Behavioral Synthesis of Fault Secure Controller/Datapaths Based on Aliasing Probability Analysis

Ganesh Lakshminarayana, *Member, IEEE*,
Anand Raghunathan, *Member, IEEE*, and Niraj K. Jha, *Fellow, IEEE*

Abstract—This paper addresses the problem of synthesizing fault-secure controller/data path circuits from behavioral specifications. These circuits are guaranteed to either produce the correct output, or to flag an error. We use an iterative improvement-based behavioral synthesis framework that performs functional unit selection, clock selection, scheduling, and resource sharing with the aim of minimizing the area of the synthesized circuit, while allowing multicycling, chaining, and functional unit pipelining. We present a dynamic comparison selection algorithm that can be used *during* behavioral synthesis to determine which intermediate results in the computation need to be secured in order to enable maximal resource sharing. Previous work on synthesizing fault-secure data paths has focused on *ensuring* that *aliasing* (a condition when the circuit produces an incorrect output and does not flag an error) cannot occur in any part of the design. We demonstrate that such an approach can lead to unnecessarily large overheads. In order to alleviate the overheads incurred for fault security, our behavioral synthesis framework uses *ALiasing Probability analysis* (ALPS) in order to identify resource sharing configurations that reduce area while introducing a very low probability of aliasing (of the order of 10^{-10} for a bit-width of 32) in the resultant data path. Experimental results performed for several behavioral descriptions demonstrate that our techniques synthesize more compact circuits than techniques available in the literature, e.g., double modular redundancy or zero-aliasing techniques.

Index Terms—Behavioral synthesis, concurrent error detection, fault security, fault-tolerant microarchitectures.

1 INTRODUCTION

WITH the increasing use of VLSI circuits in critical applications such as automotive electronics, process control, implantable medical devices, and avionics, the need for fault-tolerant circuits is increasing. Fault tolerance can be considered a design metric at any level of the design hierarchy ranging from the circuit level to the system level. However, considering fault tolerance late in the design cycle often involves large overheads in area, delay, and power consumption. This motivates the application of fault tolerance at higher levels. Fault tolerance covers detection, diagnosis, and recovery from faults.

A well-known method to achieve fault detection is duplication with comparison [1], [2], where the problem at hand is solved by two identical circuits and the outputs of the two circuits are compared to check for errors. A disadvantage of this technique is that it involves twice the amount of resources of the simplex system, in addition to the cost of a comparison element. Previous work has explored the integration of fault tolerance during the behavioral synthesis process. In [3], algebraic transformations were used to alleviate the hardware costs of

n-modular redundancy. In [4], checkpointing and rollback were integrated into the scheduling task of behavioral synthesis. A constrained assignment for self-recovering data paths was investigated in [5].

A formal characterization of fault security for multi-processor schedules was presented in [6]. The concept of fault security was first integrated into behavioral synthesis of application-specific integrated circuits in [7], where a technique for securing intermediate results in a computation was presented in order to ease scheduling constraints and introduce additional opportunities for resource sharing. Allocation and assignment for ensuring fault security were addressed in [8]. Schemes for error recovery through checkpointing and rollback were presented in [9], [10], and [11]. Techniques for behavioral synthesis of reconfigurable or self-repairing data path structures were presented in [12].

In this work, we focus on behavioral synthesis of fault-secure controller/data paths for data-dominated circuits. The data path is made fault-secure by having two separate threads of computation and comparing the results using equality checkers. In order to reduce the overheads required to provide fault security, we allow resource sharing between the two threads and secure intermediate results by inserting comparison operations appropriately. We present an algorithm to determine which intermediate results to secure in order to maximize the benefits of resource sharing. Unlike previous work, where the decision of which intermediate results to secure was performed *statically before behavioral synthesis*, our method performs this task *dynamically (during behavioral synthesis)*. Thus, our

- G. Lakshminarayana and A. Raghunathan are with C&C Research Laboratories, NEC USA, Princeton, NJ 08540.
E-mail: ganesh@ccl.nj.nec.com.
- N.K. Jha is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544. E-mail: jha@ee.princeton.edu.

Manuscript received 29 Sept. 1997; accepted 24 Nov. 1999.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 105722.

algorithm also naturally explores cost trade-offs between comparators and the other data path components, such as functional units, registers, and multiplexers. We demonstrate that it is possible to *significantly reduce the area overhead required* for fault security if we are willing to tolerate a very small nonzero probability of aliasing. In this direction, we present an aliasing analysis procedure that guides behavioral synthesis by identifying configurations that are very unlikely to lead to aliasing. Allowing architectures with a very low probability of aliasing leads to a substantial increase in the feasible solution space and results in better solutions. Our dynamic securing and aliasing probability analysis techniques have been integrated into an iterative improvement-based behavioral synthesis framework that performs functional unit selection, clock selection, scheduling, allocation, and assignment, while exploring the trade-offs that result from the interaction of these tasks. To the best of our knowledge, no other work targets functional unit or clock selection in behavioral synthesis for fault tolerance. We also allow for loops in the behavioral description, unlike many previous methods in this area. The aliasing analysis techniques that we have developed differ in flavor from those in the built-in self test area published in the past [13]. Unlike past methods, our aliasing analysis procedure does not assume any particular error distribution at the output of a faulty module; this makes our analysis procedure much more robust than past methods and applicable to a much wider class of faults.

The controller finite-state machine (FSM) is made fault-secure by constraining the state encoding procedure to assign codes having the same parity to all the states and, in addition, constraining the primary outputs and next state outputs of the controller to be implemented by disjoint cones of logic [14]. The parity of the primary outputs of the controller is also generated as a controller output. This ensures that a failure in any one of the logic cones will produce a single error either in a next-state line or in a primary output. This is detected using a totally self-checking (TSC) parity checker which computes the parity of all the controller outputs (all the next-state lines, all the primary outputs, and the generated parity of the primary outputs).

The rest of this paper is organized as follows: In Section 2, we present some background material. In Section 3, we describe the behavioral synthesis framework. In Section 4, we discuss aliasing probability analysis and show how this analysis can be utilized in the behavioral synthesis framework. In Section 5, we discuss our dynamic securing approach. In Section 6, we present a complete illustrative example. In Section 7, we present experimental results and conclude in Section 8.

2 BACKGROUND

In this section, we introduce some basic concepts that we use in our work.

Behavioral Synthesis: Behavioral synthesis is the process of transforming a behavioral description of the system into a register-transfer level (RTL) implementation. Behavioral synthesis can be divided into several tasks, such as scheduling, allocation, assignment, functional unit selec-

tion, and clock selection. We assume that the behavioral specification, which is usually provided in a hardware description language, has been compiled into a control-data flow graph (CDFG). Vertices in the graph represent operations and edges in the graph denote data or control dependencies. The process of scheduling assigns a cycle (or set of cycles) of execution to each operation in the CDFG. Clock selection refers to the procedure of choosing a value for the system clock period. Functional unit selection involves choosing a functional unit library template or functional unit type for each operation in the CDFG. The processes of functional unit allocation and assignment decide how many instances of each functional unit template (e.g., *ripple_carry_adder*, *carry_lookahead_adder*, *array_multiplier*, *wallace_tree_multiplier*, etc.) to use and map the operations of the CDFG to the allocated instances.

Fault Security: A circuit is said to be *fault-secure* with respect to a specified class of faults if, on the occurrence of any fault from the class, the circuit produces either the correct output or an output that will be detected as erroneous (i.e., *an erroneous output never goes undetected*). The fault model that we use in this work assumes the fault to be confined to a single unit in the circuit (e.g., adder, multiplier, register, multiplexer, etc.). The fault can last for any duration (i.e., it may be permanent or transient) and can cause an arbitrary error at the unit's output.

Duplication with comparison [1] is the traditional method of providing fault security to a system. The two circuits that execute the same computation will, henceforth, be referred to as the *original* and the *copy*. This method guarantees fault security against any fault that affects either the original or the copy, but not both. However, duplication and comparison *after synthesis* can lead to excessive overhead in circuit area. Moreover, having identical implementations for the original and copy leaves the circuit highly susceptible to common mode failures [15]. We use a variant of the duplication and comparison technique where we duplicate the CDFG. We compare the outputs of the original and copy CDFGs for error detection by inserting a sufficient number of *equal-to* comparison operations in the CDFG. We assume that these comparison operations are performed by totally self-checking (TSC) equality checkers and, hence, faults in the units implementing these comparisons need not be explicitly considered during behavioral synthesis. Depending on how the duplicated CDFG is scheduled, many equal-to comparison operations can share the same equality checker. The outputs of all the TSC equality checkers can be reduced to just two outputs using a TSC two-rail checker.

Securing of Operations and Edges: Securing of operations refers to a technique in which the outputs of some operations in the original part of the duplicated CDFG are compared with the results of the corresponding operations in the copy. The operations whose results are compared are referred to as *secured* operations. Securing of operations has been used to ease scheduling constraints and create additional opportunities for resource sharing [7]. In the sections that follow, we use the term "secured edge" to mean an edge whose source node is secured.

Example 1. In Fig. 1, we have a CDFG that computes the sum of three primary inputs, a_1 , a_2 , and a_3 . Two copies

Fig. 1. Securing operations for error detection.

of the CDFG are executed and the results are compared. The operation in the copy, corresponding to operation Op in the original, is labeled as Op_c . In this example, nodes $n1$ and $n2$ are secured. The (=) nodes denote comparison operations. The outputs produced are the primary output Out and the comparison outputs $check1$ and $check2$. If the adder executing operation $n1$ fails, an error is flagged at $check1$ and $check2$. If the adder which executes $n2$ fails, an error is flagged at $check2$.

Aliasing: If a fault causes the circuit to fail in such a manner that the outputs of the original and the copy are equal, but erroneous, *aliasing* is said to have occurred.

Example 2. Consider the circuit of Fig. 1 again. Let us suppose that we had only one comparison operation, which compares the outputs of nodes $n2$ and $n2_c$. Also, assume that $n1$ and $n2_c$ are mapped to the same functional unit. If this functional unit fails, then aliasing occurs if the inputs of the comparison operation are equal but erroneous.

Given a duplicated CDFG and an RTL circuit that implements it, Observation 1 below gives the necessary condition for aliasing to occur due to a fault in any functional unit.

Observation 1. *Aliasing can occur only if there exists a pair of operations $n1$ in the original and $n2_c$ in the copy that are performed by the same functional unit such that there exist unsecured paths from both $n1$ and $n2$, the node that corresponds to $n2_c$ in the original, to the same primary output, PO, or a loopout,¹ L , in the duplicated CDFG.*

Definition 1. A set S_SET of secured operations is said to separate two sets of operations A and B in the duplicated CDFG which are mapped to the same functional unit if, upon securing of each of the operations in S_SET from the CDFG, the necessary condition for aliasing presented in Observation 1 is not satisfied for any pair of elements $a \in A, b \in B$.

Aliasing is undesirable for fault security. One way to avoid aliasing is to impose the following constraint during resource sharing: Operations in the original and the copy

cannot be assigned to the same functional unit. This defaults to the approach of duplication and comparison after behavioral synthesis. To illustrate this approach, consider the duplicated scheduled CDFG shown in Fig. 2. The operations in the CDFG are annotated with the names of the functional units that perform them. From the figure, we can see that four multipliers, two adders, and one equality checker are required to implement the duplicated CDFG.

A further refinement of the above idea presented in [7] is the following: Nodes in the original and the copy which are not separated by secured operations cannot share a functional unit. The above constraint was imposed in [7] by first scheduling the original CDFG, then “delineating” the duplicated CDFG into several “regions,” and, finally, scheduling the regions in the copy. Operations from the original and copy that belonged to different regions were allowed to be assigned to the same functional unit. These constraints were incorporated into an assignment procedure in [8].

Example 3. Consider the duplicated CDFG in Fig. 3. This CDFG was synthesized while allowing sharing between the original and the copy. We can see that a fault in functional unit $M2$ could lead to an error in the original and the copy. However, if the extra equality checker $eq2$ is added, then the unsecured path between $n1$ and output Out is secured so that the necessary condition for aliasing is not satisfied. This ensures that any fault in functional unit $M2$ that leads to an error at the final output will be caught by $eq1$ or by $eq2$. A secured operation “cuts” a path from the nodes it separates to primary outputs and loopouts, thus preventing aliasing (see Observation 1). The insertion of the extra TSC equality checker saves a multiplier, which is a trade-off worth making.

Securing of operations can also be utilized to increase the scheduling freedom available to various operations in the CDFG and, hence, the opportunities for resource sharing [7], as shown next.

Example 4. Consider the transformed CDFG on the right in Fig. 4. Here, we make use of the same instance of operation $n3$ in both the original and the copy. The corresponding operation in the copy, $n3_c$, is now required to feed only the comparison operation with output $check2$ and, hence, the scheduling freedom available to operations in the transitive fanin of this operation is significantly enhanced. For instance, operation $n3_c$, which was required to complete in the second control step initially, can now be scheduled in control steps 2 or 3. Scheduling it in control step 3 reduces the number of required multipliers to implement the duplicated CDFG from two to one and the number of required adders from four to two at the expense of an extra TSC equality checker.

The decision of which operations to secure can significantly affect opportunities for scheduling and assignment in several ways. Moreover, the TSC equality checkers that implement the securing function themselves come with an associated cost and, hence, excessive use of operation securing can result in a suboptimal design. In effect, the

1. A loopout of a CDFG is an output which serves as an input to its next iteration.

Fig. 2. Duplicate and compare with no sharing allowed between copies.

Fig. 3. Securing of nodes to enable sharing between copies.

Fig. 4. Use of securing to enhance operation mobilities.

securing of intermediate operations interacts with the various behavioral synthesis tasks extensively. Previous work has ignored this interaction by performing the tasks separately. We present a procedure for dynamically securing intermediate operations that can be used during behavioral synthesis to explore the above trade-offs.

Previous work has focused on ensuring that aliasing does not occur in the synthesized circuit. As we shall illustrate later, enforcing this (pessimistic) condition may lead to excessive overheads in several cases. We demonstrate that it is possible to significantly lower the overhead required by performing an analysis of the probability of aliasing in the circuit. Our analysis bounds the probability of aliasing for various resource sharing configurations. The information provided by this analysis is used to perform behavioral synthesis such that the area of the resultant RTL circuit is as

low as possible while still maintaining a very low probability of aliasing.

Example 5. In the example of Fig. 5, we allow sharing between nodes in the original and copy even when there is no secured operation separating them. This is because, as illustrated in Section 4, the probability of aliasing, while not zero, can be shown to be extremely low. This saves one TSC equality checker compared to Fig. 3.

While the use of comparisons to secure intermediate results can lead to area reduction, securing is not always desirable. This is owing to the fact that sharing of a functional unit by multiple operations might require more than one operation to be secured and thus, potentially, require more than one TSC equality checker. Hence, the gain in area obtained by sharing may be nullified by the

Fig. 5. Sharing based on aliasing probability analysis.

extra equality checkers. Moreover, secured operations lead to extra edges in the CDFG, i.e., the number of data values that need to be transferred is increased. This often translates to an increase in the interconnect. As feature sizes shrink, interconnect area becomes an increasingly important component of circuit area. Finally, using securing of operations to ease scheduling constraints prolongs the lifetimes of the secured variables. This can lead to an increase in the number of registers that could offset the gains due to fewer functional units.

The above arguments suggest that the decision of which variables to secure should be considered as an integral part of behavioral synthesis and its interaction with the other behavioral synthesis tasks cannot be neglected. The complexity of the problem is such that an optimum solution cannot be guaranteed. Therefore, we use an iterative improvement-based behavioral synthesis framework that interleaves scheduling with functional unit selection and resource sharing, considering the interaction among these tasks. The algorithm also explores several clock periods and selects the one that yields an RTL circuit with the lowest area that meets the sampling period constraints. While our algorithms do not guarantee optimality, they are capable of escaping local minima through hill-climbing and, hence, yield high quality solutions in practice. We have augmented the behavioral synthesis framework by developing a dynamic procedure that selects the best operations to secure during synthesis. In order to better explore the design space, we allow multicycling, chaining, and functional unit pipelining. The use of aliasing probability analysis helps us to perform hardware sharing *without* comparators to secure intermediate results when the probability of aliasing in the resulting RTL circuit is extremely low and, thus, enables us to avoid all the associated overheads. Our method makes the RTL circuit secure against faults in registers and multiplexers in addition to functional units. We also make the controller secure against single faults.

3 BEHAVIORAL SYNTHESIS FRAMEWORK

We focus on data-dominated behavioral descriptions, as are common in the digital signal and image processing domains. Two important characteristics of such descriptions are: 1) They consist mainly of arithmetic operations

like addition, subtraction, multiplication, and delay operators, and 2) the rate at which the design should process input samples is typically fixed, i.e., it is crucial to meet the input sample rate requirement, but it does not pay to be able to process input samples any faster. Thus, we attempt to provide fault security while minimizing area under hard real-time performance constraints.

We have incorporated our techniques into the framework of an existing behavioral synthesis system which is based on iterative improvement [16], which has been used with great success in the area of high-level synthesis [17]. A related methodology, based on simulated annealing, has also been published [18]. The synthesis methodology is outlined in this paper and the parts that deal with incorporating fault-security are detailed. An overview of the behavioral synthesis framework for fault-secure data paths, called ALPS, is given in Fig. 6. ALPS accepts as input a CDFG and a constraint on the input sample period. The CDFG is first duplicated and the operations that compute primary outputs and some intermediate operations are secured with a view of minimizing the number of TSC equality checkers in the RTL circuit. The algorithm explores the clock period space by considering a subset of those candidate clock periods that divide the given sample period constraint evenly. The fastest clock period cannot be any less than the register-to-register delay of the fastest functional unit in the functional unit library. Further pruning in the clock period space is performed based on the delays of various components in the library. Details and the rationale behind the clock period pruning method we use can be found in [16]. For each clock period, ALPS first derives an initial solution that meets the given sample period constraint. It then searches for a sequence of moves (as opposed to a single move) that maximizes the cumulative improvement in the quality of the solution. This improvement is measured by a quantity called *Gain*. In our case, since we want to optimize the area, the gain is the reduction in area. At any point, we choose the move that gives the maximum improvement in the gain. Note that, at some points, there might exist no moves which can improve the quality of the solution. In this case, we pick the move which causes the least degradation in solution quality. A move which temporarily degrades the solution quality might present opportunities for the application of moves which improve the overall solution quality. The various

```

ALPS (CDFG  $G$ , Sample Period  $T_s$ , Library  $L$ ){
  CDFG  $G' \leftarrow \text{replicate}(G)$ ; //Duplicate the CDFG
   $G' \leftarrow \text{secure}(G)$ ; //Secure primary outputs and loopouts
   $Best\_DP \leftarrow \Phi$ ;
   $Cur\_DP \leftarrow \Phi$ ;
  for ( $csteps = MAX\_CSTEPS$ ;  $csteps \geq MIN\_CSTEPS$ ;  $csteps = csteps - 1$ ){
    if (CLK_PRUNE( $G'$ ,  $L$ ,  $csteps$ ))
      continue;
    //Find minimum area data path for current clock
     $Cur\_DP \leftarrow \text{INITIAL\_SOLUTION}(G', L, csteps)$ ;
    ITERATIVE_IMPROVEMENT( $G'$ ,  $L$ ,  $Cur\_DP$ );
    if (AREA_EST( $Cur\_DP$ ) < AREA_EST( $Best\_DP$ ))
       $Best\_DP \leftarrow Cur\_DP$ ;
  }
}
ITERATIVE_IMPROVEMENT(CDFG  $G$ , Library  $L$ , Datapath  $DP$ ){
   $Gain = 0$ ;
  while ( $Gain \geq 0$ ){
     $Gain\_list = \text{empty\_list}$ ;
    for ( $i = 1$ ;  $i \leq MAXMOVES$ ;  $i++$ ){
       $Gain(i) = \text{GENERATE\_FAULT\_SECURE\_MOVE}(G, L, DP)$ ;
      Append  $Gain(i)$  to  $Gain\_list$ ;
    }
    Find subsequence  $Gain(1), Gain(2), \dots, Gain(k)$ 
    in  $Gain\_list$  so that  $G = \sum_{i=1}^k Gain(i)$  is maximized
    if ( $G > 0$ )
      Accept moves  $1, \dots, k$ ;
  }
}

```

Fig. 6. Overview of our fault-secure datapath synthesis method.

moves that we explore are described in Section 3.1. While exploring moves that affect resource sharing, we use our aliasing probability analysis method to ensure that the aliasing probability as a result of the move is very low. The working of this algorithm is illustrated through an example in Section 6.

3.1 Moves in the Iterative Improvement Procedure

In order to explore the functional unit selection, scheduling, and resource sharing search spaces, we define three types of moves for our algorithm: class *A*, *B*, and *C*. At every stage of iterative improvement, the best *A*, *B*, and *C* moves that can be applied to the solution are determined. The best move is picked at each stage, irrespective of type. Note that the best move can have a negative gain, i.e., it could degrade overall solution quality. The ability to select and apply such moves is a key feature of our iterative improvement algorithm and enables it to escape local minima.

Moves of Class A transform the data path by replacing an instance of a functional unit $t1$ by a functional unit $t2$.

For example, the replacement of a *ripple_carry_adder* by a *carry_lookahead_adder* is a move of class *A*.

Moves of Class B either replace two functional units $fu1$ and $fu2$ by a single functional unit fu , or split a single functional unit into two separate functional units. For merging of functional units, the following three conditions should be satisfied:

- C1. There exists a functional unit which can execute operations executed by both $fu1$ and $fu2$.
- C2. There exists a schedule which makes the sharing feasible. In general, sharing between two functional units might involve a reschedule of the operations mapped to $fu1$ and $fu2$.
- C3. The probability of aliasing due to a fault in the merged functional unit is extremely low.

The manner in which condition C3 is checked is the topic of Section 4. If this condition is violated, then some intermediate results are appropriately secured to reduce the probability of aliasing to acceptable levels. Operations are secured in such a manner that the nodes in the original,

Fig. 7. Design of the controller.

which are assigned to a functional unit, and some other nodes in the original, which correspond to those in the copy, that are assigned to the same functional unit are separated by secured operations. The procedure for selecting which intermediate operations to secure is presented in Section 5.1. Figs. 2 and 3 illustrate the application of a move of class *B*. Here, two operations, one from the original and the other from the copy, which were initially assigned to different functional units, *M2* and *M4*, are now assigned to the same functional unit, *M2*. It can be easily seen that the operations satisfy conditions C1 and C2. Our aliasing probability analysis techniques presented in Section 4 reveal that condition C3 is also satisfied in this case.

Moves of Class C secure a specific operation *Op* with an aim of easing the scheduling constraints on operations in the duplicated CDFG. Moreover, all the fanouts of *Op*, as well as of the corresponding operation in the copy, *Op_c*, are now fed by *Op*. While a move of class *C* in itself may not lead to a decrease in area, it can enable other subsequent moves of classes *A* and *B* due to the extra scheduling freedom of the operations in the transitive fanin of *Op_c* (including *Op_c*). Fig. 4 is an example application of a move of class *C* followed by a reschedule. In the figure, securing of operation *n3* enables rescheduling of operations *n1_c*, *n2_c*, and *n3_c* in the copy.

3.2 Fault Security for the Controller

Fig. 7 shows the architecture of the fault-secure controller that we synthesize. An FSM specification for the controller is generated based on the schedule and assignment information after data path synthesis. Logic synthesis tools are then used to generate an implementation. The following constraints are imposed on the logic synthesis process in order to result in a fault-secure implementation. During state encoding, all valid controller states are constrained to have the same parity [14]. The outputs are also constrained to be encoded with the same parity with the help of an extra output. The combinational logic of the controller is constrained such that no logic sharing is performed among

its various output cones. The present-state lines and the controller outputs are fed to a TSC parity checker. A fault within any one of the logic cones or a single fault at any state flip-flop or a single fault in the TSC parity checker itself is detected by the TSC parity checker. The outputs of this checker can be combined with the outputs of other equality checkers in the datapath through a TSC two-rail checker. Thus, only two extra pins are required for fault tolerance purposes.

4 ALIASING PROBABILITY ANALYSIS

In this section, we describe our aliasing probability analysis procedure which is used to identify additional resource sharing opportunities that do not significantly compromise fault security. The basic features of our procedure are as follows:

- Aliasing analysis tries to determine whether a given resource sharing configuration can cause aliasing. Its inputs are a duplicated CDFG, a resource sharing configuration, and a faulty module and the output of the procedure is a *yes* if the configuration can result in aliasing with a high probability and *no* if the probability of aliasing is extremely low.
- The primary inputs to the circuit are assumed to be independently distributed. Note that the inputs to individual functional units and multiplexers might be correlated.
- Our fault model is general. A fault can affect the functionality of a module in any arbitrary manner. If the aliasing analysis procedure returns a *no*, then the probability of aliasing is extremely low, *independent of the manner in which the fault affects the functionality of the module*.

Error Model: Consider a functional unit *fu* that is affected by a fault *F*. Let one of the operations performed by *fu* be *I1 Op I2*, where *I1* and *I2* are variables and *Op* is an operation in the duplicated CDFG. Then, the output

Fig. 8. Effect of single faulty node.

produced by fu under the influence of the fault is modeled by the equation:

$$Output(fu) = (I1 \text{ Op } I2) + f(I1, I2). \quad (1)$$

Note that the above additive error model does not forsake any generality since no assumption is made about the nature of the error function $f(I1, I2)$. No assumption is made about the nature of $\mathcal{F}$ either (i.e., we do not assume any specific fault model). We, however, do assume that the fault is combinational in nature, i.e., the effect of the fault does not change over time. If a functional unit performs several operations, the outputs of any arbitrary subset of these operations may be erroneous.

4.1 The Aliasing Equation

Our aliasing analysis procedure is based on the derivation of the *aliasing equation* that captures the conditions which need to be met for aliasing to occur. In this section, we explain how to derive the aliasing equation for a duplicated CDFG, given the candidate set of operations that we desire to assign to the same functional unit. A separate aliasing equation is derived for each primary output and loopout of the CDFG.

The aliasing equation is derived by expressing the value of each primary output and loopout of the original and the copy in terms of the primary inputs and the error function. The error function captures the effect of a fault in a functional unit on the results of the operation(s) it performs. The effect of the faulty nodes in the original and copy CDFGs appears at their respective primary outputs or loopouts.

4.1.1 Effect of Single Faulty Node

In this section, only one node in the CDFG is assumed to be faulty. We evaluate the error due to the fault in that node at a primary output or loopout. The procedure is illustrated by the following example:

Example 6. Consider the duplicated CDFG in Fig. 8.

Suppose the functional unit that executes addition operation $n1$ is faulty and $n1$ is the only operation that is performed by it. Assuming that all other operations are executed by fault-free functional units, we can find the value at the primary output as shown next. In the equations that follow, n_i stands for the output value of operation n_i . The values at the nodes are:

$$\begin{aligned} n1 &= a1 + a2 + f(a1, a2) \\ n2 &= a1 + a2 + a3 + f(a1, a2) \\ n3 &= a1 + a2 + a6 + f(a1, a2) \\ n4 &= a5(a1 + a2 + a6 + f(a1, a2)) \\ n5 &= (a1 + a2 + f(a1, a2))(a1 + a2 + a3 + f(a1, a2)) \\ n6 &= (a1 + a2 + f(a1, a2))(a1 + a2 + a3 + f(a1, a2)) + a4 \\ n7 &= (a1 + a2 + f(a1, a2))(a1 + a2 + a3 + f(a1, a2)) \\ &\quad + a4 + a5(a1 + a2 + a6 + f(a1, a2)). \end{aligned}$$

The error at the primary output can be calculated by subtracting the expression for the error-free output from that for the erroneous output. In this case, the error turns out to be $f^2(a1, a2) + f(a1, a2)(2a1 + 2a2 + a3 + a5)$.

The error at the primary output due to a faulty node n_i may not depend on the edges that feed paths from the erroneous operation to the output at *add* or *subtract* operations. This is because these terms may get cancelled out when subtracting the expression for the error-free output from that for the erroneous output. For example, consider the error at $n7$ in Example 6. The error does not depend on primary input $a6$ which feeds the add operation, $n3$, on the path from erroneous operation $n1$ to the output of operation $n7$.

4.1.2 Effect of Multiple Faulty Nodes

In this section, we derive an expression for the error at a primary output or loopout when multiple operations are affected by a fault in the functional unit that they are

Fig. 9. Effect of multiple faulty nodes.

assigned to. The calculation is illustrated by the following example:

Example 7. Consider the example CDFG shown in Fig. 9. In the CDFG, the faulty addition operations are shaded. We evaluate the error at the primary outputs, $n9$ and $n9_c$. The outputs of the faulty nodes are:

$$\begin{aligned} n1 &= a1 + a2 + f(a1, a2), \\ n2_c &= a4 + a5 + f(a4, a5), \\ n3_c &= a6 + a7 + f(a6, a7). \end{aligned}$$

The expressions for the errors at the primary outputs are:

$$\begin{aligned} \text{error at } n9_c &= f(a4, a5)(1 + a6 + a7) \\ &\quad + f(a6, a7)(a4 + a5 + 1 + f(a4, a5)) \end{aligned}$$

and $\text{error at } n9 = a9f(a1, a2)$. Aliasing would occur if these two errors are equal.

Observation 2. The error at a primary output or loopout can be written as a polynomial in the error functions of the faulty operations and the primary input variables. Mathematically, the expression can be written as follows:

$$\text{Error} = \sum_{i=1}^{T'} \prod(i), \quad (2)$$

where T' is the number of terms and $\prod(i)$ is the i th product term, that is, a product of some primary inputs and some error functions.

For the last example, for the error at $n9_c$, $\prod(1) = 1.f(a4, a5)$, $\prod(2) = a6f(a4, a5)$, and so on.

4.1.3 Summary

Now that we can find the error at a primary output or loopout in terms of the errors at the faulty nodes, we can equate the error at each such output/loopout in the original with the error at the corresponding output/loopout in the copy to obtain the aliasing equation as follows:

$$\begin{aligned} &(\text{Error due to nodes in } S' \text{ at the output}) \\ &- (\text{Error due to nodes in } S'' \text{ at the output}) = 0, \end{aligned} \quad (3)$$

where S' is a set of nodes in the original and S'' is the set of nodes in the copy that are assigned to the faulty functional unit.

4.2 Aliasing Conditions

Our aliasing analysis procedure consists of deriving the aliasing equation and then analyzing it to draw conclusions about the probability of it being satisfied. Exactly evaluating the probability of aliasing requires information about the nature of the error functions (which in turn depends on the fault model) and the joint probability density function (PDF) of the input variables. In general, such information may not be available for several applications and, even when available, the computational requirements for performing an exact analysis may be prohibitive (e.g., if multiple stuck-at fault model is assumed). Hence, we have developed methods to compute an *upper bound* for the aliasing probability. We present two conditions (*Condition 1* and *Condition 2*) that can be used as tests to check whether the probability of aliasing is low. If either test succeeds, we can share a functional unit between the candidate operations. Initially, we assume that all primary input variables of the CDFG are uniformly distributed in their entire range and uncorrelated. Later, we show how arbitrary distributions can be considered. A generalization of the aliasing

conditions concludes this section. We discuss the two conditions next.

4.2.1 Condition 1

In this subsection, we describe a condition, called Condition 1, under which sharing can be performed with a very low aliasing probability. The test for this condition hinges on the following result:

Result 1: If there exists a primary input PI on which none of the error functions depend (i.e., which does not appear in the argument of any error function), and if a subset $R1$ of the product terms in the aliasing equation depends on PI , while a subset $R1'$ does not, satisfaction of the aliasing equation implies that, with a very high probability, the following two subequations are satisfied:

$$\sum \text{Product terms in } R1 = 0 \quad (4)$$

$$\sum \text{Product terms in } R1' = 0. \quad (5)$$

If, as a result of the recursive application of Result 1, each product term in the aliasing equation becomes 0, Condition 1 is said to be satisfied. The rest of this section proves Result 1 and describes an algorithm to check for Condition 1.

The aliasing equation can be written as:

$$\sum_{i=1}^T \prod(i) = 0. \quad (6)$$

Consider again the duplicated CDFG of Fig. 9. The aliasing equation for this CDFG when the shaded operations are faulty is as follows:

$$(a9f(a1, a2)) - (f(a4, a5) + a6f(a4, a5) + a7f(a4, a5) + f(a6, a7) + a4f(a6, a7) + a5f(a6, a7) + f(a4, a5)f(a6, a7)) = 0.$$

In this case, we can see that the number of product terms, T , is equal to 8.

Consider a primary input ai that is a factor of a proper subset of the product terms and does not appear in the argument of any error function. We rewrite the aliasing equation, abstracting all the terms other than ai into coefficients C_j . In other words, we reduce the aliasing equation to a polynomial in ai as given below.

$$\sum_{j=0}^k C_j ai^j = 0. \quad (7)$$

Example 8. Consider the running example CDFG of Fig. 9.

We know that product term $\prod(1)$ depends upon $a9$ and that $a9$ does not appear at any other place in the aliasing equation. Hence, the aliasing equation can be written as $C_0 + a9.C_1 = 0$. All the other terms have been absorbed into coefficients C_0 and C_1 .

If at least one coefficient is nonzero, then the probability that a randomly chosen value of ai will solve the equation is upper bounded by $k/(2^N)$, where N is the bit-width of the

data path and k is the degree with respect to ai of the aliasing equation. (This is because (7), being a polynomial in ai , has at most k roots. The probability that a randomly chosen integer in the range $[0, 2^N - 1]$ will satisfy the equation is upper bounded by $k/(2^N)$. The actual probability could be much less because (7) may not have any integer roots, in which case the probability of the equation being satisfied by ai is 0.) For a data path which has a bit-width of 32, for example, this is negligible. The above analysis has shown that the aliasing probability is very small if at least one of the coefficients C_j is nonzero. We next analyze the aliasing probability when the assumption (at least one of the coefficients is nonzero) does not hold. To see what this means, let us apply the condition to our running example. In the following analysis, we refer to the probability of aliasing as $P_{aliasing}$.

Example 9. If $C_0 \neq 0$ or $C_1 \neq 0$, then, from the above arguments, $P_{aliasing} = 1/2^N$.

We now consider the case where $C_0 = C_1 = 0$. This reduces to the following subequations:

$$\begin{aligned} (1 + a6 + a7)f(a4, a5) + (1 + a4 + a5)f(a6, a7) \\ + f(a4, a5)f(a6, a7) = 0 \\ f(a1, a2) = 0. \end{aligned}$$

Both of these subequations must be satisfied for aliasing to occur. The aliasing analysis procedure is recursively called on these two subequations. Therefore, the $P_{aliasing}$ of each subequation bounds the $P_{aliasing}$ of the whole equation.

In general, the primary input ai can be looked upon as inducing a *partition*² on the product terms of the aliasing equation. One block of the partition consists of product terms which depend on ai and in the other block are product terms which do not depend on ai . The product of all these partitions gives a set of aliasing subequations. If each subequation consists of a single product term, then, for the aliasing equation to have a solution, each of the product terms in the aliasing equation must be zero. Note that each product term in the aliasing equation represents an additive error at a primary output or loopout. Therefore, if all the product terms are zero, it implies that the error at the primary output or loopout is zero and aliasing does not occur.

We illustrate recursive testing of the aliasing condition using the following example:

Example 10. Consider the aliasing equation given below.

$$a1a2a3f(a1, a2) + a1f(a2, a1) + a4f(a2, a1 + a2) = 0.$$

2. A partition, P , is a structure which is defined on a universal set $U = \{e1, e2, \dots, er\}$ and consists of several disjoint subsets $S1, S2, \dots, Sm$ such that $\cup_{i=1}^m Si = U$. The product of two partitions, $P1$ and $P2$, which have the same universal set U is another partition $P3$ which has the following property: Disjoint subsets $A1, \dots, An$ which comprise $P3$ are such that if any two elements $ei, ej \in U$ are in the same subset Ai of $P3$, then the two elements are in the same subset in both $P1$ and $P2$. By definition, $Zero_Partition = \{\{e1\}, \{e2\}, \dots, \{er\}\}$ and $Identity_Partition = \{e1, e2, \dots, er\}$. For example, if $U = \{a1, a2, a3, a4\}$, $P1 = \{\{a1, a2\}, \{a3, a4\}\}$, and $P2 = \{\{a1, a3\}, \{a2, a4\}\}$, then the product of $P1$ and $P2$ is $\{\{a1\}, \{a2\}, \{a3\}, \{a4\}\}$, which is the $Zero_Partition$.

```

TEST_CONDITION1 (ALIASING EQUATION Al_Eq)
{
    partition_product = Identity_Partition;
    for each product term in Al_Eq
        for each primary input PI in the product
            if (PI  $\notin$  support set of any error function)
                {
                    Partition(PI) = ({Terms containing PI}, {Terms not containing PI});
                    partition_product = PRODUCT(partition_product, Partition(PI));
                }
    return (partition_product == Zero_Partition);
}

```

Fig. 10. Test for Condition 1.

Here, $\prod(1) = a1a2a3f(a1, a2)$, $\prod(2) = a1f(a2, a1)$, and $\prod(3) = a4f(a2, a1 + a2)$, where $a1$, $a2$, $a3$, and $a4$ are primary inputs. $\prod(1)$ is the only product term that depends on $a3$. Moreover, the error functions do not depend on $a3$. Hence, the aliasing equation splits into the following two subequations: $a1a2f(a1, a2) = 0$ and $a1f(a2, a1) + a4f(a2, a2 + a1) = 0$.

It turns out that we can apply Condition 1 to the second subequation. Here, we see that the second term, $\prod(3)$, has the primary input $a4$ unique to it. So, this induces a partition on the subequation which leaves us with three distinct subequations: $a1a2f(a1, a2) = 0$, $a1f(a2, a1) = 0$, $f(a2, a2 + a1) = 0$. If a product term is 0, then one of its constituent factors must be zero. We already know that a product term is a product of some primary inputs and some error functions. The probability of a primary input being zero is $1/2^N$. This implies that, given that a product term is 0, with a high probability one of the error functions in it must be zero. (If there are no primary inputs in the product term, then one of the error functions which constitute the product term is guaranteed to be zero). From the above, it follows that all three error functions in our example are zero with a very high probability. However, this implies that all primary outputs assume their error-free values, in which case, by definition, aliasing cannot occur. Thus, we can conclude that $P_{aliasing} \leq 1/2^N$.

The pseudocode for our procedure that tests for Condition 1 is given in Fig. 10. In this code, the support set of an error function is defined as the set of all primary inputs that it depends on.

To derive the above result, we assumed that all the primary inputs are uniformly distributed. We now show how to evaluate $P_{aliasing}$ for nonuniform distributions. Let primary input ai satisfy Condition 1, i.e., it does not appear in the support of any error function and influences the coefficients of a subset of the error functions. Suppose ai is nonuniformly distributed and the most probable value of ai has a probability of $Peak(ai)$. Aliasing occurs when a randomly chosen value of ai satisfies the aliasing equation.

The number of solutions to the aliasing equation is bounded by k where k is the degree of ai in the aliasing equation. Hence, $P_{aliasing}$ is bounded by $k \times Peak(ai)$, which corresponds to a case when all roots of the equation have a probability of $Peak(ai)$.

Graph-theoretically, Condition 1 can be stated as follows: If a primary input, which is not in the transitive fanin of any of the faulty nodes, feeds a path from any faulty node to a primary output at a multiply operation, then the aliasing equation can be split into subequations. For instance, in the CDFG in Fig. 9, $a9$ does not appear in the transitive fanin of any error function, but feeds the path from $n1$ to *Out* at operation $n7$. Whatever values the error functions take, satisfaction of the aliasing equation requires, in some sense, the *knowledge* of $a9$ which the faulty nodes do not possess.

4.2.2 Condition 2

Our second test for aliasing is based on performing a set of variable transformations to recast the aliasing equation into a form where it is more amenable to an analysis similar to that for Condition 1. Condition 2 uses the following result:

Result 2: If there exists any pair of primary inputs, (am, an) , in the aliasing equation which satisfies the following conditions: 1) If the argument of any error function in the aliasing equation depends on either of them, the dependence is by means of a *common subexpression CS*, involving am and an , and 2) am influences the coefficients of some of the error functions in the aliasing equation, but an does not, then the following holds: Satisfaction of the aliasing equation implies, with a very high probability, that the terms in the aliasing equation in which coefficients of error functions depend on am must sum to 0 and the rest of the terms in the aliasing equation must also sum to 0.

If Result 2 can be recursively utilized to prove that each term of the aliasing equation is 0, Condition 2 is said to be satisfied. The rest of this section is devoted to a proof of Result 2. The algorithm to check for Condition 2 is shown in Fig. 11.

Each of the product terms in the aliasing equation given in (6) can be written in the following form:

```

TEST_CONDITION_2 (CDFG  $G$ , FAULTY_NODE_SET  $F_s$ , ALIASING_EQUATION  $A_{eq}$ )
{
  for each Primary input  $PI$  which appears in a product term
  {
    if (SUB_TEST( $PI$ ) == True)
    {
      for each primary input  $PI'$  not in
        any  $\prod(i)$  but in the support set of some faulty node
      {
        if (SUB_TEST( $PI'$ ) == True)
          Partition_induce( $PI$ );
      }
    }
  }
  for each primary input
    partition_product = Product(partition_product, Partition( $PI$ ));
  return (partition_product == zero);
}
SUB_TEST(primary input  $PI$ )
{
   $D = \text{dominator}(PI, PO)$  //Compute the dominatora of  $PI$  with respect to output  $PO$ 
  if  $D$  precedes each faulty node
    return true;
  if  $D$  precedes no faulty node
    return true;
  return false;
}

```

^aThe dominator of a primary input PI with respect to a primary output PO is the node of smallest level in the CDFG through which all paths from PI to PO pass

Fig. 11. Test for Condition 2.

$$\prod(i) = \left(\prod_j a_j^{p_j} \right)_i \times \left(\prod_k f_k^{p_k} \right)_i. \quad (8)$$

The product in j is taken over all the primary input factors in a product term, while the product in k is taken over all error function factors. The variables p_j and p_k are the powers of the primary input a_j , and error function f_k , respectively. (The subscript i alongside the brackets indicates that the corresponding product terms are derived from $\prod(i)$). *Support_Set*(i) is defined to be the set of primary inputs that occur in $(\prod_j a_j^{p_j})_i$.

Consider two distinct primary inputs, am and an such that, whenever am and an appear in the argument of any error function, they are part of a common subexpression, $CS(am, an)$. Moreover, suppose that am is part of *Support_Set*(i) for some i . In addition, we also require that an does not belong to *Support_Set*(i) for any i . Under the above assumptions, we prove that variable am splits the aliasing equations into two aliasing subequations, both of which must be satisfied for aliasing to take place.

Example 11. Consider the duplicated CDFG shown in Fig. 12. In this figure, the shaded nodes, $n3$ and $n4_c$ are faulty. We would like to analyze the aliasing equation that results when they are assigned to the same functional unit. The aliasing equation is:

$$a1a2f(a2a3, a4) - f(a2a3, a1) = 0.$$

In this equation, $T = 2$, $\prod(1) = a1a2f(a2a3, a4)$, $\prod(2) = -f(a2a3, a1)$, and *Support_Set*(1) = { $a1, a2$ }. We can see that $(a2, a3)$ is a possible candidate for (am, an) and $CS(a2, a3) = a2a3$.

Suppose that we are given arbitrary values for: 1) all primary inputs except am and an , and 2) CS .

Under the above constraints, let v be a permissible value for CS . The aliasing equation reduces to a polynomial in am and can be solved to obtain possible values of am for which aliasing occurs. Assuming that at least one of the coefficients in this polynomial is nonzero, the number of solutions is bounded by p_m , the degree of the polynomial in am (we later address the case when all the coefficients of the

Fig. 12. Application of Condition 2.

polynomial are zero). For each of the p_m values that am can take, CS becomes a polynomial in an , which can have at most p_n solutions, where p_n is the power of the polynomial in an . Thus, overall, there are at most $p_m \times p_n$ solutions to the aliasing equation for a given value v for CS . In the above example, $am = a2$ and the aliasing equation reduces to the linear equation, $a2 \times C_1 - C_0 = 0$. In this case, given a value of $CS = a2a3$, there is at most one pair of values for $(a2, a3)$ that satisfies the aliasing equation.

Note that the above arguments were made under the condition that CS was fixed to a value v . It is possible that there might be several assignments to am and an that result in CS assuming the value v . Only $p_m \times p_n$ of these result in aliasing. Therefore, the probability of aliasing, given that $CS = v$, is given by $p_m \times p_n / (\text{number of solutions to } CS = v)$.

Since in the above analysis we assumed a single value v for CS , summing the conditional probability over all possible values of CS will result in the probability of aliasing. Let us suppose that $S(v)$ represents the number of solutions to the equation $CS(am, an) = v$. The probability of aliasing can then be derived as follows:

$$E_v = \text{Event where value of } CS \text{ is } v$$

$$A = \text{Event that the aliasing equation is satisfied}$$

$$P_{\text{aliasing}} = P(A) = \sum_{\text{All } E_v} P(E_v) \times P(A/E_v).$$

(9)

The probability of event E_v is equal to $S(v)/(2^{2N})$ since the pair of primary inputs (am, an) can assume 2^{2N} values with equal probability of which only $S(v)$ values result in event E_v . We have previously evaluated $P(A/E_v)$ to be $p_m \times p_n / S(v)$. Therefore,

$$P(A) = \sum_{\text{All } E_v} (S(v)/(2^{2N})) \times (p_m \times p_n / S(v)) \quad (10)$$

$$\begin{aligned} &\leq 2^N \times p_m \times p_n / (2^{2N}) \\ &\leq p_m \times p_n / (2^N). \end{aligned} \quad (11)$$

Now, we address the case where all the coefficients of powers of am in the aliasing equation are zero. In this case, am can be viewed as splitting the aliasing equation into at most $p_m + 1$ distinct subequations, one for each power of am . These subequations can then be recursively analyzed, as in the case of Condition 1.

If am and an are nonuniformly distributed with the most probable values having probabilities $Peak(am)$ and $Peak(an)$ of occurring, respectively, then the analysis is as follows:

$$P(A) = \sum_{\text{All } E_v} P(E_v) \times P(A/E_v)$$

$$P(E_v) = P(CS = v)$$

$$= \sum_{\text{All } v_{am}, v_{an} : CS(v_{am}, v_{an}) = v} P(v_{am})P(v_{an})$$

$$P(A/E_v) =$$

$$\sum_{\text{All } \{v_{am}, v_{an}\} : \text{aliasing equation satisfied given } E_v} P(v_{am})P(v_{an})$$

$$\leq p_m p_n Peak(am) Peak(an) / P(E_v)$$

$$P(A) \leq p_m p_n Peak(am) Peak(an) (2^N).$$

(12)

Note that $Peak(am)$ and $Peak(an)$ evaluate to $1/2^N$ for the case of a uniform primary input distribution.

Condition 2 is more difficult to test than Condition 1, but is common in the case of CDFGs with heavy reconvergence. Condition 2 can be stated as follows: If there exists a node n in the CDFG with two primary inputs, am , an , in its transitive fanin such that all paths from am and an to any faulty node pass through n , and if exactly one of the two nodes feeds a path from a faulty node to a primary output at a multiply operation, then the aliasing equation can be split into aliasing subequations. In the CDFG in Fig. 12, node $n1$ is n . Condition 2 can be easily understood as a variable transformation wherein the fanout edge of n becomes a primary input, taking the place of an . Now, Condition 1 can be applied to the CDFG obtained because am does not appear in the transitive fanin of any faulty node.

4.2.3 Generalization of Aliasing Conditions

The aliasing conditions were proven in the following manner: We tried to see if the outputs of the faulty operations could provide us with some information about the inputs to the operations and, hence, about the primary inputs. With this information, we tried to see how hard it was to "guess" correct values for primary inputs so as to make aliasing occur. The extent of our difficulty provided us with an upper bound on the probability of aliasing. The similarity of the reasoning in the two cases suggests a more general aliasing condition of which Conditions 1 and 2 are special cases. In this section, we explore this generalization. The generalized aliasing condition is stated as follows:

Result 3: If there exists any set of primary inputs, $Sp = \{a1, a2, \dots, ap\}$, in the aliasing equation, which satisfies the following conditions: 1) If the argument of any error function in the aliasing equation depends on any member of the set, the dependence is by means of a *common subexpression* CSp , involving all elements of Sp , and 2) the coefficients of the error functions in the aliasing equation are influenced only by a proper subset, $Sp' = \{a1, a2, \dots, aj\}$, of Sp , then the following holds: Satisfaction of the aliasing equation implies, with a very high probability, that the terms in the aliasing equation in which coefficients of error functions depend on members of Sp' must sum to 0 and the rest of the terms in the aliasing equation must also sum to 0.

Proof. Let A denote the event that aliasing occurs. The probability of aliasing is given by

$$P_{aliasing} = P(A) = \sum_{i=0}^{2^N-1} P(CSp = i) \times P(A/CSp = i). \quad (13)$$

Let $Sp'' = Sp - Sp'$, i.e., the set of elements in Sp which do not belong to Sp' . Let m_i denote the number of p -tuples $\{v1, v2, \dots, vp\}$ such that $CSp(v1, v2, \dots, vp) = i$. Then, the probability that $CSp = i$ is given by

$$P(CSp = i) = m_i / (2^{Np}). \quad (14)$$

The probability of aliasing can be upper bounded in the following manner: There exist at most $2^{N(j-1)} \times Max_deg(Sp') \times 2^{N(p-j-1)} Max_deg(Sp'')$ solutions to the aliasing equation, given that $CSp = i$, where $Max_deg(Sp')$ is the maximum degree of any member of Sp' in the aliasing equation and $Max_deg(Sp'')$ is the maximum degree of any member of Sp'' in CSp . This is because, assuming arbitrary values for all primary inputs other than $a1, a2, \dots$, and ap , and fixing values for the outputs of faulty operations, the aliasing equation depends only upon $a1, a2, \dots$, and aj . Therefore, there can exist at most $2^{N(j-1)} \times Max_deg(Sp')$ solutions to the aliasing equation which are j -tuples of the form $\{a1, a2, \dots, aj\}$. For each unique specification of these j variables, there are $2^{N(p-j-1)} \times Max_deg(Sp'')$ ways of specifying $\{a(j+1), a(j+2), \dots, ap\}$ so that both the aliasing equation and the condition on CSp are satisfied. Since there are m_i solutions to the equation $CSp = i$, we have:

$$P(A/CSp = i) \leq \frac{2^{N(j-1)} \times Max_deg(Sp') \times 2^{N(p-j-1)} \times Max_deg(Sp'')}{m_i}. \quad (15)$$

Then from (13), (14), and (15), we obtain:

$$\begin{aligned} P(A) &\leq \sum_{i=0}^{2^N-1} \frac{m_i}{2^{Np}} \\ &\times \frac{2^{N(j-1)} \times Max_deg(Sp') \times Max_deg(Sp'') \times 2^{N(p-j-1)}}{m_i} \\ &= \sum_{i=0}^{2^N-1} \frac{Max_deg(Sp') \times Max_deg(Sp'')}{2^{2N}} \\ &= \frac{Max_deg(Sp') \times Max_deg(Sp'')}{2^N}. \end{aligned} \quad (16)$$

The above analysis assumed the primary inputs of the CDFG to be uniformly distributed. However, this restriction can be easily relaxed, as in the case of Conditions 1 and 2. Let us assume that the primary inputs are independent, but nonuniformly distributed and the probability of the most probable value of primary input ai is $Peak(ai)$. As in the uniformly distributed case, our starting point is the equation

$$P(A) \leq \sum_{i=0}^{2^N-1} P(CSp = i) \times P(A/CSp = i),$$

where

$$\begin{aligned} P(CSp = i) &= \sum_{all\ V=\{v1, v2, \dots, vp\}: CSp(V)=i} \prod_{k=1}^p P(vk) \\ P(A/CSp = i) &= \frac{\sum_{all\ \{v1, v2, \dots, vp\}: aliasing\ occurs} \prod_{k=1}^p P(vk)}{\sum_{all\ \{v1, v2, \dots, vp\}: CSp=i} \prod_{k=1}^p P(vk)} \\ &\Rightarrow P(A) = \sum_{i=0}^{2^N-1} \sum_{all\ \{v1, v2, \dots, vp\}: aliasing\ occurs\ when\ CSp=i} \prod_{k=1}^p P(vk) \\ &\Rightarrow P(A) \leq \sum_{i=0}^{2^N-1} Max_solns(i) \times \prod_{k=1}^p Peak(vk), \end{aligned}$$

where $Max_solns(i)$ is the maximum number of solutions to the aliasing equation when $CSp = i$. From (15), it follows that

$$\begin{aligned} Max_solns(i) &\leq 2^{N(j-1)} \times Max_deg(Sp') \\ &\times 2^{N(p-j-1)} \times Max_deg(Sp''). \end{aligned}$$

Therefore,

$$\begin{aligned}
P(A) &\leq 2^N \times 2^{N(p-2)} \times \text{Max_deg}(Sp') \times \text{Max_deg}(Sp'') \\
&\quad \times \prod_{k=1}^p \text{Peak}(vk) \\
&= 2^{N(p-1)} \times \text{Max_deg}(Sp') \times \text{Max_deg}(Sp'') \\
&\quad \times \prod_{k=1}^p \text{Peak}(vk).
\end{aligned} \tag{17}$$

Note that (17) reduces to (12) if primary inputs are uniformly distributed. $\square$

4.2.4 Summary

The generalized aliasing condition is more comprehensive than Conditions 1 and 2, but is too expensive to implement in the inner loop of a synthesis algorithm. However, it could be used by a method that does static analysis of aliasing probabilities for selected configurations. The two tests which have been developed are not necessary conditions for aliasing to be extremely improbable. However, they are easy to implement and detect a significant fraction of cases where aliasing is extremely unlikely. In many of the cases in which the two tests failed, we could find error functions f which corresponded to single stuck-at faults which would result in aliasing with a high probability.

Conditions 1 and 2 test for the possibility of aliasing at a specific primary output or loopout. A fault in a single functional unit may result in aliasing at multiple outputs and a very low probability of aliasing at one output does not imply a very low probability of aliasing at other outputs. This is because a faulty operation need not cause an error to occur at the output of every operation it feeds. Hence, we have to apply the two tests to all outputs that are fed by operations mapped to the functional unit under consideration. Only if all primary outputs and loopouts pass the test can the candidate operations be mapped to the same functional unit.

4.3 Faults in Registers and Multiplexers

In this section, we discuss how register and multiplexer faults are taken care of.

Register faults: The previous section described aliasing analysis procedures for faults in functional units. However, it can be straightforwardly extended to cover register faults. Here, we would like to determine whether a set of variables in the original CDFG and another set of variables in the copy can share a register. As in the case of functional units, we express the erroneous value of a variable stored in the register as $V + f(V)$, where V is the fault-free value and $f(V)$ is an arbitrary error function. Any subset of variables assigned to a faulty register can get corrupted due to the fault. Therefore, some of the edges in the CDFG are faulty (as opposed to nodes in the case of functional unit faults). The procedure for deriving and analyzing the aliasing equation, however, remains similar to the case of faulty functional units.

A register which holds a variable v in the original cannot store the corresponding variable v_c in the copy. If it does, then a fault in the register could cause identical errors in the

value of v in both the original and the copy. This error cannot be detected by a comparator. This is entirely analogous to the condition that nodes that perform the same operation in the original and the copy cannot map to the same functional unit.

Multiplexer faults: Faults in multiplexers can be classified into those that affect data selection and those that do not affect data selection. Formally, we can characterize them as follows: Consider an n -to-1 multiplexer with data inputs $I_1, I_2, \dots, I_n$ and output O . In the first case, we can write the erroneous value at the multiplexer output as: $O_{err} = f(I_1, I_2, I_3, \dots, I_n)$. In the second case, the error is: $O_{err} = f(\text{selected data input})$. Our analysis techniques handle the second class of faults and a subset of the first class. We assume that faults that affect data selection could make the multiplexer select a wrong input, but the output must be a value present at one of the multiplexer inputs. The multiplexer is not permitted to select an incorrect input and alter it.

We use a point-to-point interconnect model. A fault that does not affect data selection is equivalent (from the point of view of aliasing analysis) to a fault in the register or the functional unit that feeds the multiplexer. We make the multiplexers secure against a restricted subclass of the other class of faults, namely, those faults that cause the multiplexer to select a data input that is different from the one that it is supposed to select. It can be shown that a multiplexer routing error causes the controller/data path circuit to behave in a manner that can be modeled by deleting some nodes and edges from the CDFG and adding different nodes and edges in their place. The aliasing equation is derived by equating the expressions for primary outputs and loopouts in the original and modified CDFGs. Analysis of the aliasing equation is then performed as explained for the case of functional units. Note that, in the new CDFGs that we obtain, the original and the copy potentially compute different functions of the primary inputs.

To see this, we observe that if the multiplexer feeds a register, at some time, the multiplexer chooses the output of the wrong functional unit to feed the register. Likewise, it chooses the input for a functional unit from the wrong variable. To evaluate the probability that this will result in a detectable error for an arbitrary input, we need information about functional unit and register assignment. Since it is expensive to do the analysis at every stage of the iterative improvement process, we perform this analysis after synthesis. If the multiplexer has a routing error, the RTL circuit computes an output for a CDFG which is different from the output of the actual CDFG. Once the register and functional unit assignments are known, we can determine the actual function which is computed by the RTL circuit with a multiplexer selection error. Since we assumed a data-dominated CDFG, the output is still an arithmetic sum of products expression in the inputs. We would like to see if the outputs of the original and the copy can give identical wrong results. To do this we note that the aliasing equation would take the following form:

$$\prod_{i=1}^T \prod(i) = 0, \quad (18)$$

where $\prod(i)$ is a product of some primary inputs. Also suppose that p is the maximum degree of any of the variables in the equation (i.e., p is the highest exponent of any variable in any term of the equation). To evaluate the probability of this equation being satisfied by an arbitrary choice of primary inputs, we make the following observation:

Observation 3. Any equation of form (18) in n variables, with each having a range of $\{0, 1, \dots, 2^N - 1\}$, which is not an identity, will have at most $p * 2^{N(n-1)}$ solutions. To see this, we can simply fix the values of all but one of the variables and solve for the remaining one variable. Since this equation has a degree of at most p , the number of solutions is less than or equal to p . Since there are $2^{N(n-1)}$ ways of fixing $n - 1$ variables, the number of solutions is less than $p * 2^{N(n-1)}$.

Observation 4. The probability that a randomly chosen set of primary input patterns will satisfy this equation is $p/(2^N)$. This can easily be seen to be true because the total sample space has a cardinality of 2^{Nn} . Therefore, the probability in question is just $p * 2^{N(n-1)} / (2^{Nn})$, which is $p/(2^N)$.

Therefore, to check whether a specific data selection error creates a high probability of aliasing, we merely have to check in the RTL circuit if the original and the copy CDFGs compute the same function. We can easily check this by simulating the faulty CDFGs for a small number of input patterns. If the value of the output differs for even one input pattern, we can use the earlier observation to guarantee that the probability of aliasing is extremely low. As we mentioned earlier, we do this after the register and functional unit assignment to save CPU time. If the probability of aliasing turns out to be high, we split the functional unit into two to take care of the problem. However, in practice, it was found that multiplexer data selection errors did not pose a problem and we did not have to add hardware after synthesis.

5 DYNAMIC SECURING

There are two situations where we need to select a set of intermediate results in the CDFG to secure. The first situation occurs in moves of class B where securing is performed to allow resource sharing between the original and the copy in order to minimize area overhead. The second situation is in moves of class C where comparison operations are added in order to improve scheduling freedoms of various operations in the CDFG. We present efficient heuristics to selectively secure operations in the duplicated CDFG for both of the above situations in this section.

5.1 Dynamic Securing for Moves of Class B

The dynamic securing problem can be transformed to the following problem: *Given two sets of nodes in a CDFG, A and B , we want to find a minimum set of nodes C such that if all edges fed by the nodes in C are removed from the CDFG, then A and B have no common successor.* Here, A consists of nodes in the

original that map to a functional unit and B contains nodes in the original which correspond to those in the copy that map to the same functional unit.

Since the nature of the problem forbids an exact solution, we try to find the minimum number of nodes whose edges should be secured such that A and B have no common successor. Note that we do not directly attempt to minimize the number of comparators. This is because the schedule and assignment change as iterative improvement progresses and sharing configurations that give the best area overheads in one step may not hold in the next. Therefore, we choose to minimize the number of comparison operations. We found that, in practice, this simplification yielded good results and the total area overhead was small.

We first find the nodes in the CDFG which are not successors of either A or B . Since these nodes need not be secured, we remove these nodes from the CDFG. Then, we model the CDFG as an undirected hypergraph³ in which all the fanout edges of a node are replaced by a single edge. This is motivated by the fact that all outgoing edges of an operation in the CDFG are assigned to the same register. Securing the source node of this hyperedge secures all edges it corresponds to in the CDFG. Therefore, we are interested in finding a minimum set of edges in the hypergraph upon whose removal nodes in the hypergraph corresponding to A and B have no common successor. This minimum edge cut problem was solved in [19].

To solve this problem, the hypergraph is transformed into a normal graph using the procedure described below. Each edge E of the hypergraph is replaced by two vertices V_E^1 and V_E^2 with an edge of unit capacity connecting them. For each vertex V in the hypergraph, there is a corresponding vertex V' in the transformed graph. If an edge E in the hypergraph is incident on vertex V , then two edges are added to the transformed graph, one from V' to V_E^1 of capacity ∞ and the other from V_E^2 to V' , also of capacity ∞ . To this graph, we add a source vertex S and a sink vertex T . Vertex S is connected by edges of capacity ∞ to every vertex in the transformed graph which corresponds to a vertex in A . Vertex T is connected by edges of capacity ∞ to every vertex corresponding to a vertex in B . Solving the maximum network flow problem [20] from S to T yields the minimum edge cut of the transformed graph. From the minimum edge cut in the transformed graph, we can infer the minimum edge cut of the hypergraph using the inverse of the mapping described above. This corresponds to a minimum cardinality set of intermediate results in the CDFG to be secured.

The transformation is illustrated in Fig. 13. Here, the original CDFG is shown at the top. We would like to separate operation $n1$ in the original from operation $n2_c$ in the copy. To do this, we need to secure intermediate edges so that $n1$ and $n2$ do not have unsecured paths to the same successor, as mentioned in Definition 1. The hypergraph, which corresponds to the original CDFG, is shown in the figure, with unnecessary nodes and edges deleted. In the transformed graph shown in the figure, nodes $e1s$ and $e1t$ correspond to edge $e1$ of the hypergraph. Similarly, nodes

3. A hypergraph is like a regular graph except that a single edge can connect multiple nodes, unlike a regular graph.

Fig. 13. Transforming the hypergraph to find the minimum edge cut.

$e1s$ and $e2t$ correspond to edge $e2$ of the hypergraph. Nodes $n1$, $n2$, and $n3$ in the transformed graph correspond to nodes $n1$, $n2$, and $n3$ in the hypergraph. The edges in the transformed graph are annotated with their capacities.

5.2 Heuristic for Moves of Class C

We now present a method to identify nodes in the CDFG which cause scheduling bottlenecks and secure them with a view to increasing the scheduling freedoms of various operations in the CDFG. We would like to find a node which, when secured, “frees” itself and its predecessor nodes to be rescheduled to control steps where the hardware utilization is low, potentially increasing functional unit utilization. Example 4 illustrates an application of a class C move.

To find out which nodes of the CDFG are good candidates to secure, we use the following heuristic: For each node we compute a figure (called the *weight*) which is a measure of the gain in area that could be obtained by securing the node and rescheduling the operations in the

transitive fanin of that node. We compute the weight of a node n as follows: For each node in the transitive fanin of n , we compute the area of the functional unit that it maps to. We sum the areas of all nodes in the transitive fanin of n , including n itself. We then compute the difference between the earliest and latest control steps (span) in which any node from the transitive fanin of n is scheduled. The ratio of the cumulative area to the span of control steps is then computed. From this ratio, we subtract the ratio of the area of a register to the lifetime in cycles of the variable that is generated at node n to get the weight of node n (the subtracted term accounts for the impact of the prolonged lifetime of the variable on register area). The node with the largest positive weight is chosen for securing during a move of class C.

The rationale behind this heuristic is as follows: A node, n , with a large weight implies that it “locks away” a large number of resources. For example, if a node were scheduled in cycle 3 and the nodes in the transitive fanin consume an area of 100 units, then it can be assumed that 100 area units

have to be scheduled in the first two cycles. The securing of n frees it and allows it to be scheduled any time before the deadline and in doing so frees the operations in its transitive fanin by increasing their mobility, thus increasing the chance of obtaining a compact architecture. In this example, the 100 units can now be scheduled anytime before the cycle in which n is scheduled. If the freeing of n allowed it to migrate to cycle 10, the nodes in its transitive fanin would have to be scheduled in the first nine cycles, which is considerably more relaxed and more likely to result in a compact architecture. A node which locks away more resources in fewer cycles would clearly be a better candidate for securing than one which locks away fewer resources over more cycles.

Example 12. Consider once again the example CDFG shown in Fig. 4. Assume that the areas of the multiplier, adder, equality checker, and register are 4, 1, 1, and 0.5 units, respectively. We now compute the weight of each node in the CDFG. For node $n1$:

$$\begin{aligned} \text{Fanin Area} &= ((\text{Area} \in \text{Transitive fanin}) + \\ &\quad \text{Area of functional unit the node maps to}) = 1 \\ \text{Span} &= 1 \text{ cycle} \\ \text{Register overhead} &= (1/\text{Lifetime of } v1) * \text{Register area} \\ &= 1 * 0.5 = 0.5 \\ \text{Weight} &= 1 - 0.5 = 0.5. \end{aligned}$$

Similarly, the weights of nodes $n2$ through $n4$ can be determined to be 3.5, 4.5, and 1.8, respectively. Node $n3$, which has the maximum weight, is chosen for securing.

5.3 Handling Logical Operations

In this section, we describe how our analysis procedure handles logical operations in the behavioral description. Data-dominated descriptions, such as those that are usually found in DSP algorithms, do not typically contain many logical operations like AND, OR, and NOT. Hence, we do not need to extend aliasing analysis to cover logical operations. Instead, we secure the inputs and outputs of each logical block embedded in the CDFG. This is required because, even if the inputs to a logical block in the original and the copy are different from each other and from the correct value, the outputs of the block may be equal, but incorrect, allowing aliasing to occur. Securing the inputs ensures that any errors in the inputs are not suppressed by the logical block. Also, we do not allow sharing of functional units for logical operations in the original and the copy (within the original or within the copy, such sharing is allowed). This is because, unlike in the case of arithmetic unit faults, faults in logical operations in the original and the copy often do cause aliasing to occur with unacceptably high probabilities. In our synthesis system, though we enforce that logical operations do not share hardware across the original and the copy, the quality of our final solution is not affected because most CDFGs in the target domain contain very few, if any, logical operations.

6 ILLUSTRATIVE SYNTHESIS EXAMPLE

In this section, we trace the synthesis procedure for a small CDFG to illustrate the integration of aliasing analysis and securing into the synthesis procedure. In addition, we also trace the flow of the algorithm for a case when aliasing analysis is not applied because we require the circuit to have $P_{aliasing} = 0$.

In the example shown in Fig. 14, the library is assumed to have exactly one adder, one multiplier, and one TSC equality checker, all of which compute in one cycle. The deadline for all primary outputs is four cycles.

In the initial solution that is fed to the algorithm, each operation is bound to a separate functional unit and each variable to a separate register. In the intermediate solution for ALPS and zero-aliasing cases, sharing between different operations takes place without the use of rescheduling. If we are synthesizing a double modular redundant (DMR) architecture, no further sharing can take place because operations in the original and the copy cannot share functional units. In the case of ALPS, nodes $n2$ and $n1.c$ can share a functional unit if $n2$ is scheduled in control step 2. Application of aliasing analysis to this configuration results in the following condition for aliasing at primary output Out :

$$f(a4, a5) = a3f(a1, a2).$$

By applying Condition 1 to this equation, we can see that the aliasing probability is $1/2^N$, where N is the bit-width. The aliasing threshold is chosen as $2 \times 2^{-32} (4.66 \times 10^{-10})$ and we synthesize a 32-bit datapath ($N = 32$). Therefore, $P_{aliasing} \leq \text{threshold}$. Hence, we conclude that sharing of $n2$ and $n1.c$ requires no comparison operations. This gives the final solution for ALPS as no sequence of moves can result in any improvement in the current solution. If we want a solution with no aliasing, we would have to add a comparison operation at this stage to secure either operation $n1$ or $n2$. In this example, we chose the latter option, which gave us the final solution for the zero-aliasing case. Further details regarding the working of the area optimizer and the use of iterative improvement for area optimization can be found in [16].

7 EXPERIMENTAL RESULTS

We have implemented the behavioral synthesis framework presented in the previous sections, including the aliasing probability analysis techniques and dynamic securing procedures as the program ALPS. ALPS is written in C++. We have performed experiments to evaluate our techniques using several behavioral descriptions of digital signal and image processing applications. ALPS reads in a textual description of the CDFG and performs functional unit selection, scheduling, allocation, and assignment to result in a highly fault-secure RTL circuit that consists of a data path netlist and an FSM description of the controller. The controller is then subject to constrained logic synthesis, as described earlier, to result in a fault-secure implementation. The controller and data path netlists are merged and

Fig. 14. Iterative improvement applied to an example CDFG.

mapped to the MSU standard cell library (SCMOS 2.2) using the SIS logic synthesis system and then placed and laid out using tools from the Octtools suite.

Past literature in the area of behavioral synthesis for fault security has traditionally tabulated the numbers of registers, functional units, and multiplexers consumed by a

TABLE 1
Area Results Given as $grid_count \times 10^{-4}$

Circuit	L.F. = 1.2			L.F. = 1.7			L.F. = 2.2			L.F. = 2.7		
	DMR	0-alias	ALPS	DMR	0-alias	ALPS	DMR	0-alias	ALPS	DMR	0-alias	ALPS
<i>Chemical</i>	212	212	212	164	131	127	140	120	115	116	84	76
<i>Dct_dif</i>	124	101	96	93	84	81	70	59	56	70	59	54
<i>Dct_lee</i>	122	113	112	71	71	71	71	71	71	71	58	50
<i>Dct_pr2</i>	370	365	344	215	210	145	170	165	134	130	123	108
<i>Dct_wang</i>	225	195	154	177	152	124	102	102	86	102	102	84
<i>Dist</i>	223	218	205	175	164	153	149	113	95	101	100	93
<i>FIR</i>	84	80	74	60	57	51	60	56	50	60	56	39
<i>IIR77</i>	166	146	127	119	97	91	95	94	91	95	67	59
<i>Paulin</i>	85	81	80	84	80	57	60	45	44	46	46	31

behavior and a reduction in these numbers is taken to represent a reduction in area. We, however, present only the final area of the synthesized circuit. This break with tradition is intentional. Our algorithm dynamically trades off different circuit elements, depending upon their effect on the total area of the circuit, e.g., the savings in area obtained by merging two functional units into one, as suggested by a class *B* move, might well be offset by the overhead in terms of registers and multiplexers. We therefore believe that it is more meaningful to present the final area of the circuit, rather than to present a breakup in terms of the numbers of individual circuit elements.

We conducted experiments on nine CDFGs which perform different DSP algorithms. Among them, *Paulin* is a widely known benchmark. *Chemical*, *Dist*, and *IIR77* are IIR filters used in the industry. *Dct_dif*, *Dct_lee*, *Dct_pr2*, and *Dct_wang* perform the discrete cosine transform and are named after the inventors of their algorithms. *FIR* is an FIR filter. Of these examples, *Paulin*, *Chemical*, and *Dist* have loops in their behavioral descriptions. The number of operations in the duplicated CDFG ranges from 23 (*Paulin*) to 111 (*Dist*). Table 1 presents the results of our experiments. The example name is given in the column **Circuit**. The circuit was synthesized at four different sampling periods at a bit-width of 32. The laxity factor (L.F.) is the ratio of the given sampling period to the delay of the RTL circuit corresponding to the fastest possible implementation of the CDFG (with the given functional unit library). Major columns 2, 3, 4, and 5 are for circuits synthesized with laxity factors 1.2, 1.7, 2.2, and 2.7, respectively. For each laxity factor, we synthesized the following architectures: 1) an architecture with a duplicated RTL circuit and a TSC equality checker, 2) a fault-secure area-optimized architecture where we enforced the constraint that the probability of aliasing in functional units should be zero, and 3) a fault-secure area-optimized architecture synthesized using ALPS, where the probability of aliasing was restricted to be less than or equal to 4.66×10^{-10} . Since we synthesize a 32-bit data path and $1/2^N = 2.33 \times 10^{-10}$, the value of k in Condition 1 and the value of $p_m p_n$ in Condition 2 must be less than or equal to 2. The clock cycle time was constrained to 25ns. The area in $grid_count/10,000^4$ is reported for these three architectures under the subcolumns **DMR**, **0-alias**,

and **ALPS**, respectively. The results show that architectures synthesized using ALPS show average area improvements over DMR of 12.8 percent, 22.3 percent, 19.0 percent, and 24.9 percent at laxity factors of 1.2, 1.7, 2.2, and 2.7, respectively, while architectures synthesized with zero-aliasing requirement show improvements of 6.2 percent, 9.7 percent, 10.0 percent, and 12.1 percent for the corresponding laxity factors. On an average, the control logic accounted for approximately 4 percent of the total area of the circuit. The checking hardware accounted for approximately 15 percent, the "original" elements in the circuits about 47 percent, and the redundant elements about 34 percent. Note that all architectures were required to meet the same sampling period constraint. The results indicate that architectures synthesized using ALPS are more area-efficient than architectures synthesized with an aim to guarantee that aliasing cannot occur or compared to duplicate-and-compare architectures. The CPU times taken up for synthesis were not significant enough to be an issue. Our largest example consumed under 15 minutes of CPU time on a SPARC-20 workstation running at 62 MHz.

8 CONCLUSIONS

In this paper, we described a behavioral synthesis framework, called ALPS, for the construction of highly fault-secure, area-efficient RTL circuits. We introduced the concept of aliasing probability analysis to enhance resource sharing among operations in the duplicated CDFG. We showed that our approach results in a significant reduction in the overhead associated with the provision of fault security. Experimental results indicate that fault security can be provided by ALPS with area overheads as low as 25.5 percent (*IIR77* at a laxity factor of 2.7) over a circuit synthesized without a requirement of fault security (i.e., over a simplex system).

ACKNOWLEDGMENTS

This work was supported by the US National Science Foundation under Grant No. MIP-9423574.

4. The grid count represents the number of $\lambda \times \lambda$ squares in the circuit's layout. We used the SCMOS 2.2 library for our experiments.

REFERENCES

- [1] R.M. Sedmak and H.L. Leibergot, "Fault Tolerance of a General Purpose Computer by Very Large Scale Integration," *IEEE Trans. Computers*, vol. 29, no. 6, pp. 492-500, June 1980.
- [2] N. Saxena and E.J. McCluskey, "Dependable Adaptive Computing Systems—The ROAR Project," *Proc. Int'l Conf. Systems, Man, and Cybernetics*, pp. 2,172-2,177, Nov. 1998.
- [3] R. Karri and A. Orailoglu, "High Level Synthesis of Fault-Tolerant ASICs," *Proc. Design Automation Conf.*, pp. 419-422, June 1992.
- [4] R. Karri and A. Orailoglu, "Scheduling with Rollback Constraints in High Level Synthesis of Self-Recovering ASICs," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 519-526, June 1992.
- [5] K. Hogstedt and A. Orailoglu, "Integrating Binding Constraints in the Synthesis of Area Efficient Self Recovering Microarchitectures," *Proc. Int'l Conf. Computer Design*, pp. 331-334, June 1994.
- [6] D. Gu, D.J. Rosenkrantz, and S.S. Ravi, "Construction and Analysis of Fault Secure Multiprocessor Schedules," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 120-127, June 1991.
- [7] R. Karri and A. Orailoglu, "High Level Synthesis of Fault Secure Microarchitectures," *Proc. Design Automation Conf.*, pp. 429-433, June 1993.
- [8] S. Sokolov and R. Karri, "Allocation and Binding during Fault Secure Microarchitecture Synthesis," *Proc. Int'l Conf. Computer Design*, pp. 327-330, June 1994.
- [9] V. Raghavendra and C. Lurnisap, "A Technique for Micro Rollback Self-Recovery Synthesis," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 1,171-1,179, Sept. 1995.
- [10] D.M. Blough, F.J. Kurdahi, and S.Y. Ohm, "Optimal Recovery Point Insertion for High-Level Synthesis of Recoverable Microarchitectures," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 50-59, June 1995.
- [11] R. Narasimhan, D.J. Rosenkrantz, and S.S. Ravi, "Algorithms for Analyzing and Synthesizing Fault-Tolerant Datapaths," *Proc. Int'l Workshop Defect and Fault Tolerance in VLSI Systems*, pp. 81-89, Nov. 1995.
- [12] L.M. Guerra, M.M. Potkonjak, and J.M. Rabaey, "High Level Synthesis of Reconfigurable Data Path Structures," *Proc. Int'l Conf. Computer-Aided Design*, pp. 26-29, Nov. 1993.
- [13] P.H. Bardell, W.H. McAnney, and J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques*. New York: John Wiley & Sons, 1987.
- [14] S. Devadas, H.K.T. Ma, A.R. Newton, and A. Sangiovanni-Vincentelli, "A Synthesis and Optimization Procedure for Fully and Easily Testable Sequential Machines," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 1,100-1,107, Oct. 1986.
- [15] D.P. Siewiorek and R.S. Swarz, *Reliable Computer Systems*. Bedford, Mass.: Digital Press, 1992.
- [16] A. Raghunathan and N.K. Jha, "An Iterative Improvement Algorithm for Low Power Datapath Synthesis," *Proc. Int'l Conf. Computer-Aided Design*, pp. 597-602, Nov. 1995.
- [17] I.C. Park and C.M. Kyung, "FAMOS: An Efficient Scheduling Algorithm for High-Level Synthesis," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 1,437-1,448, Oct. 1993.
- [18] S. Devadas and A.R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 768-781, July 1989.
- [19] E.L. Lawler, "Cutsets and Partitions of Hypergraphs," *Networks*, vol. 3, pp. 275-285, 1973.
- [20] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, pp. 579-615. Cambridge, Mass.: MIT Press, 1990.

technical papers. His papers have received the Best Paper Award at the Design Automation Conference in 1999 and 2000 and at VLSI Design '98 and '00. He has served on the program committee of ASP DAC. He is currently the editor of the TTTC Newsletter. His research interests include system level design techniques, high-level synthesis, digital system testing, and fault-tolerant system design. He is a member of the IEEE.

involved in the research and development of high-level and System-on-Chip design tools and methodologies with emphasis on low power design, testing, and design-for-testability. He has coauthored a book titled *High-Level Power Analysis and Optimization* (Kluwer Academic). He has presented conference tutorials on low-power design, and considering testability during high-level design. He holds or has filed for seven US patents in the areas of digital testing and low-power design. He received best paper awards at the 11th IEEE International Conference on VLSI Design (1998) and at the ACM/IEEE Design Automation Conference (1999 and 2000) and two best paper award nominations at the ACM/IEEE Design Automation Conference (1996 and 1997). He has served on the technical program committees of the IEEE VLSI Test Symposium (1998-2000) and the International Test Synthesis Workshop (1998-1999), and the ACM/IEEE Design Automation Conference (2000). He is currently organizational vice-chair of the Tutorials & Education Group of the IEEE Computer Society Test Technology Technical Council and serves as associate editor of the *IEEE Transactions on VLSI Systems*. He is a member of the IEEE.

Princeton, New Jersey. He has served as an associate editor of *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* and is currently serving as an associate editor of *IEEE Transactions on VLSI Systems* and of the *Journal of Electronic Testing: Theory and Applications* (JETTA). He served as the guest editor for the JETTA special issue on high-level test synthesis. He also served as the program chairman of the 1992 Workshop on Fault-Tolerant Parallel and Distributed Systems. He is the recipient of the AT&T Foundation Award and the NEC Preceptorship Award for research excellence. He has coauthored two books titled *Testing and Reliable Design of CMOS Circuits* (Kluwer, 1990) and *High-Level Power Analysis and Optimization* (Kluwer, 1998). He has authored or coauthored more than 160 technical papers. He has coauthored four papers which have won the Best Paper Award at ICCD'93, FTCS'97, ICVLSID'98, and DAC'99. He has also received nominations for Best Paper Awards at DAC'96, DAC'97, and DATE'98. His research interests include digital system testing, computer-aided design of integrated circuits, fault-tolerant computing, and distributed computing. He is a fellow of the IEEE.

Ganesh Lakshminarayana (S'95-M'98) received his BTech degree in electrical and electronics engineering from the Indian Institute of Technology, Madras, in June 1994, and his MA and PhD degrees in electrical engineering from Princeton University, Princeton, New Jersey, in June 1996 and August 1998, respectively. He is currently a research staff member at Computer and Communications Research Labs, NEC USA. He has coauthored more than 20

Anand Raghunathan (S'93-M'97) received the BTech degree in electrical and electronics engineering from the Indian Institute of Technology, Madras, in 1992, and the MA and PhD degrees in electrical engineering from Princeton University, Princeton, New Jersey, in 1994 and 1997, respectively. Dr. Raghunathan is currently a research staff member in the Computers & Communications Research Laboratories, NEC USA, Princeton, New Jersey, where he is

Niraj K. Jha (S'85-M'85-SM'93-F'98) received his BTech degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India, in 1981, his MS degree in electrical engineering from the State University of New York at Stony Brook in 1982, and his PhD degree in electrical engineering from the University of Illinois, Urbana-Champaign in 1985. He is a professor of electrical engineering at Princeton University,