A Systematic Approach for Designing Testable VLSI Circuits

Sen-Pin Lin, Charles A. Njinda and
Melvin A. Breuer

CEng Technical Report 91-18

Electrical Engineering Systems

University of Southern California

Los Angeles, CA. 90089-2562

July 15, 1991

A Systematic Approach for Designing Testable VLSI Circuits*

Sen-Pin Lin , Charles A. Njinda and Melvin A. Breuer

Department of Electrical Engineering - systems
University of Southern California
Los Angeles, CA 90089-0781

Email: splin@poisson.usc.edu
Phone: (213) 740-4460

* This work was supported by the Defense Advanced Research Projects Agency and monitored by the Federal Bureau of Investigation and conclusions considered in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

A Systematic Approach for Designing Testable VLSI Circuits

Abstract

There are many ways to make a digital circuit testable. Each way can have different effects on the design in terms of area overhead and test time. Most current systems developed for making a design testable either minimize area overhead and sacrifice test time, or vice versa. A better approach should achieve both a tolerable area overhead and an acceptable test time. A designer should have the flexibility to make the necessary trade-off between area overhead and test time depending on the characteristics of the design under consideration. A systematic approach has been developed to provide the designer with a set of testable versions for a given design, ranging from the minimal test time solution to the minimal area overhead solution. By employing an expert selection system developed previously, the system can be extended to operate as an intelligent BIST design advisor. Experiments have been performed on several circuits generated by MABAL, a CAD tool, to demonstrate the performance of this approach.

Index terms: Built-In-Self-Test, synthesis for testability, test scheduling.

1 Introduction

The density of circuits and the limited I/O inherent in VLSI chips make the testing problem for physical faults very complicated and time consumming. Numerous design for testability (DFT) techniques have been proposed to simplify this problem. Among them Built-In Self-Test (BIST) techniques are often used. A common feature of most BIST techniques is the addition of extra test hardware. Due to this extra hardware, the testable version of a circuit usually needs more layout area than the original one. In the test mode, sufficient test patterns have to be generated on board in order to achieve satisfactory fault coverage. Therefore, *area overhead* and *test time* are the two most significant overheads encountered when using BIST techniques. Other overheads include performance degradation, design complexity, and extra I/O pins. How to reduce the BIST overheads without sacrificing fault coverage becomes an important issue. Many researchers have addressed this problem in recent years[1][5][6]. Unfortunately, most of the systems developed make a design testable by minimizing either test time or area overhead. From a designer's point of view, a good design should achieve both a tolerable area overhead and an acceptable test time. He/she should have the flexibility to make trade-offs between area overhead and test time depending on the design constraints.

Abadir and Breuer[1] proposed a Testable Design Expert System (TDES) to help a designer modify the original circuit and make it testable. A typical circuit contains combinational blocks, buses and registers. For each combinational block, all possible embeddings of a particular testable design methodology (TDM) are constructed and evaluated by a scoring system. A user can then choose one embedding based on the design constraints. The whole circuit is processed on this block by block basis. The sequential nature of selecting embeddings does not have a global view of the solution space, hence its performance greatly depends on the order in which the combinational blocks are processed. Due to the lack of global analysis, it is not easy to find an optimal solution for the design. Compared to their work, we use the same philosophy of generating a set of solutions

instead of one solution, but we take a global point of view to achieve better performance. Craig *et al.*[6] proposed an optimal and a suboptimal procedures for scheduling the tests of a testable design in order to minimize test time. A clique covering and a chromatic coloring algorithms are used. They made the restriction that resource allocation for each combinational block is done before the test scheduling, hence the optimality can only be applied to the specific resource allocation. In a non-trivial circuit, there are usually many ways of using test resources to make a combinational block testable, hence their solution may not necessarily be the global optimum. In the optimum version of their procedures the solution may be redundant, namely a combinational block is tested more than once, which makes further reduction in test time possible and violates the conclusion of optimality. Bhawmik *et al.*[5] used integer linear programming technique to minimize area overhead of a BIST design. Our study of integer linear programming shows little success in this problem, and a branch and bound algorithm is suggested instead.

In Section 2 the notations and graph model used to represent circuits and the preprocessing of circuits are presented. Section 3 explores the solution space of the testable design problem. Algorithms for generating a family of solutions are presented. Experimental results are obtained and listed in Section 4. In Section 5 we conclude with a brief summary and issues for future investigation.

2 Preliminaries

Given a circuit consists of combinational blocks (called *kernels*), registers, buses and MUXes. A *K-port* is an input/output port of a kernel. An I-path[1] is a data path from a primary input or a register to a K-port or from a K-port to a primary output or a register so that data can be transfered unaltered. In this report, an I-path is represented as a list $(c_1, c_2, \ldots, c_n)$ where each c_i, $1 \leq i \leq n$, is a component or a port of a component in the circuit, and data can be transfered from c_{i-1} to c_i without changes. The first and the last elements in an I-path are called the *head* and *tail*, respectively, of the I-path. The I-paths in the example circuit of Figure 1 are enumerated

below.

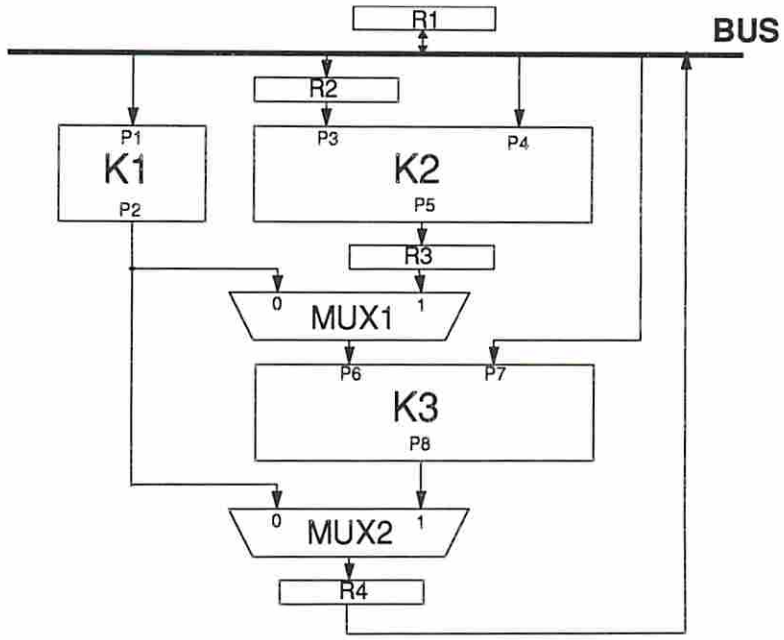

Figure 1: An example circuit

$I_1 : (R_1, BUS, K_1(P_1))$

$I_2 : (R_4, BUS, K_1(P_1))$

$I_3 : (K_1(P_2), MUX_2(0), R_4)$

$I_4 : (K_1(P_2), MUX_2(0), R_4, BUS, R_2)$

$I_5 : (K_1(P_2), MUX_2(0), R_4, BUS, R_1)$

$I_6 : (R_2, K_2(P_3))$

$I_7 : (R_1, BUS, R_2, K_2(P_3))$

$I_8 : (R_4, BUS, R_2, K_2(P_3))$

$I_9 : (R_1, BUS, K_2(P_4))$

$I_{10} : (R_4, BUS, K_2(P_4))$

$I_{11} : (K_2(P_5), R_3)$

$I_{12} : (R_3, MUX_1(1), K_3(P_6))$

$I_{13} : (R_1, BUS, K_3(P_7))$

$I_{14} : (R_4, BUS, K_3(P_7))$

$I_{15} : (K_3(P_8), MUX_2(1), R_4)$

$I_{16} : (K_3(P_8), MUX_2(1), R_4, BUS, R_2)$

$I_{17} : (K_3(P_8), MUX_2(1), R_4, BUS, R_1)$

A circuit is modeled as a directed bipartite graph (N_A, N_B, E). N_A is a set of type A nodes, each of which represents a register; N_B is a set of type B nodes, each of which represents a K-port. E is a set of edges, each of which represents an I-path between a register and a K-port. Figure 2 shows the corresponding graph model for the circuit in Figure 1.

Definition 1 A register is a *driver* of a K-port P_i if there exists an I-path from it to P_i; a register is a *receiver* of a K-port P_j if there exists an I-path from P_j to it.

It is possible that a register can be a driver of some K-port and a receiver of another K-port. In this report we consider only the BILBO TDM[2]. In this methodology a register is not allowed to be used as both a driver and a receiver in the same test session[1]. This restriction does not exist for other TDMs like CBILBO or CSTP[8]. A *test session* is defined as the execution of a set of tests for some kernels, where every test of a kernel must start before the end of tests of other kernels.

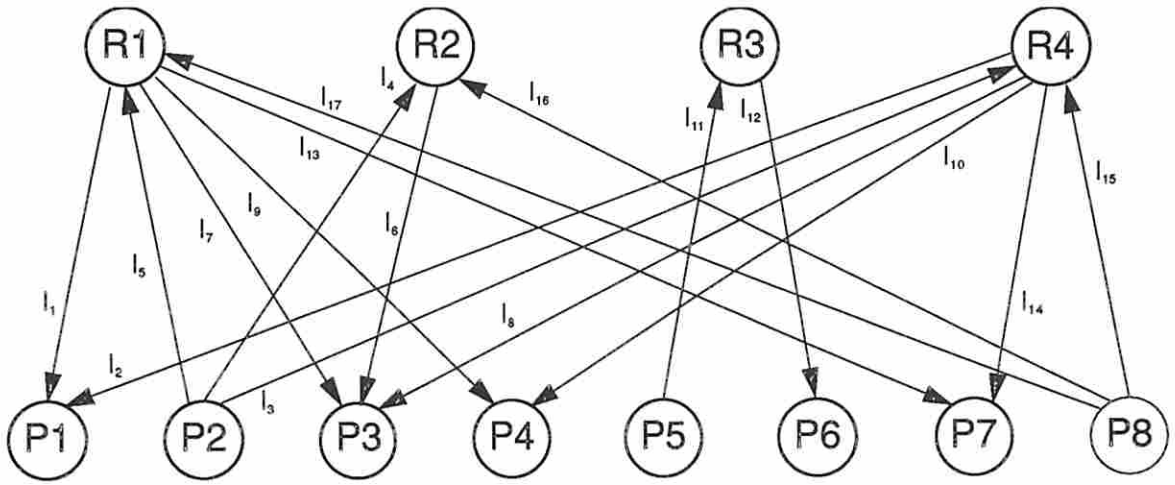

Figure 2: Graph model of the circuit

Definition 2 An I-path is a *driving path* if it starts from a primary input or a driver and ends at an input K-port; an I-path is a *receiving path* if it starts from an output K-port and ends at a receiver or a primary output.

Driving or receiving paths are said to *cover* their associated K-ports. Given the set of K-ports $\{P_1, P_2, \ldots, P_M\}$, and the set of I-paths $\{I_1, I_2, \ldots, I_L\}$ of the circuit under consideration, the

I-path covering matrix $[\mathrm{C}]_{M\times L}$ is defined as follows.

$$\mathrm{C(i,j)} = \begin{cases} 1 & \text{if } I_j \text{ covers } P_i \\ 0 & \text{otherwise} \end{cases}$$

It is obvious that each column in the matrix has exactly one 1. A K-port is said to be testable if it is covered by at least one I-path. To make a K-port testable using some I-path that covers it, it may be necessary to add extra functions like Random Pattern Generation(RPG), Signature Analysis(SA), HOLD, or LOAD to some of the registers in the I-path if these functions are absent. If there is no I-path that can cover a K-port, extra registers or MUXes may be needed. To shift in the initial seed data and shift out the signature, the SHIFT function needs to be added to all drivers and receivers when necessary. By adding an extra function to a register, the area required for its implementation increases. Suppose there are P registers $\{R_1, R_2, \ldots, R_P\}$, a modification matrix $[\mathbf{M}]_{P\times L}$ can be defined where each element of this matrix is a bit vector composed of five functional bits, [R, S, H, L, T]. In other words, $\mathbf{M}(i,k)$ represents the necessary modifications on R_k if I_i is used. The corresponding bits of $\mathbf{M}(i,k)$ are

$$\begin{cases} \mathrm{R} = 1 \text{ if the RPG function is added to } R_k, \text{ 0 otherwise.} \\ \mathrm{S} = 1 \text{ if the SA function is added to } R_k, \text{ 0 otherwise.} \\ \mathrm{H} = 1 \text{ if the HOLD function is added to } R_k, \text{ 0 otherwise.} \\ \mathrm{L} = 1 \text{ if the LOAD function is added to } R_k, \text{ 0 otherwise.} \\ \mathrm{T} = 1 \text{ if the SHIFT function is added to } R_k, \text{ 0 otherwise.} \end{cases}$$

Definition 3 Two I-paths are said to have a *Forbidden Conflict (FC)* if they need not co-exist in a testable design; two I-paths are said to have a *Hard Conflict (HC)* if they cannot be operated in the same test session; two I-paths are said to have a *Soft Conflict (SC)* if they can be operated in the same test session but restrictions on their schedules exist.

The following rules are used to identify all possible configurations that may cause I-path conflicts. We consider six cases of I-path configurations as described below, and give a corresponding

example using Figure 1.

Case 1 If two I-paths cover the same K-port, then they have a FC because it is not necessary to use multiple I-paths to test the same K-port. I_1 and I_2 fall in this category.

Case 2 Consider two driving I-paths with the same head and their tails belong to different ports of the same kernel.

(a) If they contain the same set of registers, then there exists a FC since the fault coverage of this configuration will be degraded significantly due to the strong correlation between test data to the two input ports. In the example circuit, if R_2 is absent then I_7 and I_9 are in this category.

(b) If they contain different sets of registers, then there exists a SC since we can hold data in any register not common to both I-paths, and properly schedule them. I_7 and I_9 have a SC due to this rule.

Case 3 Consider two receiving I-paths.

(a) If they have the same tail, then there exists a SC because two output patterns cannot be loaded into a register at the same time. The signature will be a compressed value of two sets of output patterns. For example, I_3 and I_{15}.

(b) If one I-path contains the tail (receiver) of the other, then there exists a HC since if they are operated in the same test session, the signature in the receiver mentioned above will be destroyed by the other I-path. I_5 and I_{15} fall in this category.

Case 4 Consider two I-paths where one is a driving path, the other is a receiving path, and they cover K-ports of the same kernel.

(a) If the head of the driving path and the tail of the receiving path are the same, then there exists a *FC* since they have to be operated in the same test session and the test pattern generation and response compression processes will interfere with each other. I_1 and I_5 fall in this category. If different TDMs are allowed (for example CBILBO, CSTP) then this rule can be relaxed.

(b) If the driving path contains the tail of the receiving path, or the receiving path contains the head of the driving path, then there exists a *FC* due to the same reason as above. I_2 and I_5 belong to this category.

Case 5 Consider two I-paths where one is a driving path, the other is a receiving path, and they cover K-ports of different kernels.

(a) If the head of the driving path and the tail of the receiving path are the same, then there exists a *HC* since the test pattern generation and response compression processes will interfere with each other if they are operated in the same test session. I_1 and I_{17} fall in this category.

(b) If the driving path contains the tail of the receiving path, or the receiving path contains the head of the driving path, then there exists a *HC* due to the same reason as above. I_5 and I_{14} belong to this category.

Case 6 If two I-paths have a MUX, bus, or register in common and none of the above configuration holds, then there exist a *SC* unless both are driving I-paths with the same head and their tails belong to different kernels. In the latter case, test

patterns are broadcasted to the K-ports, hence no conflicts exist. For example, I_1 and I_{10} have a SC but I_1 and I_9 do not conflict.

If the configuration of any two I-paths does not fall in any of the above categories, then no conflict exists between them.

Given the set of I-paths, an I-path conflict matrix $[\mathbf{IC}]_{L\times L}$ can be constructed by examining every pair of I-paths. The elements of **IC** are,

$$\mathbf{IC}(\mathrm{i1},\mathrm{i2}) = \begin{cases} F & \text{if } I_{i1} \text{ and } I_{i2} \text{ have a FC} \\ H & \text{if } I_{i1} \text{ and } I_{i2} \text{ have a HC} \\ S & \text{if } I_{i1} \text{ and } I_{i2} \text{ have a SC} \\ - & \text{otherwise} \end{cases}$$

A kernel is said to be *testable* if each of its K-ports is testable and no conflicts exist between the driving/receiving paths of the K-ports. A circuit is said to be *testable* if every kernel is testable.

Example 1 Consider the circuit in Figure 1, there are 17 I-paths and 8 K-ports. The I-path covering matrix and I-path conflict matrix are generated by our program as shown below.

$$\mathbb{C} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

$$\mathbf{IC} = \begin{bmatrix}
F & F & - & S & F & - & - & S & - & S & - & - & - & S & - & S & H \\
 & F & F & F & F & - & S & - & S & - & - & - & S & - & H & H & H \\
 & & F & F & F & - & - & H & - & H & - & - & - & H & S & H & H \\
 & & & F & F & H & H & H & S & H & - & - & S & H & H & S & S \\
 & & & & F & - & H & H & H & H & - & - & H & H & H & S & S \\
 & & & & & F & F & F & - & - & - & - & - & - & - & H & - \\
 & & & & & & F & F & S & S & - & - & - & S & - & H & H \\
 & & & & & & & F & S & S & - & - & S & - & H & H & H \\
 & & & & & & & & F & F & - & - & - & S & - & S & H \\
 & & & & & & & & & F & - & - & S & - & H & H & H \\
 & & & & & & & & & & F & H & - & - & - & - & - \\
 & & & & & & & & & & & F & - & - & - & - & - \\
 & & & & & & & & & & & & F & F & - & S & F \\
 & & & & & & & & & & & & & F & F & F & F \\
 & & & & & & & & & & & & & & F & F & F \\
 & & & & & & & & & & & & & & & F & F \\
 & & & & & & & & & & & & & & & & F
\end{bmatrix}$$

The matrix **IC** is symmetric, hence only the upper triangular portion is shown. □

3 Exploring the solution space

For a non-trivial circuit, there are many different ways to make it testable. Each testable design has a certain *area overhead* and *test time*. If area overhead and test time are the only attributes we are concerned with , then the solution space can be represented as shown in Figure 3. The point where the test time is shortest is called *the minimal test time solution (s_{mtt})*; the point where the area overhead is smallest is called *the minimal area overhead solution (s_{mao})*. Between these two points are a set of intermediate solutions with test time greater than or equal to that of s_{mtt} and area overhead greater than or equal to that of s_{mao}. This family of solutions represent the choices for making the circuit testable. It should be noted that there may be solutions where neither the area overhead nor test time are better than those of s_{mao} or s_{mtt}, for example s_1 and s_2 in Figure 3. Such solutions are not considered as members of the solution space.

The solutions s_{mtt} and s_{mao} can be obtained by using different branch and bound procedures

described in Sections 3.1 and 3.2. If the whole family of solutions is desired, a more elaborated searching scheme must be employed as descibed in Section 3.3.

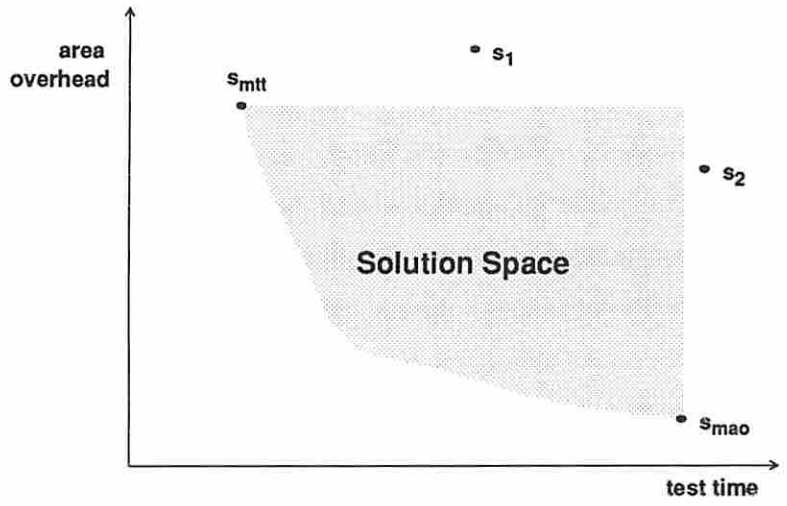

Figure 3: A plot of the solution space

3.1 The minimal test time solution

Definition 4 A *legitimate BILBO embedding* (*embedding* for short) is a structure consisting of a kernel, a driving path for each input K-port and a receiving path for each output K-port of the kernel. None of these I-paths should have FCs or HCs with each other.

In the circuit shown in Figure 1, $e = (K_2, I_7, I_9, I_{11})$ is a legitimate embedding. K_2 is the kernel of e. I_7 and I_9 are the driving paths for P_3 and P_4, respectively. I_{11} is the receiving path for P_5. To make a kernel testable, it may be necessary to add extra functions to some registers. The modification information can be obtained from the matrix $\mathbf{M}$, and a set of (register, modification vector) pairs, called *rm-pairs*, is associated with each embedding to denote the modifications required. A register R_k will appear in the set of rm-pairs if it is contained in an I-path I_i of the embedding and $M(i,k)$ is not a 0 bit vector, and the corresponding modification vector is $M(i,k)$.

Example 2 Suppose all registers in Figure 1 have the LOAD function only. To make K_2 testable using embedding e, R_1 must have RPG function while R_2 must be able to HOLD data so that P_3 and P_4 can get different test patterns. SA function has to be added to R_3. R_1 and R_3 also need the SHIFT function so that initial

seed data can be shifted in and signature can be shifted out. The rm-pairs associated with e are

$$\{(R_1, [1,0,0,0,1]), (R_2, [0,0,1,0,0]), (R_3, [0,1,0,0,1])\}$$ □

Each kernel may be contained in more than one embedding. The set of all possible embeddings in a given circuit can be obtained using the procedure shown in below.

Procedure *find_all_embeddings()*
begin
  $\mathcal{E} \leftarrow \emptyset$;
  for $i = 1$ **to** N **do**
  begin
    Let $\mathcal{P}_i$ *be the number of K-ports of* K_i;
    Let $\mathcal{I}_j$ *be the set of I-paths that can cover the* j^{th} *port of* K_i;
    $\mathcal{E}' \leftarrow \mathcal{I}_1 \times \mathcal{I}_2 \times \ldots \times \mathcal{I}_{\mathcal{P}_i}$;
    $\forall\ e \in \mathcal{E}'$, **if** *e does not contain I-paths with HCs or FCs*
      then $\mathcal{E} \leftarrow \mathcal{E} \bigcup \{(K_i) \parallel e\}$;
  end
  $\mathcal{E}$ *is the set of all legitimate embeddings;*
end

The operator '$\parallel$' denotes list concatenation. Given an embedding, a test plan for the kernel in this embedding can be generated. To achieve minimal test time, *pipeline* structures and *minimal latency* should be used[4]. Minimal latency is defined as the minimal delay between two consecutive initiations of test vectors, so that no resource conflicts can happen. The minimal latency can always be achieved by using NO-OPs[1][4] in the test plan. If the I-paths of an embedding do not have SCs, the minimal latency is always 1. That is, we can apply a new test vector every clock period. If the kernel in the embedding needs T test vectors, then the test time is approximately T clock periods if T is large, i.e. we ignore the time to shift in the initial and final states of the various shift registers.

Throughout this report we will assume each kernel requires T test vectors. If two I-paths have SCs, then they must share some bus, MUX, or register. We define a *resource* to be a bus, MUX, or register. Suppose there are m resources $\{r_1, r_2, \ldots, r_m\}$ in the circuit. Given an embedding e_j, the reservation table[4] of these m resources for e_j can be generated so that the cell in row i and column j contains an 'X' if r_i is used in the k^{th} step of the test plan for e_j. The minimal latency will be the maximal number of 'X's in any row of the reservation table. Formally, let $X_{i,j}$ denote the number of 'X's in the i_{th} row of the reservation table for e_j, then the minimal latency for e_j is $\max_{i=1}^{m} X_{i,j}$. The test time for e_j will be approximately $T \cdot \max_{i=1}^{m} X_{i,j}$.

Consider two embeddings e_{j1} and e_{j2} where the test time for e_{j1} is $t_{j1} = T \cdot \max_{i=1}^{m} X_{i,j1}$ and test time for e_{j2} is $t_{j2} = T \cdot \max_{i=1}^{m} X_{i,j2}$. e_{j1} and e_{j2} are *fully compatible* if I-paths of e_{j1} do not have conflicts with I-paths of e_{j2}. They can be executed in the same test session with a test time of $max(t_{j1}, t_{j2})$. This is obviously less than the case where the kernels are tested one after the other.

Assume I-paths of e_{j1} have SCs with I-paths of e_{j2}. If e_{j1} and e_{j2} are executed in one test session, then the test time of this joint test session is $t_{j1,j2} = T \cdot \max_{i=1}^{m}(X_{i,j1} + X_{i,j2})$. It is obvious

that $t_{j1,j2} \leq t_{j1} + t_{j2}$. In the case that $t_{j1,j2} < t_{j1} + t_{j2}$, it is beneficial to operate e_{j1} and e_{j2} in one test session. In this situation e_{j1} and e_{j2} are said to be *partially compatible*. If $t_{j1,j2} = t_{j1} + t_{j2}$, there is no benefit in operating e_{j1} and e_{j2} in the same test session since the test time is not reduced, but the control logic is more complicated. One can also verify that if e_{j1} and e_{j2} are fully compatible, the above inequality also holds.

Example 3 Consider the circuit in Figure 1. Let $e_1 = (K_1, I_1, I_3)$ and $e_2 = (K_3, I_{12}, I_{13}, I_{15})$ be two embeddings. Possible test plans for e_1 and e_2 are

Test plan for e_1	**Test plan for e_2**
$\Phi 1 : R_1(RPG)$	$\Phi 1 : R_1(RPG), R_3(RPG)$
$\Phi 2 : BUS(R_1), K_1, MUX_2(K_1), R_4(SA)$	$\Phi 2 : MUX_1(R_3), BUS(R_1), K_3, MUX_2(K_3), R_4(SA)$

It should be noted that these test plans are used for illustration purpose only as it is not necessary to generate them at this stage. The resources employed by these embeddings can be summarized by the reservation tables shown below.

embedding e_1

	$\Phi 1$	$\Phi 2$
R_1	X	
R_4		X
BUS		X
MUX_2		X

embedding e_2

	$\Phi 1$	$\Phi 2$
R_1	X	
R_3	X	
BUS		X
MUX_1		X
MUX_2		X
R_4		X

From the reservation tables, the test time to execute e_1 and e_2 one after the other is $T + T = 2T$. One test plan for executing these two embeddings in one test session is shown below.

$\Phi 1 : R_1(RPG)$

$\Phi 2 : BUS(R_1), K_1, MUX_2(K_1), R_4(SA), R_1(HOLD), R_3(RPG)$

$\Phi 3 : BUS(R_1), MUX_1(R_3), K_3, MUX_2(K_3), R_4(SA)$

The joint reservation table is as follows.

	$\Phi 1$	$\Phi 2$	$\Phi 3$
R_1	X	X	
R_3		X	
BUS		X	X
MUX_2		X	X
MUX_1			X
R_4		X	X

Since there are two 'X's in rows 1, 3, 4 and 6, the minimal latency is 2. The test time of operating e_1 and e_2 in the same test session is $2T$, which is same time required to operate these embeddings in seperate test sessions. □

Example 4 Now consider the circuit shown in Figure 4. The following I-paths are used.

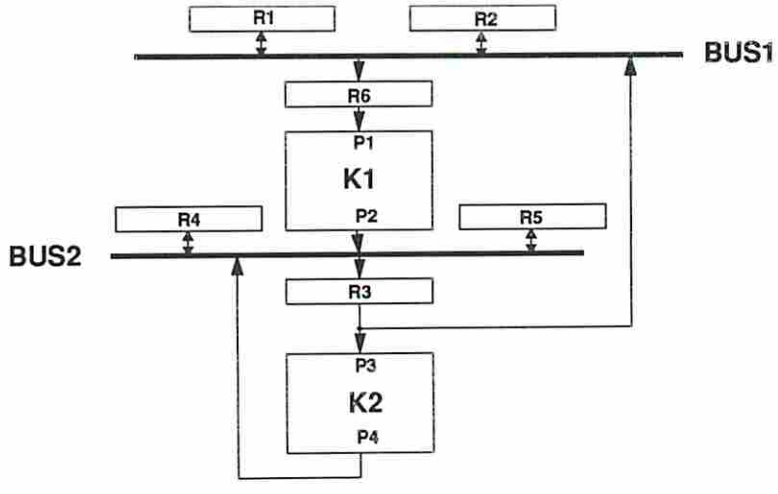

Figure 4: Circuit for Example 4

$I_1 : (R_1, BUS_1, R_6, K_1(P_1))$

$I_2 : (K_1(P_2), BUS_2, R_3, BUS_1, R_2)$

$I_3 : (R_4, BUS_2, R_3, K_2(P_3))$

$I_4 : (K_2(P_4), BUS_2, R_5)$

Let $e_1 = (K_1, I_1, I_2)$ and $e_2 = (K_2, I_3, I_4)$ be two embeddings. Since I_2 has SCs with both I_3 and I_4, they are not fully compatible. Test plans for e_1 and e_2 are

Test plan for e_1	**Test plan for e_2**
$\Phi 1 : R_1(RPG)$	$\Phi 1 : R_4(RPG)$
$\Phi 2 : BUS_1(R_1), R_6(LOAD)$	$\Phi 2 : BUS_2(R_4), R_3(LOAD)$
$\Phi 3 : K_1, BUS_2(K_1), R_3(LOAD)$	$\Phi 3 : K_2, BUS_2(K_2), R_5(SA)$
$\Phi 4 : BUS_1(R_3), R_2(SA)$	

The reservation tables for e_1 and e_2 are shown below.

embedding e_1

	$\Phi 1$	$\Phi 2$	$\Phi 3$	$\Phi 4$
R_1	X			
R_2				X
R_3			X	
R_6		X		
BUS_1		X		X
BUS_2			X	

embedding e_2

	$\Phi 1$	$\Phi 2$	$\Phi 3$
R_3		X	
R_4	X		
R_5			X
BUS_2		X	X

From the reservation tables, the test time to operate e_1 and e_2 one after the other is $2T + 2T = 4T$. Notice that to achieve the test time of $2T$ for e_1, we need to insert a NO-OP between $\Phi 3$ and $\Phi 4$ so that using minimal latency is feasible[1]. A test plan for executing the two embeddings in one test session is shown below.

$\Phi 1 : R_1(RPG)$

$\Phi 2 : BUS_1(R_1), R_6(LOAD)$

$\Phi 3 : K_1, BUS_2(K_1), R_3(LOAD), R_4(RPG)$

$\Phi 4 : BUS_1(R_3), R_2(SA), BUS_2(R_4), R_3(LOAD)$

$\Phi 5 : K_2, BUS_2(K_2), R_5(SA)$

The joint reservation table is now:

	Φ1	Φ2	Φ3	Φ4	Φ5
R_1	X				
R_2				X	
R_3			X	X	
R_4			X		
R_5					X
R_6		X			
BUS_1		X		X	
BUS_2			X	X	X

Since there are three 'X's in the last row, the minimal latency is 3. The test time of operating e_1 and e_2 in the same test session is $3T$. This is obviously better than the case where the kernels are tested one after the other. Thus e_1 and e_2 are partially compatible. □

From the above analyses, concurrent execution of compatible embeddings has the potential of reducing test time. Using these concepts we have developed an optimal procedure and a sub-optimal procedure to obtain s_{mtt}.

3.1.1 An optimal procedure

Definition 5 An embedding by itself or a set of pair-wise fully compatible embeddings is called a *Fully Compatible Embedding set(FCE)*. A set of embeddings is called a *Partially Compatible Embedding set(PCE)* if every pair of the embeddings is either fully compatible or partially compatible, but they are not pair-wise fully compatible.

Theorem 1 The test time for executing embeddings in a FCE or PCE in one single test session is shorter than that of executing them one after the other.

Proof Since each pair of embeddings in a FCE must satisfy the condition for partially compatibility, if the statement for a PCE is true, it is also valid for a FCE. Therefore it is sufficient to prove only the case of a PCE.

Let E be a PCE set. For all e_j, $e_k \in E$,

$$\max_{i=1}^{m}(X_{i,j} + X_{i,k}) < \max_{i=1}^{m} X_{i,j} + \max_{i=1}^{m} X_{i,k} \tag{1}$$

To show the test time is shorter if embeddings are operated in one test session, we need to prove that

$$\max_{i=1}^{m} \sum_{e_j \in E} X_{i,j} < \sum_{e_j \in E} \max_{i=1}^{m} X_{i,j} \tag{2}$$

If equation (2) is false, it implies that $\max_{i=1}^{m} \sum_{e_j \in E} X_{i,j} = \sum_{e_j \in E} \max_{i=1}^{m} X_{i,j}$. It is obvious that $\exists\, i_0 \ni \forall\, e_j \in E$, $\max_{i=1}^{m} X_{i,j} = X_{i_0,j}$. Hence $\exists\, e_j, e_k \in E \ni \max_{i=1}^{m}(X_{i,j} + X_{i,k}) = X_{i_0,j} + X_{i_0,k} = \max_{i=1}^{m} X_{i,j} + \max_{i=1}^{m} X_{i,k}$, which contradicts the inequality (1). Therefore, equation (2) must hold. □

Definition 6 A set of embeddings is called a *Maximal Compatible Embedding set(MCE)* if it is a FCE or PCE and is not a proper subset of any other MCEs.

Corollary 1 The test time of executing embeddings of a MCE in one single test session is shorter than that of executing them one by one.

Proof By definition a MCE is a FCE or PCE. Based on Theorem 1, the test time is shorter if the embeddings are operated in a single test session. □

A kernel is said to be *covered* by an FCE(PCE) if it is the kernel of an embedding which belongs to the FCE(PCE). A kernel can be covered by more than one FCE(PCE). A solution of the testable design problem is represented by a set of FCEs and PCEs so that every kernel is covered at least once. Each FCE or PCE in the solution needs one test session. The *minimal test time solution (s_{mtt})* of the testable design problem is a solution whose cumulative test time of the FCEs and PCEs is minimal. A set of MCEs, denoted by $\mathcal{M}$, is said to be *reducible* to a solution if every FCE or PCE in this solution is a subset of some MCE belonging to $\mathcal{M}$. The existence of reducible MCE sets for each solution is obvious. It is also clear that in a reducible MCE set, each kernel is covered at least once. The approach we employ to find s_{mtt} is to intelligently search for the MCE set which is reducible to s_{mtt}, and then reduce it to the optimum solution.

The problem of finding all $MCEs$ from the set of embeddings $\mathcal{E}$ can be solved by using classical algorithms for finding maximal compatible sets of states in the state minimization problem[3]. We have modified a classical algorithm to generate all possible $MCEs$. Let the set of all $MCEs$ be $\{MCE_1, MCE_2, \ldots, MCE_Q\}$, we can construct an MCE covering matrix $[\mathbf{C}']_{Q \times N}$ where

$$\mathbf{C}'(\mathrm{i},\mathrm{j}) = \begin{cases} 1 & \text{if } MCE_i \text{ covers } K_j \\ 0 & \text{otherwise} \end{cases}$$

A branch and bound algorithm has been developed to find the minimal test time solution. The algorithm constructs a search tree where each node in the tree is a 4-tuple ($\hat{\mathcal{K}}$, $\hat{T}$, $\hat{\mathcal{R}}$, $\hat{\mathcal{A}}$). $\hat{\mathcal{K}}$ is the set of kernels that are not yet covered, $\hat{T}$, $\hat{\mathcal{R}}$ and $\hat{\mathcal{A}}$ are the lower bound test time, cumulative rm-pairs and lower bound area overhead, respectively, from the root node to the current node. They will be explained shortly after a few concepts are introduced. If there are N kernels K_1, K_2, ..., K_N, then the root node is represented as ($\{K_1, K_2, \ldots, K_N\}$, 0, $\emptyset$, 0).

A leaf node is a node whose $\hat{\mathcal{K}}$ is empty, that is, every kernel has been covered at least once. For a non-leaf node ($\hat{\mathcal{K}}$ not empty), the next kernel to be covered is selected from the set $\hat{\mathcal{K}}$. Suppose $K_j \in \hat{\mathcal{K}}$ is selected, let $\mathcal{M}_j = \{MCE_i \mid \mathbf{C}'(i,j) = 1\}$ be the set of $MCEs$ that can cover K_j, then there are $|\mathcal{M}_j|$ branches from this node where each branch represents a different embedding for K_j. Our objective is to keep the size of search tree as small as possible, hence the next kernel to be covered is chosen so that it will expand the tree the least amount, i.e. adds the fewest number of branches. This can be formulated by the following rule.

Rule 1 *Select $K_j \in \hat{\mathcal{K}}$ as the next kernel to be covered so that $|\mathcal{M}_j| = \min_{K_{j'} \in \hat{\mathcal{K}}} |\mathcal{M}_{j'}|$.*

After using Rule 1 to select a kernel, say K_j, the order of traversing the $|\mathcal{M}_j|$ branches is also very important. Let $MCE_i \in \mathcal{M}_j$ and the number of uncovered kernels that are covered by MCE_i is $\mathcal{K}_i$, then besides K_j there are $\mathcal{K}_i$-1 extra kernels covered by MCE_i. Hence it reduces the size of $\hat{\mathcal{K}}$ by $\mathcal{K}_i$. So if we always choose the branch which covers the largest number of uncovered kernels to traverse, then we have a good chance to reach a leaf node quickly. From this argument we obtain the following rule.

Rule 2 *Given the kernel to be covered, say K_j, select $MCE_i \in \mathcal{M}_j$ as the next branch to traverse so that $\mathcal{K}_i = \max_{MCE_{i'} \in \mathcal{M}_j} \mathcal{K}_{i'}$.*

Once a leaf node is reached, the set of branches leading from the root node to the leaf node represents a reducible MCE set. It is possible that a reducible MCE set can be reduced to different solutions. For example, suppose there are 3 kernels $\{K_1, K_2, K_3\}$ in the circuit. Assume MCE_1 covers K_1 and K_2 and MCE_2 covers K_1 and K_3. The MCE set $\{MCE_1, MCE_2\}$ can be reduced to a solution where K_1 and K_2 are tested in one test session and K_3 alone is tested in the second session. However, it can also be reduced to the solution where K_2 is tested in one test session and K_1 and K_3 are tested in the other session. Our objective in this section is to minimize test time, hence given a reducible MCE set, we always reduce it to a solution with the shortest test time. The reduction problem and process will be described after we introduce the procedure *min_test_time1*.

Let u be a non-leaf node, v be the father of node u and k be the kernel to be covered by node v. Suppose MCE_i is the branch between nodes u and v, then the lower bound test time of node u, $\hat{T}$, is the summation of the lower bound test time of node v and the test time of an embedding in MCE_i which contains kernel k. The rm-pair and lower bound area overhead can be calculated in a similar fashion. An estimated test time for node u is calculated as $\hat{T} + \hat{T}'$, where $\hat{T}'$ is the anticipated test time from node u to a leaf node. A loose lower bound can be calculated as $\hat{T}' = \lceil \frac{|\hat{\mathcal{K}}|}{\mathcal{K}_{max}} \rceil \cdot tt_{min}$, where $\mathcal{K}_{max} = \max\{\mathcal{K}_k | MCE_k \text{ has not been used}\}$ and $tt_{min} = \min_{e_k \in \mathcal{E}} t_k$.

We use the following bounding techniques to prune the search space.

- Set s_B to be the best test time solution up to now, tt_B be its test time.
- Set ao_B to be the area overhead of s_B.
- If the estimated test time of a non-leaf node is greater than tt_B, or equals tt_B and the lower bound area overhead is greater than ao_B, then prune this node and its descendants.

The branch and bound algorithm for finding the minimal test time solution s_{mtt}, referred to a procedure *min_test_time1*, is shown as follows.

Procedure *min_test_time1()*
begin

```
        Initialize the stack, tt_B ← ∞, ao_B ← ∞;
        Initialize K̂, R̂, T̂ and Â;
    cont:
        Create the node (K̂, T̂, R̂, Â), push it onto the stack;
        Use Rule 1 to select the next kernel to be covered, say K_j;
        if M_j is empty, backtrack();
        else begin
            Use Rule 2 to select MCE_i ∈ M_j and traverse this branch;
            Remove MCE_i from M_j and calculate the new T̂, R̂ and Â;
            if ((T̂ + T̂' > tt_B) or (T̂ + T̂' = tt_B and Â > ao_B)), then backtrack();
            else begin
                K̂ ← K̂ - K_i;
                if (K̂ = ∅)
                    then begin
                        Obtain the set of MCEs by backtracing the path from the current node
                          to the root node;
                        Reduce the set of MCEs to a shortest test time solution;
                        Calculate the test time tt and area overhead ao of this solution;
                        if ((tt < tt_B) or (tt = tt_B and ao < ao_B))
                            then tt_B ← tt, ao_B ← ao, mark this solution as s_B. backtrack();
                    end
                else goto cont;
            end
        end
    end
Procedure backtrack()
    begin
        if stack is empty, exit min_test_time1;
        else pop the stack;
    end
```

Definition 7 Given a set of MCEs M so that every kernel is covered at least once. A *reduction* of M is a set of FCEs and PCEs where each FCE or PCE is a subset of some MCE in M, so that every kernel is covered exactly once. The *reduction problem* is to find a reduction of M which has the shortest test time.

Given a reducible MCE set M, a reduction from M can be obtained as follows: For each kernel there are at least one MCE $\in M$ that covers it. Let M_i be $\{MCE_j \mid MCE_j \in M \text{ and } \mathbf{C}'(j,i) = 1\}$. For each kernel, say K_i, assign one MCE in M_i to it and mark the embedding in this MCE that contains K_i as *used*. After every kernel is processed, the embeddings in each MCE $\in M$ that are marked form either a FCE or a PCE, and the collection of these FCE/PCEs is a reduction. All possible reductions can be obtained by trying every combination of assignments. The reduction problem is solved by first finding all possible reductions from M, then choose the one with the smallest test time.

Example 5 For the circuit in Figure 1, Procedure *find_all_embeddings* generates the following 10 embeddings.

$e_1 : (K_1, I_1, I_3)$ $e_6 : (K_2, I_7, I_{10}, I_{11})$

$e_2 : (K_1, I_1, I_4)$ $e_7 : (K_2, I_8, I_9, I_{11})$

$e_3 : (K_2, I_6, I_9, I_{11})$ $e_8 : (K_2, I_8, I_{10}, I_{11})$

$e_4 : (K_2, I_6, I_{10}, I_{11})$ $e_9 : (K_3, I_{12}, I_{13}, I_{15})$

$e_5 : (K_2, I_7, I_9, I_{11})$ $e_{10} : (K_3, I_{12}, I_{13}, I_{16})$

From these 9 MCEs can be generated. We assume initially every register in the circuit has the HOLD and LOAD functions. The area overhead of adding the RPG or SA function to a register is 2 (units), and adding both RPG and SA to a register costs 3 (units). There are 16 reducible MCE sets, only 4 of them are generated by the branch and bound algorithm. The rest are pruned due to the bounding techniques. The minimal test time solution obtained by Procedure *min_test_time1* is summarized as follow.

```
        *** Optimal solution ***
test time :  2T
area overhead :  9 (units)
use 2 test sessions
    **test session #1:
    embedding 9 is used to test kernel 3 as follows:
        I-path 12 covers port 6
        I-path 13 covers port 7
        I-path 15 covers port 8
    **test session #2:
    embedding 1 is used to test kernel 1 as follows:
        I-path 1 covers port 1
        I-path 3 covers port 2
    embedding 3 is used to test kernel 2 as follows:
        I-path 6 covers port 3
        I-path 9 covers port 4
        I-path 11 covers port 5
    the following registers need to be modified
        R1:  RPG function is added
        R2:  RPG function is added
        R3:  RPG SA functions are added
        R4:  SA function is added
```

□

Theorem 2 Procedure *min_test_time1* generates a minimal test time solution.

Proof Since the estimated test time in Procedure *min_test_time1* is always a lower bound value, we will never prune out the optimal solution[7]. If the bounding techniques are not used, the search procedure is capable of generating every MCE set that is reducible to some solutions. Based on these two observations, optimality is guaranteed. □

Although this procedure generates an optimal solution, the computation time for some moderate sized circuits can be excessive due to the extremely large amount of MCE sets. Next we will present a heuristic procedure which is significantly faster than Procedure *min_test_time1*, but does not guarantee an optimal solution.

3.1.2 A sub-optimal procedure

The main problem with Procedure *min_test_time1* is the breadth of the search tree it builds. Instead of using MCE sets, we will use embeddings to cover a kernel in the search process. We implicitly enumerate the combinations of embeddings so that every kernel is covered once. For every such combination, a greedy approach is used to schedule the embeddings with the smallest amount of test time.

Let $\mathcal{E}_i$ denote the set of embeddings that cover K_i. A branch and bound procedure has been implemented to enumerate the combinations of embeddings. The procedure constructs a search tree, where each node is a 3-tuple (PS, PT, PA). PS represents a partial schedule for the embeddings used up to this node, and PT and PA are the test time and area overhead for PS, respectively. A partial schedule consists of a set of test sessions, and each test session consists of a set of pair-wise fully or partially compatible embeddings. PT can be calculated from PS, and PA can be obtained by examining the I-paths involved. A node at level i has $|\mathcal{E}_i|$ branches to represent the embeddings that cover K_i. Once a branch (an embedding) is chosen, say e_k, a greedy scheduling approach is used to find the first test session in PS so that e_k is compatible with every embedding in this test session. If such test session exists, e_k is added to it. If e_k cannot fit into any of the test sessions in PS, a new test session is created which contains only e_k. The test time of this new partial schedule is calculated, and if it is greater than the best(smallest) test time generated so far, this node (and its descendants) are pruned. Once a leaf node (a node at level N) is reached, a complete schedule is obtained and the best test time up to now is updated if necessary.

The procedure, known as *min_test_time2*, is shown below, where s_B, tt_B and ao_B are the same as in previous section.

```
Procedure min_test_time2()
  begin
      Initialize the stack, tt_B ← ∞, ao_B ← ∞, PS ← ∅, PT ← 0, PA ← 0, i ← 1;
    cont:
      Create the node (PS, PT, PA);
      if E_i is empty, backtrack();
      else begin
          Choose the first embedding in E_i, traverse this branch and remove the embedding from E_i;
          Schedule this embedding using greedy approach, update PS, PT and PA;
          if ((PT > tt_B) or (PS = tt_B and PA > ao_B)), then backtrack();
          else begin
              if (i = N) then
                  tt_B ← PT, ao_B ← PA and mark this solution as s_B. Backtrack();
              else i = i + 1, goto cont;
          end
      end
  end
Procedure backtrack()
    begin
      if stack is empty, exit min_test_time2;
      else pop the stack, i = i − 1;
    end
```

In the worst case this procedure will construct every combination of embeddings which covers each kernel, which in general is much faster than Procedure *min_test_time1*. The greedy approach we used to schedule each combination of embeddings cannot guarantee finding the shortest test time solution. However, in the experiments we have carried out, it did generate the optimal solutions for all cases.

3.2 Minimal area overhead solution

The objective of this section is to cover every kernel (or K-port) so that the area overhead is minimal. A branch and bound algorithm, call procedure *min_area_overhead*, is used to generate a minimal area overhead solution. The I-path covering matrix C, modification matrix M, and I-path conflict matrix IC defined in Section 2 are used. The algorithm constructs a search tree that is levelized by the indices of K-ports. Namely, a node at level j represents a modified circuit where K-ports with indices less than or equal to j have already been covered and the remaining K-ports are not yet covered. The depth of the tree is M, the number of K-ports. Each node is a 3-tuple $(\tilde{\mathcal{I}}, \tilde{\mathcal{R}}, \tilde{\mathcal{A}})$. $\tilde{\mathcal{I}}$ is a set of I-paths that are not allowed to be used due to I-path conflicts. The construction of $\tilde{\mathcal{I}}$ will be explained later. $\tilde{\mathcal{R}}$ is a set of rm-pairs similar to $\hat{\mathcal{R}}$ defined in the previous section. $\tilde{\mathcal{A}}$ is the area overhead calculated from $\tilde{\mathcal{R}}$. The root node is $(\emptyset, \emptyset, 0)$.

A leaf-node is a node at level M which represents a testable design of the original circuit. $\tilde{\mathcal{A}}$ of the leaf node gives the area overhead of that particular design. For a non-leaf node at level j, the next K-port to be covered is P_{j+1}. The possible I-paths that can cover P_{j+1} without considering other K-ports is denoted by the set $\mathcal{I}_{j+1} = \{I_i \mid \mathbf{C}(i, j+1) = 1\}$. Due to I-path conflicts the set of I-paths that can be used to cover P_{j+1} in the search process is reduced to $\mathcal{I}_{j+1}$ - $\tilde{\mathcal{I}}$. Hence there are only $|\mathcal{I}_{j+1}$ - $\tilde{\mathcal{I}}|$ branches from this node. Each branch represents a unique I-path, say I_i, that covers P_{j+1}. The son of this node, obtained by traversing the branch I_i, will have a new $\tilde{\mathcal{I}}$ which is $\tilde{\mathcal{I}} \bigcup \{I_{i'} | \mathbf{IC}(i, i') = F\}$.

The following bounding technique is used in the algorithm, where let $\mathcal{A}_B$ to be the best(least) area overhead found so far; if the area overhead of a node is greater than or equal to $\mathcal{A}_B$, prune this node and its descendants. This algorithm is presented here.

```
Procedure min_area_overhead()
  begin
      Initialize the stack, A_B ← ∞;
      Initialize Ĩ, R̃ and Ã;
      j ← 1;
   cont:
      while (j ≤ M) do
      begin
          Create the node (Ĩ, R̃, Ã), push it onto the stack;
          I_j ← I_j - Ĩ;
          if I_j is empty, backtrack();
```

```
            else begin
                Let I_i be the first I-path in I_j, remove it from I_j and traverse this branch;
                Calculate the new R̃ and estimate its area overhead Ã;
                if (Ã ≥ A_B), backtrack();
                else Ĩ ← Ĩ ⋃ {I_i' | IC(i, i') = F};
                j ← j + 1;
            end
        end
        if (Ã < A_B) then
            A_B ← Ã, mark this solution as the best up to now, backtrack(), and goto cont;
    end
Procedure backtrack()
    begin
        if stack is empty, exit min_area_overhead;
        else pop the stack and goto cont of min_area_overhead();
    end
```

Since we never over-estimate the area overhead in the search process, this algorithm guarantees an optimal solution.

Example 6 The circuit in Figure 1 is used again to demonstrate this algorithm. The cost model used in Example 5 are used here also. Ignoring conflicts there are 96 I-path combinations that can cover every K-port. Out of these only 2 combinations are generated by Procedure *min_area_overhead*, the rest are pruned either by the conflict criterion or by the bounding technique. The minimal area overhead solution is summarized as follows.

```
    *** Optimal solution ***
area overhead:  7
    I-path 1 covers port 1
    I-path 4 covers port 2
    I-path 7 covers port 3
    I-path 9 covers port 4
    I-path 11 covers port 5
    I-path 12 covers port 6
    I-path 13 covers port 7
    I-path 16 covers port 8
the following registers need to be modified
    R1:  RPG function is added
    R2:  SA function is added
    R3:  RPG and SA functions are added
```

□

3.3 Intermediate solutions

In general the area overhead of s_{mtt} may be too high and the test time of s_{mao} may be too long. Neither of them is a good solution if both attributes are important for the design consideration. To develop an intelligent and user-friendly testable design system, it is important to provide the designer with a family of solutions having different test time and area overhead. It is useful if a designer can trade area overhead for test time without manually exploring the whole solution space. Thus it is useful to generate a set of intermediate solutions.

Notice that if no pruning is made in Procedure *min_test_time1*, it is capable of generating every reducible MCE set, and therefore all possible solutions can be reduced from these MCE

sets. Similarly, if no pruning is carried out Procedure *min_test_time2* is capable of generating all possible combinations of embeddings if no pruning is carried out. There may be too many solutions for a designer to consider. To make the problem more tractable, solutions can be partitioned into groups, where solutions with similar test time are put into the same group. In a group of solutions only the one with smallest area overhead is considered, which is called a *representative solution*. Let $t1$ be the test time of s_{mtt}; $t2$ be the test time of s_{mao}. Define $\Delta t = \frac{t2-t1}{G}$, where G is the number of representative solutions desired. If $\mathcal{N}$ is the set of solutions with test time no greater than $t2$, then we can partition $\mathcal{N}$ into G groups, where group i contains solutions with test time in the interval $(t1 + (i-1) \times \Delta t, t1 + i \times \Delta t]$.

As mentioned in Section 3.1, in most cases Procedure *min_test_time2* is much more efficient than Procedure *min_test_time1*. A search algorithm slightly different from Procedure *min_test_time2* is employed to generate the set of representative solutions. The same notation is used except that we keep track of the best area overhead for every group. Let $\mathcal{A}_B(i)$ denote the best area overhead of a solution in group g_i generated so far. If the partial test time of a node exceeds $t2$, this node and its descendants are pruned since they do not belong to the solution space. The procedure for generating intermediate solutions is shown below.

```
Procedure inter_sol()
  begin
      Initialize the stack, PS ← ∅, PT ← 0, PA ← 0, i ← 1;
      for i = 1 to G do
          A_B(i) ← ∞;
   cont:
      Create the node (PS, PT, PA);
      if E_i is empty, backtrack();
      else begin
          Choose the first embedding in E_i, traverse this branch and remove the embedding from E_i;
          Schedule this embedding using greedy approach, update PS, PT and PA;
          if (PT > t2), backtrack();
          else begin
              if (i = N)
              then begin
                  Identify the group where this solution belongs to, say g_k;
                  if (PA < A_B(k)) then A_B(k) ← PA and mark this solution as the
                      representative solution of group g_k, backtrack();
              end
              else i = i + 1, goto cont;
          end
      end
  end
Procedure backtrack()
  begin
      if stack is empty, exit inter_sol;
      else pop the stack, i = i − 1;
  end
```

The circuit in Figure 1 is too simple to use to illustrate Procedure *inter_sol*, because it contains only two representative solutions, namely s_{mtt} and s_{mao}. In the next section, larger circuits will be considered and results presented.

Circuit	# of kernels	# of MUXes	# of registers	# of I-paths	# of embeddings
DAC4	10	19	11	43	21
DAC8	6	18	7	39	33
DAC10	4	12	9	30	36
DAC16	3	9	8	30	92

Table 1: Summary of circuits

Circuit	# of representative solutions	s_{mtt}		s_{mao}		intermediate solutions		CPU time (sec)
		TT	AO	TT	AO	TT	AO	
DAC4	3	3T	28	5T	26	4T	27	3.4
DAC8	2	2T	14	3T	12	N/A	N/A	5.7
DAC10	3	1T	18	3T	13	2T	14	4.7
DAC16	3	1T	10	3T	6	2T	8	3.6

TT - Test time, AO - Area Overhead

Table 2: Summary of the results

4 Experimental results

We have tested these procedures on circuits generated by *MABAL* (a Module And Bus ALlocator)[9]. MABAL is a CAD tool developed by the USC design automation group for automatically synthesizing RT circuits from a behavioral description. It examines the trade-offs between functional unit allocation, register allocation, interconnect allocation and module binding while minimizing the overall hardware area. The circuits tested are summarized in Table 1 with results shown in Table 2.

The assumption that every kernel requires the same number of test vectors greatly reduces the number of representative solutions. For the purpose of illustration, we also test a circuit shown in Figure 5. The solutions generated are summarized in Table 3. The CPU time consists of both pre-processing time (i.e. finding I-paths, embeddings, etc), and the run time of the procedure itself. It is measured on a Sun-4/490 machine. Instead of separating the resource allocation and scheduling problems as most existing systems do, our system deals with them at the same time.

solution	test time	area overhead
s_{mtt}	1T	12
intermediate solution #1	3T	11
intermediate solution #2	4T	8
intermediate solution #3	6T	6
s_{mao}	7T	4

Table 3: Solutions of circuit in Figure 5

A solution contains the information about extra hardware required, area overhead, test schedule, and test time, hence no pre-processor[6] or post-processor[5] are needed. Due to the concept of exploration of the solution space, a user can select the most appropriate solution by a simple comparison process, which cannot be achieved by other systems.

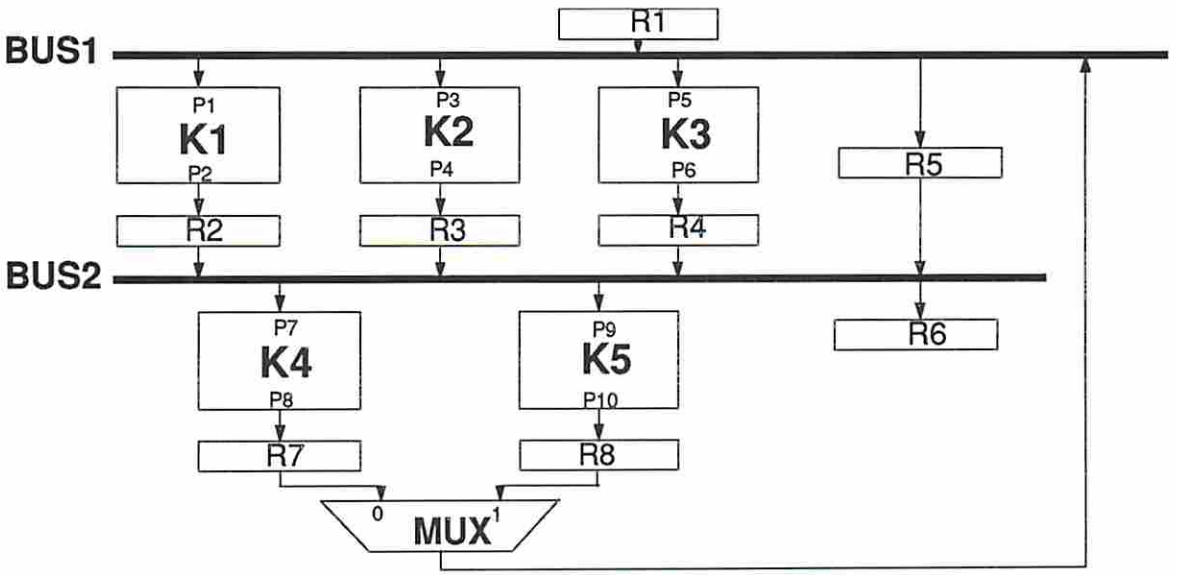

Figure 5: Example circuit for Table 3

5 Conclusion and future research

This report demonstrates a systematic approach to generate a set of testable versions of a design. The space of testable design is explored so that the minimal area overhead, minimal test time, and intermediate solutions are provided to the designer for consideration. A *Self-Adaptive*

Expert Selection System (SAESS)[10], developed by the authors, can then be employed to choose the solution which closely matches the designer's requirements in terms of area overhead and test time.

Though the BILBO architecture is the only TDM considered in this report, extension to other TDMs can easily be dealt with. Trade-off between area overhead and test time is the unique feature of this system. Other design considerations such as extra I/O pins and fault coverage will be considered in the near future. More efficient heuristics may need to be employed when very large circuits are considered in order to reduce CPU time. We are investigating the case where numbers of test vectors for kernels are not all the same currently.

References

[1] M.S. Abadir and M.A. Breuer. A Knowledge-Based System for Designing Testable VLSI Chips. *IEEE Design & Test.* Aug. 1985, pp. 56-68.

[2] B. Konemann, J. Mucha, and G. Zwiehoff. Built-In Logic Block Observation Technique. *Digest of Papers 1979 Test Conf.* Oct. 1979, pp.37-41.

[3] J. Ellison. *Techniques in Advanced Switching Theory, 2nd Printing.* Los Angeles, 1989.

[4] K.Hwang and F.A.Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.

[5] P.R. Chalasani, S. Bhawmik, A. Acharya and P. Palchaudhuri. Design of Testable VLSI Circuits with Minimum Area Overhead. *IEEE Trans. on Computer.* Vol.C-38, No.10, Oct. 1989, pp.1460-1462.

[6] G. Craig, C. Kime, and K. Saluja. Test Scheduling and Control for VLSI Built-In Self-Test. *IEEE Trans. on Computers.* Vol.C-37, No.9, Sept. 1988, pp. 1099-1109.

[7] E. Rich. *Artificial Intelligence.* McGraw-Hill, New York, 1983.

[8] A. Kraśniewski and S. Pilarski. Circular Self-Test Path: A Low-Cost BIST Technique for VLSI Circuits. *IEEE Trans. on CAD*, Vol.8, No.1, Jan. 1989, pp. 46-55.

[9] K. Kucukcakar and A.C. Parker. MABAL: A Software Package for Module and Bus ALlocation, *Int'l J. of Computer Aided VLSI Design.* Vol.2, No.4, 1990.

[10] S.P. Lin, M.A. Breuer, C.A.Njinda. A Self-Adaptive Expert Selection System(SAESS) and its Application to Selection Problems. *Third Int'l Conf. on Software Engineering and Knowledge Engineering.* June. 1991.