An Evolutionary Approach to Behavioral-Level Synthesis

G. Grewal, M. O'Cleirigh, and M. Wineberg
Department of Computing and Information Science
University of Guelph,
Guelph, Ontario, Canada,
N1G 2W1
gwg@cis.uoguelph.ca, mocleirigh@uoguelph.ca, wineberg@cis.uoguelph.ca

ABSRACT - This paper presents a novel approach to the concurrent solution of three *High-Level Synthesis (HLS)* problems and solves them in an integrated manner using a *Hierarchical Genetic Algorithm (HGA)*. We focus on the core problems of HLS: *Scheduling*, *Allocation*, and *Binding*. Scheduling consists of assigning of operations in a *Data-Flow Graph (DFG)* to control steps or clock cycles. Allocation selects specific numbers and types of functional units from a hardware library to perform the operations specified in the DFG. Binding assigns constituent operations of the DFG to specific unit instances. A very general version of the problem is considered where functional units may perform different operations in different numbers of control steps. The HLS problems are solved by applying two genetic algorithms in a hierarchical manner. The first performs allocation, while the second performs scheduling and binding and serves as the fitness function for the first. When compared to other, well-known techniques, our results show a reduction in time to obtain optimal solutions for standard benchmarks.

1.0 Introduction

As Very Large Scale Integration (VLSI) design complexities continue to increase, designers are moving to higher and higher levels of abstraction to meet the growing challenges. An example of this is the VLSI community's recent acceptance of High-Level Synthesis (HLS) tools. High-level synthesis is the process of automatically generating a Register-Transfer Level (RTL) design from a behavioral specification [1]. An RTL design consists of functional units, memory elements, and interconnections (e.g., multiplexers and buses). The functional units normally implement one or more elementary operations like addition, subtraction, etc.
The motivation for HLS stems from several factors. HLS tools not only reduce the time to design or redesign a product, they also reduce the number of iterations required to achieve a satisfactory design by guiding the designer towards a better solution with less chance of error earlier in the design process. Today, HLS tools are enabling companies to deliver competitive, reliable products in a timely manner.

The inputs to a typical HLS tool include a *behavioral description* of the digital circuit to be designed, a *library* describing available hardware resources, and a set of *design constraints*. The behavioral description is normally specified using a hardware description language [2], or a traditional programming language that is compiled into a Control/Data-Flow Graph (CDFG) [3]. The hardware library contains a description of the available unit types including functional units, registers, multiplexers, buffers, buses, and interconnects. Design constraints include constraints on performance (execution time), cost (area), and other constraints defined by the application and/or the designer.

To synthesize a desired RTL design several conflicting and interrelated subproblems must be solved. This paper focuses on the three core problems of high-level synthesis: *Scheduling*, *Module allocation*, and *Binding*. Scheduling is the process of assigning operations in the original specification to control steps (clock cycles) so that all of the precedence relationships, timing constraints, and other types of constraints are satisfied. Allocation is the job of selecting specific numbers and types of functional units from a hardware library to execute the operations. The number of functional units required for a design depends on the number and types of operations that can be performed simultaneously in any control step. If there is enough hardware available, it may be possible to perform several operations simultaneously in a single step and hence get a faster schedule. In practice, however, a designer may not be able to use as much hardware as required to obtain the fastest schedule. Instead, designers are expected to make suitable trade-offs between performance and cost. Binding refers to the mapping of operations to functional units. For example, when performing two multiplication operations in the same control step using two multipliers, the decision must be made as to which multiplier will perform which operation.

The three problems of scheduling, allocation, and binding are interdependent and notoriously difficult to solve optimally. (Optimality is defined as some combination of execution time, area, and other appropriate measures.) For guaranteed optimal results all three problems must be considered simultaneously. However, the problem of scheduling alone is NP-complete [1]. Many of the current approaches for

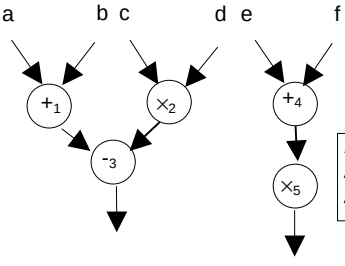
solving the problems are based upon heuristics [4-10] and the quality of results varies and may not be optimal. Recently, Integer-Linear Programming (ILP) formulations [11-14] have proven effective in handling the interdependence between the three problems, but their runtimes become unacceptable as the problem size increases.

In this paper we present an integrated solution to the HLS problems of scheduling, allocation, and binding based on two Genetic Algorithms (GAs). The main advantage of the genetic search over other approaches is a reduction in time to obtain optimal solutions. Our model differs from earlier hybrid approaches [15] that use a genetic algorithm in conjunction with well-known heuristics. We use two genetic algorithms in a hierarchical manner to solve he problems in an integrated manner. The first genetic algorithm is used to select functional units from a hardware library, while the second uses the allocations to perform scheduling and binding. The second GA performs the role of fitness function for the first. In this way, we achieve a truly integrated solution as both genetic algorithms run concurrently and cooperate with each other until a satisfactory solution is found.

The remainder of this paper is organized as follows. Section 2 describes the problems of scheduling, allocation, and binding in more detail. In section 3 we describe how we model and solve the problems using two genetic algorithms. In Sec. 4 we demonstrate the effectiveness of our approach by showing that our system can generate optimal results for standard benchmarks. Section 5 explains how we determined appropriate operators and parameters for the GAs. Finally, we present our conclusions and directions for future work in Sec. 6.

2.0 Basic Concepts

We now begin with a closer look at the three problems of scheduling, allocation, and binding. In Fig. 1 a simple data-flow graph and hardware library are shown.



Time Delay

	×	(+,-)	Area
ALU 1	2	1	600
ALU 2	3	1	400
ALU 3	3	2	300

Figure 1: Data flow graph and module library.

The DFG contains five operations: 2 multiplication operations, 2 addition operations, and 1 subtraction operation. The hardware library contains three types of functional units, each with various execution times and areas. More than one instance of each functional unit may be chosen in the final allocation. Notice that all of the functional units present in the library are complex, allowing different operations to be performed in different numbers of control steps. The faster functional units also tend to consume more space (chip area) and thus cost more to instantiate. Here each functional unit is capable of performing each operation in the DFG; in general, this may not be true.

Given a behavior specified as a DFG, as well as a hardware library, the problem is to find a suitable schedule such that:

- the operations are performed by the units having the required capability,
- a unit can execute at most one operation at a time,
- the operations are executed in the order specified by the DFG,
- the total area is limited,
- the execution occurs with a number of steps that is limited.

Figure 2 illustrates three different schedules and allocations for the behavior specified in Fig. 1. Each schedule incorporates different combinations of units. The short 3-step schedule requires 2 instances of the faster – but larger – multiplier (ALU1), resulting in a 1500 unit area design. By extending the schedule length to 4 steps, only 1 instance of the fast multiplier is required to satisfy the timing requirement using a smaller 1000 unit design. In the case of the 6-step schedule, both multiplication operations must be bound by the same unit instance if the inexpensive unit, ALU3, is to be used. Notice how the issues of schedule and area restrictions, availability of units, and unit allocation and binding are all interdependent.

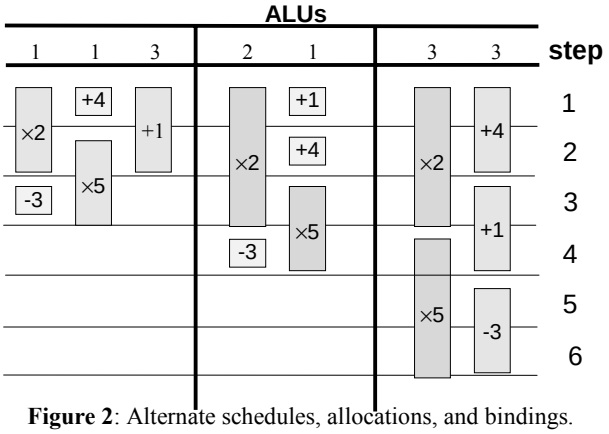


Figure 2: Alternate schedules, allocations, and bindings.

Two kinds of constraints play a crucial role during scheduling and allocation. First and most familiar are the *data-dependency* constraints. These ensure that no operation begins execution until all of its operand values have been computed. Each data-dependency constraint corresponds to an edge of the

DFG and bounds the starting time of an operation by the completion times of its predecessors on the DFG. Completion times, in turn, depend on the particular units allocated. Thus, data-dependency constraints enforce the ordering imposed by the DFG, with consideration given to the type of unit allocated to each operation.

The second important set of constraints focuses on each individual unit – its type and instance. If any unit is used by more than one operation, those operations must be implemented in some sequential order. No operation can begin until its predecessor (if any) on the same unit has completed its execution. Although this constraint also bounds starting times by the completion times of predecessors, this ordering is unrelated to data dependency. The ordering here is imposed by the mapping of operations to units and is not predetermined. It must be ascertained dynamically, if (multi-step) units are to be reused. For example, the 6-step schedule in Fig. 2 indicates that operations 4, 1, and 3, being bound to the same instance of ALU3, must occur in some order (here 4, 1, 3). We refer to the ordering among operations that are executed by the same unit as their *unit-use* ordering.

3.0 Our Approach

In practice, the total *time* and *area* available on a chip are often restricted to lie within some prespecified limits. Consequently, a designer is required to verify the existence of some feasible solution that meets both time and total area constraints; this requires no objective function at all. By progressing through a series of plausible time and area limits, the sequence of solutions clearly display the area/time tradeoffs to the designer. Our integrated approach to the scheduling, allocation, and binding problems is based on the hierarchical application of two genetic algorithms. They are hierarchical because the second GA acts as the fitness function for the first as shown in Fig. 3.

The first genetic algorithm, GA1, performs allocation; that is, it selects functional units from a general hardware library to implement the operations in the original specification. Each unit instance is assigned a unique unit number, and multiple instances of the same unit type can be selected. The only restriction is that the total area required for a given allocation cannot exceed the area limit specified by the designer.

The second genetic algorithm, GA2, performs scheduling and binding using the allocation provided by GA1. Thus, potential allocations provided by GA1 are evaluated on the basis of their ability to satisfy the timing constraint imposed by the designer. If a schedule is found that satisfies the timing constraint, a feasible solution has been found and the process terminates. Otherwise, GA2 returns the length of the best (shortest) schedule found after a predetermined number of generations.

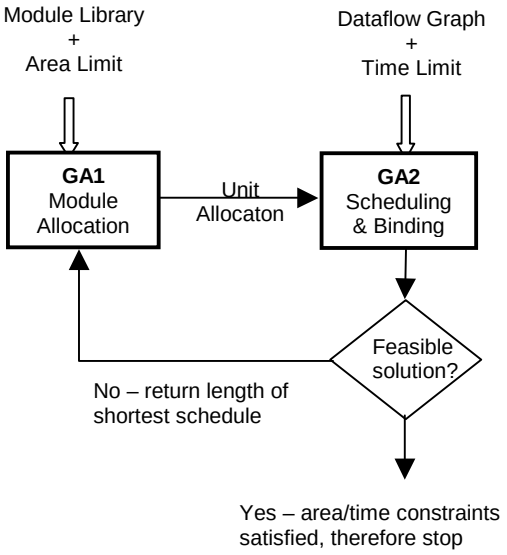


Figure 3: Hierarchical application of two GAs.

The point is that both genetic algorithms will continue to run concurrently and cooperate with each other until a solution is found that satisfies both timing and area constraints imposed by the designer.

3.1 GA1 – Allocating Functional Units

We now take a closer look at the genetic algorithm used to allocate functional units. The total area available on a chip is restricted to lie within a prespecified limit. More formally,

$$\sum_m U_m A_m \leq AreaLimit$$

where variable m indexes each instance being considered in the current allocation, A_m is the chip area consumed by including unit m in the design, and U_m is a 0-1 variable. If $U_m = 1$, unit m is selected to appear in the final design.

We employ a bit-string representation where the values of all variables U_m are directly stored in a string of length n, where n is the maximum number of functional units the designer is willing to consider using. At first glance, this representation seems to be the most direct and easiest, but infeasible solutions containing constraint violations need to be considered. One way to do this is to use penalty functions to penalize infeasible solutions [16]. However, difficulties arise in selection of the penalty function and its coefficients to prevent premature convergence and infeasible final solutions.

A second strategy, and the one employed in this paper, is based on the application of a special repair algorithm to "correct" any infeasible solutions generated [17]. For the allocation problem, this can be done by setting some U_m variable(s) to *0*. The repair procedure is outlined in Fig.4.

procedure repair (x)
begin
area-exceeded = false
$x' = x$
if $\sum_{i=1}^{n} x'[i] \cdot A[i] > AreaLimit$
then area-exceeded = true
while (area-exceeded) **do**
begin
i=randomly **select** a unit and remove it from the current allocation: i.e., $x'[i] = 0$
if $\sum_{i=1}^{n} x'[i] \cdot A[i] \leq AreaLimit$
then area-exceeded = false
end
end

Figure 4: The repair procedure.

The repair procedure transforms an infeasible allocation (x) into a feasible one (x') by removing (setting $x'[i] = 0$) randomly selected units from the current allocation until the area constraint is satisfied. Thus, all candidate allocations generated by GA1 are guaranteed to satisfy the area constraint imposed by the designer.

Figure 5 shows a template for the allocation genetic algorithm (GA1). The initial population is constructed from randomly generated individuals, with infeasible individuals in the population being repaired as described above. Routine *Evaluate()* computes the fitness of each of the given individuals. This routine calls the scheduling and binding genetic algorithm (GA2) which returns the length of the best schedule found using the given allocation. One execution of the *while* loop corresponds to the simulation of one generation. Throughout the simulation, the number of individuals (M) is kept constant. Routine *stopCriteria()* terminates the simulation when a solution (allocation, schedule, and binding) is found that satisfies both the area and timing constraint imposed by the designer. If no such solution can be found, the GA terminates after a prespecified number of generations and returns the best solution found.

In each generation, a set of offspring of size M are created. Two mates *p1* and *p2* are selected from the population independently of each other, and each mate is selected using binary tournament selection [18]. The crossover routine generates two offspring *c1* and *c2* using standard 2-point crossover. With a small probability, $P_m = 1/n$, the mutation operator randomly changes each of the components of each individual in the population. Clearly, offspring may be infeasible, in which case, they must be repaired before being evaluated.

```
Generate initial Population randomly;
Repair infeasible members of Population;
Evaluate (Population);
while not stopCriteria() Do
begin
        NewPoulation = ∅
        for j = 1 to M/2 Do
                Select p1 and p2 from Population
                {c1,c2} = crossover (p1,p2);
                mutate (c1,Pm)
                mutate (c2,Pm)
                Add c1 and c2 to newPopulation
    end
    Population = newPopulation
    Repair infeasible members of Population
     Evaluate (Population);
end
```

Figure 5: Outline of GA1.

3.2 GA2 – Scheduling and Binding Operations

The role of GA2 is to schedule the operations in the data-flow graph under the precedence and resource restrictions implied by GA1. Scheduling operations using only the units provided by GA1 involves making a choice as to the order in which operations can be executed and assigned (bound) to units. The problem is to find a schedule and binding that satisfies the timing constraint imposed on the final circuit by the designer.

GA2 uses a modified version of List Scheduling [19] to construct feasible schedules. In List Scheduling, all "ready" operations of the DFG are placed into a list ordered by relative priorities and whenever a unit becomes available the operation with the highest priority in the list is chosen and assigned to that unit. An operation is considered "ready", if and only if all of its predecessors have completed executing. Figure 6 shows the general list-scheduling algorithm.

1. Assign a **priority** to each operation in the DFG.
2. Initialize a priority queue for ready operations by inserting every operation that has no immediate predecessors. Sort the operations in increasing order of priority.
3. While the priority queue is not empty do the following:
   a. Get the operation at the front of the queue.
   b. Select an idle unit to perform the task.
   c. After all immediate predecessors of an operation are executed, insert that operation's successor(s) into the priority queue.

Figure 6: List-Scheduling Algorithm.

It is clear from Fig. 6 that the particular assignment of priorities to operations (step 1) results in different schedules (with possibly different lengths) being generated. This is because operations are selected for execution in different orders. Various heuristics [20-22] have been proposed for prioritizing the operations in the DFG. However, no one heuristic has emerged as the clear winner. Rather, recent results [23] suggest that the performance of a list scheduler can be improved by using a mixture of several heuristics.

In the approach presented here, we employ a genetic algorithm to determine the priority of the operations to be scheduled. This is a *permutation* problem by nature. We employ a priority-based encoding where the position of each gene in a chromosome is used to identify the operation to be scheduled and the value of each gene is used to denote the priority associated with the operation. The value of a gene is an integer exclusively within [1,*p*], where *p* is the number of operations in the original DFG. The larger the integer (*p*), the higher the priority.

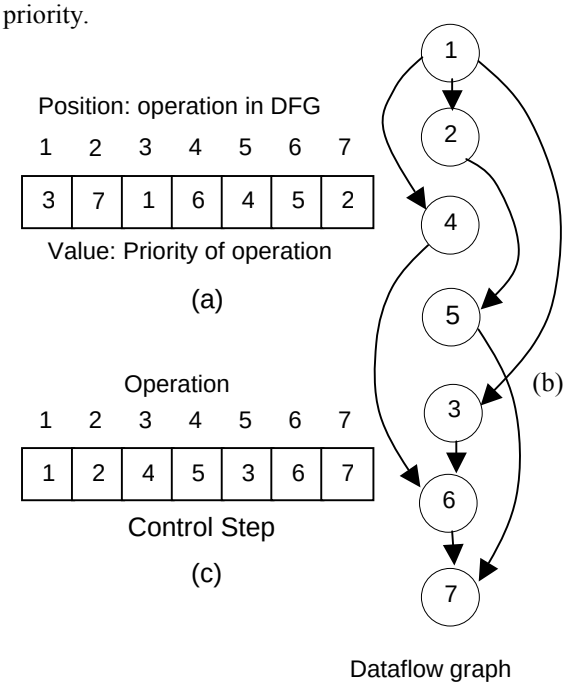


Figure 7: (a) Priority-based encoding; (b) dataflow graph; and (c) final schedule.

Figure 7(a) shows a possible encoding for the operations of the DFG in Fig. 7(b). If we assume the existence of a single functional unit capable of performing all of the operations in the DFG in a single control step, a feasible schedule can be constructed as follows. At the beginning, only operation 1 is ready to be scheduled and is therefore assigned to control step 1. Then three operations 2, 3, and 4 become ready and compete for the second control step. Their priorities are 7, 1, and 6, respectively. Operation 2 wins, because it has the highest priority. After scheduling operation 2 on control step 2, the candidates for control step 3 are operations 3, 4, and 5. Operation 4 wins and is scheduled on control step 3. The process simply repeats until all of the operations are scheduled. The final schedule is shown in Fig. 7(c).

In the previous example, a single unit capable of executing all of the operations in the DFG was used. Most designs, however, will include multiple units. It should be clear, therefore, that the decision regarding whether or not an operation can be scheduled not only depends on whether the operation's predecessors have executed, but also on the availability of a unit to perform the operation. Thus, a chromosome does not directly represent a schedule, but rather a way of resolving conflicts when more than one operation is ready for scheduling.

Unlike GA1, the scheduling and binding genetic algorithm (GA2) uses a permutation encoding as opposed to a simple bit string, Specialized crossover and mutation operators are used to preserve the encoding during reproduction. This way, a repair operator is not required. A number of crossover operators have been proposed for permutation encodings. However, through extensive experimentation we have found that *Partially-Mapped Crossover (PMX)* proposed by Goldberg et al. [24] to be most effective for our problem. PMX can be viewed as a variation of two-point crossover by incorporating a special repair procedure to resolve possible illegitimacy. PMX is implemented as follows. Choose a random cut point and consider the segments following the cut point in both parents as a partial mapping of the genes to be exchanged in the first parent to generate the offspring. Take corresponding genes from the segments of both parents, locate both these genes in the first parent, and exchange them. Repeat this process for all genes in the segment. Thus, a gene in the segment of the first parent and a gene in the second parent will define which genes in the first parent have to be exchanged to generate offspring.

In addition to PMX, a "swap" mutation operator was also used. This operator simply selects two positions at random and swaps their contents. The operator is applied with a small probability $P_m = 1/p$.

4.0 Experimental Results

The previously described Hierarchical Genetic Algorithm (HGA) is implemented in the *C* programming language in a UNIX environment. Using well known benchmarks, the HGA will be compared to other related works, including the HAL system [9], SYMPHONY [11], TASS [4], and ADPS [13]. In selecting comparison systems, a wide variety of approaches have been chosen. HAL uses the popular force-directed scheduling, heuristic, while SYMPHONY uses an integrated ILP formulation which is sensitive to the interdependence between the 3 problems. TASS uses a variant of simulated annealing. ADPS first uses force-directed scheduling

to obtain a schedule; the resulting schedule is then used for allocation and binding, all within an iterative process that attempts to refine the design. All times given represent CPU time expressed in seconds. The initial population sizes of GA1 and GA2 were selected to be 5 and 20 strings, respectively.

4.1 The Elliptical Wave Filter

An important facet of any HLS systems is the ability to determine a lower bound performance for execution of a behavioral specification, for a fixed number of resources. This first example demonstrates the ability of the HGA we are proposing to quickly minimize schedule length given a fixed bound on area.

The elliptical wave filter consists of 34 addition and multiplication operations. The hardware library contains two unit types: adders and multipliers. Each adder has a delay of 1 control step and an area of 100 units, while each multiplier has a delay of 2 control steps and an area of 250 units.

Initially, we limit area to 1050 units and seek a minimum time solution. Table 1 reveals that the HGA found an optimal 17-step schedule, requiring 3 adders and 3 multipliers. When the area limit was reduced to 700 units the HGA found an optimal 18-step schedule, and near-optimal 19-step schedule both requiring 3 adders and 2 multipliers. When the area limit was further reduced to 450 units an optimal 21-step schedule was found requiring 1 multiplier and 2 adders. Finally, when the area limit was reduced to just 350 units an optimal 28-step schedule was found requiring 1 adder and 1 multiplier. In each case, the comparison with HAL, TASS and SYMPHONY show a reduction in time to obtain the same optimal results.

4.2 Bandpass Filter

We now demonstrate the HGA's ability to minimize area given constraints on execution time. For purposes of comparison, we choose the Bandpass Filter benchmark and complex unit library presented in [9], and compare the HGA with SYMPHONY and ADPS. The DFG consists of 29 operations including multiplication, addition, and subtraction operations. The cumulative area of the units present in the library is 1620, with each unit having a delay of 1 control step.

Initially, time was limited to 8 control steps and a minimum area solution sought. Table 2 (at end of paper) reveals that the HGA and SYMPHONY found an optimal allocation (675), while ADPS produced a sub-optimal (685) solution. With the time limit relaxed to 9 control steps, the HGA and SYMPHONY again found an optimal allocation (625), but ADPS could only find a sub-optimal solution (650). This trend continued for both 10 and 11 step schedules. In every case, HGA and SYMPHONY produced better results than ADPS. However, the EGA required less time than SYMPHONY to obtain equivalent (optimal) solutions.

5.0 Tuning the HGA

In this section we wish to look at the various components and settings that comprise the Hierarchical GA to see whether different setting would produce different results. We decided to use the Bandpass Filter problem, as it was the more difficult of the two problems we have investigated.

5.1 Experimental Design

We varied the crossover rate (crt), mutation rate (mrt) and population size (ps). We also examined the 3 different permutation crossover techniques (cvr) for the scheduling level of the GA: exchange crossover (X), order crossover (O) and cycle crossover (C). We therefore are doing a multiple factor analysis with 4 factors (crt, mrt, p-s and cvr). For the crossover rate, we used 20%, 70%, 90% and 100%; for the mutation rate we used 0, 0.035 and 0.07 (no crossover, 1/lngth and 2/lngth); for the population size we used 10, 50, 100 and 1000 individuals. For a summary of the factors and factor levels see Table 3.

For the experimental design we chose a factorial design, where all factors are fully crossed. Each treatment is repeated 20 times to provide statistical significance for the various inferences obtained.

Since the fitness function used is a minimization on the number of control steps that the choice of ALU units along with a schedule found produces, it is natural that the response variable studied would be the number of control steps.

However, in the majority of runs, the actual minimum is found. So to gain more information, we also keep track of the generation that the best fitness value was discovered in a run. The better the settings for the HGA, the earlier that the minimum should be found.

Area Limit	Units Allocated	Control Steps	Time (s) HAL	TASS	SYMPHONY	HGA
1050	*,*,*,+,+,+	17	120	10.0	2	0.026
700	*,*,+,+	18	240	10.1	233	0.026
700	*,*,+,+	19	Not reported	Not Reported	358	0.026
450	*,+,+	21	360	10.2	Not Reported	0.014
350	*,+	28	480	Not Reported	Not Reported	0.008

Table 1: Results for Elliptical Wave Filter.

Table 2 Execution Times for Bandpass Filter.

Control Steps	Model	Units Allocated	Area Constraint	CPU Time (s)
8	SYMPHONY	+*,+-,+-,*	675	4.8
	ADPS	+-,*,*,+,-	685	9.0
	HGA	+*,+-,+-,*	675	0.215
9	SYMPHONY	+-,*,*,+	625	17
	ADPS	+-,*,*,+-	650	64
	HGA	+-,*,*,+	625	0.225
10	SYMPHONY	*,+-,+,*	625	132
	ADPS	+*,*,+,+-	650	15
	HGA	*,+-,+,*	625	0.25
11	SYMPHONY	+*,+-,*	600	42
	ADPS	+-*,*,+,-	630	197
	HGA	+*,+-,*	600	0.02

Unfortunately, the HGA does not *always* find the minimum number of control steps. This presents a problem when comparing runs that converged on the minimum and those that do not.

To resolve this problem we combine the two response variables into a single response variable by using the number of steps as the major factor and the number of generations as the subfactor. The new response variable follows the formula:

$$r_i = (g_{\max} + 1) \cdot s_i + g_i$$

where g_{max} is the maximum generation before the GA is halted, s_i is the number of control steps by the i^{th} chromosome, and g_i is the first generation that the i^{th} chromosome was first formed. For example, in an experiment which has a maximum generation of 99, if the best chromosome was discovered in a run first appeared in generation 52 and took 9 control steps, then

$$r_i = (99 + 1) \cdot 9 + 52 = 952$$

The resulting response variable is obviously not normally distributed. Therefore we must use non-parametric methods when performing a statistical analysis; all statistical methods used, in our case a MANOVA (multivariate analysis-of-variance), are performed using ranked values instead of the original response variable (the r_i's).

	Factor	Factor Levels
CVR	Crossover Type	Exchange Crossover (X) Order Crossover (O) Cycle Crossover (C)
CRT	Crossover Rate	0.2,0.7,0.9,1.0
MRT	Mutation Rate	0.000,0.035,0.070
P-S	Population Size	10,50,100,1000

Table 3: Factors and Factor Levels for HGA Tuning Analysis

For the experiments, we used a maximum generation cutoff value of 99 generations. We ran experiments on the bandpass filter DFG using 3 different area constraints: 675, 625 and 600.

5.2 Results

Surprisingly, tightening the constraints does not always mean increasing the level of difficulty. From the histograms of our response variable r_i (Fig. 8) we see that the "easiest' problem, i.e. the one that always finds the solution in 1 or 2 generations has an area target of 600, our tightest constraint.

The area constraint of 600 is so easy for the HGA to solve that there is not enough variance in the response data to perform a MANOVA to examine the various factors under investigation.

Furthermore, the results of the two other area constraints produce identical inferences. Consequently, we will only present the factor analysis for the 625 area constraint.

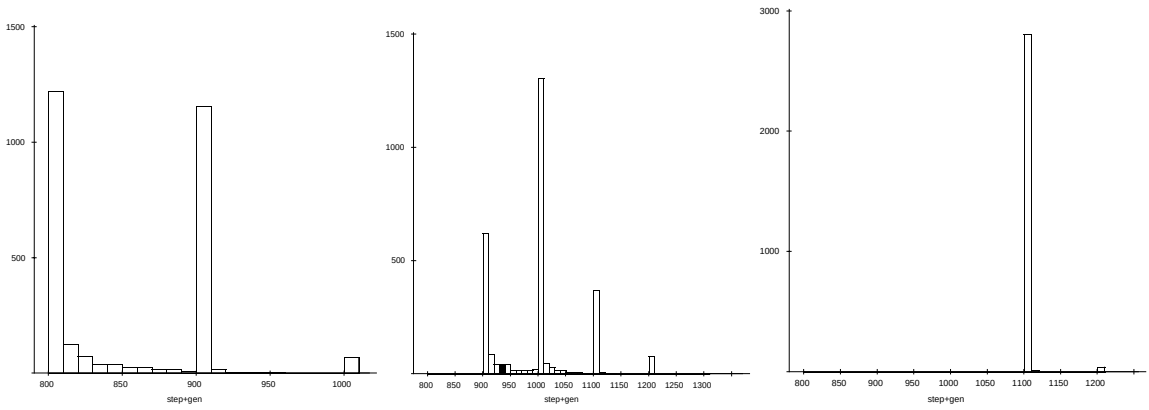
The results of the MANOVA on the bandpass filter problem with an area constraint of 625 are given in Table 4. The factors that have statistical significance are in bold.

We will now look at the statistically significant main effects one by one. First we see that all of our factors have a significant main effect with the exception of the crossover rate. This means that the crossover rate (*crt*), taken by itself, does not have an effect on the behavior of the HGA.

Next, looking at pair-wise comparison (using the Student's T test) of the factor levels of the crossover type (*cvr*) main effect using the Bonferroni post-hoc correction, we see that

Xover Type	Difference	Prob
C | O	9.68646	0.962137
X > C	395.955	0
X > O	405.642	0

std. err = 22.31



Area = 675 Area = 625 Area = 600

Figure 8: HGA Performance Frequencies on the Bandpass Filter Problem by Control Steps + Generation Found For 3 Different Area Constraints

where > means "performs better than", and | means "is not statistically different from". In other words, the exchange crossover always produces better results when used in the HGA when compared to cycle and order crossovers.

Next is the pair-wise comparison of the different mutation rates (*mrt*), again using the T test with a Bonferroni post-hoc correction:

Mutation rate	Difference	Prob
0.034483 > 0	55.1453	0.039980
0.068965 > 0	52.2062	0.056957
0.068965 | 0.034483	2.93906	0.998849

std.err = 22.31

From this we can conclude that mutation is important, as either of the mutation rates used was better than no mutation. However, we could not distinguish between the behaviors of the two different mutation rates that were used.

Analyzing the population size (*p-s*) we found that the larger the population size used, the better the behavior of the HGA, which is to be expected. However, when looking at the number of evaluations performed a different story emerges. We computed the number of evaluation by using the formula $e_i = n \cdot g_i + n$. We then combined the control steps with the number of evaluations by computing $v_i = 100{,}000 \cdot s_i + e_i$, where 100,000 is the maximum evaluation that can occur, and then used a non-parametric analysis by applying MANOVA on the ranks of the v_i.

The results were exactly the opposite found when doing the analysis on the number of generations taken:

Population Size	Difference	Prob
10 > 50	181.548	0
10 > 100	383.737	0
50 > 100	202.189	0
10 > 1000	462.213	0
50 > 1000	280.665	0
100 > 1000	78.4757	0.000034

std. err. = 17.26

Consequently, smaller population sizes, while perhaps taking more generations to find the solution, will do so using fewer evaluations and hence find the solution in much smaller computational time.

Finally, we examined the interaction terms. We found that by and large, the interaction terms just reinforced the results found for the main effects (with some very minor differences, which could have been the effect of the low number of repetitions used for time consideration).

Source	df	Mean Square	F-ratio	Prob	Source	df	Mean Square	F-ratio	Prob
Const	1	5976115920	25012	≤ 0.0001	**p-s**	**3**	**352277511**	**1474.4**	**≤ 0.0001**
xvr	**2**	**51427123**	**215.24**	**≤ 0.0001**	**xvr*p-s**	**6**	**2854217**	**11.946**	**≤ 0.0001**
xrt	3	386875	1.6192	0.1828	xrt*p-s	9	449420	1.8810	0.0503
xvr*xrt	**6**	**1396494**	**5.8447**	**≤ 0.0001**	**xvr*xrt*p-s**	**18**	**540738**	**2.2631**	**0.0018**
mrt	**2**	**924022**	**3.8673**	**0.0210**	mrt*p-s	6	238399	0.99777	0.4249
xvr*mrt	4	162643	0.68071	0.6053	xvr*mrt*p-s	12	307270	1.2860	0.2194
xrt*mrt	6	425081	1.7791	0.0994	**xrt*mrt*p-s**	**18**	**418112**	**1.7499**	**0.0258**
xvr*xrt*mrt	12	204909	0.85761	0.5905	xvr*xrt*mrt*p-s	36	278740	1.1666	0.2290

Error df = 2736 SSE = 653717184 MSE = 238932
Total df = 2879 SSE = 1884033224

Table 4: MANOVA Results for Bandpass Filter with Area Constrain of 625

6.0 Conclusions

In this paper we have presented an integrated solution to the high-level synthesis problems of scheduling, allocation, and binding. Our approach is based on the hierarchical application of two genetic algorithms that handle all the interactions among the three subproblems. The solution permits the use of complex functional units and allows operations to be implemented by a variety of functional units, possibly requiring different execution times. Our results show a reduction in time to obtain optimal solutions to standard benchmarks compared with other systems. In a straightforward way, our model can be extended to effectively use pipelined units or *chained* operations whenever there is an opportunity to do so. Furthermore, our model can also be extended to find optimal schedules and module allocations for multiple-block designs, not block-by-block or just along critical paths, but for all the blocks of a design simultaneously; we plan to publish the details of multi-block synthesis in a separate manuscript.

References

1. M. C. McFarland, A. C. Parker and R. Camposano, Tutorial on high-level synthesis *DAC*, pp. 330-336 (June 1998).
2. D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor and R. L. Blackburn, The system architect's workbench, *Proc. 25th Design Automation Conference ACM/IEEE*, (June 1998).
3. H. Trickey, Flamel: A high-level hardware complier, *IEEE Transactions on Computer-Aided Design* 6, 2 (March 1987).
4. S. Amellal and B. Kaminska, Functional synthesis of digital systems with TASS, *IEEE Transactions on Computer-Aided Design Conference*, *13*, (May 1994).
5. M. Nourani and C. Papachristou, Move frame scheduling and mixed scheduling allocation for automated synthesis of digital systems, *Proceedings of the 19Th Design Automation Conference*, pp. 99-105, (1992).
6. K. Wakabayashi and H. Tanaka, Global scheduling independent of control dependencies based on condition vectors, *Proceedings of the 29th Design-Automation Conference*, pp. 112-115, (June 1992).
7. N. Park, R. Jain, and A. Parker, Datapath synthesis of pipelined designs: Theoretical foundations in Progress in Computer-Aided VLSI Design 3 (Ablex Publishing Corporation) pp. 119-156, (1990).
8. W. Grass, J. Scheichenzuber, U. Lauther and S. Marz, Global hardware synthesis from behavioral data-flow descriptions, *Proceedings of the 27th Design-Automation Conference*, pp. 456-461, (1990).
9. P. Paulin, High-level synthesis of digital circuits using global scheduling and binding algorithms, PhD Thesis, Carleton University, Ottawa, (January 1989).
10. M. McFarland, Using bottom-up design techniques in synthesis of digital hardware from abstract behavioral descriptions, *Proceedings of the 23rd Design-Automation Conference*, pp. 474-480, (1986).
11. N. Mukherjee, An ILP solution for optimum scheduling, module and register allocation, and operation binding in datapath synthesis, *VLSI Design*, vol. 3, pp. 21-36,(June 1995).
12. G. Grewal, Scheduling, allocation, and binding in multiple-block synthesis, Master's Thesis, University of Guelph, Department of Computing and Information Science, Ontario, Canada, (April 1993).
13. C. Papachristou and H. Knouk, A linear program driven scheduling and allocation method followed by an interconnect optimization algorithm, *Proceedings of the 27th European Design-Automation Conference*, pp. 193-199, (1992).
14. c. Gebotys and M. Elmasry, Simultaneous scheduling and allocation for cost constrained optimal architecture synthesis, *Proceedings of the 28th Design Automation Conference*, pp. 2-7, 1991.
15. G. Syswerda and J. Palmucci, The application of genetic algorithms to resource scheduling, *Proceedings of the Fourth International Conference of Genetic Algorithms*, pp. 502-508, (1991).
16. Z. Michalewicz, Genetic algorithms + data structures = evolution programs, Springer, pp. 80-82, (1999).
17. Z. Michalewicz, Genetic algorithms + data structures = evolution programs, Springer, pp. 83, (1999).
18. D. Goldberg, K. Deb, and B. Korb, Do not worry, be messy, 4th International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, San Mateo, CA, pp. 24-30, (1991).
19. A. Zomaya, Parallel and Distributed Computing Handbook, McGraw-Hill, 1996.
20. H. El-Rewini, T. Lewis and H. Ali, Task scheduling in parallel and distributed systems, Prentice Hall, 1994
21. T. Lewis, H. El-Rewini, Introduction to Parallel Computing, Prentice Hall, 1992.
22. T. Adam, K Chandy, and J. Dickson, A comparison of list schedules for parallel processing systems, *Comm. ACM.*, *Vol. 17*, pp. 685-690, 1974.
23. A. Auyeung, I. Gondra, and H. Dai, Multi-heuristic list scheduling genetic algorithm for task scheduling, *The Eighteenth Annual ACM Symposium on Applied Computing*, pp.721-724, 2003.
24. D. Goldberg and R. Lingle, Alleles, Loci, and the TSP, *1st International Conference on Genetic Algorithms*, pp. 154-159, (1985).