HETEROGENEOUS BUILT-IN RESILIENCY OF APPLICATION SPECIFIC PROGRAMMABLE PROCESSORS *

Kyosun Kim, Ramesh Karri
Department of ECE
University of Massachusetts
Amherst, MA 01002
{karri,kkim}@ecs.umass.edu

Miodrag Potkonjak
Department of Computer Science
University of California
Los Angeles, CA 90095
miodrag@cs.ucla.edu

Abstract - *Using the flexibility provided by multiple functionalities we have developed a new approach for permanent fault-tolerance: Heterogeneous Built-In-Resiliency (HBIR). HBIR processor synthesis imposes several unique tasks on the synthesis process: (i) latency determination targeting k-unit fault-tolerance, (ii) application-to-faulty-unit matching and (iii) HBIR scheduling and assignment algorithms. We address each of them and demonstrate the effectiveness of the overall approach, the synthesis algorithms, and software implementations on a number of designs.*

1 Introduction

Application Specific Programmable Processors (ASPP s) provide efficient implementation for any of N specified functionalities. ASPPs preserve all the advantages of full custom designs, while providing multiple functionalities and flexibility during the design and effective usage of the ASPP circuits. For example, the ASPP in figure 1 implements a seventh order IIR filter (IIR7) and two versions of the volterra filter (VOLTERRA).

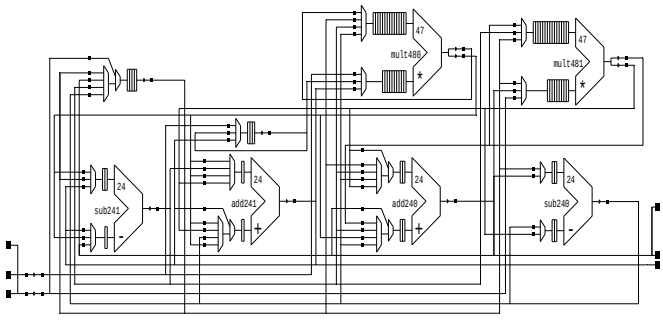


Figure 1: An example ASPP microarchitecture implementing seventh order IIR and VOLTERRA filters

The flexibility and redundancy inherent in a ASPP design is an excellent source for providing transient and/or permanent fault tolerance with no or low overhead. One can program an ASPP to execute the applications which do not use the faulty units. For example, as shown in table 1, one of the versions of the VOLTERRA filter is still operational in the presence of a single faulty unit. In cases when limited repair is economically feasible, one can program an ASPP to implement an application which requires the smallest number of repair steps.

application		add24		sub24		mult48	
	sch	0	1	0	1	0	1
IIR7	1	√	√	√	√	√	√
VOLTERRA	1		√			√	
	2	√					√

Table 1: Functional units used by IIR7 and VOLTERRA applications

In this paper, we present a behavioral synthesis approach for incorporating such **Heterogeneous Built-In Resiliency (HBIR)** into processors and present algorithms for k-unit fault-tolerant HBIR processor synthesis and discuss the relationship between ASPPs, degree of fault-tolerance and area and performance overheads.

2 HBIR: A Motivating Example

Towards illustrating the key concepts of **heterogeneous built-in resiliency (HBIR)** consider three example control data flow graphs (CDFGs) shown in figures 2(a)-(c). Also, assume that (i) all operations finish in a single cycle, (ii) the applications have identical word lengths and (iii) each application is implemented in three clock cycles.

If implemented as a dedicated ASIC, the CDFG in figure 2(a) requires three adders and three multipliers, and the CDFG in figure 2(b) requires one adder and two multipliers. Similarly, the CDFG in figure 2(c) requires two adders and one multiplier. However, these dedicated ASICs cannot tolerate any fabrication time or in-operation functional unit failures. Consider an alternate approach wherein these three applications are implemented as an ASPP that can be **programmed to run any one of these applications at any given time.** This ASPP requires three adders and three multipliers. What are the potential

*This research was partially supported by SAMSUNG Electronics Co., Ltd.

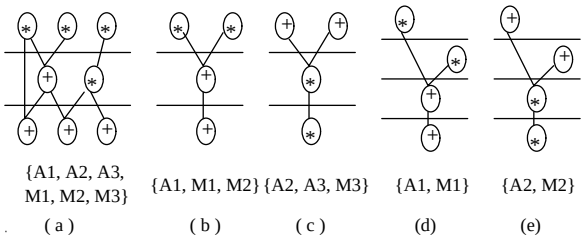


Figure 2: Heterogeneous Built-In Resiliency

benefits of such a multi-functional processor? Consider a hardware allocation in table 2.

dfg	Hardware Allocation	Tolerate Faults in	%Util'n +	%Util'n *
2a	a1,a2,a3,m1,m2,m3	–	44	44
2b	a1,m1,m2	a2,a3,m3	30	26
2c	a2,a3,m3	a1,m1,m2	26	30

Table 2: Hardware allocation for 1-unit fault tolerance

- If all modules are operational, program the processor to implement the CDFG in figure 2(a).
- If adder a2, adder a3, or multiplier m3 is faulty program the processor to implement the CDFG in figure 2(b).
- Similarly, in the presence of a faulty a1, m1 or m2, program the processor to implement the CDFG in figure 2(c).

Note that applications 2(b) and 2(c) tolerate three single-unit faults each. In addition they can tolerate a few two-unit and three-unit faults.
The case of 2-unit fault tolerance is more complex. There are 15 different ways in which two of these units can fail. The schedule in figure 2(b) can tolerate only one of the three possible two-adder-unit faults. Similarly, the schedule in figure 2(c) can tolerate one of the three possible two-multiplier-unit faults. In order to tolerate all possible two-adder-unit faults, at least three schedules similar to figure 2(b) with different operator-to-application assignments are required. Similarly, three schedules similar to figure 2(c) but with different operator-to-application assignments are required to tolerate all possible two-multiplier-unit faults. The two-unit faults and the schedules that tolerate them are summarized in table 3. Notice that three of the two unit faults ((a2, m3), (a1, m2), and (a3, m1)) are tolerated by more than one schedule.
The interconnection and controller overhead of the HBIR ASPP increases with an increase in the number of schedules implemented on it. The number of implemented schedules can be reduced without compromising the k-unit fault-tolerance of the ASPP by reducing the hardware requirements of the implemented application schedules. This can be accomplished by

dfg	Hardware Allocation	Tolerate Faults in	%Util'n +	%Util'n *
2a	a1,a2,a3 m1,m2,m3	–	44	44
2b	a3,m1,m2	(a1,a2),(a1,m3),(a2,m3)	44	30
2b	a2,m1,m3	(a1,a3),(a1,m2),(a3,m2)	44	30
2b	a1,m2,m3	(a2,a3),(a2,m1),(a3,m1)	44	30
2c	a2,a3,m3	(m1,m2),(a1,m1),(a1,m2)	30	44
2c	a1,a2,m2	(m1,m3),(a3,m1),(a3,m3)	30	44
2c	a1,a3,m1	(m2,m3),(a2,m2),(a2,m3)	30	44

Table 3: Hardware allocation for 2-unit fault tolerance

increasing the latency of some application. If the latencies of the CDFGs in figures 2(b) and (c) are increased to four as shown in figures 2 (d) and (e), their hardware requirements decrease by one multiplier and one adder, respectively.
Also, the number of implemented schedules required for two-unit fault-tolerance is reduced from six to three. The resulting hardware allocation is summarized in table 4. The number of schedules can be

dfg	Hardware Allocation	Tolerate Faults in	%Util'n +	%Util'n *
2a	a1,a2,a3 m1,m2,m3	–	44	44
2d	a1,m1	(a2,a3),(a2,m2),(a2,m3) (m2,m3),(a3,m2),(a2,m3)	28	28
2d	a2,m2	(a1,a3),(a1,m1),(a1,m3) (m1,m3),(a3,m1),(a3,m3)	28	28
2e	a3,m3	(a1,a2),(a1,m1),(a1,m2) (m1,m2),(a2,m1),(a2,m2)	28	28

Table 4: Trade-off between number of schedules and performance degradation

also reduced by adding more spare hardware units in addition to those required.

3 Related Research

The most relevant related work can be traced along the two lines of research and development: behavioral synthesis and fault tolerance techniques.
Behavioral synthesis has been an active area of research for more than two decades [3, 8], and numerous outstanding systems have been built targeting both data path oriented and control oriented applications [15, 8]. Behavioral synthesis traditionally has been addressing synthesis and optimization of a single CDFG for sampling rate, area, and more recently power and test hardware overhead minimization [8]. Recently, a few efforts have been reported on behavioral synthesis techniques for fault tolerant designs. Karri and Orailoglu [9] presented scheduling, assignment

and transformation-based methods for fault-tolerance against transient faults. Guerra et al. [4] presented the first work which concentrates on permanent faults. They showed how fault tolerance achieved using a set of spare units can be used for yield and productivity enhancement. Recently Iyer et al [5] introduced a method which explores trade-offs between performance and yield.

Automatic synthesis of self-recovering microarchitectures has been previously addressed. An algorithm that intertwines checkpoint insertion and scheduling to synthesize self-recovering microarchitectures for supporting fault-recovery in hardware was first presented in [10]. Guerra *et. al* have developed synthesis for built-in self-repair using redundant modules[4]. More recently, Blough, et. al. [2] presented an algorithm for recovery point insertion in recoverable microarchitectures. These RT-level techniques for transient and permanent fault-tolerance have been successful in certain situations. The main target for built-in-self-repair (BISR) techniques for yield enhancement are systems that are bit-, byte-, or digit- sliced, and in particular memories [14] and PLAs [6, 14].

4 Preliminaries

Computational, Timing, & HardwareModels: Majority of most popular multimedia, DSP, video, communication, control, and graphics application are defined as periodic computations on a stream of incoming data. Therefore, a natural and proper computational model for those important application domains is synchronous data flow [7]. Each application is defined by a control data flow graph (CDFG) and the set of timing constraints, most commonly throughput requirements. Modern datapath designs, both general purpose and application specific, invariably group registers in register files in order to better enable sharing of control logic and to facilitate area-efficient layouts. We assume the dedicated register file model [12, 11] where each register is connected to a single input of an execution unit, while each unit can send data to an arbitrary number of registers. This model is also exceptionally well suited for implementing fault tolerance for yield enhancement. The control is synthesized by combining different controllers into one using logic synthesis tools resulting in small hardware overhead.

Fault Model and Fault Diagnosis: We assume a widely used single stuck-at fault model [1]. The proposed HBIR approach requires fault detection and diagnosis as a preprocessing step. Any off-line testing and diagnosis scheme such as full-scan, combinational ATPG and BIST can be used. We also assume that the controller is fault free. However, since the area of the controller is usually only a few percent (1-3) of the designs, it can be easily duplicated with a very limited impact on the final area. We also assume that there is limited bus merging. All data transfers emanating from a functional unit are merged into buses. Otherwise, there exists a dedicated bus connecting any two units between which there are data transfers. Faults can occur in either an execution unit, a register file, or an interconnect. A fault in a register file prevents its corresponding execution unit from receiving data, and thus has the same effect as a fault in the execution unit. Also, a faulty interconnect can be treated as a failure in the execution unit at its data-sending connection.

5 HBIR Synthesis: Algorithms

The hardware requirements for implementing a collection of applications (the application bundle) as an ASPP are greater than or equal to those of each application in the bundle. k-unit fault tolerance can be explored by taking advantage of this inherent redundancy of ASPPs together with judicious operator-to-application assignment and application latency determination. The HBIR synthesis problem can be defined as follows:

Given an underlying hardware model and N applications, synthesize a high-performance and minimum area design so that any one of these N applications can be executed at any given time, and for any k-unit failure the design can be resilent (at least one of these N applications is still working).

The design flow is outlined in figure 5. Initially, the applications are bundled together based on their hardware and structural similarity. In the process, the area overhead is minimized. Following application bundling, the latency determination phase is entered wherein the latencies of the individual applications are determined while ensuring the desired k-unit fault-tolerance. Next, the application schedules

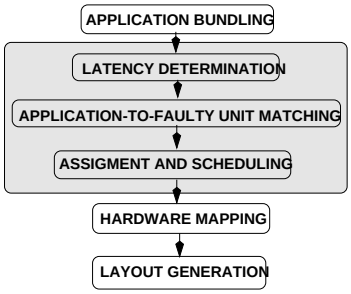


Figure 3: The Design Flow of HBIR Synthesis

are matched to the k-faulty-unit combinations using a branch and bound technique. This step determines the hardware not usable by a given schedule of an application. Based on this information, ASPP assignment and scheduling algorithms are invoked on the applications in a bundle. The HBIR constraints are incorporated during the highlighted phases in the above design trajectory.

5.1 Latency Determination

The latency of each application in the bundle is determined so as to ensure that the resulting HBIR processor can tolerate all k-unit faults. k-unit fault-tolerance is ensured by tuning the hardware utilization of individual applications. Decreasing the system throughput (by increasing the latency) reduces hardware uti-

lization and increases the fault-tolerance capability of an application.

As a first step, all possible k-unit faults are collapsed into smaller sets of faults based on the type of hardware units that fail. For example, in a design that uses adders and multipliers, the 2-unit faults are collapsed into the following sets: • 2-multiply faults, • 2-adder faults, and • 1-adder-1-multiply faults.

Collapsing faults into smaller sets can be represented by the set of tuples:

$$R = \{(r_1, r_2, ..., r_n) | \sum_{i=1}^{n} r_i = k\}$$

where, $r_1, r_2, ..., r_n$ are the number of faulty units of type $1, 2, ..., n$ respectively. For an application bundle with n hardware types and k-unit fault tolerance, the number of tuples in the set R can be computed by the recursion:

$$| R | = f(n,k) = \sum_{j=0}^{\min(H_n,k)} f(n-1, k-j)$$

$$f(1,j) = 1, 1 \leq j \leq k$$

$$f(i,1) = i, f(i,0) = 1, 1 \leq i \leq n$$

The set R together with the unused hardware distributions of the individual applications (obtained using a similar recursion) is used to quickly verify if the selected latencies for the applications can ensure k-unit fault-tolerance. It is checked if for each tuple in the set R, there is at least one application in the bundle whose unutilized hardware units for each type of hardware is larger than number of faulty units of that type in the tuple. If not, the latency of selected applications are increased. Although increasing latency succeeds for low k, additional hardware units are added to guarantee k-unit fault-tolerance.

5.2 Application to Faulty Unit Matching

Towards implementing k-unit fault tolerance, a single schedule for an application may not be sufficient. This is because, although this single schedule may cover a tuple in the set R, it may not tolerate all the k-unit fault combinations that the tuple represents. If out of the H_t hardware units of type t available for the bundle, s_t^i are not utilized by the application i, then $\prod_{t=1}^{n} {}^{H_t}C_{s_t^i}$ schedules of the application may be necessary. The problem is to match the schedules of the applications with the k-faulty-unit combinations so that:

- **there is atleast one schedule that can operate in the presence of a k-unit fault,**
- **atleast one schedule of each application is implemented** and
- **the total number of implemented schedules is minimized.**

The problem is illustrated in figure 4. Each column in the two-dimensional table corresponds to a k-faulty unit combination (denoted as $C_1, C_2, .., C_n$) and each row corresponds to a schedule of an application (denoted as $A_1S_1, A_1S_2, ..., A_mS_{mn}$). If a schedule of an application covers a k-faulty-unit combination, the corresponding cell is marked with an x. The objective is then to ensure that there is at least one x in each column while minimizing the number of rows.

Schedules in ASPP	k-unit fault combinations				
	C_1	C_2	C_3	...	C_n
A_1S_1	x	x			
A_1S_2		x		x	
..					
A_1S_{n1}		x		x	
A_2S_1	x	x			
..					
A_2S_{n2}	x		x		
..					
A_mS_1			x		
A_mS_2		x			x
..					
A_mS_{nm}		x			

Figure 4: Application-to-faulty-unit matching

This problem can be transformed to the vertex covering problem by identifying each of the k-faulty-unit combinations as a vertex and each of the possible schedules of the constituent applications as an edge. A branch and bound technique outlined in figure 5 is used to solve this problem.

The `MinSchSet` is the current best solution that tolerates all the combinations of k faulty units and includes all the schedules initially. The `CurSchSet` tolerates only some of the combinations of k-faulty units and is initially null and grows when the schedule inclusion branch in step 8 is taken. The branching step is invoked when a candidate `sch` is either included into or excluded from the `CurSchSet` by the recursive calls in lines 9 and 12. `Sch`edules are selected one after the other (step 1) from the `SchList`. Associated with each `sch`edule is a vector identifying the k-faulty-unit combinations that are tolerated (covered) by it. The `Cover`() returns this vector. Set union of the coverage vectors of all schedules in the `CurSchSet` is the `CurCover`.

Upper and lower bounds on the number of schedules are used to prune the solution branches. For example, the cardinality of the `MinSchSet` is an upper bound on the number of schedules. If the cardinality of the `CurSchSet` is greater than that of the `MinSchSet`, this schedule inclusion branch and its branches are pruned. For each tuple in the set R, there are $\prod_{t=1}^{n} {}^{H_t}C_{r_t}$ combinations of k faulty units. The sets of k-unit combinations represented by tuples are disjoint. Based on this observation, the `LowerBounds` on the number of schedules for each application are determined as follows. Let the faults represented by a tuple

```
   MinMatch(CurSchSet, SchList, CurCover)
   {
1:  if ((sch←car(SchList)) = φ) return;
2:  appl←GetApplication(sch);

    /* bound the schedule inclusion branch */
3:  if (|CurSchSet| + Σ_{i=appl+1}^{#Appls} LowerBound[i] + 1
      < |MinSchSet|) {
4:    NewCover ← CurCover ⋁ Cover(sch);
5:    if (NewCover covers all k-unit faults) {
6:      MinSchSet←CurSchSet ∪ {sch}; return;
      }
7:    if |NewCover| > |CurCover| {
8:      CurSchSet←CurSchSet ∪ {sch};
9:      MinMatch(CurSchSet,cdr(SchList),NewCover);
10:     CurSchSet←CurSchSet - {sch};
    }}
    /* bound the schedule exclusion branch */
11: if (|CurSchSet ∩ {schedules of appl-1}|
      ≥ LowerBound[appl-1] )
12:   MinMatch(CurSchSet,cdr(SchList),CurCover);
   }
```

Figure 5: Algorithm to find the minimum set of schedules tolerating all the combinations of k faulty units

$(r_1, r_2, ..., r_t, ..., r_n)$ in the set R be covered by application i. A lower bound on the required number of schedules of application i is $\prod_{t=0}^{n} \lceil {}^{H_t}C_{r_t} / {}^{s_t^i}C_{r_t} \rceil$ where H_t, s_t^i and r_t have been previously defined. If the tuple is covered by more than one application, the aggregate number of schedules of the applications which cover the tuple must be considered. Some of the schedules in the `SchList` that have not been visited are absolutely necessary to satisfy this lower bound requirement. Hence, these schedules must also be added to the cardinality of the `CurSchSet` in line 3 when comparing with the upper bound. In line 11, if the number of schedules corresponding to an application is less than the `Lowerbound`, all successive branches are pruned. `car()` and `cdr()` are two lisp-like functions used to return the first and remaining elements in a list, respectively.

5.3 Assignment and Scheduling

At the end of the matching phase, for each version of an application, hardware units that are excluded from its allocation are finalized. In turn, this determines the hardware units that are available for use by a given version of an application. An assignment and scheduling algorithm is invoked on each of the versions of the applications using its usable hardware allocation. Applications can be synthesized in any order as the hardware requirements are determined prior to this step. The resulting HBIR processor is then mapped and synthesized using the Hyper back-end system.

6 Experimental Results

no	appl'n	word len'	avail time	utilization +	utilization -	utilization *	area (mm^2)
1	ARAI	22	10	32	35	65	34.77
2	CSCADE	12	10	40	-	65	11.34
3	DIR	22	22	63	-	68	53.05
4	FFT8	16	6	55	55	33	11.21
5	FIR20	16	12	33	-	41	18.03
6	GM1M	20	14	32	21	25	21.81
7	IIR7	24	10	35	35	50	32.26
8	IIR8	11	12	66	-	75	9.97
9	LEE	22	12	35	33	83	31.51
10	MCM	22	20	38	34	75	35.26
11	PR1	22	10	43	43	55	52.16
12	PR2	22	15	43	43	71	38.50
13	VOLTRA	24	12	83	-	66	28.47
14	WANG	22	14	33	33	78	31.64
15	WVLET	16	15	42	46	47	18.68
16	WDF5	16	12	25	41	50	9.80
17	WDF7	22	12	33	29	33	28.68

Table 5: Example Applications

The HBIR synthesis techniques proposed in this paper were validated on the set of DSP, video, control and communication applications summarized in table 5. The selected applications span a wide range of complexities in computational structures. For each application, columns 3 and 4 show the word length and the input latency, respectively. The hardware utilization of each type is shown in the next three columns. The last column reports the area in mm^2 when the application is implemented as a dedicated ASIC. This is used to evaluate the area overheads of HBIR processors. Eight application bundles were synthesized into k-unit tolerant HBIR processors(k=1,2).

Appl'n Bundles	1-unit # sch	1-unit area mm^2	1-unit over head	2-unit # sch	2-unit area mm^2	2-unit over head
{2, 3}	5	58.86	10.95	7	61.29	13.72
{7, 13}	3	36.01	11.62	4	50.59	56.82
{14, 16}	4	35.35	11.73	4	42.58	34.58
{4, 9}	4	36.10	14.57	7	45.27	43.67
{10, 17}	3	40.26	14.18	7	54.01	53.01
{1, 5, 6}	4	40.35	16.05	6	51.21	35.58
{2, 12}	6	45.37	17.84	11	64.94	68.68
{8, 11, 15}	6	65.42	25.16	8	68.45	31.23

Table 6: 1- and 2-Unit Fault Tolerant HBIR designs

The results of 1-unit fault tolerant HBIR processor synthesis for eight application bundles are summarized in table 6. The number of schedules used in

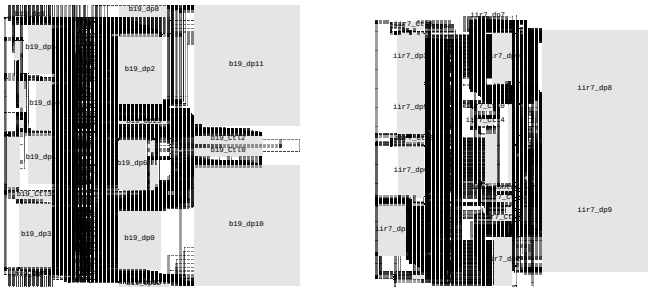
the HBIR processor are shown in column 2. As the number of schedules increases, so do the number of registers for constant coefficients and the interconnection requirement. These are the major causes of the area overhead. The hardware utilization of each type of hardware units for a 1-unit tolerant HBIR processor is shown in the next three columns. The high utilization is due to the fact that HBIR does not require the k spare units of each type for every application. The area overhead vis-a-vis a dedicated implementation of the primary application is summarized in the last column. The areas reported are all in mm^2. The area overhead is 15.3% on average and varies from 11% to 25%. The low overhead is due to udicious application-to-faulty-unit matching and (ii) substituting explicit hardware redundancy with increased utilization of the available computation cycles.

Most fault-tolerance techniques incur significant overheads -either area or performance- when supporting multiple unit fault-tolerance. In contrast, the proposed approach can support 2-unit fault-tolerance with low performance penalty and less than seventy percent area overhead. This is one of the first synthesis systems that has demonstrated the feasibility of automatically synthesizing multiple-fault-tolerant designs with modest area overheads.

On an average, the area overhead is 42.2% and varied from a minimum of 14% to a maximum of 69%. The increase in area overhead is mainly due to (i) additional units that were added during the latency determination phase and (ii) the dedicated coefficient register files used in the current implementation.

6.1 Layout Based Validation

Towards validating the proposed HBIR synthesis approach, we show the layout of an application bundle corresponding to row 2 in table 6. The microarchitecture for this HBIR design is shown in figure 1. The area of the HBIR processor implementation (figure 6(a)) is only 13.6% larger than the dedicated ASIC implementation of the IIR7 filter (figure 6(b)). Towards comparing their relative sizes, the layouts are shown to scale.



(a) (b)

Figure 6: VLSI Layout of an 1-unit fault tolerant HBIR processor implementing IIR7 and VOLTERRA

7 Concluding Remarks

We have presented a novel behavioral level fault-tolerance technique called heterogeneous built in resiliency that combines the flexibility afforded by implementing multiple applications with judicious application-to-faulty-unit matching. The proposed techniques have been implemented and the resulting system has been used to synthesize numerous industrial strength HBIR processors that are k-unit tolerant. The reported system results in very low area overheads for tolerating single functional unit failures and modest overheads for tolerating multiple functional unit failures.

References

[1] M. Abramovici, M.A. Breuer, A.D. Friedman, Digital Systems Testing and Testable designs, Computer Science Press, New York, NY, 1990.

[2] D. M. Blough, F. J. Kurdahi, and S. Y. Ohm, "Optimal Recovery Point Insertion For High Level Synthesis of Recoverable Microarchitectures," *FTCS*, 1995.

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

[4] L.M. Guerra, et al.," High Level Synthesis Techniques for Efficient Built-in Self Repair", IEEE Workshop on DFT in VLSI systems, pp. 41-48, 1993.

[5] B. Iyer, R. Karri, I. Koren, "Phantom Redundancy: A High-Level Synthesis Approach for Manufacturability", ICCAD, pp. 658-661, 1995.

[6] I. Koren, D.K. Pradhan, "Introducing Redundancy into VLSI Designs for Yield and Performance Enhancement", FTCS 15, pp. 330-335, 1985.

[7] E. A. Lee and D. G. Messerschmitt: "Static Scheduling of Synchronous Data flow Programs for Digital Signal Processing", IEEE Trans. on Computers, Vol. 36, No. 1, pp. 24-36, 1987.

[8] M.C. McFarland, A.C. Parker, R. Camposano, "The High-Level Synthesis of Digital Systems," Proc IEEE, Vol. 78, No. 2, pp. 301-317, 1990.

[9] A. Orailoglu and R. Karri, "Automatic Synthesis of Self-Recovering VLSI Systems, IEEE Trans on Computers, Feb 1996.

[10] A. Orailoğlu and R. Karri. "Coactive Scheduling and Checkpoint Determination during the High Level Synthesis of Self Recovering Microarchitectures," *IEEE Trans on VLSI Systems*, 2(3):304-311, 1994.

[11] D.A. Patterson, J.L. Henessy, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[12] J. Rabaey et al., "Fast Prototyping of Data Path Intensive Architectures," IEEE Design & Test, Vol. 8, No. 1, pp. 40-51, 1991.

[13] J.J. Raffel *et. al.*, "A wafer-scale digital integrator using restructurable VLSI", IEEE Trans. on Electronic Devices, No. 32, pp. 479-486, 1985.

[14] D.P. Siewiorek, R.S. Swartz, Reliable Computer Systems: Design and Evaluation, Digital Press, MA.

[15] R. A. Walker and D. E. Thomas. Behavioral transformation for algorithmic level IC design. *IEEE Trans on CAD*, 8(10):1115–1128, 1989.