An Improved Method for RTL Synthesis with Testability Tradeoffs[†]

Haidar Harmanani and Christos A. Papachristou
Department of Computer Engineering
Case Western Reserve University
Cleveland, Ohio 44106

Abstract

A method for high-level synthesis with testability is presented with the objective to generate self-testable RTL datapath structures. We base our approach on a new improved testability model that generates various testable design styles while reducing the circuit sequential depth from controllable to observable registers. We follow the allocation method with an automatic test point selection algorithm and with an interactive tradeoff scheme which trades design area and delay with test quality. The method has been implemented and design comparisons are reported.

1 Introduction

The complexity of VLSI circuitry has complicated the design and test process. It has been generally recognized that in order to have a good design quality there should be tight coordination between both processes. The field of high-level synthesis has made significant progress in addressing the needs of synthesis methods at the register-transfer level (RTL) [5, 13, 8, 16]. However, most high-level synthesis research has neglected the investigation of synthesis methods with test considerations.

Recently, a new trend in high-level synthesis has emerged, with researchers being aware of the importance of testability at the system level. Abadir and Breuer [1] presented an expert system (TDES) for designing testable circuits by modifying or adding new structures. TDES was used as a back-end tool in the ADAM system [8]. Gebotys [6] suggested an allocation method with a back-end module to convert the resulting RTL structure into a testable one for the scan methodology. The disadvantage of the above approaches is that they perform post-synthesis test embedding, which may reveal serious problems too late — problems with area and delay that the system could have avoided if testability were considered at a higher level. Using the Built-In-Self-Testing (BIST) methodology, these authors [17] first proposed a combined register and ALU allocation method generating self-testable designs without self-loops. Also, Avra [11] proposed a register allocation method generating self-testable designs using BIST, aiming at minimizing the number of self-adjacent registers in the datapath. This method assumes that the ALU binding is given; thus, does not allow much coordination between the register allocator and the module and multiplexer binder. Majumdar et al. [14] proposed an allocation method for testable designs based on the scan methodology. Lee et al. [12] suggested a method for "easy testability" that reduces the sequential depth and allocates test registers to primary inputs or outputs.

1.1 Problem Description and Significance

We present in this paper an improved model and method to solve the following problem: given a behavioral level description of a circuit represented in the form of a scheduled data flow graph (DFG), a technology library and a set of constraints, generate a self-testable RTL datapath structure such that: 1) the datapath conforms to all the user constraints; 2) the overhead of test registers in the datapath is minimized[1]. What distinguishes our approach from other high-level synthesis systems is the consideration of testing cost in addition to the conventional constraints during the design process. The approach is based on a *synthesis for testability* model that addresses the shortcomings of previous works, in particular [11, 17]. The ultimate goal is to *explore the tradeoffs that exist between the design and test processes.* Our method has the following contributions:

- A new improved model for the testable synthesis of RTL datapath structures driven by a technology library and based on the BIST methodology.
- The flexibility during the allocation phase to handle various testability design styles.
- A test point selection scheme which provides the capability to tradeoff design area and delay with test quality (fault coverage).

In section 2 we describe the test methodology and model. Section 3 discusses the testable allocator, while section 4 describes the test point selection. Section 5 illustrates the test tradeoff scheme by an example. Results are in section 6.

2 Design and Test Methodology

2.1 Background

Design for Testability (DFT) has two main aspects, controllability and observability; the control and observation of a circuit are central to implementing its test procedures. We base our method on pseudorandom BIST where the test patterns are generated using *Test Pattern Generation Registers* (TPGRs), based on autonomous Linear Feedback Shift Register (LFSR) design. The test responses are evaluated using *Multiple Input Signature Registers* (MISR) [2]. We determine the fault coverage using *logic level* fault simulation and select the test length (number of test patterns) so as to achieve an acceptable level of fault coverage.

One of the difficulties in implementing BIST techniques is the register self-adjacency problem. A register is self-adjacent if an output of that register feeds through combinational logic and back into itself. This situation is depicted in Figure 1(a). In this circuit, a Built-In Logic Block Observer (BILBO) is a register that, in addition to its normal operation mode, operates during test mode as an MISR or as a TPGR [9]. However, it is not possible to assign the output register as both a pattern generator and a signature analyzer, at the same time [7]. This problem can be rectified using a *concurrent built-in logic-block observation* (CBILBO) register [2] which can operate simultaneously as an MISR and a TPGR. The disadvantage of the CBILBO is that it is very costly in area (about 1.75 times the size

[†]This work is supported in part by the Semiconductor Research Corporation (SRC) under contract 92-DJ-147.

[1]Test overhead refers here to the additional area required to convert *normal* registers to test, i.e. *BIST* registers.

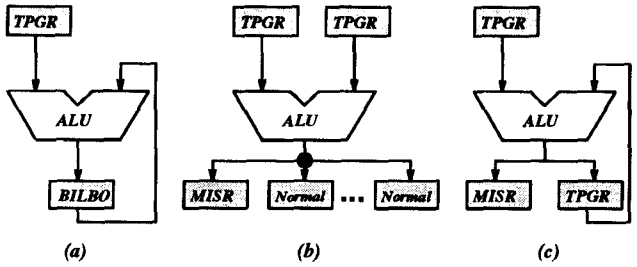

Figure 1: (a) Non-Observable ALU due to Self-Adjacency, (b) Extended Testable Functional Block, (c) Self Testable ALU with Self-Adjacent register

of a BILBO [11]) and induces more delay during normal operation mode.

2.2 Improved Model for Synthesis with Testability

We base our allocation model on the notion of *structural testability* at the RT level. The key element of structural testability is the Testable Functional Block (TFB) that we introduced in [17]. The DFG operations are mapped to the ALU of the TFB, while the variables are mapped to the register at the TFB output. The disadvantage of this model is the inability to map operations whose variables life spans overlap to the same TFB; thus, the final design may use more TFBs than necessary.

We now introduce a new *improved model* for structural testability based on the notion of Extended Testable Functional Blocks (XTFBs), shown in Figure 1(b). An XTFB consists of an ALU and a set of input and output registers. The two registers at the input ports are configured as TPGRs during test mode. The output port of an XTFB is connected to a set of registers, one of which is configured as an MISR in test mode. The number of registers at the XTFB output port is optimized during the synthesis process. Although a basic XTFB contains only three BIST registers, namely two TPGRs and one MISR, it should be noted that these BIST registers may be shared by other XTFBs in the datapath. This means that some of these registers may have to be configured as BILBO registers. Furthermore, some other non-BIST registers at the XTFB output port may have to be configured as TPGRs *only* if they control the input port of other XTFBs.

The main advantage of the model is that it avoids the dislocation of variable instances from their corresponding operations, keeping them close together during the entire allocation process. This proximity is preserved topologically in the generated data path and it contributes to its structural testability. Conflicting variables are mapped to different registers at the output layer of the XTFB. The resulting datapath is guaranteed to be self-testable by *construction*.

Another advantage of our method is its ability to control the *sequential depth* in the circuit, which we define informally as the maximum depth from a controllable to an observable register for a given module.[2] Since we always observe the faults at the output port of the ALU, we guarantee a sequential depth of *one* from the TPGRs to the MISR. However, this depth can be relaxed by the system, or interactively by the designer, using the notion of functional bypassing to perform design and test tradeoffs (sections 4 and 5).

[2]Note that this definition is different from the one in [12, 3] who are not using the BIST methodology.

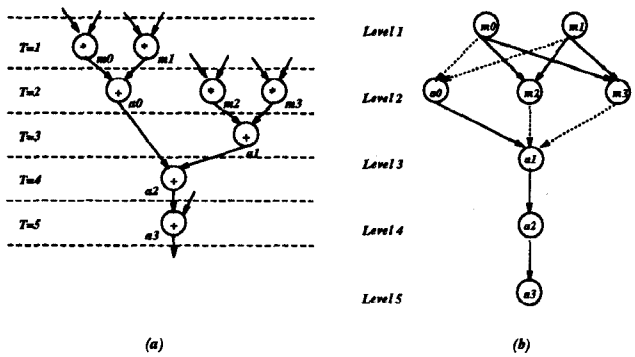

Figure 2: Biquad: (a) DFG, (b) Module Allocation graph (dotted edges correspond to incompatibilities due to library considerations).

3 Testable Datapath Allocation

Our testable datapath allocation starts with a scheduled DFG, generated by any scheduling method reported in the literature. Several key elements characterize our allocation technique: 1) A levelized compatibility graph; 2) a technology library of components; 3) a cost function. In what follows, we describe our method in reference to the simple biquad filter DFG, shown in Figure 2(a).

3.1 Requirement Analysis

Based on our model, two XTFBs are compatible if there is no resource conflict between the *operations* of the DFG nodes; that is, the nodes are assigned to different time steps in the schedule. However, in addition to the above restriction, we impose two further merging constraints expressed in the following two rules:

- *Rule 1:* Do not merge XTFBs if such merging will result in a module which does not exist in the technology library. For example, if the resulting ALU has an operation set such as {+,*,/} which is not in the library, such merging can not take place from the technology library standpoint.
- *Rule 2:* Allow self-adjacent registers *only* if the resulting structure is testable. We do not generate designs such as the ones depicted in Fig. 1(a), while designs as in Fig. 1(c) are testable and thus acceptable.

Rule 2 in our model is responsible for the structural testability of our RTL datapath designs.

3.2 System Library

Minimizing the number of components is not sufficient to guarantee a good design since some components may be more expensive than others under a given technology. We use a *technology library* as an input to our allocation method. The technology library provides information about the layout area of every individual component (width height and break points) along with the delay propagation. The library components, based on an NCR [15] library, are parametrized by their bit length and some may come in different implementation styles, for example, a slow but area efficient multiplier versus a fast but area expensive multiplier. Furthermore, each component is associated with a set of test metrics which we obtained by fault simulation. These include the fault coverage of individual ALUs along with the module *randomness* and *transparency*, detailed in [4].

3.3 Module Allocation Graph

In order to illustrate the compatibility relations among the DFG nodes, we use a special graph, *the module allocation*

graph (MAG). This is a levelized directed acyclic graph with nodes and levels corresponding to the ones of the scheduled DFG, respectively. An edge from node A to node B of the MAG indicates that these nodes are compatible; however, node A is scheduled before node B. A given path in the MAG corresponds to a list of XTFBs that can share resources.

The motivation for the MAG is twofold. First, by having directed edges, a top-bottom optimization process is possible by considering two levels at a time, pruning the number of paths in the MAG. Second, we associate a *local cost (gain)* function with every path in the MAG. The cost function guides the algorithm in order to select the best merging possibility at every iteration. The MAG is technology dependent and driven by the system library (Rule 1). Thus, for different libraries, we have different MAGs, and consequently different bindings.

The construction of the MAG is quite simple: we assign each node in the MAG to the level at which it was scheduled. We add directed edges between the compatible nodes using a top-bottom fashion by considering two levels at a time, k and $k+1$. The reason, as stated above, is to prune the number of paths in the MAG and thus reduce the number of edges in the graph. For example, in Figure 2(b), path $m0\ a1$ is redundant since it is covered by path $m0\ a0\ a1$. Edges are added to the MAG as long as they do not cause resource conflicts. The module allocation graph for the DFG in Figure 2(a) is shown in Figure 2(b) with dotted edges corresponding to incompatibilities due to library considerations.

3.4 Resources Allocation with Testability Consideration

The ultimate goal of the allocation phase is to optimize the cost of the datapath. We accomplish this by merging operations, variables and interconnects, *simultaneously*. Thus, we construct an initial datapath structure which corresponds to an initial design point. Guided by local cost functions, we improve the design through incremental merging of XTFBs. The local cost functions are associated with every path in the MAG. We define the local *cost gain* of the datapath resulting from merging XTFBs A and B into a third XTFB C as follows:

$g_{local}(C) = g_{ALU} + g_{Mux} + g_{Reg}$

where $g_{ALU}, g_{Mux}, g_{Reg}$ are described shortly in sections 3.4.1, 3.4.2, and 3.4.3.

We define next the local cost function f associated with a directed edge from A to B in the MAG as:

$f = \sum g_{local}(x) + \sum g_{local}(y) - g_{local}$

where x and y are all the nodes connected to A and B, respectively, and g_{local} is the local cost function associated with merging A and B.

The optimization algorithm starts with the nodes in the first two levels of the MAG and finds the level with the fewest outgoing edges. Obviously, at this level, the nodes with the fewest number of outgoing edges are more restricted than the others. Among these restricted nodes, we choose the nodes with a minimum cost f. The local gain functions g_{local} can be used to break any ties. After exhausting all the nodes at the level under consideration, we proceed to the following level and update the local cost functions so as to reflect the new structure in the datapath. We repeat this process until all nodes in the MAG have been processed.

3.4.1 Module Allocation

Every time we merge two ALUs we are reducing the data path cost by the cost of one ALU. Assuming that the combined operation set, $ALU1 \cup ALU2$, already exists in the library, we define the local cost gain function as:

$g_{ALU} = Cost(ALU_1) + Cost(ALU_2) - Cost(ALU_1 \cup ALU_2)$

3.4.2 Mux Allocation

When mapping a commutative operation to an XTFB, there exist two possible configurations to assign the input ports to the XTFB, left or right mux. For non-commutative operations such as subtraction, the configuration is unique.

When considering two XTFBs for merging, we assign the non-commutative operations to the multiplexers at the input ports of the resulting XTFB first. The remaining assignment is done so as to reduce the number of multiplexer inputs, right or left, through *incremental register alignment*. If $Cost(MUX_1)$ and $Cost(MUX_2)$ are the multiplexer cost of the original XTFBs, and $Cost(MUX)$ is the multiplexer cost, then the local cost gain function is defined as:

$g_{Mux} = Cost(MUX_1) + Cost(MUX_2) - Cost(MUX)$

3.4.3 Registers Allocation

The register cost gain function estimates the registers cost resulting from merging two XTFBs. The idea is to simply estimate the cost of registers at the output port of the XTFB. We apply locally the *Left Edge Algorithm* [10] incrementally at the output ports of the *resulting* XTFB. The restriction is that we do not merge registers that may create some type of self-adjacent registers (Rule 2). Thus, we guarantee the datapath self-testability. The Left-Edge running time is very fast since we are looking at a small list of registers corresponding to only two XTFBs at a time. The register cost function chooses controllable registers (TPGRs) at the *resulting* XTFB input ports and an observable register (MISR) at the output port.

Let n be the number of registers at the output port of the XTFB (Figure 1(b)), and let l be the number of TPGRs. Then, we estimate the cost of registers by:

$Cost_{reg} = Obs\_Cost + l * TPGR\_Cost + (n - l - 1) * Normal\_Cost$

where Obs_Cost is the cost of either an MISR or a BILBO, depending on whether the register needs to control another port in the datapath.

Merging two XTFB sets implies merging their output registers as well. We apply the Left Edge Algorithm to the register sets of both XTFBs. If $Cost_{reg1}$ and $Cost_{reg2}$ are the register cost of the two initial XTFBs, and $Cost_{reg}$ is the register cost of the resulting one, then the local register cost function is defined as:

$g_{Reg} = Cost_{reg1} + Cost_{reg2} - Cost_{reg}$

4 Test Point Selection

Our allocation method guarantees self-testability while keeping the controllable and the observable registers "close" to their individual modules and resulting in a sequential depth of one. To reduce further the test overhead, we perform test point (register) selection while keeping the fault coverage at an acceptable level. The selection is done in two phases. In the initial phase, described shortly, some TPGRs which are not necessary to test the datapath ALUs are eliminated. In the second phase, illustrated by an example in section 5, the system provides the ability to interactively remove/inject BIST registers to explore the design cost, test overhead and fault coverage. As a result of both reduction phases, the datapath area decreases significantly.

Let n be the number of ALUs in the data path. Let $m = 2n$ be the number of distinct input ports, assuming that every ALU has a $Left$ and $Right$ input port. Let l be the number of registers connected to the m input ports. The test point selection problem then reduces to one of covering every ALU port with a TPGR. However, we do not cover two input ports of an ALU with a single register to avoid adverse fault coverage effects (Equation 3). Formally:

Constraints:

$$\sum_{i=1}^{l} Left_{i,j} * X_i \geq 1 \qquad j = 1, 2, ...m \qquad (1)$$

$$\sum_{i=1}^{l} Right_{i,j} * X_i \geq 1 \qquad j = 1, 2, ...m \qquad (2)$$

$$\sum_{i=1}^{l} (Right_{i,j} * Left_{i,j}) * X_i = 0 \qquad j = 1, 2, ...m \qquad (3)$$

$$X_i = 0 \ or \ 1 \qquad (4)$$

Where

$$Left_{i,j} = \begin{cases} 1 & \text{if } Reg\ i \ covers \ left \ port \ j \\ 0 & \text{otherwise} \end{cases}$$

$$Right_{i,j} = \begin{cases} 1 & \text{if } Reg\ i \ covers \ right \ port \ j \\ 0 & \text{otherwise} \end{cases}$$

Cost function to minimize

$$C = \sum_{j=1}^{l} X_j \qquad (5)$$

We note that an *ILP solution* to the above problem is very efficient and very fast due to the small size of the problem. For our running example of Figure 2(a), the initial phase reduced the number of TPGRs by 6, a reduction of 61.53% of the original number of controllable registers.

5 Test Tradeoffs Illustration Using an Example

The test point selection described above reduces the number of TPGRs in the datapath. However, we can further reduce the number of BIST registers (TPGRS and MISRs) using the test metrics introduced by [4]. The idea is to remove an observable register at the output port of a XTFB if the faults can be sensitized through intermediate modules [4]. Of course, this will increase the sequential depth in the circuit. To illustrate our tradeoff scheme, we use the TRIGO example which computes the Taylor series for the trigonometric functions $sin(t/T)$ and $cos(t/T)$. All the operations for this example exist in our library ({+},{*},{/}). The test hardware overhead was significantly reduced (by 17.96 %) after applying our test point selection.

We have experimented with the fault coverage of several design styles for the trigo example versus the number of test patterns as shown in Figure 3. We notice that there is a measurable difference in fault coverage quality between the *BILBO* and the *test point selection option*, in favor of the *BILBO*. This difference is about 2% but it should be measured against the area increase required by the *BILBO* designs when making tradeoff decisions. We also note another interesting tradeoff measure that exists between *test area overhead* and *test time* or *fault coverage*. To be specific, our tradeoff scheme has the capability to "inject" observable points (MISRs) during the design process for the benefit of increasing the fault coverage (or reducing the number of test patterns). Thus, we can generate designs between the *BILBO* and the *test point selection* options by performing tradeoffs between test area overhead and fault coverage. By injecting an observable register we can "break" a chain in the circuit, used to bypass test patterns from a controllable point to an observable point. Breaking such chains improves the fault coverage of the circuit at the expense of increasing the circuit overhead; however, this increase is not very large. These tradeoffs are depicted in Figure 3 which illustrates three fault coverage/test pattern curves corresponding to one, two and three test registers injected. The data points in each curve were derived by injecting a test register from one data point to the next one.

Going back to our running example (Figure 2(a)), we have the following comments:

- The tradeoff scheme reduced the number of observable points by 2.
- The first design, based on no test point selection, has the highest fault coverage at the expense of larger area. The second and third designs present a good tradeoff or balance between area and test quality (fault coverage).

6 Results

We implemented the improved allocation and tradeoff scheme on a Sun SPARCstation IPC. The method is very fast and all reported results are produced in at most 12 CPU seconds (wave filter) including scheduling time.

We validate our methods using two sets of data. In the first set we use two examples, a differential equation from HAL [18] and an example from FACET [19]. Results are shown in Table 2 and Table 3, respectively. We compare with [11, 16, 19, 18] based on RALLOC [11] cost model. We note that the designs generated by [16, 18, 19] were modified by Avra [11] using CBILBOs in order to make them testable. As the results clearly show, the designs generated by our method, while fully testable, have a better cost. The data path cost in this case was defined [11] as:

$Cost = 20 * BILBOs + 35 * CBILBOS + MuxIn + Control\ Signals + Interconnect.$

We account for two more factors in the above cost formula, the cost of TPGR's and MISR's defined respectively as: $14 * TPGR + 16 * MISR$. These cost factors are dependent upon the technology and implementation used.

The second set of data consists of two relatively large benchmark examples, *the fifth order elliptical wave filter* and the *AR filter*. For the wave filter, first introduced by [18], we derive four testable designs based on four different schedules, i.e. 17, 19, 21 and 28 time steps. We assume multicyle operations in case of the multiplication while we assume additions take just one cycle. We show the detailed results for this example in terms of components, number and types of test registers, and the overall area of the datapath in Table 4. Table 1 shows the overhead using BILBO designs, and the overhead after applying our tradeoff scheme. The overhead was computed with respect to the overall area of the datapath which includes the area of ALUs, registers (normal and test registers) and multiplexers. In the case of 17 time steps, we notice a substantial overhead reduction, from 23.22 % to 2.18 %. The sequential depth was one in the case of BILBO option as well after the initial selection, while it increased to two after the final selection. The fault coverage versus the number of random patterns is shown in Fig. 4.

Design example	Initial Overhead	Final Overhead
AR Filter, T = 8	20.21 %	5.63 %
AR Filter, T = 10	25.23 %	6.08 %
Wave Filter, T = 17	22.13 %	3.94 %
Wave Filter, T = 19	23.22 %	2.18 %
Wave Filter, T = 21	27.88 %	6.74 %
Wave Filter, T = 28	44.90 %	9.26 %

Table 1: Test overhead before and after tradeoffs

The AR Filter was used initially by [8]. We derived two schedules using time steps 8 and 10 assuming that all operations execute in one cycle. We show detailed results for this example in Table 5. Table 1 illustrates the overhead using BILBO designs, then the overhead after applying our tradeoffs. We note that the sequential depth was one in case of the BILBO option and initial selection, but increased to two after the final selection. The fault coverage for this example versus the number of patterns is shown in Fig. 5.

To conclude, we have integrated all the software of our presented method into *SYNTEST*, a high level synthesis for testability system that we have developed at CWRU, whose details are reported in [20].

References

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

[2] M. Abramovici, M. Breuer, A. Friedman, *Digital Systems Testing and Testable Designs*, Computer Science Press, 1990.

[3] K. Cheng, V. Agrawal,"Synthesis of Testable Finite State Machines," *ISCAS-90*, 1990.

[4] S. Chiu, C. Papachristou, "A DFT Scheme with Applications to Data Path Synthesis," *DAC-91*, 1991.

[5] G. De Micheli et al., "The Olympus Synthesis System for Digital Design," Tech. Report, Stanford University.

[6] C. Gebotys, M. Elmasri, "VLSI Design Synthesis with Testability," *DAC-88*, 1988.

[7] C.L. Hudson, G. Peterson, "Parallel Self-Test With Pseudo-Random Test Patterns," *ITC-87*, 1987.

[8] R. Jain, K. Kukukcakar, M. Mlinar, A. Parker, "Experience with The Adam Synthesis System," *DAC-89*, 1989.

[9] B. Koenemann, J. Mucha, G. Zwiehoff, "Built-In Logic Block Observation Techniques," *ITC-79*, 1979.

[10] F. Kurdahi, A. Parker, "REAL: A Program for Register Allocation," *DAC-87*, 1987.

[11] LaNae Avra, "Allocation and Assignment in High-Level Synthesis for Self-Testable Data Paths," *ITC-91*, 1991.

[12] T. Lee, N. Jha, W. Wolf, "Behavioral Synthesis of Highly Testable Data Paths under the Non-Scan and Partial Scan Environments," *DAC-93*, 1993.

[13] J. Lis, D. Gajski, "Synthesis from VHDL," *ICCD-88*, 1988.

[14] A. Majumdar, K. Saluja, R. Jain, "Incorporating Testability Considerations in High-Level Synthesis," *FTCS-92*, 1992.

[15] *The NCR ASIC Data Book*, 1989.

[16] B. Pangrle, "Splicer: A Heuristic Approach to Connectivity Binding," *DAC-88*, 1988.

[17] C. Papachristou, S. Chiu, H. Harmanani, "A Data Path Synthesis Method for Self-Testable Designs," *DAC-91*, 1991.

[18] P. Paulin, J. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Trans. CAD*, Vol 8, 1989.

[19] C. Tseng, D. P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. CAD*, 1986.

[20] H. Harmanani, C. Papachristou, S. Chiu and M. Nourani, "SYNTEST: An Environment for System-Level Design for Test," *EURO-DAC 92*, Sept. 1992.

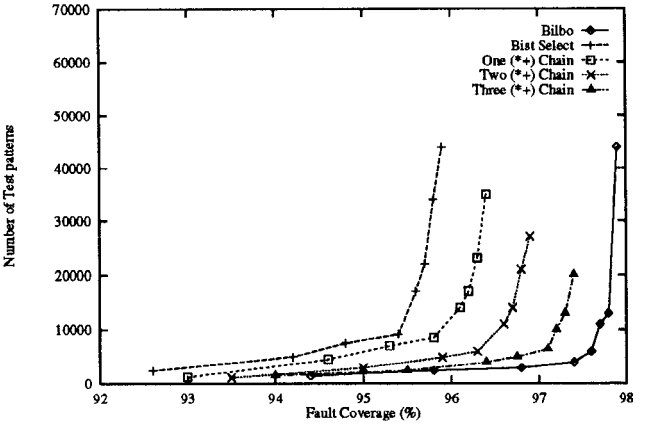

Figure 3: Fault coverage for the trigo example after injecting one, two, and three additional test points

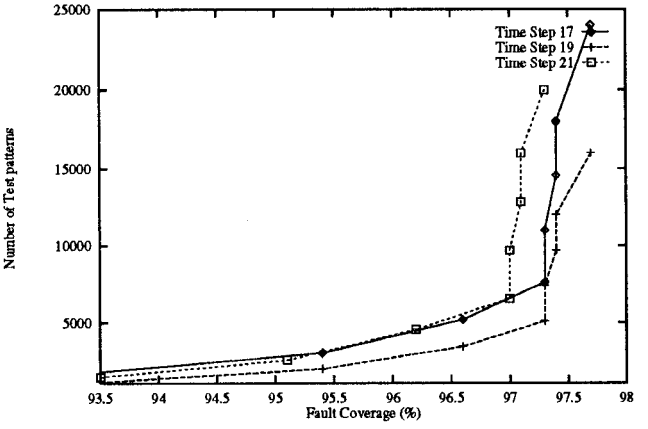

Figure 4: Fault coverage for the wave filter example

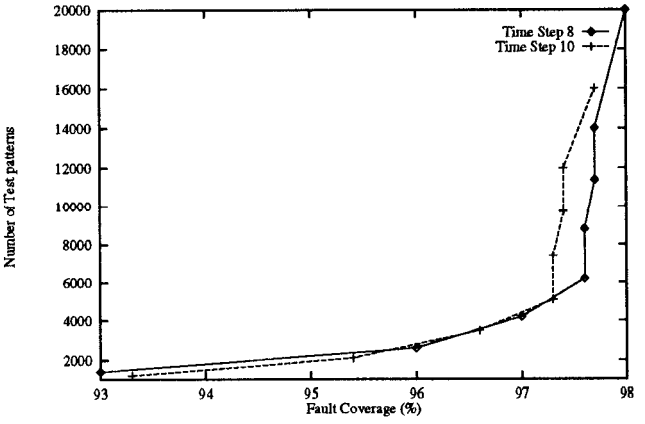

Figure 5: Fault coverage for the AR filter example

System	ALUs	BILBO	CBILBO	TPGR	MISR	Mux In	Inter-Connect	Control Signals	Cost
Splicer	(*)(*)(+)(-)(>)	1	5	0	0	17	34	23	269
HAL	(*)(*)(+)(-)(>)	1	4	0	0	19	35	24	238
RALLOC	(*)(*) (-) (+)	4	1	0	0	22	38	27	202
Ours	(+*)(*->) (*+)	0	0	4	1	15	28	21	136

Table 2: Designs Comparisons for the HAL Differential Equation example

System	ALUs	BILBO	CBILBO	TPGR	MISR	Mux In	Inter-Connect	Control Signals	Cost
Facet	(/)(-&+)(*|+)	3	5	0	0	15	31	23	304
Splicer	(/)(-&+)(*|+)	4	3	0	0	11	26	18	240
HAL	(/)(-&+) (*|+)	4	1	0	0	13	23	17	168
RALLOC	(+/|)(-&+) (*+)	4	1	0	0	16	29	21	181
Ours	(+/)(-&+) (*|+)	0	0	3	1	10	21	16	105

Table 3: Designs Comparison for the FACET Example

C_Step	*Mode*	ALUs	BILBO	TPGR	MISR	# *Reg*	Mux In	Area (μ^2)
17	Normal	3 (*), 3 (+)	0	0	0	15	32	101,500
	BILBO	3 (*), 3 (+)	6	3	0	15	32	127,480
	Initial Selection	3 (*), 3 (+)	2	3	4	15	32	110,964
	Final Selection	3 (*), 3 (+)	2	3	2	15	32	105,500
19	Normal	2 (*), 2 (+)	0	0	0	14	29	76,760
	BILBO	2 (*), 2 (+)	4	5	0	14	29	98,100
	Initial Selection	2 (*), 2 (+)	1	2	3	14	29	83,904
	Final Selection	2 (*), 2 (+)	1	2	1	14	29	78,440
21	Normal	(*), 2 (+)	0	0	0	11	31	65,260
	BILBO	(*), 2 (+)	3	3	0	11	31	87,668
	Initial Selection	(*), 2 (+)	1	2	4	11	31	72,396
	Final Selection	(*), 2 (+)	1	2	2	11	31	69,664
28	Normal	(*), (+)	0	0	0	17	32	60,246
	BILBO	(*), (+)	5	2	0	17	32	87,302
	Initial Selection	(*), (+)	0	2	2	17	32	65,830
	Final Selection	(*), (+)	0	2	2	17	32	65,830

Table 4: Results Summary from the wave filter

C_Step	*Mode*	ALUs	BILBO	TPGR	MISR	# *Reg*	Mux In	Area (μ^2)
8	Normal	3 (*), 3 (+)	0	0	0	16	31	103,292
	BILBO	3 (*), 3 (+)	6	8	0	16	31	125,568
	Initial Selection	3 (*), 3 (+)	2	2	4	16	31	114,356
	Final Selection	3 (*), 3 (+)	0	4	4	16	31	109,116
10	Normal	2 (*), 2 (+)	0	0	0	14	28	70,540
	BILBO	2 (*), 2 (+)	4	5	0	14	28	91,152
	Initial Selection	2 (*), 2 (+)	2	1	2	14	28	77,564
	Final Selection	2 (*), 2 (+)	1	2	2	14	28	74,832

Table 5: Results Summary from the AR Filter