Area-Efficient Fault Detection During Self-Recovering Microarchitecture Synthesis*

Ramesh Karri
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst
Amherst MA 01003

Alex Orailoğlu
Department of Computer Science and Engineering
University of California, San Diego
La Jolla CA 92093

Abstract: We will present the *area-efficient* fault-detection synthesis component of $\mathcal{SYNCERE}$, an integrated system for synthesizing area-efficient self-recovering microarchitectures. In the $\mathcal{SYNCERE}$ model for self-recovery, transient fault detection is based on duplication and comparison, while recovery from transient faults is accomplished via checkpointing and rollback. $\mathcal{SYNCERE}$ minimizes the overhead of duplication using two complementary area-optimization techniques. Whereas imposing inter-copy hardware disjointness at a sub-computation level instead of at the overall computation level ameliorates the dedicated hardware required for the original and duplicate computations, restructuring the pliable input representation of the duplicate computation further moderates the overall hardware.

1 Introduction

The current generation of automotive electronic ICs have to meet the military-quality and fault-tolerance goals at *commodity prices*[1]. Likewise, life-critical applications such as medical life-support units, and industrial process controls mandate fault-tolerance. This growing demand for fault-tolerance coupled with the inherent unreliability attendant upon VLSI has elevated the design of fault-tolerant VLSI systems into a research problem of immediate practical relevance.

$\mathcal{SYNCERE}$ automates the design of self-recovering microarchitectures from high level behavioral descriptions by supporting duplication based detection and checkpointing based recovery. $\mathcal{SYNCERE}$ supports a multidimensional force-directed scheduler that performs area optimization by focusing on the area overhead of intermediate computations in addition to the area overhead of the overall computation. $\mathcal{SYNCERE}$ also restructures an input representation so as to exploit the raggedness of its hardware utilization profile. Restructuring of an input representation is automatic and is steered by a *multidimensional force* metric. Finally, it exploits the unique features of a checkpointed computation by identifying selected intermediate computations and dedicating hardware for each of them and their counterparts. Such a strategy fosters inter-copy hardware sharing without however compromising the 100% fault detection capability.

In most high-level synthesis systems, only trade-offs between performance and area are explored[2]. Only recently, newer quality metrics such as testability and fault-tolerance have been considered at the microarchitectural level. In [5, 3] testability issues were explicitly addressed at the microarchitectural level, while in [4, 6, 8] fault-tolerance issues were explored. Orailoğlu and Karri[6] have developed heuristic and optimal strategies for coactive scheduling and checkpoint insertion during self-recovering microarchitecture synthesis. This is in contrast to the heuristic techniques for scheduling followed by checkpoint insertion proposed in [8]. Along a different dimension, Karri and Orailoğlu[4] have also developed a system for synthesizing reliable microarchitectures.

The rest of this paper is organized as follows. Section 2 outlines our model for self-recovery and presents the methodology. Section 3 describes an area-efficient fault detection mechanism and a scheduling algorithm that performs area optimization. Additional area optimization via flowgraph restructuring is explored in Section 4. Section 5 then describes incorporation of essential fault-detection constraints such as hardware disjointness between the original and the duplicate computations. Section 6 summarizes the results of synthesis experiments.

2 The model and the methodology

In our model for self-recovery, partial results from two copies are compared at a checkpoint (duplication and comparison), and if they agree, are written into the checkpoint registers (checkpointing). On the other hand, if the results disagree, the computation rolls back to the previous checkpoint and retries. Checkpoint insertion[6] completely determines the checkpoints as well as the edges assigned to each of them. Checkpointing groups clock cycles into *checkpoint zones*. A *checkpoint zone* is the set of clock cycles between two adjacent checkpoints. Nodes belonging to the same checkpoint zone are *coeval*, while nodes that are voted upon at a checkpoint are *secured*. Finally, a secured node together with its coeval predecessors form a $\nabla-$*subgraph*. We will illustrate these concepts

*This research was supported by NSF grant MIP-9308535.

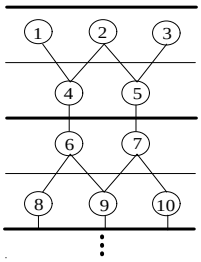


Figure 1: Model for self-recovery

using the checkpointed flow graph shown in figure 1. The first checkpoint zone comprises of clock cycles 1 and 2 while the second checkpoint zone comprises of clock cycles 3 and 4. Nodes 4,5,8,9, and 10 are secured since their outputs are voted upon at a checkpoint. The nodes 1,2,3,4, and 5 are coeval. Similarly, the nodes 6,7,8,9, and 10 are coeval. Subgraphs {1,2,4}, {2,3,5}, {6,8}, {6,7,9}, and {7,10} are $\nabla-$subgraphs. Although each secured node belongs to one and only one $\nabla-$subgraph, two or more $\nabla-$subgraphs can have common nodes. For example, $\nabla-$subgraphs {6,8}, and {6,7,9} have node 6 in common.

$\mathcal{SYNCERE}$ uses a two-pass scheduler. In the first pass, checkpoints are inserted using an edge-based scheduler[6]. Moreover, the voting area overhead is explicitly optimized. In the second pass fault-detection constraints are incorporated. Initially, the original flow graph is scheduled using an aggressive multidimensional force-directed scheduler which performs fine-grain area optimization in addition to coarse-grain area optimization. Subsequently, the duplicate flowgraph is restructured using a force-directed transformational subsystem. However, inorder to compare results from identical computations at a checkpoint, such flowgraph restructuring is confined to coeval subgraphs. Finally, the duplicate flowgraph is scheduled using a retentive scheduler that considers (i) hardware utilization characteristics of the original flowgraph, (ii) sharing between the original and the duplicate computations, and (iii) disjointness between the original and the duplicate $\nabla-$subgraphs.

3 Multidimensional Scheduling

Traditionally, scheduling algorithms have optimized the peak hardware consumption of the overall flowgraph (we will refer to this as *coarse-grain area optimization*). However, since the area-efficient fault detection mechanism enforces hardware disjointness at the $\nabla-$subgraph level, *the peak hardware of each of these $\nabla-$subgraphs have to be minimized as well.* We will outline a *multidimensional* analog of the well known force directed scheduling algorithm[7] that minimizes the maximum hardware used both by the original flowgraph as well as by all of its constituent $\nabla-$subgraphs (we will refer to this as *fine-grain area optimization*).

In the basic force-directed scheduler, a *distribution graph* is set up for each clock cycle i and for each operation type j as given by equation 1 where F_k is the number of clock cycles that node k (of type j) can be feasibly assigned to.

$$DG(i,j) = \sum_{type(k)=j} \frac{1}{F_k} \qquad \forall k \tag{1}$$

Next, a *node* is assigned to a *clock* if it yields the minimum *force*[7]. The *multidimensional force-directed scheduler* extends and supplements the traditional coarse-grain area optimization with fine-grain area optimization. Fine-grain area optimization is accomplished by assigning a *node* to a *clock* that additionally minimizes the peak hardware usage of each of the $\nabla-$subgraphs to which the node belongs. Fine grain distribution graphs called the $\nabla-$*distribution graphs* are computed for each $\nabla-$subgraph. $\nabla-$*forces* of assigning a *node* to a *clock* are computed –one for each $\nabla-$subgraph to which the *node* belongs to– by using the corresponding $\nabla-$distribution graphs. Let $\mathcal{R}_1$, $\mathcal{R}_2$, .., $\mathcal{R}_n$ be the $\nabla-$subgraphs to which the *node* belongs to. Let $\mathcal{F}_1$, $\mathcal{F}_2$, .., $\mathcal{F}_n$ be the corresponding $\nabla-$forces resulting from assigning *node* to *clock*, and $\mathcal{F}_O$ be the overall force. The total force $\mathcal{F}(node, clock)$ of assigning *node* to *clock* is given by equation 2.

$$\mathcal{F}(node,clock) = \mathcal{F}_O(node,clock) + \sum_{node \in \mathcal{R}_i} \mathcal{F}_i(node,clock) \tag{2}$$

The benefits of such *fine grain area optimization* can be illustrated. Consider figure 2a with a checkpoint inserted at the clock boundary 4-5. A coeval subgraph (belonging to the first checkpoint zone) comprising of three disjoint $\nabla-$subgraphs ($\nabla-$subgraph1 = {8,6,3,4,1,2}, $\nabla-$subgraph2 = {9,6,7,3,4,5,1,2}, and $\nabla-$subgraph3 = {13,12,11,10}) is also shown. All nodes are color coded to highlight their respective $\nabla-$subgraph membership. Observe that $\nabla-$subgraph1 and $\nabla-$subgraph2 have some operations common to them. All of these $\nabla-$subgraphs should be scheduled into the first checkpoint zone alone. Traditional force-directed scheduling does not discriminate between the assignment of operation 3 to clock cycle 1, operation 5 to clock cycle 1, operation 10 to clock cycle 1 and operation 11 to clock cycle 1 as all of these assignments have the same force. Hence, a possible schedule resulting from such a scheduler is shown in figure 2b. However, such an assignment results in peak hardware usage of 3 (for $\nabla-$subgraph1), 4 (for $\nabla-$subgraph2), and 2 (for $\nabla-$subgraph3). However, a superior schedule that additionally minimizes the peak hardware usage of each of the $\nabla-$subgraphs (2 for $\nabla-$subgraph1, 3 for $\nabla-$subgraph2, and 1 for $\nabla-$subgraph3) is shown in figure 2c.

4 Force-Directed Restructuring

The benefits of flowgraph restructuring in the context of self-recovering microarchitecture synthesis can

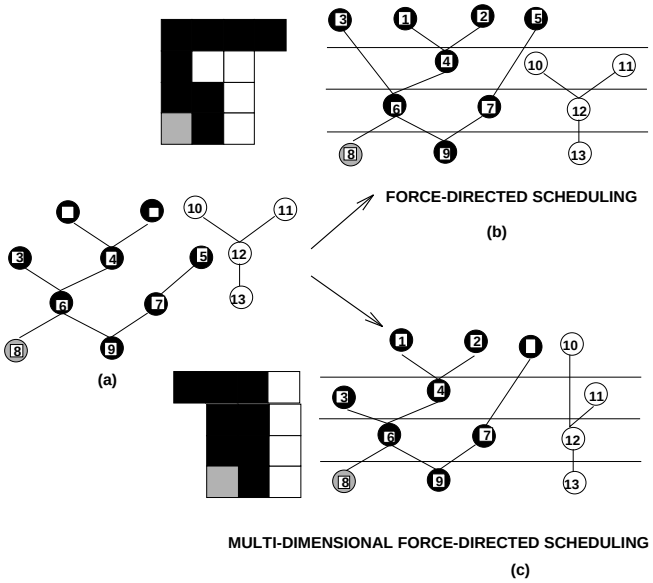
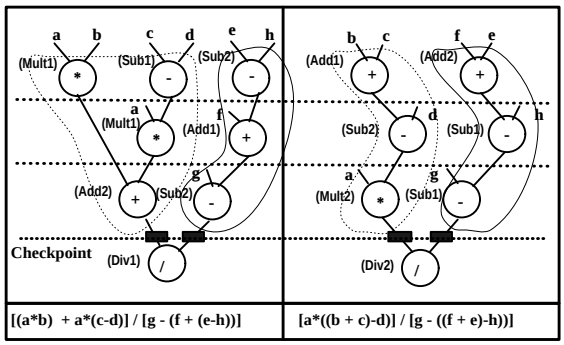


Figure 2: Multidimensional force directed scheduling

be illustrated using an example shown in figure 3. Consider the flowgraph on the left hand side box with



Total Hardw Usage: 2 Add, 2 Multiply, 2 Divide, 2 Subtracts, 2 Voters

Hardw Usage w/o Trans: 2 Add, 2 Multiply, 2 Divide, 4 Subtracts, 2 Voters

Figure 3: Impact of Flowgraph Restructuring on area overhead of hardware redundancy based detection

a checkpoint inserted at clock cycle boundary 3-4. The checkpoint partitions the original computation into two $\nabla-$subgraphs enclosed in solid and dotted lines respectively (in the left hand side box). Straightforward duplication requires four subtractors. However, distributivity followed by associativity on the left hand side $\nabla-$subgraph and associativity on the right hand side $\nabla-$subgraph has resulted in a functionally equivalent flowgraph (shown in the right hand side box) requiring only two subtractors. Also, transformations have been confined to within a checkpoint zone. Furthermore, we have annotated all nodes with a functional unit to demonstrate the existence of a feasible operator binding satisfying the hardware disjointness constraint.

We implemented a force-directed approach to restructuring the duplicate coeval subgraphs. The candidate transformations are evaluated using a *multidimensional-force* metric and a transformation with the minimum negative force is then invoked. Initially, global transformations are applied as they improve hardware utilization across clock cycles in addition to uncovering flow graph structures amenable to local transformations. The multidimensional force computation described in section 3 is generalized to derive the force associated with the invocation of a transformation as follows: Let $DG_{untrans}(i,j)$ and $DG_{trans}(i,j)$ be the distributions of the j^{th} operator type in the i^{th} clock cycle prior to and after the invocation of a transformation respectively and let $\Delta DG(i,j)$ be the difference between these distributions computed using equation 3.

$$\Delta DG(i,j) = DG_{trans}(i,j) - DG_{untrans}(i,j) \quad (3)$$

In order to incorporate the hardware utilization characteristics of the original scheduled flowgraph, the distribution graph of the duplicate flowgraph is supplemented with the hardware utilized by the original copy (say $H(i,j)$). The force $\mathcal{F}$ of the transformation, is then obtained using equation 4.

$$\mathcal{F} = \sum_{\forall i} \sum_{\forall j} (H(i,j) + DG(i,j)) \times \Delta DG(i,j) \quad (4)$$

Since the overall hardware is influenced by the peak rather than the per-clock utilization, the force computation is further enhanced to explicitly optimize the peak hardware utilization as follows: $\Delta DG(i,j)$ is modified by substituting the per-clock hardware distribution $DG(i,j)$ with the current peak committed hardware as the reference (approximated by the peak hardware $H(j)$, $(= \max_{\forall j} H(i,j))$ of the scheduled original flowgraph) as given by equation 5.

$$\begin{aligned} \Delta DG(i,j) &= DG_{trans}(i,j) - DG_{untrans}(i,j), \\ &\quad if \quad (DG_{trans}(i,j) > H(j)) \\ &= 0, \quad otherwise \end{aligned} \quad (5)$$

Next, we will describe the incremental derivation of $DG_{trans}(i,j)$ for the distributivity transformation.

Consider the untransformed flowgraph on the left hand side of the arrow in figure 4a. Assuming a time constraint of two clock cycles, the distribution graphs for the adder and the multiplier are shown below the flow graph. On the right hand side of the arrow, a transformed flow graph (resulting from the invocation of distributivity) and the corresponding distribution graph are shown. Note that the transformed flow graph has a better distribution graph and consequently a lower force when compared to the original flow graph. Consider the more general flowgraph structure within which distributivity can be applied. In figure 4b, if $L1$ and $R1$ are the left and right predecessor subgraphs respectively of the head node (of type $\oplus$)and if $R2$ is the right predecessor subgraph of the tail node (of type $\otimes$)

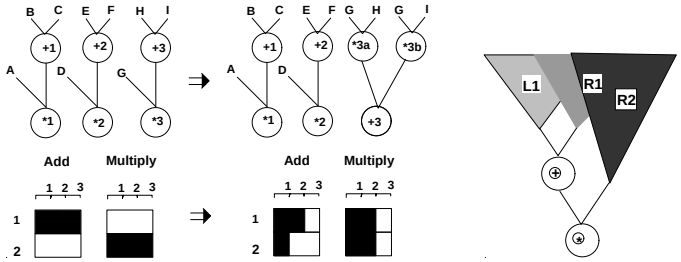


Figure 4: (a) Force of Distributivity (b) General Formulation

of the transformable flowgraph structure. Then, the incremental mobilities of the head node, n_h, of type $\oplus$, with F_{n_h} feasible clocks t_1 through t_{n_h} and the tail node, n_t, of type $\otimes$, with F_{n_t} feasible clocks t_1 through t_{n_t} can be computed as shown in figure 5.

$$DFG_{trans}(i,\otimes) = DFG_{untrans}(i,\otimes) + \frac{2}{F_{n_h}} \quad t_1 \leq i \leq t_{n_h}$$
$$DFG_{trans}(i,\oplus) = DFG_{untrans}(i,\oplus) - \frac{1}{F_{n_h}} \quad t_1 \leq i \leq t_{n_h}$$
$$DFG_{trans}(i,\oplus) = DFG_{untrans}(i,\oplus) + \frac{1}{F_{n_t}} \quad t_1 \leq i \leq t_{n_t}$$
$$DFG_{trans}(i,\otimes) = DFG_{untrans}(i,\otimes) - \frac{1}{F_{n_t}} \quad t_1 \leq i \leq t_{n_t}$$

Figure 5: Mobility Computation for Distributivity

5 Retentive Scheduling

In this final step, the multidimensional force-directed scheduler described in section 3 is supplemented with *retention* and *discrimination* capabilities. The retention capability fosters hardware sharing between the original and the duplicate flowgraph, by exploiting the compatibility of their hardware utilization profiles. If $H^1(i,j)$ is the hardware usage of the j^{th} type in clock i in the original schedule, and $DG^2(i,j)$ is the hardware distribution graph of the duplicate computation alone, then $DG(i,j)$, the *retentive distribution graph*, is given by equation 6.

$$DG(i,j) = DG^2(i,j) + H^1(i,j) \tag{6}$$

Similarly, $H^1_k(i,j)$ is the usage of the j^{th} type of hardware in clock i in the schedule of the k^{th} $\nabla-$subgraph in the original computation, and $DG^2_k(i,j)$ is the hardware distribution graph of the corresponding duplicate $\nabla-$subgraph alone, then $DG_k(i,j)$, the *retentive* $\nabla-$*distribution graph*, of the k^{th} $\nabla-$subgraph is given by equation 7.

$$DG_k(i,j) = DG^2_k(i,j) + H^1_k(i,j) \tag{7}$$

The retention capability is supplemented with *discrimination* that promotes hardware sharing only if such sharing does not compromise 100% fault-detection capability of the resulting microarchitecture. 100% fault-detection necessitates dedicated hardware for the original and the duplicate computations, and can be enforced by substituting the per-clock cycle hardware usage in equation 6 with the peak hardware usage. However, since straightforward duplication entails significant hardware overhead, $\mathcal{SYNCERE}$ imposes the inter-copy hardware disjointness only at the granularity of $\nabla-$subgraphs as given in equation 8.

$$DG_k(i,j) = DG^2_k(i,j) + \max_{\forall i} H^1_k(i,j) \tag{8}$$

Such a *discriminating retention* promotes hardware sharing between the original and the duplicate computations without however compromising on the 100% fault-detection capability of the resulting microarchitecture.

6 Experimental Results

We will now evaluate the $\mathcal{SYNCERE}$ approach to optimizing the area overhead of hardware redundancy based detection on three high level synthesis benchmarks namely, fifth order elliptic filter, AR Filter, and 16-tap FIR filter.

The reduction in hardware due to the proposed fault-detection scheme are summarized in table 1. The voting overhead reported in column 3 is an artifact of the checkpoint insertion phase. Reduction in hardware vis-a-vis straightforward duplication is then computed in column 6. A schedule for the elliptic filter, corresponding to the second row in table 1, is shown in figure 6. The hardware savings were 16.67% for the adders and 25% for the multipliers. From this experiment it can be concluded that performance remaining invariant the savings in hardware varies inversely as

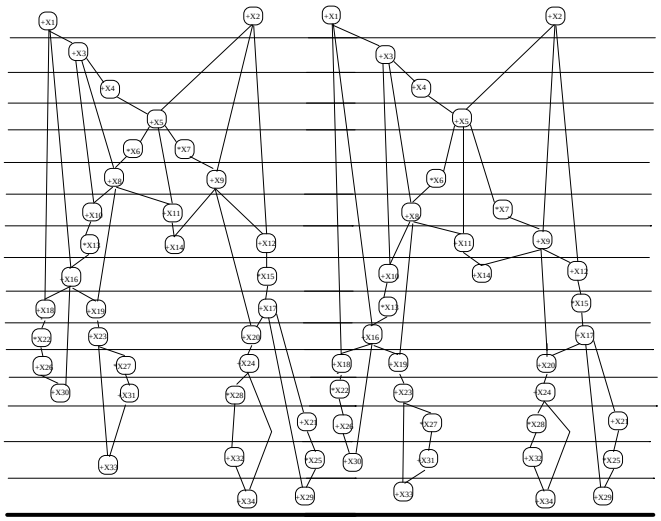


Figure 6: 16 Clocks; Retry period=14

EX	# Clk	Retry	# V's	# of Modules Copy 1	# of Modules Overall	% Red'n in Hardw
Ell	16	16	4	3A/1M	5A/2M	16.67/0
ipt	16	14	4	3A/2M	5A/3M	16.67/25
ic	16	12	5	3A/2M	5A/3M	16.67/25
	16	8	5	3A/1M	4A/2M	16.67/00
	19	19	4	2A/2M	4A/3M	0.00/25
	19	18	3	2A/2M	4A/3M	0.00/25
	19	7	6	2A/2M	4A/4M	0.00/00
	21	21	5	2A/2M	3A/3M	25.00/25
	21	16	5	2A/1M	3A/2M	25.00/00
	21	10	5	2A/1M	3A/2M	25.00/00
	21	8	5	2A/2M	4A/4M	0.00/00
FIR	10	10	1	2A/2M	4A/4M	00/0.00
	10	9	2	2A/2M	4A/3M	00/25.00
	10	8	3	2A/2M	4A/3M	00/25.00
	10	7	4	2A/2M	4A/3M	0/25.00
	9	9	1	2A/2M	4A/4M	0/0
	9	8	2	2A/2M	4A/4M	0/0
	9	7	4	2A/2M	4A/4M	0/0
AR	10	10	2	2A/3M	4A/5M	0/16.67
	10	9	4	2A/4M	4A/8M	0/0
	10	8	4	2A/4M	4A/8M	0/0
	10	7	4	2A/4M	4A/8M	0/0

Table 1: Area efficient fault detection

the number of inserted checkpoints. This is because as more and more checkpoints are inserted, the size of a checkpoint zone as well as the mobility of the nodes in the resulting coeval subgraphs decreases.

We will now evaluate the benefit of flowgraph restructuring. The results of this set of synthesis experiments are summarized in table 2. Reduction in hardware due to flowgraph restructuring vis-a-vis *area-efficient duplication* is computed in column 6. Restructuring proved extremely beneficial in synthesizing area-efficient designs in the presence of tight performance constraints. The benefits of restructuring are however limited by the inserted checkpoints. Specifically, both the clock cycle boundaries where checkpoints are inserted as well as the number of checkpoints inserted impact the effective application and exploitation of restructuring. Insertion of too many checkpoints precludes the application of global transformations such as associativity.

7 Conclusion

In $\mathcal{SYNCERE}$ the synergies arising out of an area-efficient fault-detection technique, efficient manipulation of algorithmic level design structures, and aggressive area optimization at the microarchitectural level have been exploited to synthesize area-efficient self-recovering microarchitectures. Furthermore, since scheduling is the most important phase in high level synthesis, aggressive area optimization has been incorporated into this phase of fault-tolerant microarchitecture synthesis.

EX	# Clks	Retry	# V's	# of Modules no trans	# of Modules trans	% Red'n in Hardw
Ell	14	13	4	6A/4M	6A/4M	0/0
ipt	14	12	4	6A/4M	6A/4M	0/0
ic	14	10	4	6A/4M	6A/3M	0/25.00
FIR	10	9	2	4A/4M	4A/3M	0/25.00
	10	8	4	4A/4M	4A/3M	0/25.00
	10	3	6	4A/4M	4A/4M	0/00.00
	9	9	1	4A/4M	4A/3M	0/25.00
	9	8	2	4A/4M	4A/3M	0/25.00
	9	7	2	4A/4M	4A/3M	0/0
AR	9	8	6	4A/8M	4A/6M	0/25.00
	9	6	6	4A/8M	4A/6M	0/25.00
	9	2	6	4A/8M	4A/8M	0/0

Table 2: Impact of Flowgraph Restructuring

References

[1] P. Ames, P. Miles, E. Milham, C. Pilch, W. Rodda, L. Tedesco, and B. V. Tine. Automotive Electronics: A Design and Test Roundtable. *IEEE Design and Test of Computers*, 35(8):84–93, 1992.

[2] D. D. Gajski, N. D. Dutt, A. Wu, and S. Lin. *High-Level Synthesis: Introduction to chip and system design*. Kluwer, 1992.

[3] I.G. Harris and A. Orailoğlu. Microarchitectural Synthesis of VLSI Designs with High Test Concurrency. In *Proceedings of the 1994 Design Automation Conference*, June 1994.

[4] R. Karri and A. Orailoğlu. High-Level Synthesis of Fault-Tolerant ASICs. In *Proceedings of IEEE International Symposium on Circuits and Systems*, May 1992.

[5] T. C. Lee, W. H. Wolfe, and N. K. Jha. Behavioral Synthesis for Easy Testability in Data Path Scheduling. In *Proceedings of the International Conference on Computer Aided Design*, pages 616–619, November 1992.

[6] Alex Orailoğlu and Ramesh Karri. Coactive Scheduling and Checkpoint Determination during the High Level Synthesis of Self Recovering Microarchitectures. *IEEE Transactions on VLSI Systems*, 1994.

[7] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Transactions on CAD*, 8(6):661–679, 1989.

[8] V. Raghavendra and C. Lursinsap. Automated Micro-Rollback Self-Recovery Synthesis. In *Proceedings of the 28th Design Automation Conference*, pages 385–390, 1991.