International Journal of Computers and Applications

ISSN: 1206-212X (Print) 1925-7074 (Online) Journal homepage: http://www.tandfonline.com/loi/tjca20

Concurrent BIST Synthesis and Test Scheduling Using Genetic Algorithms

H.M. Harmanani & A.M.K. Hajar

To cite this article: H.M. Harmanani & A.M.K. Hajar (2007) Concurrent BIST Synthesis and Test Scheduling Using Genetic Algorithms, International Journal of Computers and Applications, 29:2, 132-142

To link to this article: http://dx.doi.org/10.1080/1206212X.2007.11441841

International Journal of Computers and Applications, Vol. 29, No. 2, 2007

CONCURRENT BIST SYNTHESIS AND TEST SCHEDULING USING GENETIC ALGORITHMS

H.M. Harmanani* and A.M.K. Hajar*

Abstract

This paper presents an efficient method for concurrent built-in self-test *synthesis* and *test scheduling* in high-level synthesis. The method maximizes concurrent testing of modules while performing the allocation of *functional units*, *test registers*, and *interconnects*. The method is based on a *genetic algorithm* that efficiently explores the testable design space and finds a sub-optimal test registers assignment for each k-test session. The method was implemented using C++ on a Linux workstation. Several benchmark examples have been implemented and favorable design comparisons are reported.

Key Words

Testable synthesis, test scheduling, genetic algorithms

1. Introduction

High-level synthesis is the translation of a behavioral description into a structural one by finding the structure that best meets the constraints while minimizing other costs. From the input specification, the synthesis process produces a description of a register-transfer level (RTL) circuit, that is, a network of registers, functional units, multiplexers, and buses [1]. The data path is usually synthesized in two steps: *operations scheduling* and *resources allocation*. There are usually constraints on the design that may limit the total area, throughput, delay, or power consumption. The high-level synthesis process involves navigation through the design space making appropriate trade-offs until the best solution satisfying the constraints is reached. High-level synthesis has several advantages. First, it increases designers productivity by exploring the design space in a relatively short time; the resulting synthesized designs are correct by construction. Another advantage is that it reduces silicon literacy requirements as designers do not have to be familiar with the details and peculiarities of technology-specific aspects. The main difficulty in this process, however, is that most behaviors have no implied architecture. Thus, it is rather difficult to develop synthesis algorithms that will generate the same quality from all possible design descriptions.

VLSI chips designers, on the other hand, have long realized the importance and advantages of incorporating testability early in the design process. It is estimated that the cost of the test process is growing and that it is between 30% and 40% of the total production cost. Design for testability (DFT) methodologies emerged to reduce testing cost and they range from ad-hoc techniques to structured approaches such as *scan design* and *built-in self-test* (BIST) design. BIST is a "test per clock" DFT technique that allows a circuit to test itself by embedding external tester features into the chip itself [2]. The main idea of BIST is to configure the data path, during test mode, into a number of self-testable *combinational logic blocks* (CLBs) or *kernels* that are fed directly or indirectly by pseudorandom pattern generators, and each output feeds either directly or indirectly a signature analyzer. To reduce test time, kernels are organized into *test sessions*. Obviously, the minimal test time would be achieved by simultaneously testing all kernels; however, design constraints may prevent this full parallelism. For example, tests maybe conflicting because they share common test resources such as a test bus or a test response compactor [3]. A test session brings together the tests of compatible kernels; this compatibility is checked with respect to the test resource sharing needs.

Consider the circuit shown in Fig. 1(a). To test this circuit under the BIST methodology, we *configure* registers R_1, R_3, and R_4 as test pattern generators (TPGRs) while register R_2 is configured as a multiple input signature register (MISR). Register R_5 is configured as a *built-in logic block observation* (BILBO) register, which is a test register that in addition to its normal operation mode, operates during test mode as an MISR or as a TPGR but in different test sessions [4]. The modified circuit can now be tested in two test sessions; $T_1 = \{ModuleA, TPGR_3, TPGR_4, BILBO\}$ and $T_2 = \{ModuleB, TPGR_2, BILBO, MISR\}$. The test time for this circuit is equal to the test time needed to test both sessions, $T_1 + T_2$.

To explore further trade-offs between test time and hardware overhead, the circuit in Fig. 1(a) can be re-

* Department of Computer Science and Mathematics, Lebanese American University, Byblos, 1401 2010 Lebanon; e-mail: haidar@lau.edu.lb, aouni.hajar@gmail.com

Recommended by Dr. Sunil Das
(paper no. 202-1774)

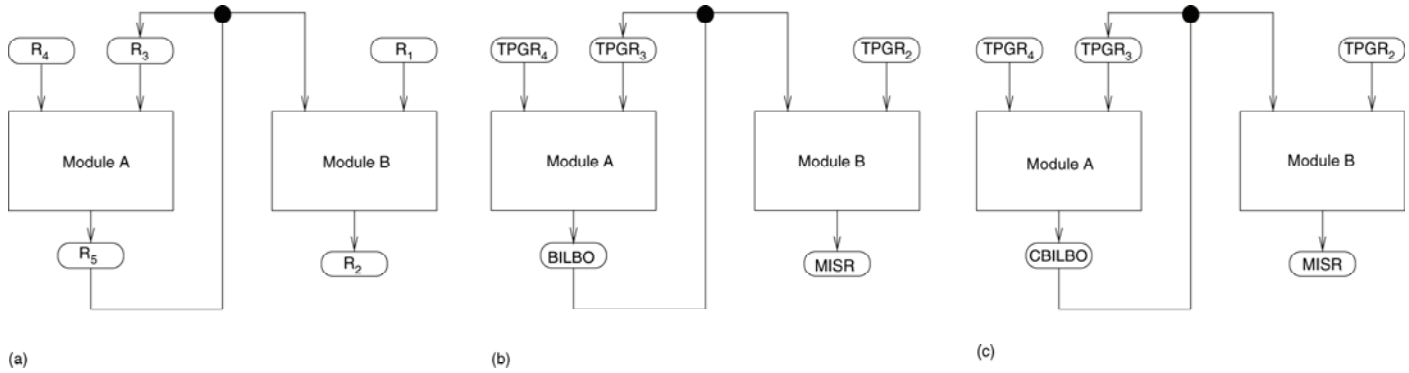


Figure 1. (a) Simple circuit; (b) testable circuit in two test sessions; and (c) testable circuit in one test session.

designed by reassigning the registers' test attributes. For example, R_5 maybe reassigned as a CBILBO [2] where a CBILBO is a test register that can operate *simultaneously*, in the same test session, as an MISR and a TPGR. This leads to the circuit shown in Fig. 1(c), which can be tested in one test session, $T = \{ModuleA,\ TPGR_3,\ TPGR_4,\ CBILBO,\ ModuleB,\ TPGR_2,\ MISR\}$. The disadvantage of the CBILBO register is that it is very costly in area (about 1.75 times the size of a BILBO [5]) and induces more delay during normal operation mode.

1.1 Genetic Algorithms

Genetic algorithms (GAs) [6] are stochastic combinatorial optimization techniques based on evolutionary improvements. A operate on a population of knowledge structures, chromosomes, that represent candidate solutions. During every generation, a number of chromosomes with the worst fitness values is removed from the population and replaced by new chromosomes obtained from applying genetic operators. The execution of the algorithm is iterated until either the best chromosome, representing an optimal solution, is found or a predetermined stopping condition, such as maximum number of generations or maximum number of fitness evaluations, is satisfied. In the later case, the chromosome with the best fitness in all generations is returned.

1.2 Related Work

Several deterministic approaches for high-level synthesis with test have been investigated [7]. One of the earliest high-level BIST synthesis was proposed by Papachristou *et al.* [8, 9] where a method that generates self-testable designs without self-adjacent registers was proposed. The method was later improved in Harmanani *et al.* [10]. Avra [11] proposed a register allocation method that minimizes the number of self-adjacent registers in the datapath and resolves the conflicts due to remaining self-adjacency by using CBILBOs. Parulkar *et al.* [5, 12] proposed a method that minimizes the sharing of test registers to reduce BIST area overhead. During the register assignment phase, input and output variables of data flow graph (DFG) are merged in a maximal sharing of the registers to avoid self-adjacent registers. The aim of the method is to ensure that the functional area is not compromised in the quest for low BIST area overhead. Olcoz *et al.* [13] proposed a method that explores area trade-offs during testable data path allocation. The method is based on the premise of no design space restriction such as avoiding self-adjacent registers. Recently, Zwolinski *et al.* [14] proposed a BIST method with design and time exploration.

The above approaches investigated test synthesis *only* and did not approach the problem of minimizing test time. While Craig *et al.* [15] proposed one of the most classical work in test scheduling through optimal and sub-optimal procedures, Harris *et al.* [16] investigated synthesis techniques to synthesize data paths with BIST such that the time required for testing is minimized by examining conditions that prevent concurrent testing of modules. Recently, Kim *et al.* [17, 18] proposed a BIST datapath synthesis approach based on ILP that performs testable synthesis in addition to test scheduling. Other researchers tackled the synthesis problem using non-deterministic approaches. Dhodi [19] proposed an GA for datapath synthesis where the problem is altered using an GA and then transformed into solution space by means of a heuristic.

1.3 Problem Description

This paper presents a concurrent method for *testable high-level synthesis* and *test scheduling*. Given a behavioral description of a digital circuit and a set of design constraints, we use an GA to generate a *self-testable RTL datapath* that: (1) implements the original behaviour; (2) minimizes the overhead of test registers; and (3) minimize test overhead within a given test session. The main features of the proposed method are:

- *A model for testable RTL synthesis datapath* based on the BIST methodology. The method discriminatingly incorporates all kind of test registers including BILBOs and CBILBOs.
- Concurrent allocation of *functional units*, *interconnects*, and *test registers* during the synthesis process. The method finds a sub-optimal test registers assignment for each k-session. This allows the designer to trade-off design area, test time, and the number of test sessions.
- Rapid exploration of the complex design space using an efficient GA.

The rest of the paper is organized as follows. Section 2 formulates the *genetic testable datapath synthesis problem* including the chromosomal representation, genetic operators, and cost function. Section 3 describes the genetic algorithm while experimental results are presented in Section 4. We conclude with remarks in Section 5.

2. Genetic Test Synthesis Formulation

Consider a DFG node associated with a variable instance V. It corresponds to an *operation* that must be assigned to a functional unit and to a *value* that must be assigned to a register for the duration of the variable's life span $L(V)$. The input edges of node V are connected to the outputs of other nodes.

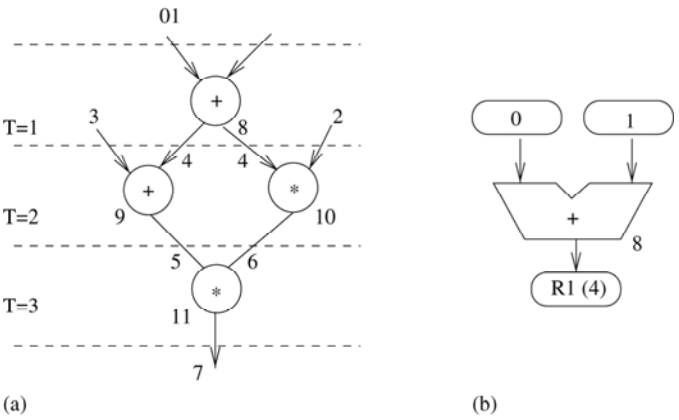


Figure 2. (a) Simple DFG and (b) kernel that corresponds to node 8.

Fig. 2(a) shows a simple DFG where node 8 corresponds to the test kernel shown in Fig. 2(b). The kernel has an adder functional unit, two registers at the input ports, and one register at the output port. The test attributes for the registers depend on other kernels as kernels' inputs are connected to other kernel's outputs. The objective of our method is to map the DFG nodes onto *test kernels* and to incrementally explore the design space by minimizing the area of the k-sessions that are generated where $k = 1, 2, 3, \ldots, N$ and N is the number of test kernels in the circuit. The algorithm finds a range of designs with different area and test time where every BIST circuit for a k-test has the least area. There are two conditions that should be met for the successful merging of two test kernels:

1. There is no conflict in the use of the kernels' functional units. Furthermore, the merger should result in a feasible functional unit that exists in the library.
2. Test registers are assigned concurrently with test scheduling. This provides a trade-off among area, delay, and test time.

The first condition implies that DFG nodes are scheduled at different clock cycles while the second condition is accomplished through the use of test registers including CBILBOs. Formally, kernels that can be merged under the above conditions are called *compatible*.

2.1 Chromosomal Representation

Each chromosome represents a candidate datapath solution that implements the original behavior DFG. The chromosomal representation consists of a vector whose length is equal to the number of test kernels in the DFG. A gene or to be precise its position in the chromosome represents a test kernel as shown in Fig. 3. The kernel's inputs and outputs are modeled using a vector whose size is a function of the total number of clock cycles and contains references to other test kernels. Thus, if n is the total number of clock cycles, a kernel is connected to an array of size n at each port.

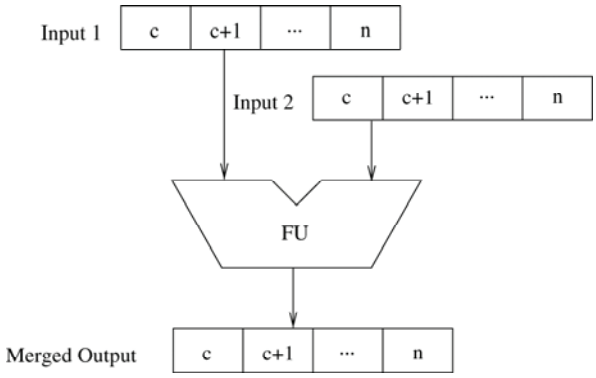


Figure 3. General *gene* representation.

Whenever two nodes are merged and assigned to the same hardware resource, the index of the genes are updated as follows. Assume that kernel K_i is scheduled before K_j. Then, if K_i is merged with K_j, the inputs of K_j are appended to K_i's inputs at an index equal to the K_j's clock cycle. Therefore, all indexes in the K_i's arrays that are smaller than K_i's cycles are not used. Fig. 4 illustrates the simple merger of two test kernels that correspond to DFG nodes 10 and 11.

2.2 Chromosomal Implementation

To improve the computational efficiency, the algorithm uses a dual representation to represent a chromosome. Thus, merged kernels in a chromosome are represented using a parent/child relationship using a doubly *linked list*. The list maintains references to the next child node, to the parent node, and to the absolute parent node. The list is sorted based on the clock cycles of each node. Thus, the search and insertion order complexity in the list is $O(N)$.

As the GA requires heavy access to all kernel's merged inputs and outputs, a second level of indexing is added using a hashed representation. The *hash table* reduces the complexity caused by the linked representation and increases the efficiency of inserting as well as locating parent/children nodes in a kernel. The advantage is that kernels' search, insertion, merge, and split are now done in constant time, $O(1)$.

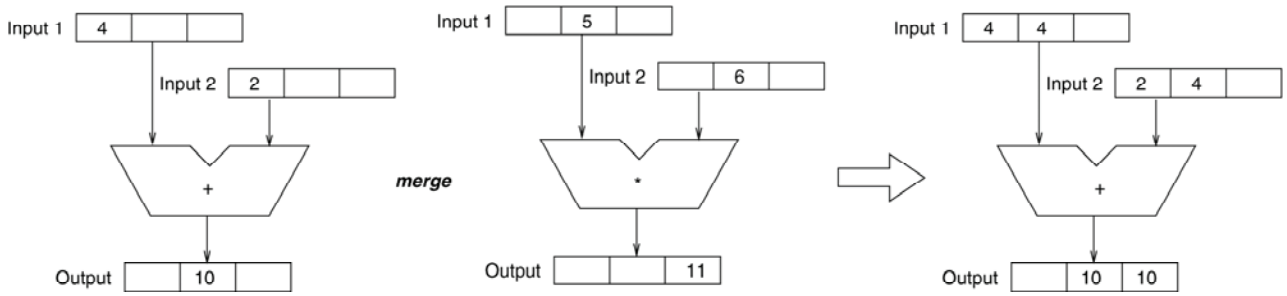


Figure 4. Kernels merger example: node 10 and node 11.

2.3 Test Registers Representation

The data path consists of modules, registers as well as the connections among these entities. The usage of each test register can be *isControllable*, *isObservable*, or *isConcurrent*. The final implementation of a register is the union of the underlying register usage. For example if a register is a TPGR for one module and an MISR for another module in the same test session, then this register will have a final implementation attribute as a CBILBO. This implementation can indicate the hardware overhead before and after merging data flow nodes by computing the difference in cost over two test sessions. Another advantage of having three different attributes is that the algorithm incrementally assigns one attribute at a time and evaluates the trade-off before it finally commits to one solution.

Finally, registers are represented using a binary tree structure where each node is connected to one vertical edge and one horizontal edge rather than two diagonal edges; the vertical edge represents a list of all merged registers while the horizontal edge links all physical registers.

To determine the number of registers at a port, we loop over the hash table attached at that port. If the register does not have a parent, that means it is a input/output register. To determine the type of a given register, we OR all test attributes in the list of that register.

2.4 Initial Population

At the beginning of each run, an analysis of the DFG is performed and DFG compatibility relations as well as registers life spans are analyzed. Compatibility relations are stored in a *compatibility graph*, $G_{comp}(V, E)$, that consists of vertices V denoting operations and edges E denoting the compatibility relations among DFG nodes. Compatible nodes are connected with edges in the graph and correspond to a *partial binding*. The initial population is generated based on problem-specific data in two steps. First, an initial chromosome is generated where each DFG node is mapped to a test kernel. Second, the remaining chromosomes are generated through a random enumeration of partial feasible bindings that are directly derived from the compatibility graph $G_{comp}(V, E)$.

2.5 Cost Function

The objective function measures the fitness of each chromosome in the population and is crucial for the transmission of gene information to the next generation. The cost of the datapath in terms of *area* is the sum of the cost of registers, multiplexers, and functional units for a given k-session and is given as follows:

$$F = \sum_i A_{fu}(i)N_{fu}(i) + A_r N_r + \sum_j A_{tr}(j)N_{tr}(j), \quad \text{for each } k\text{-session}, \quad \text{where } k = 1, 2, \ldots, N \qquad (1)$$

where $N_{fu}(i)$ is the number of functional units of type i and $A_{fu}(i)$ is the corresponding area. N_r is the number of normal registers and A_r is the area of such a register. $N_{tr}(j)$ is the number of test registers of type j (TPGR, MISR, BILBO, CBILBO) while $A_{tr}(j)$ is the area of test register of type j. Finally, $M_{mux}(l)$ is the number of multiplexers of type l and M_l is the corresponding area.

2.6 Selection

The selection process is important to maintain a mixture of chromosomes based on fitness. Our objective was to have a selection scheme that ensures that the best individuals are maintained while weaker individuals have a lower probability of being selected. The motivation for keeping the bad chromosomes is to help finding better solutions by disturbing the state of the design space.

We have explored with various selection schemes that ensure a good mixture of chromosomes based on fitness including *roulette wheel* selection. However, we have determined empirically that the best selection technique was based on a weighted, biased selection based on the following:

1. Select 2% of the best chromosomes and 10% of the worst chromosomes.
2. Select 25% of those chromosomes whose fitness is between 90% and 100% of the best chromosomes.
3. Select 40% of those chromosomes whose fitness is between 25% and 75% of the best chromosomes.
4. Select 20% of the chromosomes whose fitness is between 17% and 30% of the best chromosomes.
5. Randomly select 5% of the remaining chromosomes.

2.7 Genetic Operators

To explore the design space, we use two genetic operators: *mutation* and *crossover*. The genetic operators are applied iteratively in each generation and by taking turns.

2.7.1 Mutation

Mutation is a generic operator that is used for finding new points in the search space. We use a novel mutation operator based on a *split/merge* mechanism. Thus, a kernel is *split* into two kernels or two kernels are *merged* into one. It should be noted that the algorithm encourages more mergers than splits as the split operation is destructive and has more of a hill-climbing effect. The mutation operator merges 80% of the time and splits 20% of the time subject to the mutation probability P_m.

2.7.2 Crossover

To improve the quality of the solutions found, one needs to overcome the information loss that occurs when the algorithm converges to a solution. This was done through the use of a uniform *crossover* operator. Crossover is a reproduction technique that mates two parent chromosomes and produces two child chromosomes. The algorithm randomly selects two chromosomes and then selects two compatible kernels pseudo-randomly from the compatibility graph. The operator checks the status of the edge between the selected nodes. If there is an edge, then a *merge* operation is applied otherwise the kernel is *split*. If the probability is lower than the specified threshold, no operation is applied.

3. Algorithm

Every chromosome represents an intermediate data path that has different number of registers, multiplexer inputs, functional units, and controller cost. During every generation, chromosomes are selected for reproduction, resulting in new datapaths. This is accomplished by merging compatible kernels or splitting kernels within chromosomes.

An initial population is first selected and all input registers are assigned a $isTPGR$ attribute while the $isMISR$ attribute is assigned to all output registers. The $isConcurrent$ attribute is assigned to the remaining registers. The system then loops for $N_g \times iSteps$ times where N_g is the maximum number of generations and $iSteps$ is the number of incremental steps that reproduces and evaluates a new population (Fig. 5). The incremental step invokes the kernel evolution process and monitors the solution feasibility by checking a set of compatibility rules.

The genetic optimizer selects one operator per incremental step, either *mutation* or *crossover*. The selected operator is applied on randomly selected chromosomes and genes. There are six operations that are required every time two kernels are either merged or split (Fig. 6). If a violation occurs during the process, the whole process is aborted. The kernel evolution process commits one action at a time. The operations are: *node merge*, *register merge*, and *MUX merge*. Furthermore, the algorithm

```
Evolutionary_Synthesis(Scheduled_DFG)
{
  Read the scheduled DFG and the resource library.
  Get the population size (P_size) and the nb. of
    generations (N_g).
  Generate an initial population, current_pop.
  Evaluate(current_pop)
  Keep_the_best()
  for i = 0 to N_g
  {
    for j = 0 to iSteps do
    {
      if (i % 2 == 0)
        Select a random chromosome from current_pop
          for mating.
        apply crossover with probability P_xover
              Select two compatible genes
              Split one of the two genes
      else
        Select a random chromosome from current_pop
          for mating.
        apply mutation with probability P_m
              Select two compatible genes
              Split one of the two genes
      kernel_evolution_process()
      Evaluate the population fitness using equation 1
      Selection and reproduction()
    }
  }
  Output best chromosome
}
```

Figure 5. Genetic test synthesis and scheduling algorithm.

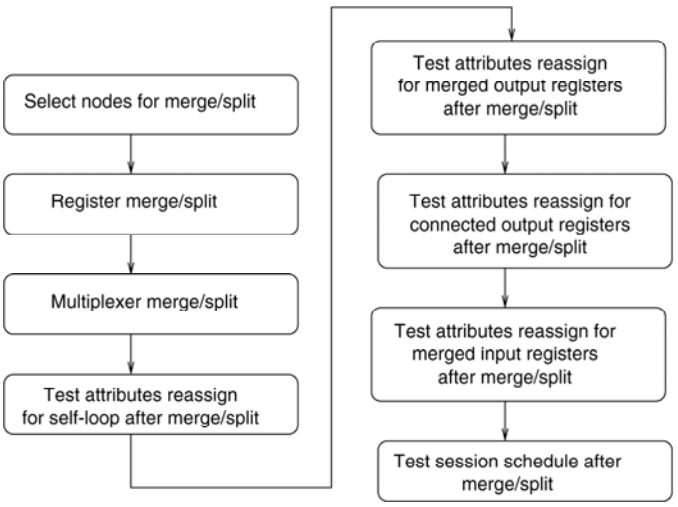


Figure 6. Kernel evolution process.

reassigns all test attributes after *merge* or *split* to resolve *self-loop* conditions, *update output and input* attributes, and reassigning test points in order to *minimize the number of test sessions*. Similar operations are applied in the

```
TestSessionSchedulingAfterMerge()
{
  For each register in the hash table
    If the register is a BILBO
      t_c ← select the next not-used test session
      if t_c is feasible
        Kernel's test session ← t_c
        Reset isConcurrent attribute in Kernel
    else
      t_c ← select test session that has minimum BILBO
      Set isConcurrent attribute for registers where
        their kernel's test session are equal to t_c
        (change all BILBO to CBILBO)
}
```

Figure 7. Test session scheduling after merging two test kernels.

```
TestAttributeReassignSelfLoop()
{
  If no self-loop then exit.
  Reset isMISR and isConcurrent test attributes for all
    merged output registers.
      If self-loop at port i then
        reset isTPGR and isConcurrent test attributes
          for all merged input registers at port i
      If self-loop at port i + 1 then
        reset isTPGR and isConcurrent test attributes
          for all merged input registers at port i + 1
        set isTPGR, isMISR and isConcurrent for this
          test kernel
}
```

Figure 8. Test attributes reassignment after merger in order in the case of self-loops.

case of the split operation. The *merge* and *split* operations work as follows:

1. *Merge:* Node merge operation selects a random parent node and a random child node from the compatibility graph. The kernels are merged by first binding the operations to a feasible functional unit. Registers are next merged and test attributes are assigned; self-loops are resolved (if needed) using CBILBOs. Merged nodes are stored in a sorted link list using a parent child relationship; children are inserted in the list based on their scheduled clock cycle. Thus, the hash table is updated by copying all input and output kernel's references at index c_i from the child node to index c_j of the parent node. Register merge includes a complete update of the tree structure for the selected kernel. The multiplexers are merged and register alignment is applied to reduce the number of multiplexers inputs. Finally, the operation applies incremental test scheduling.
2. *Split:* Node split operation is used in two possible cases. The first case occurs if a GA operation requests splitting during *mutation* or *crossover*. The second case occurs if a merge operation causes a conflict with an already existing merged node. Once a split is applied, all associated variables and inputs are split causing a split in the corresponding nodes, multiplexers, and registers. This would also trigger a reassignment for the registers test attributes. Finally, incremental test scheduling is applied.

The test session scheduling pseudo-code after merging two kernels is shown in Fig. 7 while test attributes reassignment to avoid self-loops is shown in Fig. 8. The order complexity for the above operations is shown in Table 1.

Table 1
GA Operations Order Complexity

Operation	Complexity
GA evaluation	$O(P_{size} \times N_k)$
Node merge	$O(N)$
Mux merge per port	$O(R \log_2 R)$
Register merge per port	$O(R \log_2 R)$
Node split	$O(N + R)$
Mux split	$O(1)$
Register split	$O(1)$
Test attributes reassignment	$O(R)$
Test attribute assignment	$O(R)$

4. Experimental Results

We implemented the proposed method using C++ on a Pentium 1.4 GHz PC with 128 MB of RAM running Linux. The method is very fast and all reported results were produced in at most *two* CPU minutes including scheduling time. We measured the performance of our BIST system using a suite of benchmark examples that include the *5th order elliptical wave filter*, *AR filter, the 6th order finite impulse response filter (FIR 6)*, *the 3rd order infinite impulse response filter (IIR 3)*, *the 4-point discrete cosine transform (DCT 4)*, and *the 6-tap wavelet filter*. The GA running parameters are shown in Table 2 while the DFG details for the above circuits are shown in Table 3.

Table 4 shows the implementation details by our system where for every k-test session, the system successfully generated a sub-optimal testable datapath in terms of area. The area of the circuit was computed based on the transistor area in the datapath registers and multiplexers estimated by [18, 20] and shown in Tables 5 and 6. For every example, we show the number of test sessions, functional units types, number of multiplexer inputs, and the number and type of registers. The table also shows the test overhead that the specific datapath incur due to testability considerations. The area overhead in this case ranges from 7.49% to 81.55%. The area overhead monotonically decreased with the increase of the number of test sessions which illustrates the trading-off involved in this process.

Table 2
Parameter settings

Parameter	Value
Merge/mutation	80%
Split	20%
Crossover	60%
Population size	50
Number of generations	50,000
Incremental step (*iStep*)	1,000
Max time	0–4 s

Table 3
DFG Details

Design Example	# Nodes
6th order finite impulse response filter (FIR6)	27
4-point discrete cosine transform (DCT4)	23
3rd order infinite impulse response (IIR3)	26
6-tap wavelet filter (wavelet6)	28
5th order elliptical wave filter	34
Discrete cosine transform (DCT)	48
AR filter	54

We compare the performance of our method to four other synthesis systems: ADVBIST [18], ADVAN [17], RALLOC [11], and BITS [21]. Results comparisons are shown in Table 7. Our system outperformed all other systems in the case of *FIR6* and *wavelet6*. In the other two cases, our system was outperformed by ADVBIST which is based on ILP by 1% and 13%.

Table 8 compares the performance of our system to ADVBIST [18] for $k = 1, 2, 3$, the only system that we are aware of, other than ADVAN [17], that implements concurrent test synthesis and test scheduling. ADVBIST is based on integer linear programming, which has exponential running time. Though our system was slightly outperformed in some of these cases, our system always outperforms ADVBIST in terms of time. Thus, our genetic-based synthesis technique runs in the order of seconds while ADVBIST cannot process large designs and runs in the order of hours (such as in the case of DCT4 that required up till 24 hours).

Finally, Tables 9–12 present the results for three large benchmark examples: *the AR filter*, *the 5th order elliptical wave filter*, and *the discrete cosine transfer* which were synthesized less than 18 CPU seconds. We could not compare these three cases to ADVAN or ADVBIST as

Table 5
Number of Transistors for 8-Bit Test Registers

Type	Reg	TPGR	MISR	BILBO	CBILBO
# Trs	208	256	304	388	596

Table 6
Number of Transistors for 8-Bit Multiplexers

# Mux in	2	3	4	5	6	7	Average
# Trs	80	176	208	300	320	350	—
# Trs/input	40	59	52	60	53	50	52

Table 4
Benchmark Results

Design Name	Clock Cycles	Test Sessions	ALUs	# Mux Inputs	Normal	TPGR	MISR	BILBO	CBILBO	OH (%)
FIR6	7	1	2(*), 1(+)	14	3	2	1	0	2	30.43
		2	2(*), 1(+)	14	3	2	1	2	0	14.28
		3	2(*), 1(+)	14	5	2	1	2	0	14.28
IIR3	8	1	2(*), 1(+)	21	2	1	1	2	2	48.92
		2	2(*), 1(+)	19	1	2	1	2	0	30.03
		3	2(*), 1(+)	20	1	3	2	0	0	20.32
DCT4	6	1	2(*), 1(+), 1(−)	22	1	3	1	0	3	81.55
		2	2(*), 1(+), 1(−)	23	3	1	1	3	0	52.76
		3	2(*), 1(+), 1(−)	22	3	1	0	4	0	54.13
wavelet6	11	1	1(*), 1(+), 1(−)	24	2	2	2	0	1	17.36
		2	1(*), 1(+), 1(−)	24	2	2	2	1	0	10.14
		3	1(*), 1(+), 1(−)	24	2	2	2	1	0	10.14

Table 7
Performance of Various High-level Synthesis Systems

Ckt	System	# Regs	# TPGR	# MISR	# BILBO	# CBILBO	# Mux Inputs	Area	OH
FIR6	Ours	8	2	1	2	0	14	2944	14.28
	ADVBIST	7	4	1	0	0	26	3040	18.01
	ADVAN	7	2	1	0	0	28	3308	28.42
	RALLOC	8	1	1	2	0	36	4212	63.66
	BITS	7	1	0	0	1	24	3280	27.2
IIR3	Ours	5	2	2	1	0	20	2676	20.32
	ADVBIST	6	5	1	0	0	32	2656	19.42
	ADVAN	6	3	1	0	0	32	3432	54.31
	RALLOC	7	1	0	2	0	38	4212	89.38
	BITS	6	2	0	2	0	29	3176	42.81
DCT4	Ours	8	1	1	3	0	23	3544	52.76
	ADVBIST	6	3	1	1	0	32	3236	39.48
	ADVAN	6	3	1	0	0	35	3420	47.41
	RALLOC	6	1	1	2	0	37	3812	64.31
	BITS	7	1	1	0	1	38	4180	80.17
wavelet6	Ours	7	2	2	1	0	14	3172	10.14
	ADVBIST	7	2	2	0	0	31	3248	12.78
	ADVAN	7	2	1	0	0	46	4182	31.10
	RALLOC	8	1	0	3	0	50	5186	45.21
	BITS	7	1	0	2	0	40	3946	37.01

Table 8
Comparison with ADVBIST [15] Synthesis System for $k = 1, 2, 3$

Ckt	System	# Test Sessions	# Regs	# TPGR	# MISR	# BILBO	# CBILBO	# Mux Inputs	Area	Time
FIR6	ADVBIST	1	7	3	2	0	1	29	3684	17 m 34 s
		2	7	3	1	0	1	23	3268	40 m 16 s
		3	7	4	1	0	0	26	3040	23 h 56 m 4 s
	Ours	1	8	2	1	0	2	14	3360	0.76 s
		2	8	2	1	2	0	14	2944	0.76 s
		3	8	2	1	2	0	14	2944	0.76 s
IIR3	ADVBIST	1	6	3	3	0	0	27	2912	3 h 11 m 8 s
		2	6	4	2	0	0	24	2688	2 h 6 m 26 s
		3	6	5	1	0	0	23	2656	2 h 50 m 8 s
	Ours	1	6	1	1	0	2	22	3312	0.52
		2	6	2	1	2	0	21	2892	0.57
		3	6	3	2	0	0	21	2676	0.52
DCT4	ADVBIST	1	6	2	4	0	0	27	3024	24 h
		2	6	3	2	0	0	34	3088	24 h
		3	6	2	2	0	0	59	4256	24 h
	Ours	1	8	3	1	0	3	22	4212	1 s
		2	8	1	1	3	0	23	3544	1 s
		3	8	1	0	4	0	22	3576	1 s
wavelet6	ADVBIST	1	7	2	3	0	0	31	3344	17 m 34 s
		2	7	2	2	0	0	31	3248	40 m 16 s
		3	7	2	2	0	0	31	3248	23 h 56 m 4 s
	Ours	1	7	2	2	0	1	24	3380	0.81 s
		2	7	2	2	1	0	24	3172	0.81 s
		3	7	2	2	1	0	24	3172	0.81 s

Table 9
AR Filter Results (Max Time 1 s)

Design Characteristics	Clock Cycles	Test Sessions	ALUs	# Mux Inputs	TPGR	MISR	BILBO	CBILBO	Normal	OH (%)
Non-pipelined multicycled	11	1	4(*), 2(+)	48	5	4	0	2	1	11.96
		2	4(*), 2(+)	47	5	3	2	1	1	10.53
	15	1	3(*), 2(+)	44	2	2	0	3	3	15.75
		2	3(*), 2(+)	48	2	2	3	0	3	8.53
	18	1	2(*), 2(+)	37	2	3	0	1	3	10.97
		2	2(*), 2(+)	37	3	2	2	0	3	8.65
	34	1	1(*), 1(+)	33	1	1	0	1	4	11.21
		2	1(*), 1(+)	35	2	1	1	0	4	6.32
Pipelined multicycled	11	1	4(*), 2(+)	50	5	4	0	2	5	11.26
		2	4(*), 2(+)	50	5	3	1	2	5	11.78
	13	1	2(*), 2(+)	44	3	2	0	2	3	14.14
		2	2(*), 2(+)	45	3	2	2	0	3	8.15
	16	1	2(*), 2(+)	43	2	1	0	2	5	12.72
		2	2(*), 1(+)	42	3	0	3	0	6	7.70
	20	1	1(*), 1(+)	37	1	1	0	1	6	10.20
		2	1(*), 1(+)	37	1	0	2	0	6	7.04

Table 10
Elliptic Wave Filter Results (Max Time 1 s)

Design Characteristics	Clock Cycles	Test Sessions	ALUs	# Mux Inputs	TPGR	MISR	BILBO	CBILBO	Normal	OH (%)
Non-pipelined multicycled	17	1	2(*), 3(+)	28	2	4	0	1	5	12.10
		2	2(*), 3(+)	36	1	3	2	0	4	9.53
	18	1	2(*), 3(+)	35	1	3	0	2	3	15.56
		2	2(*), 3(+)	41	1	4	0	1	2	11.68
	19	1	2(*), 2(+)	32	1	2	0	2	3	15.40
		2	2(*), 2(+)	33	1	2	2	0	3	8.97
	20	1	2(*), 2(+)	32	1	2	0	0	3	15.40
		2	2(*), 2(+)	33	1	2	2	2	3	8.97
	21	1	1(*), 2(+)	30	1	1	0	2	3	18.19
		2	1(*), 2(+)	29	1	1	2	0	3	9.95
Pipelined Multicycled	17	1	2(*), 3(+)	34	1	3	0	2	4	15.27
		2	2(*), 3(+)	34	1	3	2	0	4	9.64
	18	1	1(*), 3(+)	36	1	1	0	3	3	21.79
		2	1(*), 3(+)	30	1	2	1	1	4	14.57
	19	1	1(*), 2(+)	30	1	1	0	2	4	17.74
		2	1(*), 2(+)	34	2	1	2	0	4	9.24
	20	1	1(*), 2(+)	30	1	1	0	2	4	17.74
		2	1(*), 2(+)	34	2	1	2	0	4	9.24
	28	1	1(*), 1(+)	27	1	1	0	1	9	9.99
		2	1(*), 2(+)	33	1	2	1	0	6	7.49

these results were not reported. Fig. 9 shows the speed of convergence for our algorithm and shows at every generation the best, average, and worst fitness of chromosomes in a population of 300,000 for the elliptic wave filter.

5. Conclusion

A datapath allocation problem was presented based on an evolutionary algorithm. The proposed system can

Table 11
DCT Results (Max Time 15 s)

Design Characteristics	Clock Cycles	Test Sessions	ALUs	# Mux Inputs	TPGR	MISR	BILBO	CBILBO	Normal	OH (%)
Single cycle FUs	7	1	5(*), 5(+), 4(−)	64	1	6	0	8	8	20.91
		2	5(*), 5(+), 4(−)	67	1	5	6	3	8	15.78
	10	1	2(*), 3(+), 2(−)	66	1	3	0	4	10	17.22
		2	2(*), 3(+), 2(−)	68	1	3	3	1	10	11.63
	13	1	2(*), 2(+), 1(−)	62	1	3	0	2	12	11.57
		2	2(*), 2(+), 1(−)	67	1	3	2	0	11	6.93
	18	1	1(*), 2(+), 1(−)	58	1	2	0	2	11	12.99
		2	1(*), 2(+), 1(−)	57	1	2	2	0	11	7.55
	25	1	1(*), 2(+), 1(−)	54	2	3	0	1	15	9.07
		2	1(*), 2(+), 1(−)	53	1	1	2	1	17	11.41
Non-pipelined multicycled	14	1	3(*), 3(+), 1(−)	65	1	4	0	3	13	13.23
		2	3(*), 3(+), 1(−)	64	1	2	3	2	13	12.60
	18	1	2(*), 3(+), 1(−)	64	1	3	0	3	12	14.43
		2	2(*), 3(+), 1(−)	66	1	3	3	0	12	8.33
	34	1	1(*), 2(+), 1(−)	56	1	2	0	2	16	13.02
		2	1(*), 2(+), 1(−)	56	1	3	1	0	18	6.47
Pipelined multicycled	12	1	2(*), 3(+), 1(−)	63	1	4	0	2	11	12.17
		2	2(*), 3(+), 1(−)	63	1	3	3	0	11	8.53
	19	1	1(*), 2(+), 1(−)	61	1	2	0	2	16	11.67
		2	1(*), 2(+), 1(−)	61	1	2	2	0	16	6.71

Table 12
DCT Results Where Adders and Subtractors are Compatible (Max Time 15 s)

Design Details	Clock Cycles	Test Sessions	ALUs	# Mux Inputs	Regs	TPGR	MISR	BILBO	CBILBO	Normal	OH (%)
Single cycled FUs	7	1	5(*), 8(+−)	74	22	1	7	0	6	8	17.56
		2	5(*), 8(+−)	76	21	1	5	5	3	7	14.92
	10	1	2(*), 5(+−)	71	17	1	4	0	3	9	14.86
		2	2(*), 5(+−)	71	17	1	4	2	1	9	11.11
	13	1	2(*), 3(+−)	61	17	2	3	0	2	11	11.94
		2	2(*), 3(+−)	61	17	1	2	2	1	11	10.31
	18	1	1(*), 3(+−)	52	15	1	2	0	2	10	13.72
		2	1(*), 3(+−)	52	14	1	1	2	1	9	11.66
	25	1	1(*), 3(+−)	52	15	1	2	0	2	10	13.72
		2	1(*), 3(+−)	52	14	1	1	2	1	9	11.66
Non-pipelined multicycled	14	1	3(*), 4(+−)	68	19	1	3	0	4	11	15.50
		2	3(*), 4(+−)	69	20	1	4	2	1	12	9.81
	18	1	2(*), 4(+−)	60	18	1	3	0	3	11	14.90
		2	2(*), 4(+−)	68	18	1	3	3	0	11	8.34
	34	1	1(*), 2(+−)	54	19	1	1	0	2	15	11.70
		2	1(*), 2(+−)	60	19	1	1	2	0	15	5.75
Pipelined multicycled	12	1	2(*), 4(+−)	67	18	1	4	0	2	11	12.05
		2	2(*), 4(+−)	60	19	1	3	3	0	12	8.52
	19	1	1(*), 3(+−)	62	18	1	2	0	2	13	12.22
		2	1(*), 3(+−)	59	19	1	2	2	0	14	7.17

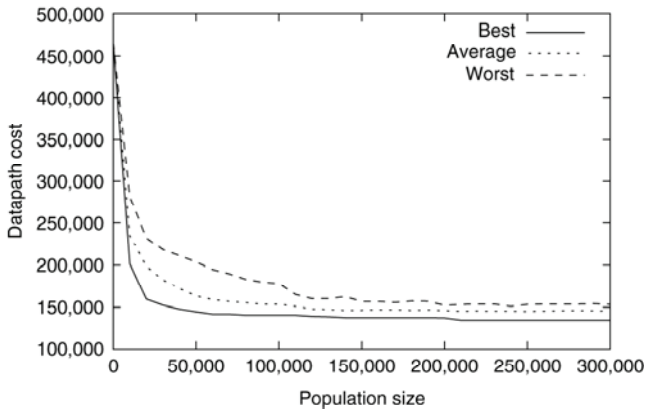


Figure 9. Best, average, and worst fitness of chromosomes in a population of 300,000 for the elliptic wave filter.

handle pipelined and multicycled operations and creates an RTL VHDL description for a least cost self-testable datapath. The system provides the best solution in terms of the number and types of functional units, the number of registers, and the number of multiplexer inputs for each k-session. The method was implemented on a Linux PC using C++ and several benchmarks were attempted. Future work includes the incorporation of power constraints during the synthesis process.

References

[1] G. De Micheli, *Synthesis and optimization of digital circuits* (New York: McGraw Hill, 1994).

[2] M.L. Bushnell & V.D. Agrawal, *Essentials of electronic testing for digital, memory, and mixed signal VLSI circuits* (Boston: Kluwer Academic Publishers, 2000).

[3] C.E. Stroud, *A designer's guide to built-in self-test* (Boston: Kluwer Academic Publishers, 2002).

[4] B. Koenemann, J. Mucha, & G. Zwiehoff, Built-in logic block observation techniques, *Proc. of the Int. Test Conf.*, Cherry Hill, NJ, October 1979, 37–41.

[5] I. Parulkar, S. Gupta, & M. Breuer, Scheduling and module assignment for reducing BIST resources, *Proc. DATE 98*.

[6] A.E. Eiben & J.E. Smith, *Introduction to evolutionary computing* (Berlin: Springer-Verlag, 2003).

[7] K. Wagner & S. Dey, High-level synthesis for testability: A survey and perspective, Las Vegas, Nevada, *Proc. DAC*, 1996, 131–136.

[8] C. Papachristou, S. Chiu, & H. Harmanani, A data path synthesis method for self-testable designs, *Proc. of the 28th Design Automation Conf.*, San Francisco, CA, June 1991, 378–384.

[9] C. Papachristou, S. Chiu, & H. Harmanani, SYNTEST: A method for high-level synthesis with self-testability, *Proc. Int. Conf. on Computer Design*, Cambridge, MA, 1991, 458–462.

[10] H. Harmanani & C. Papachristou, An improved method for RTL synthesis with testability trade-offs, *Proc. ICCAD*, Santa Clara, CA, 1993.

[11] L. Avra, Allocation and assignment in high-level synthesis for self-testable data paths, *Proc. ITC*, Nashville, TN, 1991.

[12] I. Parulkar, S. Gupta, & M. Breuer, Allocation techniques for reducing BIST overhead of datapaths, *Journal of Electronic Testing & Theory Application*, 13, 1998, 149–166.

[13] K. Olcoz, F. Tirado, & H. Mecha, Unified data path allocation and BIST intrusion, *Integration, the VLSI Journal*, *28*, 1999, 55–99.

[14] M. Zwolinski & M. Gaur, Integrating testability with design space exploaration, *Microelectronics Reliability*, *43*, 2003, 685–693.

[15] G. Craig, C. Kime, & K. Saluja, Test scheduling and control for VLSI built-in self-test, *IEEE Trans. on Computers*, *C-37*, 1988, 1099–1109.

[16] I. Harris & A. Orailoglu, SYNCBIST: Synthesis for concurrent built-in self-testability, *Proc. EDTC*, Cambridge, MA, 1994, 101–104.

[17] H. Kim, T. Takahashi, & D. Ha, Test session oriented built-in self-testable data path synthesis, *Proc. Int. Test Conf.*, Washington, DC, October 1998, 154–163.

[18] H. Kim, D. Ha, T. Takahashi, & T. Yamaguchi, A new approach to built-in self-testable datapath synthesis based on ILP, *IEEE Trans. on VLSI*, *8*, 2000, 594–605.

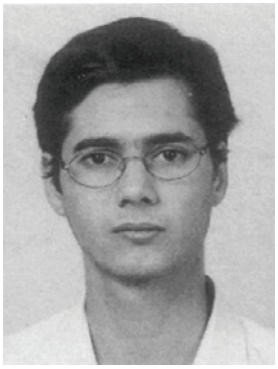
[19] M. Dhodhi, F. Hielscher, R. Storer, & J. Bhasker, Datapath synthesis using a problem-space genetic algorithm, *IEEE Trans. on CAD*, *14*, 1995, 934–944.

[20] L.T. Wang & E.J. McCluskey, Concurrent built-in logic block observer, *Proc. ISCAS'86*, San Jose, CA, 1986, 1054–1057.

[21] P. Bukovjan, L. Ducerf-Bourbon, & M. Marzouki, Cost/quality trade-off in synthesis for BIST, *JETTA*, *17*, 2001, 109–119.

Biographies

Haidar Harmanani received his B.S., M.S., and Ph.D. all in Computer Engineering from Case Western Reserve University, Cleveland, Ohio, in 1989, 1991, and 1994, respectively. He joined the Lebanese American University (LAU), Lebanon, in 1994 as an Assistant Professor of Computer Science. Currently, he is an Associate Professor of Computer Science and the Chair of the Computer Science and Mathematics Division at LAU, Byblos Campus. Prof. Harmanani has been on the program committee of various International Conferences including the IEEE NEWCAS Conference (NEWCAS 2006, 2007), the IEEE International Conference on Electronics, Circuits, and Systems, (ICECS 2000, 2006, and 2007), and the 14th IEEE International Conference on Microelectronics, 2002. His research interests include electronic design automation, high-level synthesis, SOC Testing, design for testability, and cluster parallel programming. He is a senior member of IEEE and ACM.



Aouni Kamal Hajar received his Bachelor of Engineering (BE) in Computer Engineering and his MS in Computer Science from the Lebanese American University, Lebanon, in 1999 and 2002, respectively. Mr. Hajar has worked with various regional companies as a software engineer. Currently, he is a software engineer with Netways Corporation, Lebanon.