An Efficient Algorithm to Integrate Scheduling and Allocation in High-Level Test Synthesis *

Tianruo Yang and Zebo Peng
Department of Computer and Information Science
Linköping University, S-581 83, Linköping, Sweden

Abstract

This paper presents a high-level test synthesis algorithm for operation scheduling and data path allocation. Contrary to other works in which scheduling and allocation are performed independently, our approach integrates these two tasks by performing them simultaneously so that the effects of scheduling and allocation on testability are exploited more effectively. The approach is based on an algorithm which applies a sequence of semantics-preserving transformations to a design to generate an efficient RT level implementation from a VHDL behavioral specification. Experimental results show the advantages of the proposed algorithm.

1 Introduction

The main objective of this paper is to develop a systematic technique to integrate testability consideration into high-level synthesis and make it possible for an automatic synthesis tool to predict testability of the synthesized circuits accurately in the early stage and optimize the designs in terms of test cost as well as performance and area cost. Our high-level test synthesis system takes a VHDL behavioral specification of a digital system and a set of design constraints as input and generates a Register-Transfer Level (RTL) hardware implementation which consists of a data path and a controller. In this paper, we mainly deal with data path testability improvement, assuming that the controller can be modified to support the test plan.

There are several approaches to high-level test synthesis. Papachristou et al. [10] present a data path synthesis technique for self-testable design in which the built-in self-test (BIST) technique is employed. Two allocation techniques are presented that map a given scheduled dataflow graph into a self-testable data path. The first one is based on a graph-heuristic algorithm while the second one is based on integer linear programming formulation. Flottes et al. [2] have developed a high-level test synthesis system to generate testable data path in which as many modules (memory elements, functional units, muxes, and even wires) as possible will be testable using parallel test patterns propagated from primary data input ports. The goals of their system are to take advantage of synthesis possibilities in order to establish test paths of each module and to generate designs that are easily testable. Mujumdar et al. [8] present a technique to eliminate as many self-loops as possible by altering the register and module binding during high-level synthesis. All the above approaches and even Avra's technique on self-testable data path synthesis [1] are based on the assumption that the schedule of operation has been decided. Such an assumption limits the possibility of enhancing testability of the design. To address this problem, Papachristou [9] presents several rescheduling transformations to locally transform a given schedule to improve the testability of a data path when the BIST technique is used.

Recently in [6, 7], Lee et al. propose several new data path allocation scheduling methods for testability. Two heuristic rules are used to guide data path allocation and scheduling, which are: (1) whenever possible, allocate a register to at least one primary input or primary output variable, and (2) reduce the sequential depth from a controllable register to an observable register. They develop a scheduling called mobility path scheduling which takes these two rules into account. Once a schedule is obtained by the mobility path scheduling algorithm, a modified left-edge algorithm is used for register and module allocations. In this way, scheduling and module/register allocation are carried out in separate steps, and the possibility of testability enhancement can not be fully exploited.

Our high-level test synthesis approach differs from the prior work in which the scheduling and allocation tasks are performed independently. It introduces scheduling constraints imposed by data path allocation and performs scheduling and allocation simultaneously in an iterative fashion so that their effects on testabil-

*This work has been partially sponsored by the Swedish National Board for Industrial and Technical Development (NUTEK).

ity are exploited more effectively. A similar approach to integrate scheduling and allocation has been used by Kim [4] without testability enhancement.

The paper is organized as follows. Section 2 describes briefly the high-level test synthesis system with emphasis on the design representation and the testability analysis algorithm. The controllability/observability balance allocation technique for operation modules and registers is presented in section 3. The stepwise refinement synthesis algorithm and the introduction of scheduling constraints imposed by the process of data path synthesis is described in section 4 . Finally, section 5 describes some experimental results and section 6 contains our conclusions and remarks.

2 Preliminaries

The kernel of our high-level test synthesis system is an intermediate design representation, called Extended Timed Petri Net (ETPN), which can be used both for testability analysis and high-level synthesis. In ETPN, the structural properties of the data path and controller are explicitly captured in order to facilitate accurate analysis of the intermediate design in term of performance, area and testability.

The ETPN design representation consists of two parts: data path and control part. The data path is a directed graph with nodes and arcs. The node represents storage (registers) and manipulation of data. The arc connecting two nodes represents the flow of data. The control part, on the other hand, is captured as a timed Petri net with restricted transition firing rules [14]. These two parts are related through the control states in the control part controlling the data transfers in the data path, and the condition signals in the data path controlling some transitions in the control part.

Given a design represented in ETPN, its testability can be estimated accurately, assuming that it is going to be implemented directly. Our testability analysis is mainly based on Gu et al. [3]. The testability definition assumes that a stuck-at fault model is used and ATPG is random and/or deterministic. These assumptions are made based on the observation that the stuck-at fault model is the mostly used fault model and many ATPG's start by using random test generation to cover as many faults as possible and then switch to deterministic test generation.

The testability of a circuit is mainly measured by its controllability which reflects the cost of setting up any specific value on a line and its observability which, on the other hand, measures the cost of observing any specific value on a line. They are used to measure the fault sensitization and fault propagation costs respectively during test generation and application procedures. Further the controllability/observability is defined by a combinational factor which is used mainly to reflect the cost of generating a test and fault coverage, and a sequential factor which is used to measure the sequential complexity of using a sequential test generation algorithm and the cost (time and memory) of executing the test. Our testability metric consists of, therefore, four measures: combinational controllability (CC), sequential controllability (SC), combinational observability (CO) and sequential observability (SO). The testability metrics (CC, SC, CO, SO) can be obtained by the testability analysis algorithm given in [3]. It assigns first ones to CCs and zeros to SCs for all primary inputs in the data path of the ETPN. These values will then be propagated according to the algorithm proposed in [3] until the primary outputs are reached. A similar approach can be used for calculating observability in the reverse direction.

3 Data Path Allocation Principle

Our approach uses a controllability/observability balance allocation [13] technique to perform the date path allocation task. We begin with a default allocation generated by the VHDL compiler which assumes that each operation instance in the VHDL specification is mapped into an individual data path node. Then re-allocation is mainly carried out by *merger* transformation which compacts the data path nodes until a one-by-one mapping to physical hardware is feasible.

Conventional allocation approaches often select and merge the data path nodes according to their connectivity or closeness, which aims to minimize interconnections and multiplexors. This usually results in a very hard to test design because many loops, especially self-loops, are generated. Further, nodes with good controllability and bad observability are merged together since they are very close to the primary inputs. Similar merger for nodes with good observability and bad controllability will also occur. As a result, the data path consists of many nodes which are very difficult either to control or observe.

In our approach, the selection of nodes to be merged is based on the testability measures generated by the testability analysis algorithm. The main goal is to generate a data path with good controllability and observability for all the nodes and with as few loops as possible. The basic idea is to fold nodes with good controllability and bad observability to nodes with good observability and bad controllability. Note that the controllability of a node is defined as the best controllability of any of its input lines. While the observability of a node is the best observability of any of its output

lines. In this way, the new node will inherit the good controllability from one of the old nodes and the good observability from the other.

4 The Test Synthesis Algorithm

4.1 The basic algorithm

Our synthesis algorithm accepts an unscheduled ETPN design representation as input and generates a highly testable data path. It iteratively applies transformation to the current ETPN. In each iteration, the algorithm selects pairs of modules and registers in the data path and merges them to generate a new data path. Here merging two modules into one implies that all the operations which were assigned to the two modules are reassigned into one single module. Therefore, merging two modules into one imposes a scheduling constraint that all the operations which were assigned to these modules must be scheduled in different control step. A similar argument holds for the case of register merger. Merging two registers into one imposes a scheduling constraint that the lifetime of all variables which were assigned in these registers must be disjoint. Note that merger of nodes (modules and registers) leads to a reduction in the number of hardware components in the design, whereas the additional scheduling constraints may lead to an increase in the execution time of the data path. The iterative synthesis algorithm is described as Algorithm 1.

Algorithm 1 The Synthesis Algorithm

1: Perform a simple default scheduling/allocation
2: **repeat**
3: **for** all modules and registers **do**
4: Run the testability analysis algorithm
5: **end for**
6: Select k pairs of mergable nodes according to the controllability/observability balance principle
7: **for** k pairs modules or registers **do**
8: Estimate the incremental execution time cost ΔE
9: Estimate the incremental hardware cost ΔH
10: **end for**
11: Select the pair with smallest $\Delta C = \alpha \cdot \Delta E + \beta \cdot \Delta H$
12: Merge the selected pair and modify the data path
13: Do lifetime analysis of variables
14: Perform rescheduling imposed by data path synthesis using a merge-sort algorithm based on a controllability and observability enhancement strategy
15: **until** no merger exists

In each iteration, our algorithm runs the testability analysis algorithm to select k pairs modules and registers according to the controllability/observability balance allocation principle. Here k is a number chosen by the user which is used to control the trade-offs between the testability and execution time and hardware cost. A small value of k means that more emphasis is placed on improving the testability measure. For each of k pairs of modules and registers, we will estimate the incremental execution time cost ΔE and the incremental hardware cost ΔH. Then we choose the pair with the smallest value of $\Delta C = \alpha \Delta E + \beta \Delta H$, where α and β are two user-controlled parameters. We will present the details of estimation the execution time and hardware cost later. The selected pair is merged and the data path is modified accordingly. Rescheduling if needed is performed by a merge-sort algorithm based on a controllability and observability enhancement strategy, to be presented later, after we finish the lifetime analysis of variables. This process is then repeated until no merger is possible in the data path.

4.2 Estimation of performance and cost

For a given data path, the minimum execution time E is equal to the length of the critical path which consists of a sequence of control places which dominating the time needed for a token to flow from the initial place to the final place. The method to detect the critical path is based on the reachability tree of the Petri net model [15]. A reachability tree represents the reachability set of the given Petri net. That is, it shows all markings which can be reached from the initial markings. The critical path algorithm first constructs such a reachability tree and, analyzing the reachability tree, extracts the critical path.

The hardware cost H is based on calculation of the cost of the data path. The cost of data path is calculated according to the cost of each data path unit. The cost of data path units which performs logic, arithmetic, or storage operations is given by the corresponding module parameters stored in the module library. The cost of data path units for communication such as arcs and bus vertices depends on the placement of the components as well. To make a more accurate estimation, we follow the floorplanning algorithm proposed by Peng et al. [14] to estimate the hardware cost which takes into account the geometrical information. This algorithm basically makes use of a simple heuristics based on the connectivity between the data path vertices. These heuristics provide a relatively precise indication of how much it will cost in hardware implementation.

Given a floorplan, the cost for an ETPN data path

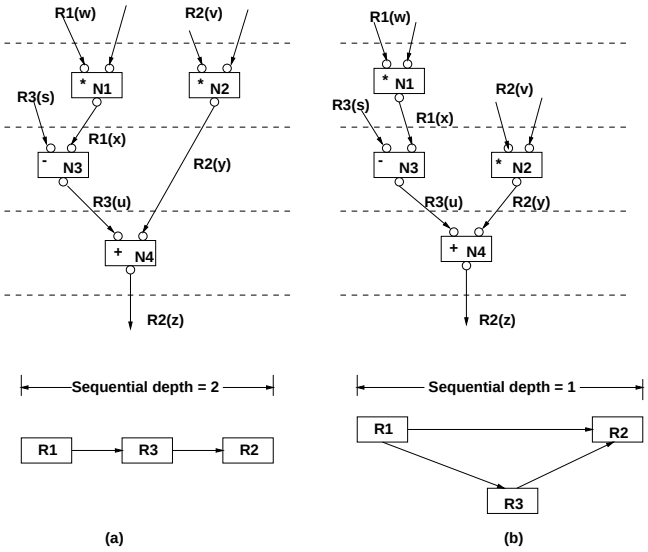


Figure 1: An example of controllability/observability enhancement strategy

is estimated as follows:

$$H = \sum_i Area(V_i) + \sum_j Len(A_j) \times Wid(A_j),$$

where

- $Area(V_i)$ is the area cost of the module corresponding to a data path node V_i.
- $Len(A_j)$ is the length of the connection represented as a data path connection A_j.
- $Wid(A_j)$ is the width of the connection represented as A_j, which is the bit width of the connection multiplied by a given weighted factor.

Based on the above estimations, when a pair of modules or registers is merged, ΔE is equal to the increase in the critical path length and correspondingly ΔH is equal to the increase in data path area cost.

4.3 Operation scheduling

As pointed out before, when two modules are merged, the operations executed on these modules must be scheduled in different control steps so that they can share the same component. Similar for registers, the variables stored in these registers must be disjoint. We will present the rescheduling transformation which is performed by a merge-sort algorithm based on a controllability/observability enhancement strategy. These transformations change locally the execution orders of some operations in the current schedule in order to improve the testability and satisfy the scheduling constraints imposed by the merger.

4.3.1 Rescheduling by module merger

Suppose we would like to merge two modules m_i and m_j. Assume the s operations scheduled for execution in module m_i are $o_{i_1}, o_{i_2}, \cdots, o_{i_s}$ and t operations scheduled for execution in module m_j are $o_{j_1}, o_{j_2}, \cdots, o_{j_t}$. Let us first consider the simplest case where $s = t = 1$, then operations o_{i_1} and o_{j_1} must be scheduled in different control steps for execution. If they have been scheduled already in different control steps, we do not need to introduce any scheduling constraints. Otherwise, we have to consider two possibilities: (1) execute o_{i_1} before o_{j_1}, or (2) execute o_{j_1} before o_{i_1}. To decide the order, we can use the controllability and observability enhancement strategy suggested by Lee et al. [6]. They propose an effective allocation rule for good testability:

- SR1: reduce the sequential depth from a controllable register to an observable register.

We will arrange the order of o_{i_1} and o_{j_1}, to support the application of SR1 using the following rule:

- SR2: schedule operations to support the application of SR1.

These two heuristic rules are called controllability/observability enhancement strategy. If these two rules can not be applied, we will select the pair which results in the smallest increase in the length of the critical path of data path. Rescheduling is performed by introducing dummy control steps (places in Petri net) [14] if necessary so as to change the default scheduling of operations.

In Figure 1, we give an example to illustrate these heuristic rules. Suppose register $R1$ is used for variables w and x, $R2$ for variables v, y and z, and $R3$ for s and u as described in Figure 1(a). Also suppose we want to merge operation nodes N1 and N2. Since they are in the same control step, we have to introduce a scheduling constraint. If we take the execution order N1 before N2 as described in Figure 1(b), where we reschedule operation node N2 to the next step. The sequential depth from register R1 to R2 is reduced from 2 to 1 by sharing nodes N1 and N2.

For the general case in which operations $o_{i_1}, o_{i_2}, \cdots, o_{i_s}$ are scheduled for execution in module m_i and $o_{j_1}, o_{j_2}, \cdots, o_{j_t}$ for execution in module m_j, respectively, we know that there is a sequential order among $o_{i_1}, o_{i_2}, \cdots, o_{i_s}$ because they already share the module m_i. Without loss of generality, we assume the order to be $o_{i_1} \rightarrow o_{i_2} \rightarrow \cdots \rightarrow o_{i_s}$. Similarly, let the sequential order for operations in module m_j to be $o_{j_1} \rightarrow o_{j_2} \rightarrow \cdots \rightarrow o_{j_t}$. The main goal is to merge these two sequential orders into one. First we examine the pair of operations o_{i_1} and o_{j_1} using the controllability/observability enhancement strategy to decide the most suitable order for testability. Then we decide the rest using a merge-sort heuristic to achieve a single execution order for testability.

4.3.2 Rescheduling by register merger

With regards to the scheduling constraints imposed by a register merger, we can use a similar merge-sort algorithm helped with the controllability and observability enhancement strategy. For this case, we suppose we want to merge two registers r_i and r_j. Let $v_{i_1}, v_{i_2}, \cdots, v_{i_s}$ denote the s variables which are stored in register r_i and $v_{j_1}, v_{j_2}, \cdots, v_{j_t}$ denote the t variables which are stored in register r_j. Let us also first consider the simplest case where $s = t = 1$, then the operations which determine the lifetimes of variables v_{i_1} and v_{j_1} must be scheduled in a way that the lifetimes of v_{i_1} and v_{j_1} do not overlap.

First we should examine whether some operations, which determine the lifetime of v_{i_1} and v_{j_1}, have been scheduled to be executed in a certain order. Based on our observations, it is very natural to check first whether these operations are scheduled such that two lifetimes are never disjoint. There are two cases: (1) there are two arcs, one is from some of the operations that determine the lifetime of v_{i_1} to the operations that determine the lifetime of v_{j_1}, the other is from some of the operations that determine the lifetime of v_{j_1} to the operations that determine the lifetime of v_{i_1}. or (2) there is an operation which uses both of the value of v_{i_1} and v_{j_1} as inputs.

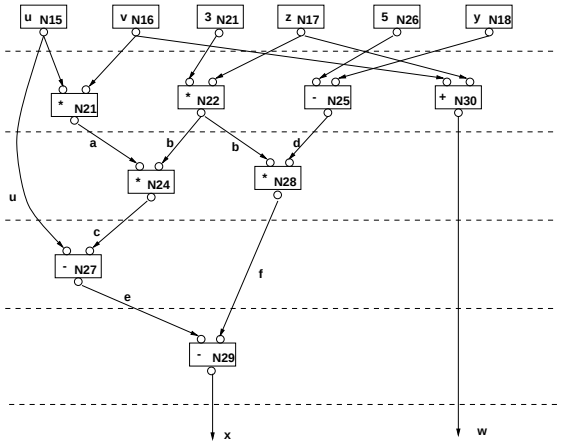


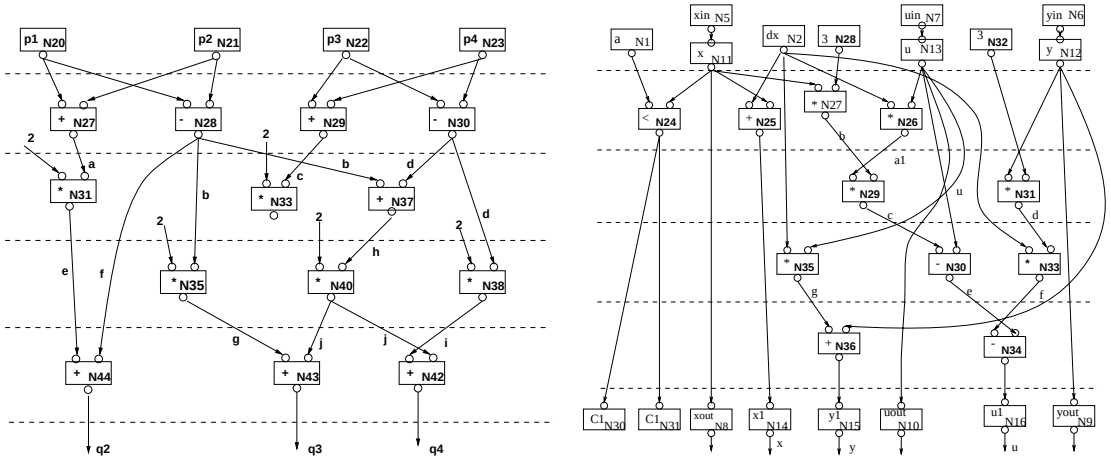
Figure 2: The schedule for the Ex benchmark

If both cases do not hold, a scheduling constraint to ensure that the lifetime of v_{i_1} and v_{j_1} will be disjoint is added to the representations. If there are no arcs between any of the operations which determine the lifetime of v_{i_1} and any of the operations which determine the lifetime of v_{j_1}, we consider two possibilities: (1) variable v_{i_1} expire before variable v_{j_1} is created. (2) variable v_{j_1} expire before variable v_{i_1} is created. Our algorithm selects, between the two possibilities, the one which results in a shorter sequential depth from a controllable register to an observable register based on the controllability and observability enhancement strategy described above.

For the general case in which variables $v_{i_1}, v_{i_2}, \cdots, v_{i_s}$ and $v_{j_1}, v_{j_2}, \cdots, v_{j_t}$ share registers r_i and r_j, respectively, the procedure for adding lifetime disjoint arcs is similar to that of the case of module merger.

5 Experimental Results

We have tested our synthesis algorithm manually on Diffeq [12], DCT [5], Ex [6, 7], EWF [6, 7], Paulin [12] and Tseng [16] benchmarks with different parameters k, α, β carried out on MentorGraphics. We also have compared the experimental results concerning control steps, module and register allocation, and numbers of multiplexers with the schedule and allocation schemes produced by other approaches. The selected results (due to the space limitation) are summarized in Table 1, 2 and 3, where the rows denoted Approach 2 are results produced by Lee's algorithm [6, 7]. The rows denoted Approach 1 present results given by the force-directed scheduling (FDS) [11] without testability consideration followed by the same allocation algorithm as in Approach 2 [7]. The last results are generated by



(a) The Dct benchmark

(b) The Diffeq benchmark

Figure 3: The schedules for Dct and Diffeq benchmarks

the CAMAD high-level synthesis system [14] without testability consideration. The corresponding fault coverage, test generation time, test generated cycle for 4, 8, 16-bit implementations and area (hardware cost) are compared. Our main experimental results are based on the chosen parameters which achieve the same allocation and scheduling where (k, α, β) are equal to $(3, 2, 1)$, $(3, 10, 1)$ and $(3, 1, 10)$ respectively for 4, 8, 16-bit implementations. From the following experimental descriptions, it seems that the chosen parameters do not influence so much the final results.

The first area-optimized benchmark, denoted as Ex, shows the effect of scheduling and allocation on testability in term of fault coverage, test generation time, and test generated cycle in Table 1. Figure 2 shows the schedule after our synthesis algorithm for this benchmark. In Figure 2, since operation node pairs for example, (N21, N24), (N22, N28), and (N25, N27, N29), can share the same ALU and are scheduled in different control steps, they can be merged to a common functional module respectively. Correspondingly those variable pairs such as, (a, c, x), (b, f, v), (d, e, z) and (y, w) can also share the same register respectively. We can see that for the 4, 8, 16-bit implementations, our synthesis algorithm can result in Ex benchmark the highest fault coverage with smaller total test generation time. In addition, an interesting observation is that our synthesis approach requires fewest registers as Approach 2.

The second selected example, denoted as Dct, is taken from a portion of an 8-point DCT signal flow graph. The experimental results on the area-optimized benchmark compared with other approaches in term of fault coverage, test generation time, area (hardware cost), and test generated cycle are given fully for 4, 8, 16-bit implementations in Table 2. Figure 3(a) shows the schedule after our synthesis algorithm for this benchmark. In Figure 3(b), since the operation nodes in the groups, (N31, N40), (N33, N38), (N27, N44), and (N29, N37, N43), can share the same ALU and are scheduled in different control steps, they can be grouped into a common functional module respectively. Similarly the variables in the groups, (a, j, q2), (c, h, q3), (f, p1), (e, p2), (b, i, p3) and (d, g, p4, g4) can also share the same register respectively. We also can observe that the schedule and allocation produced by our synthesis algorithm achieve better fault coverage and area (hardware cost) with shorter test generation time than other approaches listed in Table 2.

The third selected example is the benchmark Diffeq. Figure 3(b) depicts its schedule determined by our proposed synthesis algorithm. Table 3 also shows the experimental results for the area-optimized benchmark concerning fault coverage, test generation time, area (hardware cost) and test generated cycle where the bit width of the data path are 4, 8 and 16 bits respectively. In Figure 3(a), since the operation nodes in the groups, (N26, N31, N35), (N27, N29, N33), (N25, N36), and (N30, N34), can share the same ALU and are scheduled in different control steps, they can be grouped into a common functional module respectively. Similarly the variables of the groups, (u, u1, e), (x, a1, d, g), and (y1, b, c, f), can also share the same register respectively. With the same module allocation and scheduling but different register allocation,

Table 1: Experimental results on the area-optimized Ex benchmark

Different Synthesis	Module allocation	Register allocation	#Mux	#Bit	Fault coverage	Test generation time	Test generated cycle
CAMAD	(*): N21, N22 : N24, N28 (±): N25, N27 : N29, N30	R: a, R : b R: c, R : d R: e, R : f R: u, R : v R: w, R : x R: y, R : z	4	4	81.27%	27	1081
				8	89.89%	81	912
				16	93.74%	279	691
Approach 1	(*): N21, N24 (*): N22, N28 (-): N25, N27, N29 (+): N30	R: d, f, x R: b, y R: v R: a, c, e, w R: u R: z	10	4	86.41%	24	707
				8	90.87%	74	943
				16	92.58%	191	1070
Approach 2	(*): N21, N24 (*): N22, N28 (-): N25, N27, N29 (+): N30	R: y, d, f, x R: w R: u R: a, c, e, v R: b, z	10	4	88.19%	11	824
				8	92.49%	37	1654
				16	93.91%	115	1054
Ours	(*): N21, N24 (*): N22, N28 (-): N25, N27, N29 (+): N30	R: a, c, x R: u R: b, f,v R: d, e, z R: y, w	10	4	90.66%	13	366
				8	94.48%	43	1383
				16	96.11%	112	1122

our synthesis algorithm produces the design not only with fewer registers and area (hardware cost) but also with better testability properties such as fault coverage and shorter test generation time.

6 Conclusion

In this paper, we present a high-level test synthesis system which carries out the scheduling and allocation tasks in an integrated fashion. A data path allocation, namely controllability/observability balance allocation technique which is based on testability analysis at register-transfer level is proposed. Contrary to other works in which the scheduling and allocation tasks are performed independently, our approach introduces scheduling constraints imposed by the data path allocation and performs them simultaneously so that the effects of scheduling and allocation on testability are exploited more effectively. Experimental results evaluating our algorithm on benchmarks are given and indicate that our algorithm produces good designs with respect to testability and hardware costs.

References

1. L. Avra. Allocation and assignment in high level synthesis for self-testable data paths. In *Proceedings of the International Test Conference*, pages 463–472, 1991.
2. M. L. Flottes and B. Rouzeyre. Testability driven synthesis of non-scan data pths. In *IEEE European Test Workshop*, pages 136–142, Montpellier, France, 1996.
3. X. Gu, K. Kuchcinski, and Z. Peng. Testability analysis and improvement from VHDL behavioral specifications. In *Proceedings of the European Design Automation Conference with EURO-VHDL*, 1994.
4. T. Kim. *Scheduling and allocation problems in high level synthesis.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1993.
5. G. Krishnamoorthy and J. A. Nestor. Data path allocation using an extended binding model. In *Proceedings of the Design Automation Conference*, pages 279–284, June 1992.
6. T. C. Lee, W. H. Wolf, and N. K. Jha. Behavioral synthesis for easy testability in data path scheduling. In *Proceedings of the International Conference on Computer-Aided Design*, pages 616–619, 1992.
7. T. C. Lee, W. H. Wolf, N. K. Jha, and J. M. Acken. Behavioral synthesis for easy testability in data path allocation. In *Proceedings of the International Conference on Computer Design*, pages 29–32, 1992.
8. A. Mujumdar, R. Jain, and K. Saluja. Incorporating testability considerations in high level synthesis. *Journal of Electronic Testing: Theory and Applications*, 5:43–55, 1992.
9. C. A. Papachristou. Rescheduling transformation for high level synthesis. In *Proceedings of the International Symposium on Circuits and Systems*, pages 766–769, 1989.
10. C. A. Papachristou, S. Chiu, and H. Harmanani. A data path synthesis method for self-testable designs. In *Proceedings of 28th Design Automation Conference*, pages 378–384, 1991.
11. P. G. Paulin and J. P. Knight. Forced-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8:661–678, June 1989.
12. P. G. Paulin, J. P. Knight, and E. F. Girczyc. HAL: a multi-paradigm approach to automatic data path synthesis. In *Proceedings of Design Automation Conference*, pages 263–270, June 1986.
13. Z. Peng. High-level test synthesis using design transformations. The 2nd International Test Synthesis Workshop, 1995. Santa Barbara.
14. Z. Peng and K. Kuchcinski. Automated transformation of algorithms into register-transfer level implementations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 150–166, 1994.
15. J. Peterson. *Petri Net Theory and the Modeling of System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
16. C. Tseng and D. P. Siewiorek. Automated synthesis of data path in digital systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 5:379–395, July 1986.

Table 2: Experimental results on the area-optimized Dct benchmark

Different Synthesis	Module allocation	Register allocation	#Mux	#Bit	Fault coverage	Test generation time	Test generated cycle	Area
CAMAD	(*): N31, N33 : N35, N38 : N40 (±): N27, N28 : N29, N30 : N37, N42 : N43, N44	R: a, R : b R: c, R : d R: e, R : f R: g, R : h R: i, R : j R: p1, R : p2 R: p3, R : p4 R: q2, R : q3 R: q4,	4	4	70.44%	49	846	$0.607mm^2$
				8	81.60%	121	841	$1.488mm^2$
				16	85.00%	785	604	$3.320mm^2$
Approach 1	(*): N31, N35 (*): N33, N38 (*): N40 (+): N27, N37 : N44 (+): N29, N43 (+): N42 (-): N28 (-): N30	R: c, f, p1, q3 R: p2 R: d, h, j, q2 R: g, p3 R: a, b, e, q4 R: i, p4	14	4	88.96%	32	552	$0.592mm^2$
				8	95.15%	52	2902	$1.388mm^2$
				16	94.73%	286	10283	$2.634mm^2$
Approach 2	(*): N31, N35 (*): N33, N38 (*): N40 (+): N27, N37 : N44 (+): N29, N43 (+): N42 (-): N28 (-): N30	R: a, e, p1, q4 R: b,h,j,p2,q2 R: d, i, p4 R: g, p3 R: c, f, q3	14	4	91.73%	16	602	$0.575mm^2$
				8	93.36%	110	1088	$1.363mm^2$
				16	96.11%	177	8149	$2.584mm^2$
Ours	(*): N31, N40 (*): N33, N38 (*): N35 (+): N27, N44 (+): N29, N37 : N43 (+): N42 (-): N28 (-): N30	R: a, j, q2 R: c, h, q3 R: f, p1 R: e, p2 R: b, i, p3 R: d, g, p4, q4	14	4	93.13%	16	802	$0.571mm^2$
				8	96.01%	47	2278	$1.336mm^2$
				16	96.99%	118	6753	$2.531mm^2$

Table 3: Experimental results on the area-optimized Diffeq benchmark

Different Synthesis	Module allocation	Register allocation	#Mux	#Bit	Fault coverage	Test generation time	Test generated cycle	Area
CAMAD	(*): N26, N27 : N29, N31 : N33, N35 (±): N25, N30 : N34, N36 (<): N24	R: dx, R : x R: y, R : u R: u1, R : a1 R: e, R : g R: x1, R : y1 R: b, R : c R: f, R : d	7	4	72.40%	143	304	$0.573mm^2$
				8	87.15%	311	2321	$1.366mm^2$
				16	88.40%	2091	1827	$3.064mm^2$
Approach 1	(*): N26, N31 : N35 (*): N27, N29 : N33 (+): N25, N36 (-): N30, N34 (<): N24	R: y1, b, d, g R: x, x1 R: dx R: u1, a1, c, f R: u, e R: y	13	4	90.51%	9	350	$0.559mm^2$
				8	92.79%	49	959	$1.161mm^2$
				16	94.11%	162	676	$2.124mm^2$
Approach 2	(*): N26, N31 : N35 (*): N27, N29 : N33 (+): N25, N36 (-): N30, N34 (<): N24	R: x, y1, b,d,g R: x1 R: dx R: u1, a1, c, f R: u, e R: y	12	4	91.11%	15	504	$0.521mm^2$
				8	95.56%	55	920	$1.112mm^2$
				16	94.64%	164	1546	$2.150mm^2$
Ours	(*): N26, N31 : N35 (*): N27, N29 : N33 (+): N25, N36 (-) : N30, N34 (<): N24	R: u, u1, e R: x, a1, d, g R: y R: y1, b, c, f R: x1	12	4	95.28%	11	510	$0.470mm^2$
				8	97.31%	46	982	$1.054mm^2$
				16	99.79%	141	1663	$2.045mm^2$