Microarchitectural Synthesis of VLSI Designs with High Test Concurrency

Ian G. Harris and Alex Orailoğlu
Department of Computer Science and Engineering
University of California, San Diego, La Jolla, CA 92093-0114

The testability of a VLSI design is strongly affected by its register-transfer level (RTL) structure. Since the high-level synthesis process determines the RTL structure, it is necessary to consider testability during high-level synthesis. A synthesis system composed of scheduling and binding components minimizes the number of hardware sharing conflicts between tests in the test schedule. Novel test conflict estimates are used to direct the synthesis process. The test conflict estimation is based on examination of the interconnect structure of the partial design state during synthesis. Test conflict estimates enable our synthesis system to select design options which increase test concurrency, thereby decreasing test time. Experimental results show that designs generated by this approach are testable in a highly concurrent manner.

1 INTRODUCTION

The cost of chip testing has become a large fraction of the total chip production expenditure as other cost components have improved. Furthermore, increasing gate-to-pin ratios limit the feasibility of testing chips externally. The incorporation of test structures into the design ameliorates the testability of hardware which is not easily testable through external pins. The Built-In Self-Test BIST approach ([2], [11]) tests chip components using pseudo-random patterns which are generated on-chip. Consequently, testing can be performed on-site with minimal additional testing equipment, and at chip speed. BIST requires the placement of pseudo-random pattern generators (PRPG) and multiple-input signature registers (MISR) on the chip. PRPGs generate pseudo-random patterns to test the combinational modules on the chip and MISRs compact the results of the tests. All test registers are in a shift register chain so that seeds to the PRPGs can be shifted in at the beginning of testing, and compacted results can be shifted out of the MISRs after testing.

High chip test time reduces chip production throughput and increases chip production cost. In order to minimize test time, as many tests as possible should be executed in parallel, yet total parallelization is often impossible due to hardware sharing conflicts between tests. Hardware sharing conflicts occur because the test results of two different hardware modules may be forced to propagate through the same hardware in order to arrive at a MISR. Such conflicts occur as a result of the nature of the interconnect structure between different modules in the datapath. Considering test concurrency during high-level synthesis may greatly improve test time since the structural representation is determined at that stage.

Previous research concentrates on the effect of self-loops and sequential depth from test registers. Such approaches may increase the testability of a design, but they only indirectly address the problem of conflicts between tests. It is the test conflicts which increase test time by requiring that tests be executed sequentially. Since reduction of self-loops and sequential depth only resolves part of the test conflict problem, those approaches may degrade the datapath area and delay characteristics, while providing little increase in testability. A new metric for testability is needed which reflects the conflict properties of the tests.

In this paper we present metrics for the estimation of the number of test conflicts in the final design, and we present a scheduling and binding algorithm which uses these metrics to make intelligent high-level synthesis decisions. This novel approach introduces the possibility of including test concurrency and test time issues in microarchitectural synthesis. The proposed approach enables, for the first time, reasoning about evolving test concurrency in tandem with the evolving microarchitectural design. The results of the described algorithm are demonstrated.

This work is supported by the Semiconductor Research Corporation under contract number 93-DJ-538, the National Science Foundation under grant number CDA-9314748, and a graduate fellowship from Brooktree Inc.

2 PROBLEM DEFINITION

In this paper we describe a scheduling and binding algorithm which produces microarchitectural designs with high levels of test concurrency. The algorithm is applicable to a range of Built-In Self-Test methodologies, such as Partial-Intrusion BIST [1] and Full BIST [11]. The algorithm uses a testability metric which minimizes test application time by reducing hardware sharing conflicts between tests.

Under the assumption used throughout this paper, flip-flops are not necessarily placed at each control line, so random data is not sent on these lines. It is the responsibility of the test controller to select the mux configurations at each step of testing. We use this assumption because control of the mux configurations allows hardware sharing conflicts to be reduced by enabling test data to be directed through non-conflicting paths of the datapath.

In order to test a module, each of its inputs must receive test data from PRPGs, and its output must send test results to a MISR. The test data may pass through a number of modules between the output of the PRPGs and the input of the MISR. The subgraph of the datapath through which test data flows from PRPGs to MISRs in order to test a module is called a *test path*. When two or more test paths share hardware, they are said to *conflict*. Conflicts between test paths restrict the test scheduling [6], thereby reducing the throughput. The concurrency of conflicting test paths is restricted differently depending on the type of hardware being shared. The two types of conflicts are listed below.

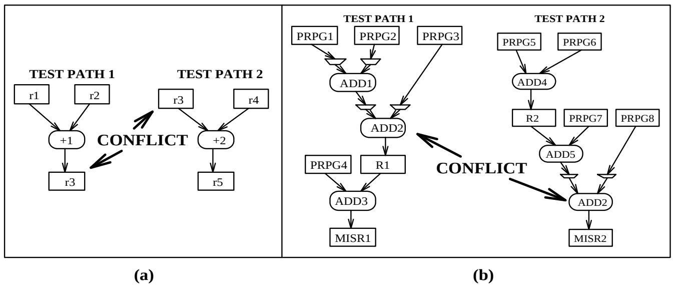

Figure 1: (a) Hard Conflict (b) Soft Conflict

Hard Conflicts occur when one test path uses a register as a MISR, while another test path uses the same register as a PRPG. Figure 1a shows a hard conflict between two tests paths due to their sharing of register r3. Since a register with both PRPG and MISR capabilities entails large area overhead, we disallow this option. This assumption forces the two test paths in figure 1a to be executed in different *test sessions*. All of the test paths in one test session are executed concurrently with one another, but each test session must be executed sequentially. Additional test sessions increase test time.

Soft Conflicts exist when two test paths share intermediate registers, muxes, buses, or functional units at the same control step. Soft conflicts can be avoided by scheduling into different control steps, operations which conflict. In the example of figure 1b, by scheduling the use of ADD2 in test path 1 to the first control step, and the use of ADD2 in test path 2 to the second control step, the conflict has been avoided and the two test paths can be executed in parallel. The work in this paper assumes that test scheduling will be performed in a single test session which contains no hard conflicts. For this reason, references to conflicts in this paper refer to soft conflicts.

We identify two goals which should be satisfied during synthesis to ensure testability: (a) each module should be *covered* (included in a test path), and (b) full coverage should be achieved with as few test conflicts as possible. Since test paths are not defined until after the datapath is complete, it is necessary at this stage of synthesis to design a datapath which allows the definition of non-conflicting test paths. It is obvious that a module must be included in a test path in order to be tested, making this goal minimally necessary in order to test the chip. The number of conflicts between tests is important due to its effect on test time. Since all tests which share hard conflicts must be performed in different test sessions, the maximum number of mutually conflicting tests is a lower bound on the number of test sessions. Soft conflicts can be avoided, but their avoidance often requires an increase in the number of control steps required to perform a test. Both hard and soft conflicts can be avoided through proper datapath definition during high-level synthesis. The effects of test conflicts are also apparent when testing is performed in a pipelined fashion. During pipelined testing, test conflicts must be avoided between different test paths, as well as different instantiations of each path. Consequently, soft conflict consideration is crucial to maximizing throughput during pipelined testing.

3 MOTIVATION

The work presented here attempts to: (a) cover each module in a test path, and (b) achieve full coverage with the minimum number of test sessions. Since every module must participate in at least one test path, each module must be covered. Coverage is reduced if a module's input port is reachable only by registers which must act as MISRs, or if a module's output port can only reach registers which must act as PRPGs. A self-loop is perhaps an extreme case of a lack of coverage since an input port and an output port cannot be independently covered. Test time is greatly increased by reduced test concurrency caused by hardware sharing conflicts between tests. The

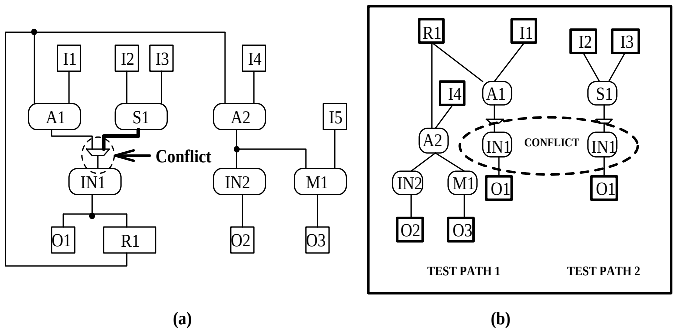

Figure 2: (a) Datapath with Conflict Problem (b) Conflicting Test Paths

datapath in figure 2a has a test conflict because both modules A1 and S1 must be observed through module IN1. The test paths in figure 2b show that the outputs of modules A1 and S1 must both be connected to the input of IN1. This conflict forces A1 and S1 to be tested sequentially, in different test sessions, increasing test time.

Coverage and conflict problems may be inadvertently created during high-level synthesis unless care is taken to avoid them. Evolving concurrency problems can be estimated during high-level synthesis to determine which decisions will reduce the probability of conflicts. Scheduling and binding decisions which do not consider the conflicts between tests can limit the testing options of many modules to single options which share hardware. When the test options of two or more modules are limited in this way, a conflict is forced to occur which may have been avoidable.

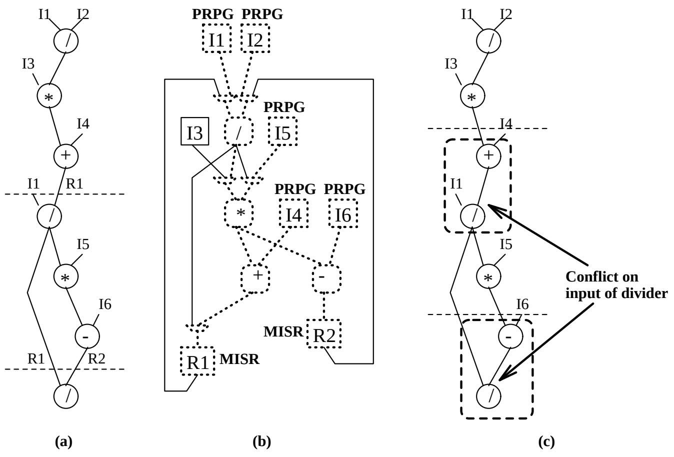

Figure 3: (a) Scheduling Which Allows Conflict to be Avoided (b) Datapath for Scheduling (c) Scheduling Resulting in Minimum Two Test Sessions

We use figure 3 to illustrate the necessity of considering test conflicts during scheduling. Two schedulings are shown for the dataflow graph in figure 3. Dashed horizontal lines denote clock cycle boundaries. In the scheduling in figure 3a, at least one binding exists as shown which results in the concurrently testable datapath shown in figure 3b. All functional units are testable in a single test session through the test path which is shown in dotted lines in figure 3b. The scheduling in figure 3c does not allow a binding to be generated that avoids a test conflict because the outputs of both the adder and the subtracter are connected only to the right input of the divider. In order for the subtracter and adder to be tested concurrently, the divider's right input would need to receive input from both the adder and subtracter at the same time. The scheduling in figure 3a allows binding to be performed without creating a conflict. Consequently, it is more advantageous, from a self-test perspective, than the scheduling shown in figure 3c.

Similarly, the example in figure 4a illustrates the necessity of considering test conflicts during binding. Figure 4a shows the dataflow graph from which the datapath in figure 2a was generated. Node inc1 in figure 4a has the option of being bound to IN1 or IN2. With respect to interconnect, these two options are equivalent, but from a testing standpoint, one option causes the conflict shown in 2a, and the other option generates the conflict-free datapath in figure 4b whose test paths (which are testable in one session) are shown in figure 4c. Notice that there is no conflict at the input of IN2 because A2 may be observed through M1 as shown in figure 4c.

Suboptimal scheduling and binding decisions can easily increase test time as shown above, while reduced test time solutions may exist which incur no additional interconnect overhead. It is clear that an effort must be made during scheduling and binding to avoid these conflicts.

4 PREVIOUS WORK

The use of high-level synthesis as a technique which allows fast exploration of microarchitectural design possibilities has

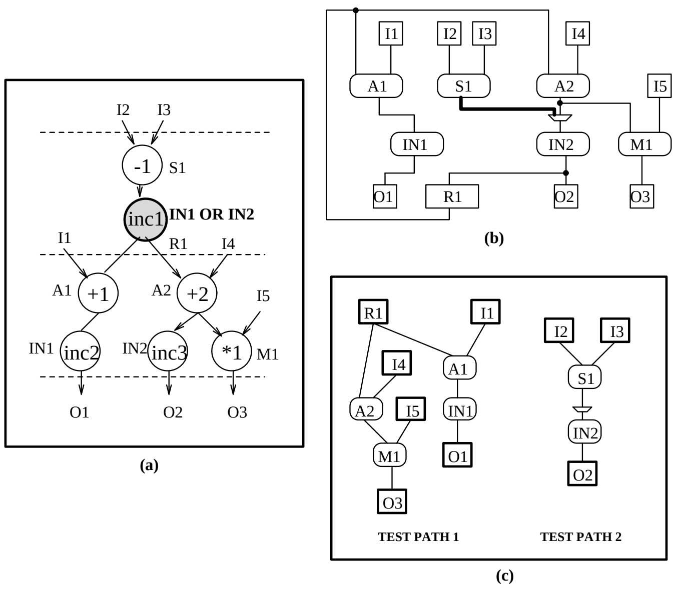

Figure 4: (a) Dataflow Graph (b) Datapath without Conflict (c) Test Paths

been thoroughly studied. Recently, efforts have been made to incorporate new design constraints into high-level synthesis such as fault-tolerance [7, 8] and testability. Testability constraints have been included into high-level synthesis in [10], wherein synthesis is performed to reduce sequential depth between registers and primary I/O pins. Reduction of sequential depth will reduce test time by reducing the number of patterns necessary to control and observe each possible fault, but it does not directly reduce the number of conflicts between tests. In [12], an algorithm is presented to perform register and operator binding to remove self-loops in the datapath which can cause problems during testing. Work performed in [5] is an initial attempt at incorporating test conflict considerations in conjunction with synthesis. Test conflicts are removed in [5] by modifying high-level synthesis binding decisions to redefine RTL interconnect.

Various metrics have been used to estimate the testability of a datapath. In [3], a metric is proposed to estimate the controllability/observability of chip modules based on adjacency to registers and external pins. Chiu and Papachristou [4] present metrics to estimate the fault coverage of circuit modules based on the locations of test registers in the datapath.

5 DESIGN REPRESENTATION

It is necessary to maintain a representation of the partial design state during synthesis in order to estimate the effect of synthesis decisions on the testability of the design. While performing binding, we model the design as a graph whose nodes are functional units and registers with a number of input ports, and a single output port. The edges of the graph represent point-to-point connections between those ports.

During scheduling (before binding has been performed), no distinction can be made between hardware modules with the same functionality, so each node in the *Scheduling Design Representation* corresponds to the set of all modules with the same functionality. Each node in the scheduling design representation is annotated with an allocation value to indicate the number of modules with the corresponding functionality that will exist in the final design. Each edge in the scheduling design representation is annotated with a *connection weight* which is the total number of connections which exist between the two classes of modules that the edge spans. The *Binding Design Representation* distinguishes each hardware module as a different node, and each connection between modules as a different edge with unit weight. Figure 5 shows an example of a scheduling design representation and one binding design representation to which it maps.

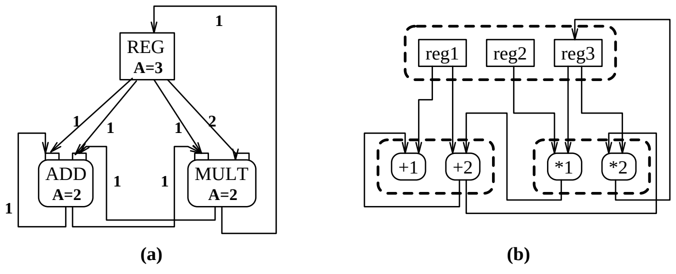

Figure 5: (a) Scheduling Design Representation (b) Binding Design Representation

6 ALGORITHM OVERVIEW

The algorithm performs sequentially scheduling and binding of a dataflow graph under a clock cycle and clock duration (performance) constraint and an allocation (area) constraint. Synthesis is performed to minimize the estimated number of test conflicts in the final design.

The goal during scheduling is to selectively break connections between adjacent dataflow graph nodes by scheduling them to different clock cycles. This causes the insertion of a register between the functional units to which the dataflow graph nodes are eventually mapped during binding. In this way, connections between functional units and registers are distributed, to allow each operator to have sufficient conflict-free test options. During scheduling, test conflicts can only be avoided between *groups* of functional units of the same type. It is not until the binding phase that conflicts between individual functional units can be examined. As was illustrated in the example of figure 3, unless scheduling creates a good distribution of interconnections between hardware types, it may be impossible for binding to avoid a conflict.

A scheduling decision is chosen by first finding a connection in the scheduling design representation whose connection weight should be increased. The connection is chosen using the proposed test conflict metric to evaluate the design representation as a result of increasing each connection individually. Then a scheduling decision is selected which will cause the weight of the chosen connection in the design representation to be increased. The chosen scheduling decision is performed and the effects of the decision on the design are propagated by pruning design options which have been made infeasible due to the scheduling decision. Scheduling decisions are selected in this manner until scheduling is complete.

Binding decisions distribute interconnections over the modules to allow all modules to have a multiplicity of test options. Operator and register binding are performed in an intertwined fashion. At each decision step, all possible operator and variable decisions are enumerated and evaluated. The options are evaluated using the proposed test conflict metric to evaluate the testability of the binding design representation as a result of each option. The selected binding option is then performed and its effects are propagated throughout the design state to prune away infeasible design options. Additional binding decisions are selected until binding is complete.

The rest of the paper is organized as follows: Section 7 describes the test conflict metrics which are used to guide synthesis. Sections 8 and 9 describe the use of the test conflict metric during scheduling and binding. Experimental results are discussed in section 10.

7 METRICS FOR TEST CONCURRENCY

We propose the following testability metrics which examine an incomplete design and estimate the number of hardware sharing conflicts that will occur between tests in the final design. These metrics are used in the algorithm to evaluate the testability of proposed scheduling and binding decisions, and thus guide the algorithm towards designs with superior testability characteristics.

The two aspects that influence test time are: (a) concurrency between tests, and (b) the number of patterns required for each test path. The degree of chaining affects the number of patterns for each test path. This issue has been explored previously in [13]. The research described here maximizes the test concurrency by avoiding the creation of test hardware sharing conflicts in the structural representation. Consequently, we propose metrics for modeling the number of test conflicts during high-level synthesis.

7.1 Metric Definition

We identify two characteristics of the I/O ports of a module that indicate the testability of that I/O port.

- **Coverage Probability**: The probability that *at least one* of the incoming edges will be included in a test path which connects it to LFSRs. If at least one of the incoming edges is covered (included in a test path), then the I/O port will be covered also.
- **Conflict Probability**: The probability that a hardware component will need to be multiply included in some test path(s). Conflict probability is only valid for the input ports of a component, since an output port can fan out to many different modules, while an input port can only select a single input.

To evaluate the testability quality of a datapath, we examine the ratio between the average coverage probability and the maximum conflict probability over all I/O ports. The *average coverage* is maximized because all I/O ports are equally important since they must all be tested. The *maximum conflict probability* is used because the largest conflict in any I/O port determines the maximum degree of allowable concurrency. For example, in the datapath of figure 2a, although there is a conflict on the input of only a single module, IN1, that single conflict causes two test sessions to be the minimum number.

7.2 Coverage Calculation

The coverage of an I/O port is the probability that the port will be contained in a test path which connects it to either an LFSR or a MISR. The coverage probability of a port is a path based metric which evaluates the paths between a port and a test register. The coverage can be computed as a function of the coverages of the neighboring edges and nodes.

Each input port i of a module m has a coverage $C_{in}(m,i)$ which is a function of the coverage probabilities of its incoming edges. The output port is similarly associated with a coverage probability $C_{out}(m)$, which is a function of the output coverages of the outgoing edges. Each edge has an input coverage value, $C_{in}(e)$, which represents the probability that it will be needed by its predecessor module to send test data to a MISR. Each edge also has an output coverage, $C_{out}(e)$, which is the probability that its successor module will need it to connect to a PRPG. The formulas for $C_{in}(e)$ and $C_{out}(e)$ are shown in equation 1,

$$C_{in}(e) = \prod_{i=0}^{iMax} \frac{C_{in}(m_p(e),i)}{OutDeg(m_p(e))},\ C_{out}(e) = \frac{C_{out}(m_s(e))}{InDeg(m_s(e),e)} \tag{1}$$

where $m_p(e)$ is the predecessor module of edge e, $m_s(e)$ is the successor module of edge e, $iMax$ is the number of input ports that module m_p has, $OutDeg(m)$ is the number of outgoing edges of m, and $InDeg(m,e)$ is the number of edges entering module m at the input through which edge e is connected.

The *input coverage probability* of an input port is the probability that at least one of the incoming edges is covered. This can be computed as shown in equation 2.

$$C_{in}(m,i) = 1 - \left(\prod_{e \in ein(m,i)} 1 - Cin(e) \right) \tag{2}$$

where $ein(m,i)$ is the set of all edges entering module m at input i.

The *output coverage probability* at an output port is computed in a similar way and is shown in equation 3.

$$C_{out}(m) = 1 - \left(\prod_{e \in eout(m)} 1 - Cout(e) \right) \tag{3}$$

where $eout(m)$ is the set of all edges exiting module m.

7.3 Conflict Calculation

Since conflicts can only occur on the inputs of a module, only input ports have conflict probabilities. The conflict probability $X(m,i)$ for input port i of module m is the probability that two incoming edges will need to be covered. This is modeled in the binding design representation by finding the probability that the two incoming edges with the highest coverage probabilities will both be covered. The coverage probabilities of two edges are assumed to be independent, so the probability of their coincident occurrence is modelled as the product of their respective probabilities. In the binding design representation the conflict probability is calculated as shown in equation 4.

$$\begin{aligned}
X(m,i) &= C_{in}(eM1(m,i)) * C_{in}(eM2(m,i)) && (4)\\
eM1(m,i) &= e_1 \ni (e_1 \in eList(m,i) \cap && (5)\\
&\quad C_{in}(e_1) = \max_{e \in eList(m,i)} C_{in}(e)) \\
eM2(m,i) &= e_1 \ni (e_1 \in eList(m,i) \cap && (6)\\
&\quad C_{in}(e_1) = \max_{(e \in eList(m,i) \cup e \neq eM1(m,i))} C_{in}(e))
\end{aligned}$$

where $eList(m,i)$ is the set of edges entering module m at port i.

Each edge in the scheduling design representation represents as many connections in the datapath as the edge weight indicates. The conflict estimate must reflect conflicts between edges in the actual datapath, so the conflict estimate for the scheduling design representation is formulated as shown in equation 7.

$$X(m,i) = \begin{cases} C_{in}(eM1(m,i)) * C_{in}(eM2(m,i)) & \\ & \text{if } CW(eM1(m,i)) \leq 1 \\ C_{in}(eM1(m,i))^2 & \text{if } CW(eM1(m,i)) > 1 \end{cases} \tag{7}$$

In the equation above, CW denotes the connection weight of an edge.

8 SCHEDULING FOR TEST CONCURRENCY

During scheduling, in order to minimize test conflicts it is necessary to distribute connections between different hardware types in such a way that all of the hardware types can be covered with minimal increase to the conflict probability. It is important that each hardware module participate in at least one test path, but not two or more *conflicting* test paths. The scheduling algorithm distributes connections between different module types to minimize the probability that a single hardware type is required to participate in multiple test paths.

Each node in the scheduling design representation corresponds to the set of modules with a particular functionality.

The edge weight of each edge in the representation is the number of connections between the sets of modules represented by the nodes adjacent to the edge. Before any scheduling decisions have been performed, only the connections to constants and architectural variables can be determined. Consequently, the weights of the edges to the variable and constant nodes in the representation are initialized with the number of connections to variables and constants respectively. Since connections to nodes other than variables and constants cannot be determined before scheduling, the edges to these nodes are initialized with zero weight.

The two types of scheduling decisions which may be performed at each step are: (a) scheduling two adjacent nodes in the same clock cycle, and (b) scheduling two adjacent nodes in different clock cycles. Adjacent pairs of nodes are scheduled together so that at least one connection will be determined at each step. An adjacent pair of nodes is simultaneously considered to ensure a well-defined effect on the interconnect definition in the structural representation.

Each scheduling decision increments the weights of edges in the scheduling design representation. The quality of a scheduling decision is measured by increasing the weights of the edges which will be affected, and evaluating the resulting design using the proposed testability metric for estimating coverage and conflict probability. The algorithm applies one of the scheduling options which most increases the coverage/conflict ratio. There may be many scheduling options which equally improve the coverage/conflict ratio. A scheduling option is chosen from the set of candidate options which best preserves the degrees of scheduling freedom of adjacent nodes.

9 BINDING FOR TEST CONCURRENCY

During binding, it is necessary to distribute connections between individual modules in such a way that all of the modules can be covered, yet no single module is required to participate in more than one test path. Each node in the binding design representation corresponds to a unique module in the structural representation, and each edge corresponds to a point-to-point connection between modules in the datapath. Before binding has been performed, no module connections can be determined, so the binding design representation contains no edges.

The two types of binding decisions that can be made are: (a) binding a node to a functional unit, and (b) binding a variable to a register. Each binding decision necessitates the addition of new edges to the binding design graph. The quality of each binding decision is measured by adding edges to the design representation which would need to be added if the binding decision were performed, and using our testability metric to evaluate the resulting design.

As a result of a binding decision, the binding options of the remaining unbound nodes need to be restricted. No other operations may be bound to a module if there is already a module bound which is assigned to the same control step. The binding possibilities of each unbound node are updated to avoid these hardware usage conflicts. A similar algorithm is applied for updating register binding possibilities; in this case, register lifetimes are additionally considered.

10 EXPERIMENTAL RESULTS

The results are illustrated in two parts. We first show detailed results for selected high-level synthesis benchmarks. We then summarize extensive results in tabular form, applied to a number of high-level synthesis benchmarks. To illustrate the performance of this system in detail, we examine the design results for the differential equation example [15] and the design of the 16 point elliptic filter benchmark[9] (with chaining). Chained execution of operations in a clock cycle is used in the elliptic filter design, but not in the differential equation design.

The differential equation dataflow graph is scheduled and bound by our system in 4 clock cycles, with 2 adders, 2 multipliers, and 1 relational operator. The resulting datapath is shown in figure 6. The rectangles in the datapath are architectural registers used to store input and output variables, and intermediate registers used to store intermediate values.

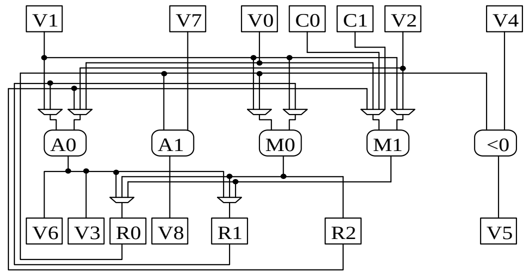

Figure 6: Differential Equation Datapath

The scheduling and binding enable each functional unit to be tested in a single test session using the test paths shown in figure 7 (the test registers are shown shaded). Test conflicts are successfully avoided during synthesis by giving each functional unit a non-conflicting test option.

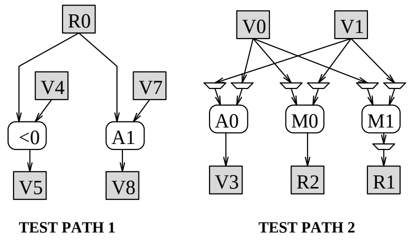

Figure 7: Differential Equation Test Paths

We also explore a design for the elliptic filter example in figure 8 which is scheduled and bound by our system. For clarity we have only included the dataflow operations and the non-recursive edges in figure 8. The full dataflow graph and signal flow graph can be found in [9]. Synthesis was performed in 9 clock cycles, with 5 adders, and 2 multipliers, with chaining. This example also enables test conflicts to be avoided, and testing to be performed in a single session. The non-conflicting test paths are shown in figure 9. An interesting feature is that chaining in a test path, which would cause an increased number of test patterns, is not necessary due to the range of test options allowed by the structure of the design.

In addition to the two designs described above, we have generated multiple designs for the AR-filter[14] the FIR-filter[14], the fifth-order elliptic filter, and the differential equation flowgraphs under various constraints. The results of these synthesis experiments with different allocations and clock cycle limits are summarized in table 1. The *Chain* column indicates the degree of chaining allowed during scheduling. The last column of the table contains the number of test sessions in which each design can be tested. In all experiments, the high-level synthesis system achieved a single test session, independent of the synthesis constraints given, or the existence of chaining in the scheduling of the dataflow graph.

11 CONCLUSIONS

The importance of test time as a component of chip cost has caused test time to become a design attribute that needs to be considered at the earliest stages of design. The strong effect of test conflicts on the test time of a design makes the use of a test conflict metric necessary in order to reduce test time during high-level synthesis. The work presented here is the first to integrate test conflict information into scheduling and binding in this way. The use of the proposed test conflict metric

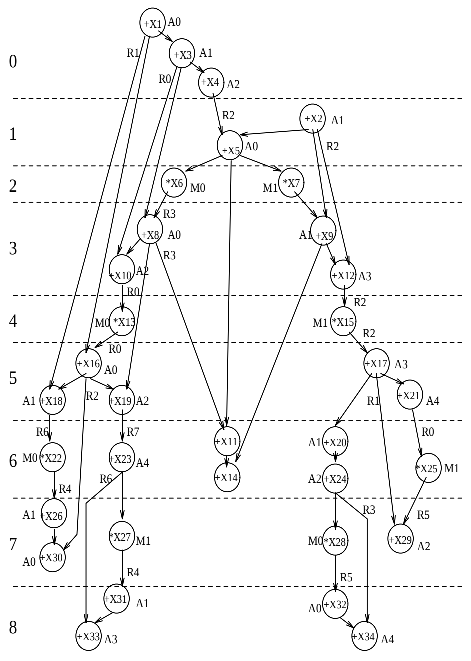

Figure 8: Elliptic Filter Scheduling and Binding

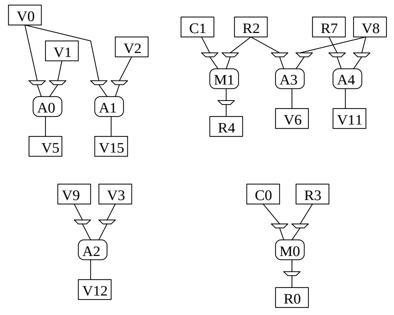

Figure 9: Test Paths to Test Datapath Derived from Scheduling

enabled all of the designs which we generated in our experiments to be testable with maximum test concurrency. We have thus shown that highly testable designs with high levels of test concurrency can be achieved by considering testability during microarchitectural synthesis. Future planned extensions of this work include utilization of the proposed test conflict metric to select testable registers in conjunction with selection of test paths.

	Clks	+	*	>	Chain	Sess
AR Filter	4	4	6	0	any pair	1
	8	2	4	0	none	1
	11	2	3	0	none	1
FIR Filter	7	3	2	0	2 adders	1
	9	2	2	0	none	1
	11	2	1	0	none	1
Ellipt. Filter	9	5	2	0	3 adders	1
	14	4	2	0	none	1
	15	3	2	0	none	1
Diff. EQ	2	3	4	1	any pair	1
	4	2	2	1	none	1
	5	1	2	1	none	1

Table 1: Test Session Results for Various Designs

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design.* Computer Science Press, 1990.

[2] P. H. Bardell, W. H. McAnney, and J. Savir. *Built-In Test for VLSI.* Wiley-Interscience, 1987.

[3] C.-H. Chen, C. Wu, and D. G. Saab. BETA: Behavioral Testability Analysis. *Proceedings of the IEEE Conference on Computer Aided Design*, pages 202–205, November 1991.

[4] S. Chiu and C. A. Papachristou. A Design for Testability Scheme with Applications to Data Path Synthesis. *Proceedings of the 28th Design Automation Conference, ACM-IEEE*, pages 271–277, June 1991.

[5] I. G. Harris and A. Orailoğlu. Effective Test Path Definition Assisted by High-Level Synthesis Modifications. *Proceedings of the Synthesis and Simulation Meeting and International Exchange (SASIMI)*, pages 187–195, October 1993. Nara, Japan.

[6] I. G. Harris and A. Orailoğlu. Fine-Grained Concurrency in Test Scheduling for Partial-Intrusion BIST. *Proceedings of the European Design Automation Conference*, pages 119–123, February 1994.

[7] R. Karri and A. Orailoğlu. Scheduling with Rollback Constraints in High-Level Synthesis of Self-Recovering ASICs. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 519–526, July 1992.

[8] R. Karri and A. Orailoğlu. Area-Efficient Fault Detection During Self-Recovering Microarchitecture Synthesis. In *Proceedings of the 31st Design Automation Conference*, June 1994.

[9] S. Y. Kung, H. J. Whitehouse, and T. Kailath. *VLSI and Modern Signal Processing.* Prentice-Hall, 1985.

[10] T.-C. Lee, W. H. Wolf, N. K. Jha, and J. M. Acken. Behavioral Synthesis for Easy Testability in Data Path Allocation. *Proceedings of the IEEE Conference on Computer Design*, pages 29–32, October 1992.

[11] E. J. McCluskey. Built-In Self-Test Techniques. *IEEE Design and Test*, pages 21–28, April 1985.

[12] A. Mujumdar, K. Saluja, and R Jain. Incorporating Testability Considerations in High-Level Synthesis. *22nd Fault Tolerant Computing Symposium*, pages 272–279, July 1992.

[13] A. Orailoğlu and I. G. Harris. Test Path Generation and Test Scheduling for Self-Testable Designs. *Proceedings of the IEEE Conference on Computer Design*, pages 528–531, October 1993.

[14] N. Park and A. C. Parker. Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications. *IEEE Transactions on Computer Aided Design*, 7(3):356–370, March 1988.

[15] P. G. Paulin and J. P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Transactions on Computer Aided Design*, 8(6):661–679, June 1989.