TRANSFORMING BEHAVIORAL SPECIFICATIONS TO FACILITATE SYNTHESIS OF TESTABLE DESIGNS

Sujit Dey and Miodrag Potkonjak
C&C Research Laboratories, NEC USA, Princeton, NJ 08540

ABSTRACT

Recently, several high level synthesis approaches have been proposed to synthesize testable data paths from behavioral specifications. This paper introduces a novel technique to transform behavioral specifications, such that an existing behavioral test synthesis system can generate area-efficient, testable designs with significantly lower partial scan overhead. Experimental results demonstrate the significant savings in partial scan overhead when the transformation is applied before using the behavioral test synthesis system to synthesize 100% test-efficient designs.

1.0 INTRODUCTION

Recently, several high level synthesis approaches have been proposed to generate easily testable data paths for both Built-In-Self-Test (BIST)-based testing methodology [Pap91,Avr91,Har93,Avr93], and Automatic Test Pattern Generation (ATPG) methods [Che92, Maj92, Lee92, Lee93, Dey93, Dey94]. Testability improvement using register assignment and scheduling were reported in [Lee92]. Chen and Saab [Che92] used a high-level testability analysis program to identify testable structures and synthesize them to improve testability. An approach to generate testable data paths, by minimizing the number of self-loops, was reported in [Maj92].

It is known that presence of loops in a sequential circuit makes sequential ATPG very difficult. A partial scan approach which breaks all loops, except self-loops, by scanning a subset of flip-flops [Che90, Lee90, Chi91], has evolved as a cost-effective solution to the sequential ATPG problem. A circuit synthesized from behavioral specification has a natural tendency to contain loops, partly due to the presence of loops in the Control-Data Flow Graph (CDFG) corresponding to the behavioral specification, and partly due to hardware sharing used to optimize hardware resources like execution units. Description of the high level synthesis tasks, like allocation, scheduling and assignment, involved in synthesizing RT-level designs from behavioral descriptions, can be found in [McF92,Wal91]. Several techniques have been suggested to synthesize data paths without loops, by using proper scheduling and assignment, and scan registers to break loops [Lee92, Lee93, Dey93]. It has been demonstrated that it is possible to generate designs which do not compromise resource utilization while significantly reducing the partial scan overhead by paying early attention to formation of loops in the data path [Dey93]. However, until now, the influence of only two high level synthesis tasks, scheduling and assignment, on the testability of the design has been explored.

This paper explores the relationship between the behavioral specification, or the CDFG, and the testability of the design synthesized by high level synthesis. It is shown that if the original CDFG is transformed by the technique introduced in the paper, the partial scan overhead required to synthesize a testable design from the transformed CDFG may be significantly less than the partial scan cost associated with the original CDFG. This paper introduces a novel technique to transform the original specifications, such that an existent high level synthesis for testability technique can generate area-efficient, testable designs at significantly lower partial scan overhead. Experimental results demonstrate the significant savings in partial scan overhead when the transformation is applied before using the high-level synthesis for testability technique to synthesize 100% test-efficient designs.

1.1 Transforming Behavioral Specification

Computational transformations alter the structure of control-data flow graph (CDFG) so that the functionality of the initial specification is maintained [Wal91, Fis88]. They may be applied on different levels of control-data flowgraph (CDFG) hierarchy; popular transformations include loop transformations such as loop fusion and fission, loop interchange and software pipelining as well as transformations at operation level (e.g. substitution of multiplication with constants by series of addition and shifts). Basic block transformations are applied on flat graphs (which are either at the lowest level of CDFG hierarchy or flat CDFGs themselves) and consider a number of operations simultaneously. Transformations have been applied for optimization of a great variety of goals, including area, throughput, power and fault tolerance. This paper introduces a transformation aimed at optimizing the partial scan cost for synthesis of testable designs.

Many operations used in a computation specification has an identity element associated with it. For instance, an addition operation has an identity element zero, while a multiplication operation has an identity element one. If one of the inputs of an operation is *v*, and the other input is the identity element of the operation, then the output of the operation remains *v*. Adding such an operation *op* between two operations op_1 and op_2 has the effect of deflecting the result of op_1 to *op*, before reaching op_2; hence, we term such an operation a **deflection operation**. A deflection operation can be added anywhere in a behavioral specification (CDFG), without changing the functionality of the computation. Adding such an operation *op* between two operations op_1 and op_2 has the effect of deflecting the result of op_1 to *op*, before reaching op_2. In particular, adding a deflection operation after a variable *v* has been computed, keeps the behavior invariant, because the output of the deflection operation is also *v*.

Techniques using duplicate operations, called hot potato techniques, have been applied earlier in the computer networks domain [Bar64], and pipeline optimization [Pat78]. In this paper, we introduce a new transformation technique, termed **hot potato technique**, based on adding deflection operations to the CDFG, while preserving the functionality of the computation. The transformation technique is used to facilitate an existing behavioral test synthesis system BETS [Dey93] to produce a testable design at a significant saving in the associated partial scan overhead

2.0 MINIMIZING PARTIAL SCAN OVERHEAD BY TRANSFORMATION

We assume that the underlying hardware model used is the dedicated register file model. This model assumes that all registers are grouped in certain number of register files (each register file contains one or more registers) and that each register file can send data to exactly one execution unit. At the same time each execution unit can send data to an arbitrary number of registers files. This model is used not just in a number of high level synthesis systems [Rab91], but also in many manual ASIC and general purpose data paths [Pat89]. In all motivational examples, for the sake of simplicity, it is assumed that all operations take one control step for their execution.

A data path synthesized from a behavioral specification may contain the following types of loops: CDFG loops, assignment loops, sequential false loops, and register file cliques [Dey93]. A CDFG loop is formed in the data path when there exists a cycle consisting of data-dependency edges in the CDFG. The other types of loops are introduced in the data path during behavioral synthesis, specifically hardware sharing. A comprehensive analysis of the formation of loops, in circuits synthesized by high level synthesis techniques, is presented in [Dey93], and will be briefly explained in Sections 2.1 and 2.4. The synthesis for testability technique (SFT) presented in [Dey93] first selects scan variables which can be assigned to scan registers to break all the CDFG loops present. Subsequently, it performs scheduling and assignment while trying to avoid the formation of other types of loops by reusing the selected scan registers. The SFT algorithm has been implemented in the **BETS** behavioral test synthesis system.

We demonstrate how the hot potato transformation technique can be used to minimize the number of scan registers needed to break CDFG loops, and avoid the formation of other types of loops during assignment. The original behavioral specification (CDFG) is first transformed by adding suitable deflection operations, so that the number of scan registers needed to break CDFG loops is minimized. Subsequently, a second set of deflection operations are added so that the selected scan registers can be effectively used to avoid formation of loops during assignment. The SFT algorithm [Dey93], implemented in BETS, is then applied to the transformed CDFG to synthesize a testable data path with minimal number of scan registers.

2.1 An Example

Consider the CDFG of the 3rd order Continued Fraction IIR filter shown in Figure 1(a) [Mit72]. Addition operations, subtraction operations, and multiplication operations are denoted by +, -, and *, respectively. The positive input of a subtraction operation is labeled by P. To break the CDFG loops, the behavioral test synthesis system BETS [Dey93] selects the scan variables (D1,+2), (D2,+1) and (-2,*4), requiring 3 scan registers, as the selected scan variables cannot be shared. Note that the CDFG loops cannot be broken using fewer scan registers. One more scan register will is needed to avoid formation of loops during assignment. Assume that each operation in the CDFG takes one control cycle. Figure 1(a) shows the schedule and assignment obtained by BETS [Dey93], satisfying the available time of 7 control steps, and using minimal number of execution units. For instance, the operation +1 is scheduled in control step 3, and assigned to

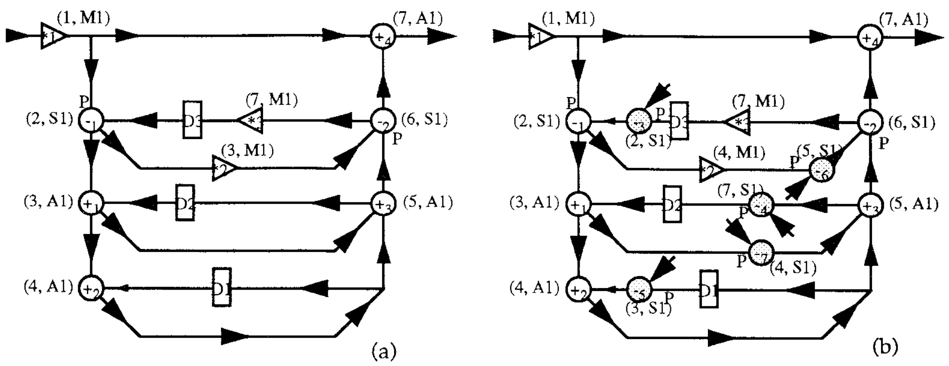

Figure 1: CDFG of the continued fraction IIR filter: (a) the original CDFG, and (b) the transformed CDFG, after the addition of deflection operations

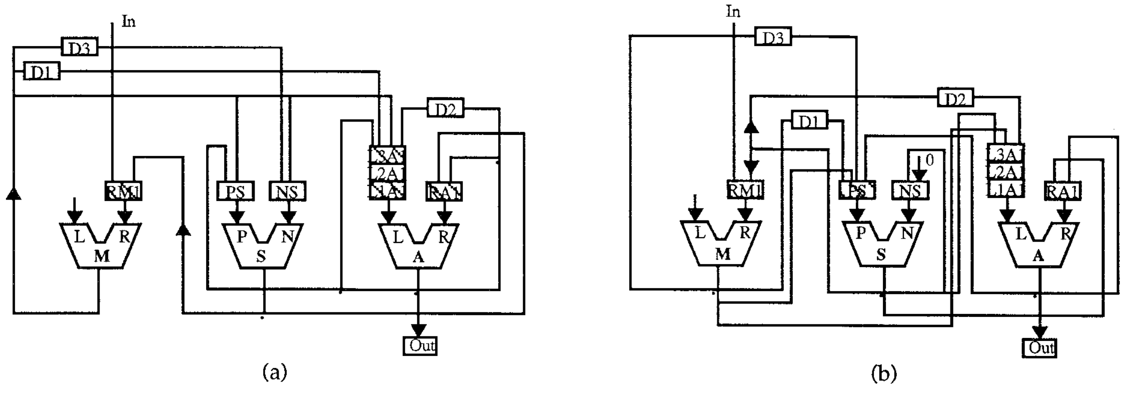

Figure 2: The data paths synthesized by BETS [Dey93] for the CDFGs of Figures 1(a) and 1(b). The scan registers are shown shaded.

adder A1, shown by the tuple (3,A1). The resultant data path is shown in Figure 2(a). The selected scan registers to break the CDFG loops are L3A1, L1A1, RA1, and RM1.

During the assignment phase, further loops can be introduced. For instance, in the case of the CDFG in Figure 1(a), both the operations +3 and +4 had to be assigned to the same adder, thus creating an *assignment loop* (RA1, PS, RA1) in the data path in Figure 2(a). When the operations along a CDFG path from operation u to operation v are assigned *n* separate modules, and u and v are assigned the same module, an assignment loop of length *n* is created in the data path [Dey93]. Though there is no path in the CDFG from a "-" operation back to a "-" operation through an addition operation, a loop, (NS,RA1,NS) is formed in the data path in Figure 2(a). This is a *sequential false loop*, introduced because -1 and +1 are assigned to S1 and A1, while +3 and -2 are assigned to A1 and S1 respectively. A comprehensive analysis of the formation of loops in the data path is presented in [Dey93]. The scan register selected by BETS to avoid the assignment and false loops during the assignment phase is RA1, for a total of 4 scan registers. Note that scanning the 4 registers L3A1, L1A1, RM1 and RA1, shown shaded in Figure 2(a), leaves no loops, besides self-loops, in the data path.

Figure 1(b) shows the transformed CDFG, derived from the original CDFG of Figure 1(a) by adding deflection operations -3, -4, -5, -6, and -7 (shown shaded). The behavioral test synthesis system BETS [Dey93] is applied to the transformed CDFG (Figure1(b)). All CDFG loops can be broken by selecting the scan variables (D1,-5), (+3,-4) and (D3,-3). All these scan variables can share the same scan register. Addition of deflection operations reduces the minimum number of scan registers required to break all CDFG loops from 3 in the original CDFG to 1 in the transformed CDFG. Moreover, all assignment and false loops can be avoided by assigning the variables (*3,-1) and (+3,-2) to the partial scan register selected, PS. The data path generated by BETS is shown in Figure 2(b). Note that if the register PS, shown shaded, is scanned, the data path does not have any loops. Consequently, synthesizing a testable data path from the CDFG transformed by adding deflection operations is significantly more economical than from the original CDFG, requiring only 1 scan register as opposed to 4 scan registers needed for the original CDFG. We describe the process of adding the suitable deflection operations to the original CDFG in the following sections.

2.2 Selecting Scan Variables to Break CDFG Loops

The problem of breaking CDFG loops using a *minimal* number of scan registers was introduced in [Dey93]. The minimum *hardware-shared cut* (HSC) problem [Dey93] is to select a set of scan variables such that the following criteria are simultaneously satisfied:

HSC1 All CDFG loops, except self-loops, are broken,

HSC2 The selected scan variables can be assigned to a minimum number of scan registers, and,

HSC3 Reusability of the scan registers, to break the other loops formed during the subsequent scheduling and assignment phase, is maximized;

Two measures can be used to capture the effectiveness of a variable in satisfying the three criteria of the minimum HSC problem. The loop cutting effectiveness (LCE) measure helps to satisfy the first criteria, HSC1, of the minimum HSC problem. The LCE measure of a variable estimates the number of loops that will be broken by assigning the variable to a scan register. Since the number of loops can be exponential, and there is no known algorithm to count them efficiently, a random walk [Haj91] based technique was proposed to calculate the LCE measure of the variables of a CDFG.

The hardware sharing effectiveness (HSE) measure satisfies criteria HSC2 and HSC3 of the minimum HSC problem. The HSE of a variable v estimates the likelihood that v can share a scan register with other variables to break the different types of loops in the data path.

In this paper, we introduce the following HSE measure. Let f_v be the type of operation to which variable v is an input. Variable v can share a scan register with all variables x such that (i) x is a corresponding input to an operation of the same type f_v, and (ii) the lifetimes of v and x do not overlap. Let set X contain those variables x which can share a scan register with variable v. Also, let Y be a subset of X such that variables in Y can cut one or more loops not broken by scanning variable v. We define HSE(v) as: $HSE(v) = w_1|Y| + w_2|X - Y|$.

The first component reflects the number of variables that can share the same scan register assigned to v, and can be used to cut CDFG loops left unbroken by v. The second component measures the likelihood that the scan register SR1, to which the variable v will be assigned, can be effectively reused later to avoid forming loops during the subsequent scheduling and assignment phase. To favor selection of scan variables good for minimizing the number of scan registers needed to break CDFG loops, as opposed to scan variables which are good to avoid formation of loops during assignment, the weight w_1 is set to be much higher than the weight w_2.

After calculating the LCE and HSE measures for each edge e which belongs to some strongly connected component (SCC), we calculate $eff_{HSC}(e) = \alpha * LCE(e) + \beta * HSE(e)$. For each SCC, we select as scan variable the edge e with highest $eff_{HSC}(e)$, and delete the selected edges from the CDFG. We repeat the process until all CDFG loops are broken. After the scan variables have been selected to cut the CDFG loops, a minimum set of scan registers is identified to which all the scan variables can be assigned. This can be done optimally by assigning all scan variables with disjoint lifetimes to the same scan register.

2.3 Using Deflection Operations to Minimize Scan Registers Needed to Break CDFG Loops

Deflection operations can be used to minimize the number of scan registers required to break CDFG loops. This is achieved by adding the deflection operations in such a way that more of the selected scan variables can share the same scan register.

Two scan variables v and w *cannot* share the same scan register if one or both of the two conditions, termed **hardware sharing bottlenecks**, is true. Note that the hardware sharing bottleneck B1 is imposed due to the hardware model adopted.

B1. Variables v and w are inputs to two operations of different types.

B2. Variables v and w have overlapping lifetimes.

2.3.1 Adding Deflection Operations to Eliminate Bottleneck B1

Consider the CDFG shown in Figure 3(a). It has two CDFG loops: loop L1 consisting of - and + operations, the other loop L2 containing operations of the type * and >>. Variables (D_1, $-_1$) and (D_2, $*_1$) have overlapping lifetimes (bottleneck B2), and hence selecting them to break the CDFG loops will require two scan registers. Any other variable v in loop L1 and variable w in loop L2, selected to break the corresponding loops L1 and L2, cannot be shared, since they have the bottleneck B1. Hence, selection of any two scan variables to break the two CDFG loops will require two scan registers.

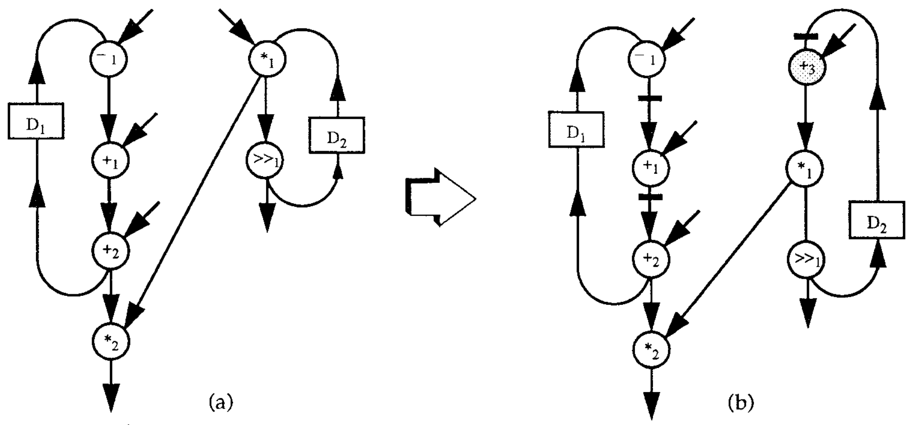

Figure 3: Using Deflection Operation to Eliminate Hardware Sharing Bottleneck B1. (a) The original CDFG, (b) The transformed CDFG, after adding deflection operation +3.

However, if the deflection operation $+_3$ is added at the input of operation $*_1$ in loop L2, as shown in the modified CDFG in Figure 3(b), the variables $(-_1, +_1)$ in loop L1 and $(D_2, +_3)$ in loop L2, do not have either of the bottlenecks, and can share the same register. Consequently, selecting $(-_1, +_1)$ and $(D_2, +_3)$ as scan variables breaks the two CDFG loops, and requires only one scan register.

Note that for any variable v that can be used to break a CDFG loop in Figure 3(a), the set Y of variables that can be used to cut other CDFG loops not broken by v, and can share the same scan register with v, is empty. On the other hand, adding the deflection operation in the CDFG shown in Figure 3(b) increases the value of $|Y|$ to 1 for the variable $(-_1, +_1)$, and adds a new variable $(D_2, +_3)$ with $|Y| = 1$. At this point, the above two variables have the highest HSE measure. The LCE measure remaining the same for all the variables in the loops, the variables $(-_1, +_1)$ and $(D_2, +_3)$ are selected as scan variables by the scan selection process outlined in the previous section, and can be assigned to one scan register. This example demonstrates how deflection operations can be used to minimize the number of scan registers required to break CDFG loops, by eliminating bottleneck B1, and increasing the HSE measure of candidate scan variables.

2.3.2 Adding Deflection Operations to Eliminate Bottleneck B2

Deflection operations can be used to eliminate the hardware sharing bottleneck B2 also. Consider a pair of variables v and w which are inputs to the same type of operation, but cannot share the same scan register due to bottleneck B2, that is, they have overlapping lifetimes. A deflection operation can be introduced on the edge w, so that the lifetime of variable w is split, and the variables v and the new input u to the deflection operation do not have any hardware sharing bottleneck.

Consider the CDFG shown in Figure 4(a), consisting of two CDFG loops L1 and L2. Each pair of variables, (v,w), v in loop L1, and w in loop L2, needed to break the CDFG loops, have one hardware sharing bottleneck, and cannot be shared. For instance, one possible pair that are inputs to the same type of operations is: $(D_1,+_1)$ and $(D_2,+_2)$. However, they have overlapping lifetimes. Similarly, variables $(-_1,*_1)$ and $(+_2,*_2)$ are inputs to the same operation type, but have overlapping lifetimes. Any pair of scan variables selected to break the CDFG loops in Figure 4(a) would require two scan registers.

The LCE and HSE measures of the variables are shown by a tuple (LCE,HSE), with the HSE part showing both the components. For example, the variable $(D_1,+_1)$ breaks one loop, and hence has an LCE value of 1. Also, $X(v) = \{*_1,+_3\}, Y(v) = \{\}$. Hence, the first component of HSE measure is 0, and the second component is 1, as shown by the tuple (1, 0+1) at the side of variable $(D_1,+_1)$.

Consider each variable in the CDFG which can break loop L2, not broken by scanning variable $(D_1,+_1)$. Consider one such variable $(D_2,+_2)$. The pair of variables can break all the CDFG loops, and are inputs to the same type of operation +, but cannot be shared because they have the hardware sharing bottleneck B2. Let us add a deflection operation, $*_3$, on the edge $(D_2,+_2)$, such that the lifetime of the split variable $(*_3,+_2)$ no longer overlaps with the lifetime of variable $(D_1,+_1)$. The transformed CDFG is shown in Figure 4(b). The bottleneck B2 is eliminated, and variables $(D_1,+_1)$ and $(*_3,+_2)$ can share the same scan register.

In terms of the HSC problem formulation, introduction of the deflection operation helps increase the HSE measure of variables, while keeping the LCE measure fixed. After

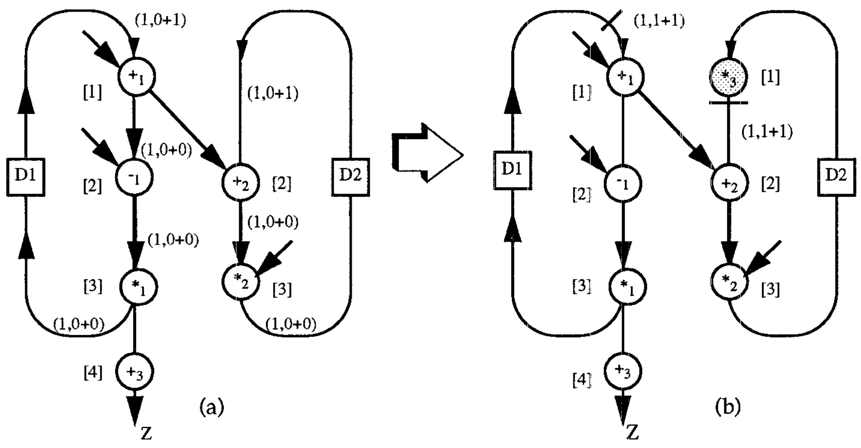

Figure 4: Using Deflection Operations to Eliminate Hardware Sharing Bottleneck B2. (a) The original CDFG, (b) the transformed CDFG, after adding deflection operation *3.

the introduction of the deflection operation, shown in Figure 4(b), variables (D_1,$+_1$) and ($*_3$,$+_2$) can now share the same scan register, and hence, the HSE measure of variables (D_1,$+_1$) increases from (0+1) to (1+1). Similarly, the split variable ($*_3$,$+_2$) now has a HSE value (1+1). The scan selection process outlined in the previous section would select variables (D_1,$+_1$), ($*_3$,$+_2$) to break the CDFG loops, requiring only one scan register.The above examples illustrate how deflection operations can be added to the CDFG to eliminate hardware sharing bottlenecks.

2.3.3 Algorithm for Adding Deflection Operations

We briefly outline the process of adding deflection operations, so that the number of scan registers required by the scan variables to break the CDFG loops is minimized. Let *Z(v)* be the set of variables *w*, such that *w* breaks some other CDFG loops not broken by *v*, and either both *v,w* are inputs to the same operation type, or *v,w* do not have overlapping lifetimes, but not both. That is, *Z(v)* is the set of all variables *w* which break other loops not broken by *v*, but cannot share the same scan register because the pair (*v,w*) have one of the two bottlenecks, B1 or B2. The aim of adding the deflection operations will be to eliminate the hardware sharing bottlenecks of those pairs of variables which are potentially good candidates for scan variables. This is achieved by attempting to increase the HSE measures of those variables *Z(v)* which can be potentially shared, after eliminating bottlenecks, with variables *v* having high LCE measure (cuts many loops). We outline below the algorithm for adding deflection operations to minimize the number of scan registers needed to break CDFG loops:

add_deflection_ops_to_break_CDFG_loops ()

```
1. for each variable v with positive LCE measure, but low HSE measure {
  2. for each w in Z(v) {
    3. add deflection operation op_d on w;
    4. check feasibility of adding op_d, with respect to scheduling and assignment;
    5. if deflection operation op_d feasible {
      6. for all variables, calculate nHSE(u) = new HSE(u) - hardware cost of adding op_d;
      7. retain the highest nHSE values for each variable;
    }
  }
}
8. select scan variables (virtual operation) assuming nHSE values;
9. for each variable v selected, add the deflection operations necessary to obtain the new HSE value, nHSE(v);
```

2.4 Detecting Formation of Loops in the Data Path During Assignment

The formation of different types of loops in the data path during the assignment phase can be captured by using a Data-Dependency and Compatibility Graph (DDCG). The DDCG models the data dependencies and the compatibilities of the operations of the CDFG. Each node in the graph represents an operation of the CDFG. There is a (undirected) compatibility edge between two operations if there is a non-zero probability that both the operations can be assigned to the same module. There is a (directed) data-dependency edge from operation *v* to operation *w* if operation *w* depends on data produced by operation *v*. We

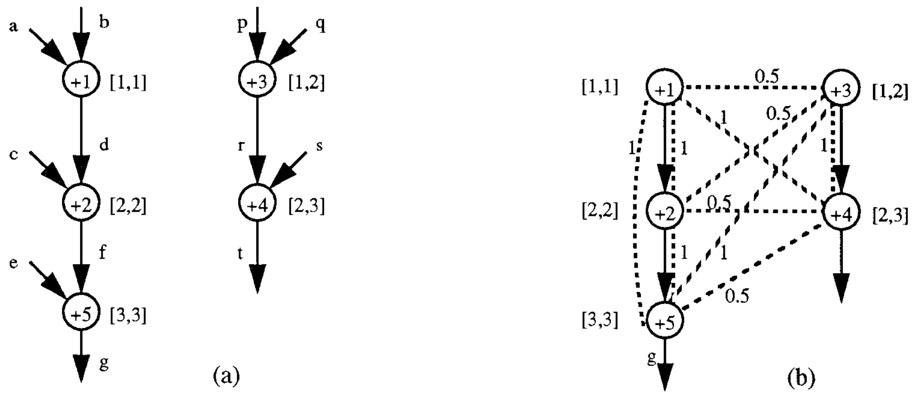

Figure 5: CDFG (a) and the corresponding DDCG (b)

will denote a compatibility edge between v and w as $c(v,w)$, and a data-dependency edge from v to w as $d(v,w)$.

Figure 5(a) shows segments of two paths in a CDFG, and Figure 5(b) shows the corresponding data-dependency and compatibility graph. Assuming that each operation in the CDFG takes one control cycle, the longest path is 3 control steps long. For a schedule which requires 3 control steps, the As Soon As Possible (ASAP) and the As Late As Possible (ALAP) control steps in which each operation can be scheduled [Wal91,McF92] is shown by the tuple [ASAP,ALAP] in Figure 5(a). In Figure 5(b), each compatibility edge $c(v,w)$, shown dotted, is weighted by an estimate of the compatibility, Comp(v,w), between two nodes v and w, as discussed below.

The mobility of an operation v, **mobility(*v*)**, is the number of control steps in which it can be scheduled:

mobility(v) = ALAP(v) -ASAP(v) + 1;

The overlap between two operations, v and w, **overlap(*v*,*w*)**, is the number of control steps in which both v and w can be scheduled. Referring to Figure 5(a), mobility(+1) = 1, mobility(+3) = 2, and overlap(+1,+3) = 1.

The estimate of the compatibility between two operations v and w, **Comp(*v*,*w*)**, is the probability that v and w can be assigned to the same module:.

$$\mathrm{Comp}(v,w) = \left\{\begin{array}{ll} 1 & \exists d(v,w) \\ 1 - \dfrac{\mathrm{overlap}(v,w)}{(\mathrm{mobility}(v) * \mathrm{mobility}(w))} & \text{otherwise} \end{array}\right\}$$

The compatibility between nodes $+_1$ and $+_3$, Comp($+_1$,$+_3$) = 1 - (1/2) = 0.5. All the compatibility edges $c(v,w)$ are weighted by Comp(v,w), as shown in the DDCG in Figure 5(b).

The paths of the DDCG can be represented using regular expressions. d^+ represents a path consisting of a sequence of one or more data-dependency edges d. The concatenation of two paths p and q is represented by $p.q$, or simply, pq. For example, the regular expression cd^+ represents a path starting with a compatibility edge c, followed by one or more occurrence of data-dependency edges. In Figure 5(b), the path ($+_5$,$+_1$), ($+_1$,$+_2$), ($+_2$,$+_5$) can be represented by cd^+.

An assignment loop is formed in the data path if there exists a cycle of the form cd^+ in the DDCG, and the compatibility edge c is used during module assignment. Similarly, if there is a cycle of the form $(cd^+)(cd^+)^+$ in the DDCG, and the compatible edges c in the cycle are used during module assignment, a sequential false loop is formed in the data path.

2.5 Adding Deflection Operations to Avoid Formation of Loops During Assignment

Deflection operations can also be added to avoid the formation of loops during assignment, namely assignment loops and sequential false loops. We adopt the approach of adding all the deflection operations required in the beginning, and performing the three phases of allocation, selecting scan variables for CDFG loops, and scheduling and assignment, on the transformed CDFG. Since we want to add the deflection operations before we know which loops can be formed during assignment, we use the DDCG and the compatibility estimates to insert deflection operations. The DDCG has information about all possible loops that can be formed during assignment. We consider only those loops that have a high possibility of formation during assignment, as reflected by the high values of comp(v,w) on their c edges. We ignore those loops that have a low possibility of forming, reflected by low values of comp(v,w) on their c edges. To achieve this, we delete all those compatibility edges which have compatibility estimate less than a threshold.

We insert deflection operations in the CDFG so as to be able to break as many loops remaining in the DDCG. Every time a deflection operation is added, the corresponding data dependency edge in the DDCG is broken, in anticipation that

Design	Method	Bits	CS	EXU	Reg	Mux	Int
Continued Fraction IIR	Orig	20	7	1M,1S,1A	11	11	15
	BETS	20	7	1M,1S,1A	11	11	15
	HP+BETS	20	7	1M,1S,1A	10	11	15
Modem	Orig	16	10	3M,2A,1S	18	18	25
	BETS	16	10	3M,2A,1S	18	18	25
	HP+BETS	16	10	3M,2A,1S	17	17	23
5th Order Elliptical Wave Digital Filter	Orig	20	17	3M,3A	23	29	20
	BETS	20	17	3M, 3A	24	32	27
	HP+BETS	20	17	3M, 3A	24	32	27
5th Order IIR Cascade Filter	Orig	16	7	3M,1S,2A	14	14	24
	BETS	16	7	3M,1S,2A	14	11	13
	HP+BETS	16	7	3M,1S,2A	14	11	13

Table 1: Performance and Hardware Characteristics of the Designs

the input of the deflection operation will be scanned. If the resultant DDCG contains no loops of the form cd^+ or $(cd^+)(cd^+)^+$, we are guaranteed no loops will be formed during assignment. Else, addition of the deflection operations have minimized the chances of formation of loops during the scheduling and assignment phase.

add_deflection_ops_to_avoid_loops_during_assgn ()

```
1. create DDCG from the given CDFG;
2. remove data dependency edges from DDCG, corresponding to scan vars selected for CDFG loops;
3. delete compatibility edges having compatibility estimate comp(v,w) < threshold;
4. for every loop L of the form $cd^+$ or $(cd^+)(cd^+)^+$ in DDCG {
5.   attempt to add a deflection operation op_d on a d edge in CDFG, such that loop L in DDCG is broken;
6.   check if adding op_d is feasible, in terms of scheduling, assignment and RU cost;
7.   if deflection operation op_d is feasible {
8.     add $op_d$ in CDFG;
9.     delete the d edge in DDCG;
     }
   }
10. return transformed CDFG, with added deflection operations.
```

3.0 EXPERIMENTAL RESULTS

To evaluate the effectiveness of the hot potato transformations on the testability of the designs, we applied the technique on the following examples [Ell87, Mit72]: the 3rd order Continued Fraction IIR filter, the Modem filter, the 5th order Elliptical Wave Digital Filter, and the 5th order IIR Cascade Filter. We first synthesize the examples using the behavioral test synthesis system BETS [Dey93] for both testability and resource utilization. Next, we transform the examples by adding deflection operations to minimize partial scan cost, using the algorithms presented in this paper, and then apply BETS on the transformed CDFG. In the following tables, **Orig** refers to the design synthesized by conventional high level synthesis (BETS without testability considerations), **BETS** refers to the design synthesized by BETS, while **HP+BETS** refers to the design synthesized by BETS after the addition of deflection operations for testability.

Table 1 shows the physical characteristics of the designs. The bit width of the synthesized designs is indicated in the column **Bits**. The number of control steps (**CS**) needed by the implementation after adding the deflection operations remains same, ensuring that the performance of the designs is not compromised while improving testability. Similarly, the number of execution units (**EXU**) needed remains same. The columns **Reg, Mux, and Int** represent the number of registers, multiplexers, and interconnects, respectively. Note that while attempting not to compromise resource utilization, BETS may increase the hardware resources used while making a design testable with reduced partial scan cost. However, deflection operations can be added such that the number of resources, like registers and interconnects, is reduced simultaneously with the number of scan registers used.

Hence, in all the design cases, the hardware costs, like registers, multiplexers, and interconnects, are not compromised by the addition of deflection operations for testability.

Table 2 shows the number of Flip-Flops (**FFs**), and the number of scan FFs (**Scan FFs**) needed to break all the loops of the design for all the three implementations, Orig, BETS, and HP+BETS versions. In the case of the original implementation, a gate-level partial scan tool, OPUS [Chi91], is used for scan FF selection. For instance, in the case of the Modem filter, the original data path synthesized without testability considerations has 288 FFs, the data path synthesized by BETS from the original CDFG has 288 FFs, while the data path synthesized by BETS from the transformed CDFG, after addition of deflection operations, has 272 FFs. To avoid loops except self-loops in the data path, the gate-level partial scan tool OPUS needs to scan 64 FFs in the original data path, while the behavioral test synthesis system BETS needs 48 scan FFs. However, when BETS is given the transformed CDFG with deflection operations, it requires only 16 scan FFs.

Design	Method	FFs	Scan FFs
Continued Fraction IIR	Orig	220	80
	BETS	220	80
	HP+BETS	200	20
Modem	Orig	288	64
	BETS	288	48
	HP+BETS	272	16
5th Order Elliptical Wave Digital Filter	Orig	460	300
	BETS	480	60
	HP+BETS	480	40
5th Order IIR Cascade Filter	Orig	208	48
	BETS	208	32
	HP+BETS	208	16

Table 2: Effect of Adding Deflection Operations on Partial Scan Overhead

The testability of the synthesized designs was evaluated using the gate-level sequential ATPG tool, HITEC [Nie91], shown in Table 3. For each design, besides the rows **Orig**, **BETS**, and **HP+BETS**, the row **Orig+OPUS** refers to the data path obtained from the original data path after scanning the FFs selected by the gate-level partial scan tool OPUS [Chi91]. For each version of a design, the total number of FFs and the number of scan FFs used, are reported. The results of running HITEC on each design is also shown. The total number of faults, the number of faults aborted by HITEC, the fault coverage and test efficiency achieved, and the CPU time taken by HITEC on a SUN Sparcstation2 are reported.

Design	Method	Total FFs	Scan FFs	Total Faults	Aborted Faults	Fault Cov (%)	Test Eff. (%)	ATPG CPU (secs)
Continued Fraction IIR	Orig	220	0	6050	133	96	98	4019
	Orig+OPUS	220	80	6050	5	98	100	55
	BETS	220	80	6058	4	98	100	89
	HP+BETS	200	20	6052	21	98	100	167
Modem	Orig	288	0	8579	8299	0	3	>6hr
	Orig+OPUS	288	64	8879	7	96	100	153
	BETS	288	48	8648	21	96	100	235
	HP+BETS	272	16	8629	21	96	100	258
5th Order Elliptical Wave Digital Filter	Orig	460	0	10364	10077	1	3	>72hr
	Orig+OPUS	460	300	10364	0	98	100	309
	BETS	480	60	10916	16	98	100	233
	HP+BETS	480	40	11130	2	98	100	200
5th Order IIR Cascade Filter	Orig	208	0	8740	8488	0	3	20446
	Orig+OPUS	208	48	8740	7	97	100	128
	BETS	208	32	7531	23	97	100	869
	HP+BETS	208	16	7575	19	97	100	124

Table 3: Sequential ATPG using Hitec [Nie91]

As expected, though the sequential test pattern generator HITEC [Nie91] could not achieve 100% test efficiency on the original designs without scan FFs, 100% test efficiency could be obtained by HITEC on the final designs after scanning the selected FFs. For example, in the case of Modem filter, while the original design was not testable, for the Orig+OPUS, BETS and HP+BETS designs, HITEC could achieve 100% test efficiency. For the original design, gate-level scan selection needs 64 scan FFs to make the circuit 100% testable (row Orig+OPUS). For the BETS design synthesized from the original specification (row BETS), 48 scan FFs are needed to achieve 100% test efficiency. On the other hand, only 16 scan FFs are needed to achieve 100% test efficiency for the HP+BETS design, synthesized by BETS from the specification transformed by adding deflection operations (row HP+BETS). The results demonstrate the effectiveness of the hot potato techniques to significantly reduce the scan overhead needed by BETS to synthesize testable data paths.

4.0 CONCLUSION

We have introduced a new hot potato transformation technique based on the addition of deflection operations. We have shown the application of the technique to reduce the partial scan overhead for generating eas ily testable data paths. Experimental results on several benchmarks demonstrate the capability of the hot potato transformations to significantly improve the cost-effectiveness of an existing behavioral test synthesis system, BETS [Dey93].

ACKNOWLEDGMENT: The authors wish to thank Vijay Gangaram for his help in the project.

5.0 REFERENCES

[Avr91] L. Avra: "Allocation and Assignment in High-Level Synthesis for Self-Testable Data Paths," ITC, pp. 463-472, 1991.

[Avr93] L. Avra, E. McCluskey: "Synthesizing for Scan Dependence in Built-In Self-Testable Designs", ITC, pp. 734-743, 1993.

[Bar64] P. Baran: "On Distributed Communication Networks", IEEE Trans. on Comm., Vol. 12, No. 1, pp. 1-9, 1964.

[Che90] K.T. Cheng, V.D. Agrawal: "A Partial Scan Method for Sequential Circuits with Feedback", IEEE Trans. on Computers, Vol. 39., No. 4, pp. 544-548, 1990.

[Che92] C.-H. Chen, D.G. Saab: "Behavioral Synthesis for Testability", ICCAD, pp. 612-615, 1992.

[Chi91] V. Chickermane, J.H. Patel: "A Fault Oriented Partial Scan Design Approach", ICCAD, pp. 400-403, 1991.

[Cri91] C. Cring, A.R. Newton: "A Cell-Replicating Approach to Mincut-Based Circuits Partitioning", IEEE ICCAD-91, pp. 2-5, 1991.

[Dey93] S. Dey, M. Potkonjak, R. Roy: "Exploiting Hardware-Sharing in High Level Synthesis for Partial Scan Optimization", ICCAD, pp. 20-25, November 1993.

[Dey94] S. Dey, M. Potkonjak, R. Roy: "Synthesizing Designs with Low-Cardinality Minimum Feedback Vertex Set for Partial Scan Application",VLSI Test Symposium, pp. 2-7, April, 1994.

[Ell87] D.F. Elliott: "Handbook of Digital SIgnal Processing Engineering Applications", Academic Press, San Diego, CA, 1987.

[Fis88] C.N. Fischer, R.J. LeBlanc, Jr.: "Crafting a Compiler", The Benjamin/Cummings Publishing Co. Inc., Menlo Park, CA, 1988.

[Haj91] B. Hajek: "Bounds on Evacuation Time for Deflection Routing", Distributed Computing, Vol. 5, No. 1, pp. 1-5, 1991.

[Har93] H. Harmanani, C. Papachristou: "An Improved Method for RTL Synthesis with Testability Tradeoffs", ICCAD, pp. 30-35, Nov. 1993.

[Hwa92] J Hwang, A. El Gamal: "Optimal Replication for Min-Cut Partitioning", IEEE ICCAD-92, pp. 432-435.

[Lee90] D.H. Lee, S.M. Reddy: "On Determining Scan Flip-Flops in Partial-Scan Designs", ICCAD, pp. 322-325, 1990.

[Lee92] T.C. Lee, W.H. Wolf, N.K. Jha: "Behavioral Synthesis for Easy Testability using Data Path Scheduling", ICCAD, pp. 616-619, 1992.

[Lee93] T.C. Lee, N.K. Jha, W.H. Wolf: "Behavioral Synthesis of Highly Testable Data Paths under Non-Scan and partial Scan Environment", DAC-93, pp. 292-297, 1993.

[Maj92] A. Majumdar, K. Saluja, R. Jain: "Incorporating Testability Considerations in High-Level Synthesis", FTCS-92.

[McF92] M.C. McFarland, A.C. Parker, R. Camposano: "The High Level Synthesis of Digital Systems", Proc. of the IEEE, Vol. 78, No. 2, pp. 301-317, 1990.

[Mit72] S.K. Mitra, R.J. Sherwood: "Canonic realization of digital filters using the continued fraction expansion", IEEE Trans. on Audio Electroacoustics", Vol. 21, pp. 185-194, 1972.

[Nie91] T.M. Niermann, J. H. Patel: "HITEC: A Test Generation Package for Sequential Circuits", EDAC, pp. 214-218, 1991.

[Pap91] C. Papachristou, et al.: "SYNTEST: a method for high-level SYNthesis with self TESTability, ICCAD, pp. 458-462, 1991.

[Pat78] J.H. Patel: "Pipelines with Internal Buffers", Proc. 5th Symposium Computer Architecture, pp. 249-255, 1978.

[Pat89] D.A. Patterson, J.L. Henessy: "Computer architecture: a quantitative approach", San Mateo, Calif.: Morgan Kaufman Publishers, 1989.

[Rab91] J.M. Rabey, C. Chu, P. Hoang, M. Potkonjak: "Fast Prototyping of DataPath-Intensive Architectures", IEEE Design & Test of Computers, June 1991.

[Wal91] R. Walker, R. Camposano: "A Survey of High-Level Synthesis Systems", Kluwer, Boston, MA, 1991.