Synthesis of Testable Pipelined Datapaths using Genetic Search

C.P. Ravikumar
Department of Electrical Engineering
Indian Institute of Technology
Hauz Khas, New Delhi
INDIA

V. Saxena *
Department of ECE
University of Illinois at Urbana Champaign
Urbana, IL
USA

Abstract

In this paper, we describe *TOGAPS*, a Testability-Oriented Genetic Algorithm for Pipeline Synthesis. The input to *TOGAPS* is an unscheduled data flow graph along with a specification of the desired pipeline latency. *TOGAPS* generates a register-level description of a datapath which is near-optimal in terms of area, meets the latency requirement, and is highly testable. Genetic search is employed to explore a 3-D search space, the three dimensions being the chip area, average latency, and the testability of the datapath. Testability of a design is evaluated by counting the number of self-loops in the structure graph of the data path. Each design is characterized by a four-tuple consisting of (i) the latency and schedule information, (ii) the module allocation, (iii) operation-to-module binding, and (iv) value-to-register binding. An initial population of designs is constructed from the given data flow graph using different latency cycles whose average latency is in the specified range. Multiple scheduling heuristics are used to generate schedules for the DFG. For each of the resulting scheduled data flow graphs, we decide on an allocation of modules and registers based on a lower bound estimated using the schedule and latency information. The operation-to-module binding and the value-to-register binding are then carried out. A fitness measure is evaluated for each of the resulting data paths; this fitness measure includes one component for each of the three search dimensions. We have implemented *TOGAPS* on a Sun/SPARC 10 and studied its performance on a number of benchmark examples. Results indicate that *TOGAPS* finds area-optimal datapaths for the specified latency cycle, while reducing the number of self-loops in the data path.

1 Introduction

The synthesis problem has been studied extensively in the recent past [5]. Our approach is different from the existing techniques in the following ways. We evaluate designs on not only area and time performance, but on structural testability properties as well. We use a genetic algorithm for exploring the 3-D design space characterized by area, time, and testability.

Earlier efforts in datapath have concentrated chiefly on minimization of chip area subject to timing constraints [3, 8, 9]. Recently, testability of the generated datapath has assumed importance [1, 6, 7]. Papachristou et al. introduced testable logic blocks ?TLB) in the construction of testable datapaths [7]. A TLB consists of a combinational logic block fed from a register Rl and feeding another register R2. In test mode, register Rl can be configured as a test pattern generator and R2 can be configured as a signature analyzer. Simple rules are defined in [7] to interconnect TLBs without forming self loops in the structure graph of the datapath. The structure graph of a datapath is a graph whose nodes correspond to registers in the datapath. A directed edge is drawn from a node i to node j if input of register j can be reached from output of register i through a purely combinational path. A self-loop involving a node i in the structure graph implies poor testability, requiring the register i be configured as a pattern generator and a signature analyzer at the same time, an impossibility unless an area-expensive concurrent BILBO is used [1, 7]. The binding phase in the synthesis process is most crucial in minimizing the number of self-loops in the structure graph. Avra [1] generates register bindings to minimize the number of self-loops without altering module binding. Mujumdar et al. use a network flow algorithm to carry out module and register bindings that minimize the number of self-loops [6]; they also proposed loop-breaking algorithms to further improve the testability of the resulting datapath. Existing approaches for testability-oriented synthesis address the problem of improving testability at the binding level; performance optimizations are first carried out using conventional scheduling and allocation heuristics. In *TOGAPS,* we address testability and area-time optimization in an integrated fashion at all the levels, namely, scheduling, allocation, and binding. To consider all the three design parameters simultaneously, we use a genetic algorithm, a powerful tool for exploring a large design space.

Section 2 describes the genetic algorithm for pipeline synthesis. In Section 3, we discuss the genetic operators (crossover and mutation) used in *TOGAPS.* The cost functions used in *TOGAPS,* and the techniques used to prune the large design space are discussed in Section 4. Experimental results are described in 5 and conclusions in Section 6.

***This author was a B.Tech student of IIT Delhi in EE Department when this work was carried out.**

2 Genetic Algorithm

Genetic algorithms have been used extensively in solving difficult combinatorial optimization problems [2, 10]. The idea behind a genetic algorithm is the biological principle of survival of the fittest. A population of individuals (designs in our problem) is maintained at any point of time. A new generation of individuals is created by applying genetic operators such as *crossover* and *mutation.* The crossover operator ensures that the resulting offspring (new designs) have a mixture of parental properties; the mutation operator introduces new characteristics not present in the parent population. The population pool consisting of parents and the offspring are graded (according to the cost function of the optimization problem) and the fittest individuals are retained as the next generation. Thus, over a sequence of generations, the overall properties of the population improves. The essential components of a genetic algorithm are the selection of the initial population, genetic operators, and the cost function used to grade individuals. In the subsequent sections, we describe these phases of the genetic algorithm, as applicable to the pipeline datapath synthesis problem.

2.1 Initial Population

A point in the design space can be represented as a four-tuple *(S,A,BM,BV),* where *S* represents the latency cycle and scheduling information, *A* represents the module and register allocation information, *BM* represents the operation-to-module binding, and *By* represents the value-to-register binding We view the design space as a three-level hierarchy; at the top level of this hierarchy is the latency/schedule information. Recall that one of the inputs to *TO GAPS* is the range of acceptable pipeline average latencies. There exist a number of latency cycles for a specified average latency *a.* Any of these latency cycles, can be selected and used along with any of the possible schedules for the DFG. The duple consisting of a latency cycle *L* and a schedule *S* forms the root of the design hierarchy. Specifying *(S, L)* would indirectly specify a design subspace; all designs within this subspace have similar timing performance. For a given *(S, L),* there is a lower bound on the number of resources (modules and registers). At the second level of the design hierarchy are a number of allocations that conform to this lower bound; we purposely consider designs which have more resources than the lower bound so as to perform area/testability tradeoff. Thus, specifying the allocation *A* of modules and registers further refines the design subspace defined by *\S,L).* At the lowest level of design hierarchy are the bindings *BM* and *By* • The module and the register bindings affect the structural testability properties such as the number of self loops. Specifying the binding information along with (<S,-4) completely specifies the design point. Figure 1 illustrates the design hierarchy.

2.2 Initial schedules and latencies

As an aid towards limiting the size of the search space, *TOGAPS* initially asks for a lower and an upper bound on the desired average latency. We maintain a database of possible latency cycles for each value of

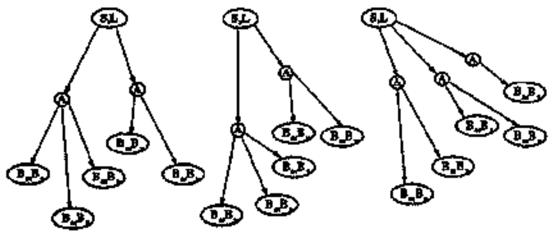

Figure 1: Hierarchical Structure of the Population

the average latency. From this database, we select *j* latency cycles randomly. For each of the *j* latency cycles *L{,* we form three schedule/latency pairs of the form *(Si, L{),* where the schedule 5,- is one of the following - a Force Directed Schedule (FDS) [9], an ASAP (As Soon As Possible) schedule and an ALAP (As Late As Possible) schedule [5]. Among the three scheduling algorithms, Force Directed Scheduling has the capability to consider the latency cycle in optimizing the number of resources [9]. We have selected the three scheduling algorithms in forming the schedule/latency pairs, with the view of providing diversity in the initial population.

2.3 Allocation

For a given schedule/latency pair *(Si,L{),* a lower bound on the number of registers and modules is determined using what we call the *Concurrent Operation Enumeration Table* (COET). Let the latency cycle in question be *Qi,* l_2, • • •, *h)*- The COET is an *F* x *P* matrix, where *F* is the number of operations in the DFG, and *P* is given by E*=i'«- The entry *COET(i,j)* is set to 1 if the operation *i* is active during time step *j.* Clearly, a lower bound on the number of functional modules of a type *q* is the maximum number of operations of type *q* that are concurrent at any particular time-step *j.* Define £>e/ia,i? to be 1 if operation *i* is of type *q;* otherwise *Deltaic* = 0. The lower bound LB_q for the number of functional modules of type *q* is given by

$$LB_q = Max_{j=1}^{P}(\sum_{i=1}^{F} COET(i,j) \cdot DeUa_{iit})$$

A lower bound on the number of registers is computed by performing a life time analysis on the values generated by each operation in the DFG. The procedure to generate the initial allocation is shown in Figure 2. The allocation procedure uses a binding algorithm *bindQ,* which is explained in the next subsection. If the allocation procedure cannot find any valid binding without violating the upper bound on the number of modules, it returns failure, and the corresponding schedule/latency pair *(Si, Li)* is discarded. For any *(Si,L{),* the *TOGAPS* system experiments with a number of resource allocations. Let A_q be the number of modules of type *q* at the end of the *Initial Allocation* procedure. Then *TOGAPS* uses $A_q + S_q$ modules of type *q* for generating more allocations, where $0 < S_q < n_q — A_,$, and n_q is the number of operations of type *q.* The δ_q are selected randomly in their respective range.

```
procedure    InltlalAlloeatlon()
(• fi is the number of initial binding* required to be generated.
  An upper bound on module of type q is the number of
  operations of type q. •)
begin
for each operation type q do
                    reaourceq  =  LBqi
nobindingt —— TRUE;
while   resources are within upper bound and   nobindings do
                    If  bind() = failure then
                                        add scarce resource
                    else
                                        nobindinga  =  FALSB;
If  nobtnding. = TRUE return   (failure)
b = 1;
while  6  <  0 do
                    If  b«nd() as success then  b  =  b + 1;
return  (success);
end
```

Figure 2: Algorithm for generating Initial Allocations

```
procedure    blnd()
 begin
   for each  operation j do
     begin
                    A_j = φ;
                    If all modules of type j have not been used at least once
                    begin
                                                  used module of type j;
                                        go  to   SKIP
                    end
                    for each  module i whose type matches with j do
                                        if  j  t  exclusion list  of  i  then
                                          Aj  =  Aj |J {i};
                    If  Aj = 0 then   return  (failure);

 SKIP:
                    Randomly seleci module k from AJ;
                    Assign operation j to module k;
                    Add all elements of j to the exclusion list of k;
                    for  the value v produced by j do  begin
                      A_v  =  0;
                      for  each  register r do
                                        It  v £ the exclusion list of r then
                                        A_v = A_v |J { r};
                                        if A_v a <p then  return  (failure);
                                           Randomly select a register a from A_v;
                                           Assign value v to register »;
                                           Add all elements of v to
                                            exclusion list of *;
                                        end
     end
 return  (success);
 end
```

Figure 3: Module and Register Binding

2.4 Binding

For a given operation i, we define the Exclusion List Ei as the list of operations which are concurrent with i during any of the time steps j, $1 \leq j \leq P$. Clearly, an operation k in Ei cannot be bound to the module to which i has been bound. The module binding procedure is the process of forming the reservation table R which has as many rows as there are functional modules and as many columns as the length of the critical path in the scheduled DFG. In *TOGAPS*, the operation-to-module binding is generated randomly, while ensuring that the resulting reservation table R has no conflicts.

Lemma 1 *The procedure bindQ shown in Figure 3 uses every allocated module and register at least once.*

Proof : Note that the procedure *bindf)* generates the availability list Aj of modules for each operation. If all the allocated modules of a particular type have not been used, the available-list consists of just one of the unused modules, which is hence selected by default. This ensures that all the allocated resources are used at least once. •

After generating the bindings we determine the actual connections that exist between the various modules and registers. This gives us the number of self-loops that exist in the circuit and the number of multiplexors that will be required. We assume a multiplexor-tree structure in case there are more than two inputs that must be multiplexed at the input pin of a module or a register. We assume that there is an independent latch control for each register. This permits us to optimize the use of registers and avoid register-chains, as far as possible.

3 Generation of New Designs

In this section we present genetic operators used in *TOGAPS* to produce new members from the existing population of designs. The purpose of these operators is to create new designs which retain some traits of their parent designs, while introducing some new features. This increases the diversity in the population and enables us to explore the design space better. A good genetic operator should introduce a reasonable amount of change, be fast to implement and produce valid designs. Two kinds of genetic operators are used in *TOGAPS:*

Crossover : A crossover operator merges the characteristics of two parent designs to form new design(s).

Mutation : After a crossover operator has been used to generate a new design, a mutation operator may be applied to introduce random changes in the new design. The purpose of the mutation operator is to allow hill climbing and hence avoid a local optimal solution; if mutation operators are not employed, the final solution generated by the genetic algorithm can become highly dependent on the initial population. If the mutation probability is $p^\wedge$, then, on an average, $N \bullet Pp$ of the N offspring are subjected to mutation.

We have explained in the previous section that the design population is maintained hierarchically. Accordingly, the genetic operators in *TOGAPS* also work at all the three levels of design hierarchy i.e. at the schedule and latency level, at the allocation level, and at the binding level.

3.1 Crossover Operators

TOGAPS uses three crossover operators, two of which work at the topmost level of design hierarchy, and a third one which works at the lowest level of design hierarchy.

3.1.1 Schedule/Latency Level Operators

The crossover operators 7MS and *JRS* change the schedule, utilizing the information obtained by the topological sorting of the nodes of the DFG. An ordered list T of DFG nodes is said to be topologically sorted (^-sorted for brief) if $i < j$ and $T(i) < T(j)$ imply that $T(j)$ does not depend on $T\{i)$. A t-sorted list corresponding to the DFG in Figure 4 [9], is T — $\{H,],J, K, A, B, D, E, C, F, G\}$. Our crossover operators use the concept of topological sorting.

The *JMS* **(Modify Schedule**) operator merges two designs *(Father* and *Mother)* to produce two new designs *(Son* and *Daughter).* Let n be the number of nodes in the DFG. Let T be a topologically sorted list of the n nodes in the DFG. We select a random number,

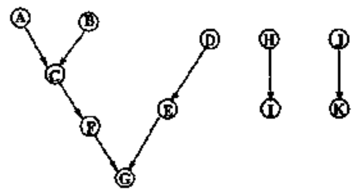

Figure 4: Example of a Data Flow Graph

```
procedure    ScheduleSon(T, i, Father, Mother)
(* Father and Mother are parent designs. T is a
  top ologic ally sorted list of nodes in the DFG. *)
begin
for  j ' s l t o  'tdo
                    schedule T(j') as in parent design Father;
for  j' = t' + 1 to  n do  begin
                    Let rjj be the time step in which
                    the schedule of Mother executes T(i);
                    Let ξj be the earliest time step in which
                    T(j) can b  cheduled w'ihout violating
                    dependenceVdue to T(l)' • • • T(i);
                    schedule T(j') in time step maxyqj, £j);
end
end
```

Figure 5: Modify-Schedule Crossover

distributed uniformly in the range 2 to n — 1 as the "pivot". The procedure of Figure 5 is used to create a schedule *Sson* from the parent designs. The latency information for *Son* is taken from *Father*. We repeat the above procedure for creating *Daughter*, with the roles of the *Father* and the *Mother* interchanged.

The *fRs* **(Randomized Schedule)** operator is similar to *IRS*, but for the fact the latencies of the children designs are determined randomly, rather than taking them from one of the parents.

3.1.2 Operator at the Binding level

This operator, which we call 7MB (Merge **Bindings),** takes two bindings *(father* and *mother)* and generates two new bindings *(son* and *daughter)*. For the son, we take the module bindings from the *father* and the register bindings from the *mother*, while for *daughter*, we take the module bindings from the *mother* and the register bindings from the *father*. Thereafter we determine the connections and the number of self-loops and multiplexors required by the two new bindings.

3.2 Mutation Operators

TOGAPS uses five different mutation operators, two of which work at the highest level of schedule & latency, while the other three work at the next level of allocation. We give a brief description of each of the five operators.

3.2.1 Operators at the highest level

MPD (Insert **Pure Delay)** : It is well known that addition of a pure delay in the pipeline reservation table can often lead to superior performance. As the name suggests, the operator */IJPD* adds a pure delay in the design schedule. This increases the total execution time required for the processing of one input instance. The selection of the time steps between which we insert the delay is random. For the new schedule/latency pair, we generate the allocations and bindings as described in the previous section.

HlDF **(Insert Delay for FDS Schedule)** : Since the control step for an operation in the FDS schedule is intermediate between its ALAP and ASAP values, adding a pure delay gives the FDS algorithm greater flexibility in scheduling the operations. This is done with the hope of achieving a more uniform distribution of operations over various control steps and hence reduce the resource requirement.

In this operator we take the parent schedule and add a pure time delay in both the ASAP and ALAP schedules corresponding to the parent schedule. The delay is inserted between the same two time steps in both the ASAP and ALAP schedules. Thereafter we generate the FDS schedule for the new ALAP and ASAP schedules with the latency taken from the parent. We take the new FDS schedule and the latency from the parent to form a new duple at the highest level.

3.2.2 Operators at the Allocation Level

PAR **(Add register**) : Addition of a register may improve the structural testability of the datapath, by giving more flexibility to the binding operator in minimizing the number of self-loops. The *(XAR* operator increases the number of registers in the allocation of the offspring, while retaining the number functional modules.

HDR **(Delete register**) : The reverse of the *\XAR* mutation operator, *\IOR* operator deletes a randomly selected register and reassigns the values to the remaining registers. The deletion of a register reduces chip area, but may affect the testability of the data path.

V-MA **(Modify Allocation**): This mutation is a generalization of *HAR* and *(J,DR-* When *\IMA* is applied, the allocation of both functional modules and registers are modified. For instance, *UMA* may increase or decrease the number of adders; this manipulation is done while respecting the lower bound and upper bound on the number of functional modules. The reallocation is followed by the generation of bindings to form new designs.

4 Pruning of the population space

Once we have generated a number of designs from the current generation of deigns, we need to prune the population space to eliminate poor designs. Genetic Algorithms (GA) make use of cost functions to select designs that shall survive until the next generation. In *TOGAPS*, the process of pruning of the population is carried out hierarchically. We start from the lowest level of bindings and proceed upwards towards the schedule/latency level, eliminating poor designs at each level. A separate cost-function formulation is used at each stage.

4.1 Pruning of Bindings

Recall that during the synthesis process, we create a number of bindings for each allocation. The first stage of pruning is carried out at the binding level. The cost function at this level considers the number of self-loops in the data path. For each allocation *Ai*, we calculate the average number of loops per binding $/^{a}V$ We then visit each of the binding *Bj,* for the current allocation *Ai,* and discard the binding if the number of self-loops in *Bj* is greater than $l^{av}i$. This procedure is repeated for each of the allocations, for all the schedule/latency duples.

4.2 Pruning at the allocation level

In the next stage of the pruning process, we reduce the number of allocations for each of the schedule/latency pairs. The cost function at the allocation level considers the area requirement and the average number of self loops. For each schedule/latency pair *(Si,Li),* we calculate the average functional area requirement *ai* using an area estimator (Section 4.4) and the average number of self-loops A,-. To calculate Aj, we first visit all the bindings *Bk* for an allocation *Aj* and compute the average number of self-loops per binding, *X'j,* for *Aj*. Then we visit each allocation for the duple *(Si, Li)* and determine A,- as the average of Aj. Thus, for each allocation *Aj* of duple *(Si, Li),* we have

$$COSt(Aj) \quad = \quad W_{ar}ea * (a(Aj)/aj) + W_{loops} * (X'j/Xf)$$

where W_{area} and $Wi_{OO}p\$$ are positive fractional weights which are user-specified and indicate the relative importance of area and structural testability in the final solution. For a schedule/latency duple *(Si, Li),* we discard all allocations *Aj* which have a cost *cost(Aj)* greater than the average cost per allocation for that (Si, L_t).

4.3 Pruning at the Highest Level

The cost function at this level has two terms, a first term which measures the area versus time performance of the design, and a second term which measures the structural testability of the design. For each schedule/latency pair *(Si, Li),*

$$\begin{aligned} Cost(i) \quad &= \quad W_{A}reaL_{a}tency(Latencyi\text{-}Areai/6) \\ &+ \quad W_{SelfLoop} * \quad (SelfLoopsi \;\; ftp) \end{aligned}$$

where *Latency,·* is the average latency for the duple *i, Area;* is the average area per allocation calculated over all allocations of the duple *i,* and *SelfLoopsi* is the average number of self-loops per binding for all the designs originating from duple *i.* *6* and V are both normalizing factors and are calculated over all the duples at the highest level of the design hierarchy. *WAreaLatency* and *Wself Loops* are the weights attached to the two parts of the cost function; both these weights are positive real.

We remove those schedule/latency pairs whose cost is greater than the average cost per unit. The above cost function may be justified as follows. In a pipeline, as the average latency decreases, the number of resources increases, since the circuit will be required to perform a larger number of operations during a shorter time-period. The first term in the cost function reflects this trade-off. Comparison of two designs purely on the basis of their areas, without considering the average latency of the designs, would be inappropriate. The second part of the cost function reflects the average structural testability of the family of designs that correspond to the duple *(Si, Li).* Thus the overall cost function reflects a tradeoff between area, delay, and structural testability properties of the designs.

4.4 Area Estimate

Since *TO GAPS* needs to estimate the area of a large number of designs, the area-estimator used should be fast, apart from being reliable. We use the following linear function which estimates the chip area in terms of the number of functional modules, registers and multiplexors used. Two types of functional modules are assumed, adders and multipliers.

$$\begin{aligned} Area \;=\; a \;\; Adder \; * \; riAdd \; + \; «M«» \; * \; \hat{}MuU \quad (1) \\ + \quad \alpha_{Reg} * TlReg \; + \; OtMux \; * \; «Mux \end{aligned}$$

where *nAdd,* "jtftiit, *nReg,* and *n&iux* are respectively the number of adders, multipliers, registers, and multiplexors required. $a_{A}dder$, $<x_{MuH}$, a_{Reg}, a_{Mux} are respectively the area of a single adder, multiplier, register, and multiplexer. The module areas are library-specific; in *TOGAPS,* we used data from a XILINX 4000 FPGA library and counted the number of Configurable Logic Blocks (CLB) required for a module as an estimate of the module's area. When a &-input multiplexer, $k > 2$, is required we assumed that the same is implemented as a multiplexer tree using $k — 1$ two-input multiplexers. Since we are interested in comparing the resource requirement of various designs, we are not concerned by the absolute values generated during the estimate, as long as the values are proportional to the final area requirement for each design.

5 Experimental Results

We ran *TOGAPS* on a number of benchmark circuits. To limit the size of the design space, we did not consider module chaining. In case of multi-step operations, we assumed that the concerned modules were in turn pipelined, ensuring that they are available to process a new input instance at every time step. In all the examples we were able to produce designs which were near-optimal in terms of the ALU resource requirement, while keeping the number of self loops small.

Tables 1 through 4 show the results obtained using *TOGAPS* on four different benchmark DFGs, namely, the AR-filter, Bandpass filter, FIR filter and Biquad filter. The number of additions and multiplications in each DFG are indicated in the table captions. In all our experiments, we used 10 generations, 5 to 10 crossovers and mutations per generation. The initial number of schedule/latency pairs varied in accordance to the size of the desired range of latencies. For each schedule/latency pair, the number of initial allocations was chosen to be 5. For every allocation, the number of initial bindings was fixed at 10. *A, M,* and *int* are the number of adders, multipliers, and registers + multiplexers used in each of the designs indicated. Area is the number of CLBs required. In each table, we show different designs obtained for different average-latencies. For a given average-latency, *lb(A)* and *lb(M)*

nvg lot	ib A	Ib M	A	M	int	e	<	x	y	κ	cpu (•)
2	6	8	6	8	30	632	6	14	6	3452	51.24
3	4	6	6	8	31	633	6	14	5	3248	42.78
4	3	4	4	4	32	379	5	14	5	3730	49.20
5	3	4	3	4	32	370	7	14	5	3872	54.90

Table 1: Results for AR Filter (12 add. and 16 mult.)

avg ut	lb A	lb M	A	M	int	θ	ζ	x	y	κ	cpu (•)
2.5	7	5	9	6	31	450	5	18	5	3098	43.93
3.5	6	4	6	5	35	462	5	15	4	3326	49.20
4.0	5	3	6	3	38	353	6	16	5	3872	43.83
4.5	4	3	7	4	29	391	5	15	5	3234	47.64

Table 2: Bandpass Filter (17 add. and 12 mult.)

are the lower bounds on the number of adders and number of multipliers, respectively. *lb(A)(lb(M^)= \n/al\,* where *n* is the # of additions (multiplications) in the DFG and *al* is the average latency. The number of self loops in the resulting data path £, the functional area estimate *9*, the number of designs searched *K*, and the CPU requirement (SUN SPARC10) of *TOGAPS* are also indicated in the tables. During each experiment, we also kept track of the maximum number of self-loops and the minimum number of self-loops over the space of all the data paths examined by *TOGAPS* (*x* and *y* in the tables). It can be observed from the tables that the majority of the designs generated using *TOGAPS* are indeed equal or close to the lower bounds on the number of functional modules, while keeping the number of self-loops in the data path at a low value. The number of self-loops in the final designs generated through *TOGAPS* were, as a rule, close to the minimum number of self-loops encountered.

6 Conclusions

The optimization technique employed in *TOGAPS* is based on genetic search, which can handle multiple objective functions and constraints (such as testability, number of resources, and timing information) through an appropriately designed cost function. The relative importance of each objective function can be controlled through user-specified weights. A novel hierarchical representation of a design was used in *TOGAPS* to achieve efficient search, storage, and pruning of the design space. This concept of hierarchy can be extended in useful ways; for instance, we are now exploring the possibility of introducing a further level of hierarchy at the top which allows us to explore *behavioral level transformations* of the data flow graph.

a.vg l&t	Ib A	Ib M	A	M	int	8	C	x	y	κ	cpu («)
3.0	5	3	5	3	30	320	4	15	4	5750	63.23
3.5	5	3	6	3	24	309	4	14	4	5310	58.40
4.0	4	2	4	2	32	256	5	15	3	4984	61.58
4.0	4	2	6	2	28	262	4	13	3	3066	34.37

Table 3: Results for FIR Filter (15 add. and 8 mult.)

Ut	Ib A	it M	A	M	int	S	<	x	V	κ	cpu (•)
2.0	8	4	6	4	27	399	5	13	5	4034	43.58
3.0	5	3	5	3	36	343	6	12	5	1170	13.00
4.0	4	2	5	3	26	308	5	13	4	5186	53.21
5.S	5	2	4	3	28	307	5	13	5	5020	53.75

Table 4: Biquad Filter (15 add. and 7 mult.)

Acknowledgements

We thank Prof. R. Jain and Mr. A. Mujumdar of the Department of Elect, and Comp. Engg., Univ. of Wisconsin, for the benchmark circuits and helpful discussions.

References

[1] L. Avra. Allocation and assignment in high-level synthesis for self-testable data paths.' In *Proc. of Int. Test Conf,* pages 463-472, 1991.

[2] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, Reading, Massachusetts, 1989.

[3] K. Kucukcakar. *System-Level Synthesis Techniques With Emphasis on Partitioning and Design Planning.* PhD thesis, University of Southern California, October 1991.

[4] S.-P. Lin, C. Njinda, and M. Breuer. Generating a Family of Testable Designs using the BILBO Methodology. *Journal of Electronic Testing: Theory and Applications,* pages 71-89, 1993.

[5] A.C. Parker M. McFarland and R. Camposano. The high level synthesis of digital systems. *Proceedings of the IEEE,* 78, February 1990.

[6] A. Mujumdar, K. Saluja, and R. Jain. Incorporating testability considerations in high-level synthesis. In *Proceedings of the Fault-Tolerant Computing Symposium,* pages 272-279, 1992.

[7] C.A. Papachristou et al. A framework for high-level synthesis with self-testability. Tech. Rep., Comp. Engg. and Sci. Dept., Case Inst. of Tech., CWRU, Feb. 1991. CES-91-03.

[8] N. Park and A.C. Parker. SEHWA - A program for synthesis of pipelines. In *Proc. of the 23rd Des. Aut. Conf,* pages 454-460, 1986.

[9] P.G. Paulin and J.P. Knight. Force directed scheduling for the behavioral synthesis of ASICs. *IEEE Transactions of CAD,* 8(6):661-679, 1989.

[10] C.P. Ravikumar and A.K. Gupta. A genetic algorithm for mapping tasks onto a reconfigurable parallel processor. IEE Proceedings(E) (142) Mar 1995, pages 81-86.