	10.2s
Vital-NS	33	40	10	—	$3+$, $1\star$	140s
GABIND	31	35	11	$< 2, 6 >$ $< 1, 1 >$	$3+$, $1\star$	251s
EWF in 19 time steps, pipelined multiplier						
HAL	26	—	12	—	$2+$, $1\star$	360s
SAM	21	40	12	—	$2+$, $1\star$	—
PSGA_Syn	—	—	9	—	$2+$, $1\star$	10.2s
Vital-NS	29	40	11	—	$2+$, $1\star$	200s
STAR	28	—	11	—	$2+$, $1\star$	—
GABIND	27	33	14	$< 2, 4 >$ $< 1, 2 >$	$2+$, $1\star$	255s

TABLE II
Results of running GABIND of Facet, Diffeq. and EWF.

data path to implement the schedule. The generated data path is organized as architectural blocks (A-block), and optional global memory blocks. Each A-block has a local functional unit (FU), local storage and internal interconnections. The A-blocks and the memory blocks, if any, are interconnected by a few global buses. The structure of the data path is characterized by a set of architectural parameters, such as, the number of A-blocks, the number of global memories, the number of global buses, the number of access links which connect an A-block to the global buses and the maximum number of writes per time step to storage locations in an A-block. The last parameter becomes relevant if a memory with a fixed number (e.g. one or two) of write ports is to be used to implement storage in an A-block. SAST delivers the following: *i*) a schedule of operations, *ii*) the A-block in which each operation is scheduled, *iii*) the schedule of all transfers over the global buses, satisfying the architectural constraints, and *iv*) the composition of the FU in each A-block, in terms of specific implementations of operators from a module database. The option to pick up modules from a data base permits the flexibility of using units which are pipelined or combinational and also units varying in speed and size. SAST can handle specifications with multiple basic blocks [7]. This requires certain variables carrying data across basic blocks to be located at predetermined locations. If the value destined for such a variable is defined or available only outside the A-block or memory where the variable is supposed to be located then, a transfer from a suitable A-block or memory to the appropriate destination for its assignment needs to be made.

The main feature of this work is that random long-distance interconnects between data path elements are avoided. This makes this technique attractive for synthesizing designs targeted towards programmable structures, where global wiring resources are limited. The experimental results indicate that this technique compares favorably, in terms of schedule time and component cost with other synthesis techniques that do not attempt to generate data paths free of random long distance interconnects. In section XI the structured architecture synthesis problem is discussed. The GA based synthesis algorithm is presented in section XII. Some results for SAST are given in section XIII.

XI. The Structured Architecture Synthesis Problem

It is necessary to find a schedule of operations such that each operation is scheduled in one of the A-blocks. The composition of an FU is determined by all the operations that it has to perform. It is also necessary to find a schedule of transfers of values between the A-blocks using the permitted buses as access links. It is assumed that sufficient storage is available in an A-block. There are a set of global buses interconnecting the A-blocks to permit the transfer of data between them. Each A-block is connected to the global buses by means of a specific number of *access links*. The number of access links limit the maximum transfer bandwidth between an A-block and the global buses.

A functional unit in an A-block is a set of one or more hardware operators such that in any time step only one operation can be initiated and in any time step only one result can be generated. Operations scheduled on an FU are not permitted to have input or output conflicts. Similarly, execution conflicts are not permitted in which operations try to execute simultaneously on the same hardware. It may be noted that multiple operations may execute on a pipelined unit without execution conflict.

If a variable is required by an operation scheduled in an A-block, it should either be available in that A-block or it should be transferred from another A-block or memory where it is already available. A variable becomes available in an A-block at a particular time step if it is either defined there or transferred therein, in that time step.

Certain variables, referred to here as *program variables* are meant to reside at specific storage locations in specific A-blocks, as explained later in section XII-A. These are initialized as being available for use in the appropriate A-block from the first time step. Variables in an A-block are stored in local storage elements. Any two variables which are live [7] at the same time need to be assigned to distinct locations. The present implementation also permits the use of multiple implementations of an operator, such as a slow adder or a fast adder. Use of pipelined operators, such as pipelined multipliers is also supported.

Thus several decisions need to be taken, which are as follows: *i*) The time step where an operation is to be scheduled. *ii*) The A-block in which the operation is to execute. *iii*) The particular module that will implement an operation in the FU in an A-block. *iv*) The time step when an input for an operation is to be transferred over a global bus, if it is not already available in the local A-block. *v*) If such a transfer is required, then the A-block from where the value should be obtained. It may be noted that a value may be present in more than one A-block. *vi*) Transfers between A-blocks that may be required for defining *program variables* (explained in section XII-A) – indicating the time step, source and destination.

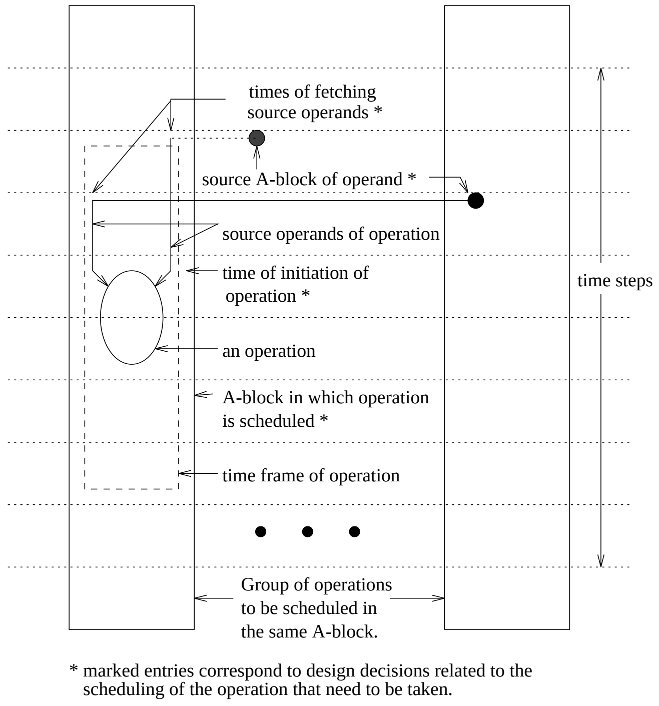
XII. GA Based Scheduling Algorithm

A genetic algorithm has been designed and implemented for solving the scheduling problem. A brief overview of the GA is given now. The detailed description follows in the sub-sections that follow. In view of the complex nature of the problem a structured solution representation has been used, as against a simple bit string. An initial population of solutions is generated at random. New solutions are obtained by inheriting values of decision variables from parent solutions, selected from the population. The decision values of the solution attributes are not independent and so the solution representation resulting from inheritance could correspond to an infeasible solution. To handle this situation a completion algorithm has been used to obtain a feasible solution from the solution representation resulting from a crossover. The completion algorithm is also used to obtain a feasible solution from a solution representation obtained by randomly assigning values to solution attributes, while generating the initial population of solutions. A scheduling heuristic has been used in the completion algorithm and this has been found to improve the performance of the genetic algorithm. A population control mechanism had to be employed to sustain diversity in the population, while at the same time retaining solutions with good overall and partial fitness. The genetic algorithm is run up to a fixed number of iterations and this serves as the stopping criterion. The last improvement in solution cost (i.e. when the best solution is obtained) usually occurs well before all the iterations are completed.

In the rest of this section the solution representation, the cost function, the parent selection scheme, the crossover scheme, the completion algorithm, the replacement scheme and the heuristic to enhance the performance of the genetic algorithm are explained.

A. Solution representation

Each solution comprises of several decisions which are required for the proper implementation of the design. Figure 5 indicates the decisions required for scheduling an operation. For each operation the time when it is to be scheduled and the A-block where it has to be scheduled are stored. For each input operand of an operation the A-block from



times of fetching source operands *
source A-block of operand *
source operands of operation
time of initiation of operation *
an operation
A-block in which operation is scheduled *
time frame of operation
time steps
Group of operations to be scheduled in the same A-block.
* marked entries correspond to design decisions related to the scheduling of the operation that need to be taken.

Fig. 5. Decisions for scheduling an operation.

where this value is to be obtained and the transfer time are given. If the operand is present in the same A-block then the time of transfer is redundant, as no transfer is necessary between A-blocks.

With loop based computations, which are very common, some of the variables defined in some basic block are required for subsequent iterations of a loop. Such variables are referred to as *program variables.* A program variable needs to reside at a fixed location before the basic block in which it is used starts executing. For each program variable the time step of assignment and the A-block from where the value is to be obtained are indicated.

The period after which the result of an operation becomes available after it has been initiated on an FU depends on how long the particular module implementing the operation in the FU takes to deliver the result. For example, an addition could be implemented by a fast adder in a single time step or by a slow adder in two time steps. Similarly, a multiplication could be implemented by a combinatorial multiplier or by a pipelined multiplier. The decisions involved in determining the composition of the FU need to be represented. It is necessary to indicate which operations an FU can implement and also the modules used for implementing these operations. The former need not be stored explicitly because it is fully implied by the union of all the types of operations that are scheduled on it. However, the module information needs to be stored explicitly.

Thus there are three types of information to be represented, which are as follows: *i*) Information directly related to the scheduling of operations, *ii*) information indicating the scheduling of variable transfers and *iii*) information regarding the composition of FUs. A structured representation is used for storing the above information. This is suitable for performing the algorithmic crossover (described in the section XII-E), which leads to a feasible solution representation.

It is often desirable to partially normalize a representation to reduce redundancies in the representation arising from permutation of attribute assignments. It may be noted that permutations of operation to A-block bindings alone do not correspond to equivalent solutions because the program variables are also bound to specific A-blocks. Such a permutation would, in general, lead to distinct transfer requirements.

B. Cost function

The scheduling algorithm tries to find a schedule of operations and transfers within a specified number of time steps. The solution cost is constructed to indicate the cost of the hardware and the extra time steps used in the schedule. It is of the form

$$C = (penalty)(extra\ \ time\ \ steps) + (cost\ \ of\ \ FUs).$$

The penalty is chosen to accord priority to finding a solution within the specified number of time steps. The penalty on the extra number of time steps is a constant chosen to be an order of magnitude higher that maximum possible cost of the FUs. In addition the cost of FUs is also separately accessible for performing population control, to be explained later in section XII-G.

C. Parent selection

The parents are selected on the basis of their costs using the roulette wheel technique [13]. This being a minimization problem, the selection probability of a parent is computed taking into account the maximum cost of solutions in the population as follows: $p_{s_i} = \dfrac{C_{max} + \delta - C_i}{N_{sols}(C_{max} + \delta) - \sum_i C_i}$, where p_{s_i} is selection the probability for solution i, $\delta \geq 0$, C_i is the cost of the solution, C_{max} is the maximum solution cost in the current population and N_{sols} is the number of solutions in the population. Solutions with higher cost are selected with lower probability. If $\delta = 0$ then the solutions with cost C_{max} will never be selected. Selection is done with replacement so that a member solution of the population may participate more than once in crossovers, in one generation.

D. Crossover

New solutions are generated through crossover. An outline of the crossover mechanism used in SAST is given in figure 6. An example illustrating the formation of operation scheduling attributes through crossover and its subsequent completion is given in example 3. First two parent solutions are selected. These go through a mutation and then the actual crossover takes place to generate a raw offspring. The crossover proceeds with inheritance of solution attributes values from each of the two parents. These attributes include schedule times and A-block bindings of operations, transfer times for operation inputs and the defined program variables. The FU configuration of the solution is also formed by inheritance from the parents. Inheritance of the attributes from either of the two parents proceeds in the (inverse) ratio of their solution costs. This may be considered to be a discrete multi-point crossover scheme. The solution representation available after inheritance, in general, not feasible. This is corrected by applying the completion algorithm.

```
procedure crossover()
1. chose two parents from the population
     of solutions.
2. mutate a each parent according to the mutation
     probability.
3. for each operation to schedule do
4.     inherit the various scheduling information
         of the operation  (such as, the A-block
         where it is to be scheduled, the time when
         the operation is to be initiated, for each
         input operand, the source A-block and the
         transfer time) from the two parents.
5. for each of the program variables do
6.     inherit the time of assignment and the
       source A-block from the the two parents.
7. for each of the A-blocks
8.     inherit library module to implement
       operations to be realized in the FU of
       this A-block from the two parents.
```

Fig. 6. Generating initial attributes of offspring by crossover.

E. Solution completion

It was noticed that optimization obtained only by applying the genetic operators of mutation and crossover, with small enough population sizes to be practical, do not perform very well. This is because of the vast numbers of solution representations generated that do not correspond to a feasible solution. A procedure for *solution completion* is applied to the raw solution resulting from attribute inheritance during crossover. Solution completion is also applied while generating new solutions because the randomly generated attributes used to construct the initial solutions may not correspond to feasible solutions either. The procedure is essentially a list scheduling algorithm with some programming intricacies to support the various features for structured architecture synthesis. A simplified version is shown in figure 7. The main data structures are a pair of lists, the ready list and the active list. A pair of these lists are used for scheduling operations and another pair for scheduling assignments. Operations or assignments in both types of lists are ready for scheduling in the current time step. However, it is only attempted to schedule operations or assignments from the corresponding active list. In each iteration the ready lists are processed to transfer some operations or transfers to the corresponding active lists. It is first attempted to schedule operations in the active list on the unit indicated in the solution representation for that operation. If this attempt to schedule the operation fails then it is attempted to schedule these operations on other available FUs. This is done to utilize FUs which may otherwise go unutilized in the current time step and is done only after it has been attempted to schedule all the operations on the active list on the designated FU. If any operation gets scheduled then the process of transferring operations to the active list from the ready list and then scheduling them is repeated. The intention of maintaining an active list of operations is to give priority to the operations in this list over the operations in the ready list for scheduling in the current time step. Assignments are normally handled after all the operations in the current time step have been scheduled. To avoid any excessive adverse effect of such a bias, assignments are sometimes attempted before trying the second round of scheduling operations, as indicated above, on other available FUs. When no more scheduling is possible, data structures are updated to close the current time step, and scheduling proceeds from the next time step. Data structures have been chosen so that single step, multi-cycle and pipelined operators implementing operations are handled homogeneously as the scheduling is done.

A scheduling heuristic is also used intermittently with the intention of improving the quality of the solutions in the population. The heuristic may be used while transferring operations from the ready list to the active list (line 4, in figure 7). Normally operations are selected from the active list for scheduling at random (line 5, in figure 7). However, if the heuristic is being used then operations are chosen from the list on the basis of the scheduling heuristic. The application of the heuristic is explained in the section XII-F.

While trying to schedule an operation in an A-block at a specific time, first it is checked whether the FU can be used without input-conflict, output-conflict or execution-conflict. Next the availability of operands is checked. If an operand is not present in the current A-block then it needs to be transferred from another A-block, in the current or a preceding time step. For an operand or variable to be transferred at a particular time a free transfer path from the source to the destination needs to be identified. Thus a free bus and a free access link at the source and destination A-blocks have to be found. An operation can be scheduled in an A-block only if the FU can be used without conflict, and the operands are available or can be made available.

The inward transfer of a variable currently unavailable is made as follows. The variable can be transferred any time between the first time step and the current time step. It can be transferred from any A-block where the variable is available at the time the transfer is being attempted. The transfer is first attempted at the time and from the A-block indicated for that value in the solution. If the transfer cannot be satisfied this way then other times and A-blocks are considered in the following order: $t_s+1, t_s-1, t_s+2, \ldots$ and $(b_s+1) \bmod \mathrm{tot}_b, (b_s+2) \bmod \mathrm{tot}_b, \ldots$, respectively, where t_s is the desired time of transfer, b_s is the desired source A-block and tot_b is the total number of A-blocks. The order of scanning is block major (i.e. the block index changes slower).

Example 3: Consider an operation having inputs *v0* and *v1*. Table III shows hypothetical scheduling attributes values of the operation in the two parent solutions (column ‘P1’ and ‘P2’), and those of the resulting offspring solution (column ‘CS’). The parent from which the attribute is inherited is shown in column ‘SP’. There are obvious inconsistencies in the inherited attribute values. These may be corrected by the completion algorithm as follows.

TABLE III
CROSSOVER OF SCHEDULING ATTRIBUTES OF A HYPOTHETICAL OPERATION.

Attribute	P1	P2	CS	SP
Initiation time	3	4	3	1
A-blk. where scheduled	1	2	1	1
Source A-blk. of left operand	1	2	2	2
Transfer time of left operand	3	4	4	2
Source A-blk. of right operand	2	1	2	1
Transfer time of right operand	3	3	3	1 or 2

Assume that this operation occurs in the active list while scheduling for time step ‘3’. Let us assume that A-block ‘1’ is available for this operation. The algorithm would find that it is not feasible to transfer the left operand into the A-block in time step 4, and would consider all feasible time steps for transferring in this operand so as to monotonically recede from the time step indicated in the offspring. Thus if the feasible transfer times for the left attribute were time steps 2 and 3 then the algorithm would first consider time step 3 and then time step 2. Let us assume that it is feasible to transfer in the first operand in the third time step from A-block ‘3’. Now while considering the second operand suppose that it is not feasible to transfer it from A-block ‘2’, as indicated in the offspring attribute. The algorithm would then try to source the operand from other A-blocks. Let us assume that it succeeds in sourcing the operand from A-block ‘1’. This operation is now scheduled in time step 3. □

```
procedure complete_solution()
1. prepare initial ready lists of operations and
      variable assignments.
2. while (operations and assignments remain
      to be scheduled)
3. { decide whether heuristic scheduling is to be used
        <sch_heur_flg> or priority will be given to
        transfers <priority_trn_flag>.
4.   transfer some operations to active list from
        ready list.
5.   try to schedule active operations on units
        indicated in the chromosome.
6.   if (priority_trn_flag)
7.     try to schedule active assignments.
8.   try of schedule remaining operations on
       other units.
9.   if (an operation has been scheduled)
10.    redo iteration.
11.  if (not priority_trn_flag)
12.    try to schedule active assignments.
13.  update ready list of operations.
14.  update status of FUs.
15.  bring in ready transfer candidates to active
       transfer list.
16.  move some transfers from ready list to active list.
17.  update data structures and flags.
18.  increment the time step.
19. }
```

Fig. 7. Completion algorithm.

F. Application of Heuristic

The heuristic assists the completion algorithm. It is applied stochastically. The heuristic is based on a weight computed for each operation, which is defined as $w_i = \sum_{o_j \succ o_i}(d_j + W)$, where o_i and o_j are operations, o_j is a successor of o_i and W is a fixed positive value. While selecting an operation to schedule using the heuristic, it is chosen at random in proportion of its computed weight. A stochastic choice is made to avoid excessive bias to a particular decision.

The heuristic is applied at two places, while selecting operations from the active list and while transferring operations from the ready list to the active list. While completing a solution it is applied with a certain probability that is taken as a parameter. Even when it is being applied it is turned on and off at random as scheduling progress through the time steps to avoid excessive bias from the heuristic which might undo the evolutionary process.

G. Replacement

The replacement policy is designed to ensure that all solutions generated stay in the population for at least one iteration. This is done by introducing all the new solutions generated through crossover during one generation of the GA into the population, and replacing an equal number of existing solutions. The offsprings are stored in an adjoint pool, to be introduced into the main population once all the offsprings from the current generation are produced. The solutions to be replaced are mostly chosen at random. This could lead to removal of apparently good solutions, with low cost, from the population. To counter this a scheme has been used, at the same time, to retain the solutions with better costs, and also maintain a diversity of FU configurations in the population.

During implementation it had been observed that solutions with low cost FU configurations initially have schedules requiring more time steps than are desirable. These, therefore, have a higher cost and tend to get displaced. The population is then left mostly with solutions having expensive FU configurations. In order to retain low cost FU configurations a fixed number of *buckets* of a certain capacity are used to retain solutions having the same FU cost, although they may differ in their solution costs. Solutions which are in these buckets do not get replaced by a newly generated solution. These buckets are used to forcibly retain solutions with a range of low FU costs, even if their solution cost is high.

When a new solution is generated, first a check is made to see whether it can be placed in one of these buckets. If the cost of the solution matches FU cost of one of the buckets then it is introduced there if there is space in that bucket. Otherwise it replaces an inferior solution from that bucket, if any. In the absence of a matching bucket, the solution is placed in a free bucket, if one is available. Otherwise, solutions from the most expensive bucket are released. If the FU cost of these exceed the that of the new solution under consideration and this solution is put in.

TABLE IV
Comparison of results with other synthesis techniques.

System	No. time steps	No. +	No. *	No. Bus, Blk., A. link	No. Reg.
Elliptic wave filter scheduled in 18 steps using multi-cycle multipliers					
SAST	18	3	2	1, 3, 1	13
COBRA	18	3	2	3, 3, -	12
CASS	18	3	2	5, 4, -	16
HAL	18	3	2	—	12
PSGA_Syn	18	3	2	—	10
Elliptic wave filter scheduled in 19 steps using multi-cycle multipliers					
SAST	19	2	2	2, 3, 2	12
COBRA	19	3	2	3, 3, -	13
CASS	19	2	2	4, 4, -	17
HAL	19	2	2	—	12
PSGA_Syn	19	2	2	—	9
Elliptic wave filter using pipelined multipliers					
SAST	18	2	1	2, 3, 2	12
COBRA	18	2	1	3, 3, -	13
HAL	18	3	1	—	12
PSGA_Syn	18	3	1	—	10
SAM	19	2	1	—	12
STAR	19	2	1	—	11
PARBUS	19	2	1	—	12

System	No. time steps	No. +	No. −	No. *	No. Bus, Blk., A. link	No. Reg.
Discrete Cosine Transform scheduled in 20 steps.						
SAST	20	2	3	1	2, 3, 2	18
COBRA	20	2	2	2	3, 3, -	12

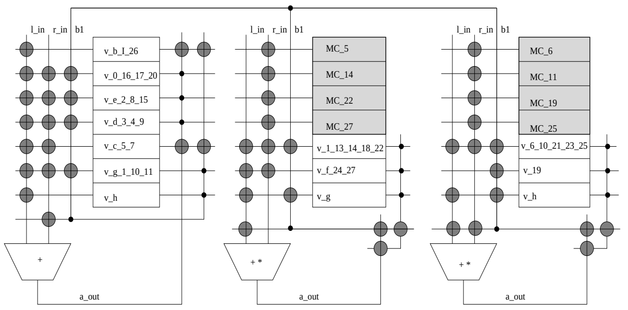


Fig. 8. Structured architecture for Elliptic Wave Filter (EWF) in 18 time steps.

XIII. Experimentation for SAST

SAST has been implemented in 'C' in a SUN SPARC-5 under Solaris. It has been used successfully to synthesize designs satisfying given architectural requirements. In particular the differential equation solver [14], fifth order elliptic wave filter (EWF) [12] and discrete cosine transform (DCT) [15] examples were worked out and the results have been given in table IV. The experimentation has been done to investigate the effectiveness of the basic scheduling algorithm, the ability to use the appropriate implementation of an operator when many are possibly available and to find schedules under tight architectural constraints. All the designs require up to two concurrent writes per A-block. The run times for the tabulated examples vary between two to five minutes, depending on the difficulty of the problem. The run time is determined by the number of generations of the GA that need to be executed before a desirable solution is obtained. Each generation is completed quickly.

The differential equation example was synthesized with a choice of fast and slow adders. SAST synthesized a data path of three A-blocks and one global bus. The FUs configuration in the three A-blocks were: ⟨slow +,<, −⟩, ⟨2-cycle *, +⟩ and ⟨2-cycle *⟩. SAST uses a slow adder to make use of the available slack time and a fast adder otherwise. This is achieved by scheduling the operations such that such in one of the A-blocks *all* the scheduled additions have a slack time. Seven storage cells are used for the three program variables and other intermediate results.

The elliptic wave filter example has been scheduled in 18 and 19 time steps using two-cycle multipliers and single cycle adders. It has also been scheduled in 18 time steps using pipelined multipliers. The usage of adders, multipliers and storage for the various cases are indicated in table IV. The architectural characteristics of the solutions are indicated in the column labeled 'No. Bus, Blk., A. link,' to indicate the number of buses, A-blocks and access links per block. The structured architecture for EWF in 18 time steps is given in figure 8.

The architecture for DCT was chosen to have three A-blocks, two global buses and two access links per block. SAST was permitted to use both a pipelined and a multi-cycle multiplier and it finds a schedule using only one pipelined multiplier, two subtracters and three adders, which is a desirable solution.

We present here SAST, a technique for synthesizing structured architectures with a simple and predictable layout structure. It relies on a GA for scheduling and allocation. The target architecture is characterized by the number of A-blocks, global memories, global buses, access links an A-block can have and the number of write ports used in the local storage for an A-block. SAST is able to handle multiple implementations of operations varying in speed, including multi-cycle and pipelined implementations. In all cases the FU cost of designs synthesized by SAST compare very favourably with those of other systems. An important feature of this work is that random long-distance interconnects between data path elements in the synthesized design are avoided. Designs produced by SAST compare favorably with other systems that do not attempt to generate structured data paths, and the run times for the tested examples are reasonable.

XIV. Conclusions

We have described GAs used to solve three problems of increasing intricacy: minimum node deletion, allocation and binding for data path synthesis and the synthesis of structured data paths, in the domain of HLS for VLSI design. In all three cases we have relied on an enhanced crossover mechanism. The crossover used for MND was the most elegant, in the sense that we were able to get a theoretical guarantee that it would generate an optimum solution with high probability. In case of MND the enhanced crossover was able to contribute significantly to obtaining the desired optimal solutions. We were able to ensure diversity in the population just by altering the mutation rate.

The next two problems dealing with the synthesis of data paths are far more intricate. For both these problems generating a reasonably good solution was a primary concern. During the development of the GAs for this problems we realized the importance of cutting down on the generation of infeasible solutions. In the unconstrained representation space, for the chosen representation scheme, on a minute fraction of the possible solution representations constitute feasible solutions. Thus without the feature of bypassing the generation of infeasible solutions, there was little scope of finding good solutions. Another problem was the proliferation of copies of a slightly better solution in the population, thereby diminishing the diversity within the population. This problem was solved by enforcing diversity using the relatively more complex replacement mechanism used for GABIND and SAST.

Our study of the three GAs in this paper indicates that a combination of a robust crossover and a diversity sustaining replacement mechanism represent a powerful scheme for solving hard and intricate combinatorial optimization problems. GA also offered the benefit of using multiple heuristics and having a set of solutions around the identified optimal cost, which is highly useful in VLSI design.

References

[1] L. Davis, *Handbook of Genetic Algorithms.* New York: Van Nostrand Reinhold, 1991.

[2] A. M. Frieze, *Probabilistic Analysis of Graph Algorithms.* Springer-Verlag, 1989.

[3] M. D. Vose, "Generalizing the notion of schema in genetic algorithms (research note)," *Artificial Intelligence*, vol. 50, pp. 385–396, 1991.

[4] C. A. Mandal, P. P. Chakrabarti, and S. Ghose, "Port assignment for dual and triple port memories using a genetic approach," in *Procs. of The Third Asia Pacific Conf. on Hardware Description Languages (APCHDL) 96*, pp. 60–64, 1996.

[5] C. A. Mandal, P. P. Chakrabarti, and S. Ghose, "A probabilistic estimator for the vertex deletion problem," *Computers and Mathematics with Applications*, vol. 35, no. 6, pp. 1–4, 1998.

[6] S. Minton, M. D. Jhonson, A. B. Philips, and P. Laird, "Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems," *Artificial Intelligence*, pp. 161–205, 1992.

[7] A. V. Aho, R. Sethi, and J. D. Ullman, *COMPILERS Principles, Techniques and Tools.* Addison-Wesley Publishing Company, June 1987.

[8] C. A. Mandal, P. P. Chakrabarti, and S. Ghose, "Some new results in the complexity of allocation and binding in data path synthesis," *Computers and Mathematics with Applications*, vol. 35, no. 10, pp. 93–105, 1998.

[9] P. G. Paulin and J. P. Knight, "Algorithms for high-level synthesis," *IEEE D. & T. of Computers*, pp. 18–31, Dec. 1989.

[10] C. Mandal and R. M. Zimmer, "High-level synthesis of structured data paths," in *IFIP TC10 WG 10.5 International Conference on Computer Hardware Description Languages and Their Applications*, pp. 92–94, April 20-25 1997.

[11] C. J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital-systems," *IEEE Trans. on C. A. D.*, vol. 5, pp. 379–395, July 1986.

[12] S. Y. Kung, H. J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing.* Prentice Hall, 1984.

[13] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Pub. Co. Inc., 1989.

[14] P. G. Paulin and J. P. Knight, "Force-directed scheduling for ASICs," *IEEE Trans. on C. A. D.*, June 1989.

[15] J. P. Neil and P. B. Denyer, "Simulated annealing based synthesis of fast discrete cosine transform blocks," in *Algorithmic and Knowledge Based CAD for VLSI* (G. Taylor and G. Russel, eds.), ch. 4, pp. 75–93, Peter Peregrinus, 1992.