HETEROGENEOUS BUILT-IN RESILIENCY OF APPLICATION SPECIFIC PROGRAMMABLE PROCESSORS

Abstract

Application Specific Programmable Processors (ASPP) provide efficient implementation for any of k specified functionalities. Using the flexibility provided by multiple functionalities we have developed a novel approach for permanent fault-tolerance called ***Heterogeneous Built-In-Resiliency (HBIR)***. The proposed HBIR technique combines the flexibility provided by multiple functionalities with judicious operation-to-operator assignment either to maximize the permanent fault-tolerance of such ASPP designs or to guarantee that the ASPP remains operational in the presence of all possible $k-$unit faults (for a designer specified k). The proposed multifunctionality based 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 industrial-strength designs.

1 Introduction

Traditionally, high level design tools have been focusing on synthesizing a specific implementation for a given computation which meets a combination of design constraints, such as throughput, latency, and power, while optimizing a primary design goal, area. Recently, it has been recognized that application specific programmable processors (ASPP) are an attractive alternative. ASPPs provide efficient implementation for any of k 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. In fact, the application specific programmable microarchitecture shown in figure 1 implements a seventh order IIR filter (IIR7) and two versions of the volterra filter (VOLTERRA)[1].

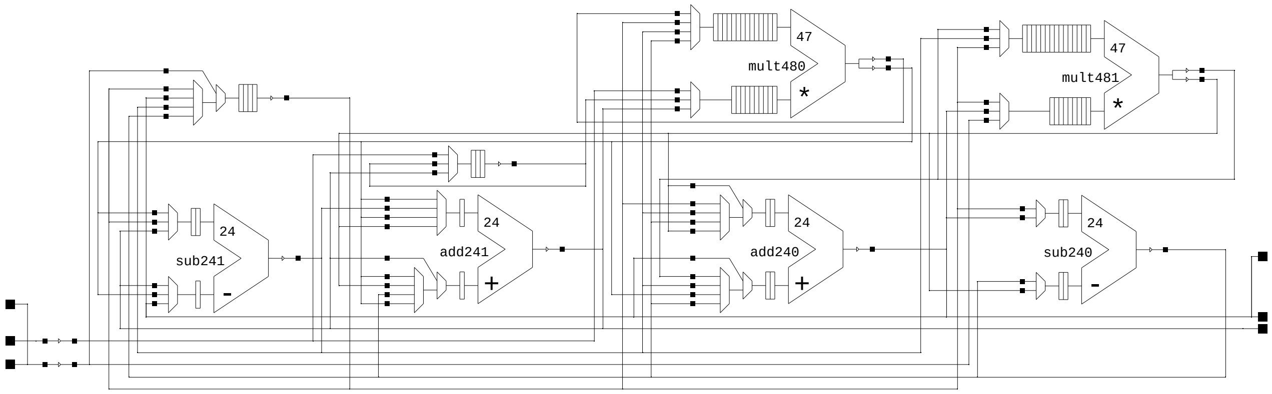


Figure 1: An example ASPP microarchitecture implementing seventh order IIR and VOLTERRA filters

The flexibility and redundancy inherent in such ASPP designs (due to the diverse hardware usages of the implemented applications) are an excellent source for providing transient and/or permanent fault tolerance with no or low overhead. For example, smaller applications which do not use all resources during all control steps in ASPP design can use those resources for providing fault-tolerant functions. In particular, it appears that the advent of the combined scheme with built-in self repair (BISR), yield and productivity enhancement and permanent fault-tolerance will be exceptionally cost effective. For example, on a particular IC one can execute only those applications that do not use the faulty hardware units. In cases when only

[1] this has been synthesized automatically using the reported HBIR synthesis system

limited repair is economically feasible, one can invoke an application on the chip requiring the smallest number of repair steps [Raf85]. Statistically, we still have an IC which realizes all of the required functionalities, but both yield and repair costs will be reduced [Neg89].

Returning to figure 1, if all units are operational, the ASPP microarchitecture can execute either the IIR7 filter or the VOLTERRA filter. In the presence of a single faulty unit, only one of the versions of the VOLTERRA filter is still operational as shown in table 1. Consequently, this ASPP microarchitecture is resilient to all single functional unit failures.

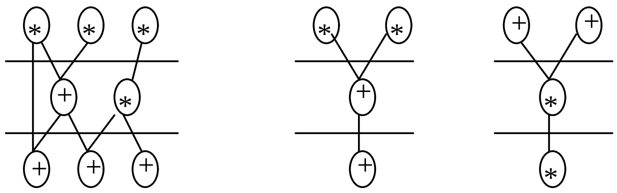
type	instance	IIR7	VOLTERRA 1	VOLTERRA 2
+	add240	$\surd$		$\surd$
	add241	$\surd$	$\surd$	
-	sub240	$\surd$		
	sub241	$\surd$		
*	mult240	$\surd$	$\surd$	
	mult241	$\surd$		$\surd$

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, the degree of fault-tolerance and area and performance overheads.

2 HBIR: A Motivating Example

Towards illustrating the key concepts and importance 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.



{A1, A2, A3, M1, M2, M3} {A1, M1, M2} {A2, A3, M3}

(a) (b) (c)

Figure 2: Heterogeneous Built-In Resiliency

If implemented as a dedicated ASIC, the CDFG shown in figure 2(a) requires three adders and three multipliers. On the other hand, the CDFG shown in figure 2(b) requires one adder and two multipliers. Similarly, the CDFG shown 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 on a single processor that can be ***programmed to run any one of these applications at any given time.*** The resulting heterogeneous programmable processor requires three adders and three multipliers. An obvious question that needs to be addressed is: "What are the potential benefits of such a multi-functional processor?". Consider the hardware allocation shown in figure 2. There are ${}^{6}C_1$ possible ways that a single functional unit can fail. Consider the reconfiguration strategy summarized in table 2:

- 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 k-unit fault tolerance is more complex. Assuming that $k = 2$, there are 15 different ways in which two of these units can fail. The schedule in figure 2(b) can tolerate

DFG Used	Hardware Allocation		Tolerate Faults in	% Utilization	
	Add	Mult		Add	Mult
(a)	A1, A2, A3	M1, M2, M3	–	44.4	44.4
(b)	A1	M1, M2	A2,A3,M3	29.6	25.9
(c)	A2, A3	M3	A1,M1,M2	25.9	29.6

Table 2: A Reconfiguration strategy for 1-unit Fault Tolerance

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. Towards tolerating all possible two-adder-unit faults, at least three schedules similar to figure 2(b) with different operation-to-operator assignments are required. Similarly, three schedules similar to figure 2(c) but with different operation-to-operator assignments are required to tolerate all possible two-multiplier-unit faults. Owing to a judicious operation-to-operator assignment, these six schedules can in addition tolerate all other two-unit faults. The two-unit faults and the schedules that tolerate them are summarized in table 3. Notice that three of the two unit

DFG Used	Hardware Allocation		Tolerate Faults in	% Utilization	
	Add	Mult		Add	Mult
(a)	A1, A2, A3	M1, M2, M3	–	44.4	44.4
(b)	A3	M1, M2	(A1, A2), (A1, M3), (A2, M3)	44.4	29.6
(b)	A2	M1, M3	(A1, A3), (A1, M2), (A3, M2)	44.4	29.6
(b)	A1	M2, M3	(A2, A3), (A2, M1), (A3, M1)	44.4	29.6
(c)	A2, A3	M3	(M1, M2), (A1, M1), (A1, M2)	29.6	44.4
(c)	A1, A2	M2	(M1, M3), (A3, M1), (A3, M3)	29.6	44.4
(c)	A1, A3	M1	(M2, M3), (A2, M2), (A2, M3)	29.6	44.4

Table 3: A reconfiguration strategy for 2-unit fault tolerance

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 increasing the latency of the application. If the latencies of the CDFGs in figures 2(b) and (c) are increased to four as shown in figures 3 (b') and 3 (c'), their hardware requirements decrease to one adder and one multiplier.

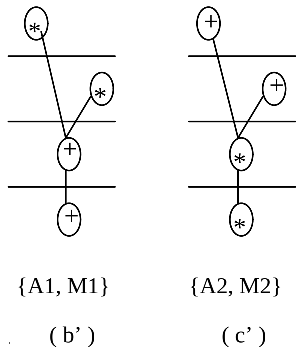


Figure 3: Graceful Degradation of Performance

In turn, the number of implemented schedules required for two-unit fault-tolerance is reduced from six to three. The ensuing reconfiguration strategy is summarized in table 4. The

DFG Used	Hardware Allocation Add	Mult	Tolerate Faults in	% Utilization Add	Mult
(a)	A1, A2, A3	M1, M2, M3	–	44.4	44.4
(b')	A1	M1	(A2, A3), (A2, M2), (A2, M3) (M2, M3), (A3, M2), (A2, M3)	27.8	27.8
(b')	A2	M2	(A1, A3), (A1, M1), (A1, M3) (M1, M3), (A3, M1), (A3, M3)	27.8	27.8
(c')	A3	M3	(A1, A2), (A1, M1), (A1, M2) (M1,M2), (A2,M1), (A2,M2)	27.8	27.8

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

tradeoff between the number of schedules and the hardware utilization can be observed. The number of schedules can be also reduced by adding more spare hardware units in addition to the inherent redundancy.

In a nutshell, the synthesized HBIR processor can be reconfigured even when one or more modules are faulty mainly due to:

1. ***implementing multiple functions on a processor***
2. ***implementing multiple schedules of an application***
3. ***judiciously assigning hardware modules to individual schedules ensuring in the process that at least one application is operational in the presence of k-unit failures***
4. ***trading-off performance with the number of implemented schedules***

The rest of the paper is organized in the following way. We first briefly survey the related work along several dimensions. Next we will discuss the hardware and fault models. In section 5 we formulate the HBIR synthesis problem and describe the proposed algorithms for HBIR synthesis. After presenting experimental results in section 6 we conclude by summarizing the results and outlining the future directions in design of HBIR processors.

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 for yield and productivity enhancement.

Behavioral synthesis has been an active area of research for more than two decades [16, 15, 8], and numerous outstanding systems have been built targeting both data path oriented and control oriented applications [20, 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, 15]. Recently, a few efforts have been reported on behavioral synthesis techniques for fault tolerant designs. Karri and Orailoglu [5] presented scheduling, assignment and transformation-based methods for fault-tolerance against transient faults. Guerra et al. [3] 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 [4] 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 (of operations in the input algorithm to clock cycles) to synthesize self-recovering microarchitectures for supporting fault-recovery in hardware was first presented in [19, 18]. Guerra *et. al* have developed synthesis for built-in self-repair using redundant modules[3]. More recently, Blough, Kurdahi and Ohm [14] presented an algorithm for recovery point insertion in recoverable microarchitectures that minimizes the number of rollback points given constraints on the number of registers and maximum number of time steps between any two rollback points (the stride). 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 [9, 12] and PLAs [6, 12].

4 Preliminaries

Computational and Timing Models: 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.

Hardware Model: 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 [11, 10] shown in figure 4 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].

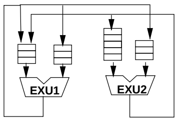


Figure 4: Register File Hardware Model

• 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.

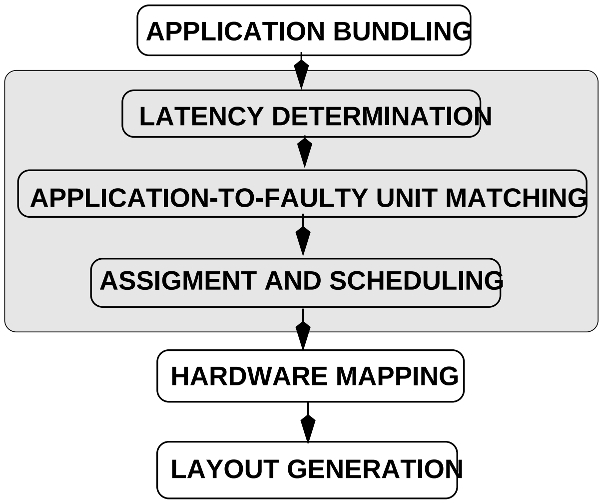
Although in our implementation we followed the presented models, it is important to stress that the approach can be easily retargeted to models of other design platforms.

5 Towards Guaranteeing k-Unit Fault-Tolerance

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-operation assignment and application latency determination. The HBIR synthesis problem for k-unit fault-tolerance 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 are matched to the k-faulty-unit combina-



tions 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. Towards completing the HBIR synthesis trajectory, hardware mapper and layout generator from Hyper are invoked to obtain the layout. Since the HBIR constraints are incorporated during the highlighted phases in the above design trajectory, they form the subject matter of this paper.

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 utilization 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
- 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. Towards this end, 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 Matching Applications with Faulty Units

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 5. 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.

This problem can be transformed to the vertex covering problem [17] 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 6 is used to solve this problem.

The `Minimum_Set_Of_Schedules` is the current best solution that tolerates all the combinations of k faulty units and includes all the schedules initially. The `Current_Set_of_Schedules` 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 `schedule` is either included into or excluded from the `Current_Set_of_Schedules` by the recursive calls in lines 9 and 12. `schedules` are selected one after the other (step 1) from the `List_Of_All_Schedules`. Associated with each `schedule` is a vector identifying the k-faulty-unit combinations that are tolerated (covered) by it. The `Cover()` returns this

Schedules	k-unit fault combinations				
in ASPP	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_2			x		x
..					
A_2S_{n2}	x		x		
..					
A_mS_1			x		
A_mS_2		x			x
..					
A_mS_{nm}		x			

Figure 5: Application-to-faulty-unit matching problem

vector. Set union of the coverage vectors of all schedules in the `Current_Set_Of_Schedules` is the `CurrentCover`.

Upper and lower bounds on the number of schedules are used to prune the solution branches. For example, the cardinality of the `Minimum_Set_Of_Schedules` is an upper bound on the number of schedules. If the cardinality of the `Current_Set_Of_Schedules` is greater than that of the `Minimum_Set_Of_Schedules`, 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 `Lower_Bounds` on the number of schedules for each application are determined as follows. Let the faults represented by a tuple $(r_1, r_2, ..., r_t, ..., r_n)$ 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 `List_Of_Schedules` that have not been visited are absolutely necessary to satisfy this lower bound requirement. Hence, these schedules

```
    Minimum_Match(Current_Set_Of_Schedules, List_Of_Schedules ,CurrentCover)
    {
1:    if (( schedule ← car(List_Of_Schedules)) = φ) return;
2:    appl ← GetApplication(schedule);

      /* Bound the Schedule Inclusion Branch */
3:    if ( |Current_Set_Of_Schedules| + Σ_{i=appl+1}^{#Applications} Lower_Bound[i] + 1
      < |Minimum_Set_Of_Schedules| ) {
4:      NewCover ← CurrentCover ⋁ Cover(schedule);
5:      if (NewCover covers all the combinations of k faulty units) {
6:        Minimum_Set_Of_Schedules←Current_Set_Of_Schedules ∪ {schedule}; return;
        }
7:      if |NewCover| > |CurrentCover| {
8:        CurrentSet←CurrentSet ∪ {schedule};
9:        Minimum_Match(Current_Set_Of_Schedules, cdr(List_Of_Schedules), NewCover);
10:        CurrentSet←CurrentSet - {schedule};
      }}
      /* Bound the Schedule Exclusion Branch */
11:   if (|Current_Set_Of_Schedules ∩ {schedules of appl-1 }| ≥ Lower_Bound[appl-1] )
12:     Minimum_Match(Current_Set_Of_Schedules, cdr(List_Of_ Schedules), CurrentCover);
    }
```

Figure 6: Algorithm to find the minimum set of schedules tolerating all the combinations of k faulty units

must also be added to the cardinality of the `Current_Set_Of_Schedules` 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 `Lower_bound`, 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.

5.4 Experimental Results

application	# of nodes	# of edges	word len'	crit' path	avail time	allocation				utilization			area
						+	-	*	reg	+	-	*	(mm^2)
ARAI	51	63	22	8	10	7	5	2	43(13)	32	35	65	34.77
CASCADE	34	47	12	10	10	4	0	2	39(13)	40	-	65	11.34
DIR	124	138	22	9	22	4	0	4	83(38)	63	-	68	53.05
FFT8	30	31	16	5	6	3	3	1	22(1)	55	55	33	11.21
FIR20	32	42	16	3	12	5	0	2	46(10)	33	-	41	18.03
GM1M	20	27	20	14	14	2	1	2	24(7)	32	21	25	21.81
IIR7	29	39	24	10	10	2	2	2	35(10)	35	35	50	32.26
IIR8	39	57	11	9	12	3	0	2	47(18)	66	-	75	9.97
LEE	57	66	22	10	12	5	3	2	32(11)	35	33	83	31.51
MCM	102	111	22	9	20	4	4	2	68(17)	38	34	75	35.26
PR1	56	65	22	8	10	3	3	4	42(17)	43	43	55	52.16
PR2	66	73	22	9	15	2	2	3	34(12)	43	43	71	38.50
VOLTERRA	30	39	24	12	12	1	0	2	28(9)	83	-	66	28.47
WANG	56	65	22	8	14	4	3	2	44(17)	33	33	78	31.64
WAVELET	53	65	16	14	15	3	1	2	57(14)	42	46	47	18.68
WDF5	23	29	16	12	12	2	2	1	25(6)	25	41	50	9.80
WDF7	31	37	22	12	12	2	4	2	33(7)	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. These examples include Arai's fast DCT algorithm (ARAI), decimation in frequency fast algorithm (DIF), S. Winograd's small-N DFT for N = 8 (FFT8), and ninth degree bi-reciprocal WDF with Butterworth response (WDF9). For each application, columns 2-5 show the number of nodes, the number of edges, the word length, and the critical path, respectively. The input latency for each application is shown in column 6. The next four columns give the hardware allocation. The column titled "reg" shows the number of registers used in the implementation. The numbers in parentheses are the register counts for constants. 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.

Application bundling was invoked on these set of applications with the objective of minimizing the ASPP overhead for each IC. The resulting bundles were synthesized into k-unit tolerant HBIR processors for two different k.

5.4.1 HBIR Synthesis Targeting One Unit Fault-Tolerance

Application Bundles	# sch'	allocation				utilization			area	over
		+	-	*	reg	+	-	*	(mm^2)	head
{CASCADE, DIR}	5	4	0	4	118(68)	46	-	41	58.86	10.95%
{IIR7, VOLTERRA}	3	2	2	2	68(28)	39	18	60	36.01	11.62%
{WANG, WDF5}	4	3	3	2	65(30)	22	32	45	35.35	11.73%
{FFT8, LEE}	4	4	3	2	44(15)	38	50	35	36.10	14.57%
{MCM, WDF7}	3	4	5	2	95(27)	24	27	52	40.26	14.18%
{ARAI, FIR20, GM1M}	4	5	4	2	89(37)	23	19	34	40.35	16.05%
{CASCADE, PR2}	6	3	2	3	81(38)	46	22	60	45.37	17.84%
{IIR8, PR1, WAVELET}	6	3	3	4	141(69)	47	16	36	65.42	25.16%

Table 6: HBIR Synthesis: 1-Unit Fault Tolerance

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 next four columns are the hardware allocations used in synthesizing the HBIR processor. 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 the 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

- ***the judicious application-to-faulty-unit matching*** and
- ***substituting explicit hardware redundancy with increased utilization of the available computation cycles.***

5.4.2 HBIR Synthesis Targeting Two Unit Fault-Tolerance

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 negligible 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. The results are summarized in table 6 for the previously used application bundles.

Application Bundles	# sch'	allocation				utilization			area	over head
		+	-	*	reg	+	-	*	(mm^2)	
{CASCADE, DIR}	7	4	0	4	139(81)	50	-	43	61.29	13.72%
{IIR7, VOLTERRA}	4	3	2	3	84(37)	31	9	49	50.59	56.82%
{WANG, WDF5}	4	3	3	3	91(36)	23	34	29	42.58	34.58%
{FFT8, LEE}	7	4	4	3	52(16)	44	44	24	45.27	43.67%
{MCM, WDF7}	7	4	5	3	108(33)	22	27	33	54.01	53.01%
{ARAI, FIR20, GM1M}	6	5	4	3	100(44)	20	10	28	51.21	35.58%
{CASCADE, PR2}	11	5	2	4	122(59)	31	20	52	64.94	68.68%
{IIR8, PR1, WAVELET}	8	3	3	4	173(83)	42	12	31	68.45	31.23%

Table 7: HBIR Synthesis: 2-Unit Fault Tolerance

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.

The one-unit and two-unit fault-tolerant HBIR processors have significantly superior hardware utilization characteristics when compared to the less than 5% utilization of general purpose processors [10].

5.4.3 Layout Based Validation of HBIR Processors

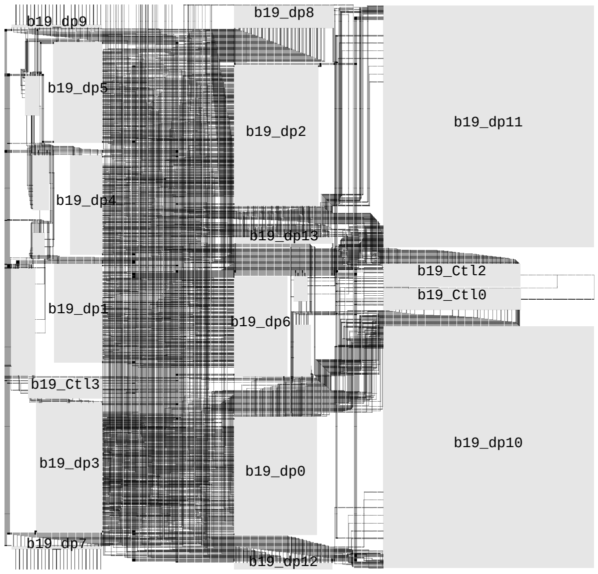
Towards validating the proposed HBIR algorithms and synthesis approach, we have generated the layout of an example bundle as shown in figure 7 using LagerIV for 1-unit fault-tolerance. The application specific programmable microarchitecture of the example is shown in figure 1. Figure 7(a) is the layout implementing the HBIR processor of a bundle consisting of the seventh order IIR filter and two versions of the Volterra filter. Figure 7(b) is the dedicated ASIC implementation of the primary application namely the seventh order IIR filter. The area of the HBIR processor implementation is only 13.6% larger than the dedicated ASIC implementation of the IIR7 filter. This is consistent with the area estimates in the table. Towards comparing their relative sizes, the layouts are shown to scale.

6 Towards Maximizing Fault-Tolerance of ASPPs

Consider the scenario wherein k-unit fault tolerance is expensive and need not necessarily be guaranteed. Fault-tolerance is a a secondary objective and can be maximized using the available hardware resources. The corresponding synthesis problem can be formulated as follows:

Given N hierarchical control data flow graphs, each with its own execution time bound an underlying hardware model, synthesize a minimum area design so that any of these N CDFGs can be executed at any given time and the ***fault-tolerance of the ASPP design is maximized****.*

Let $\{g_1, g_2, ..., g_N\}$ be the N application CDFGs and let U_i be the set of hardware units



(a) (b)

Figure 7: VLSI Layout of a 1-unit fault tolerant HBIR processor implementing IIR7 and VOLTERRA

that are not used in implementing the CDFG g_i. Then the fault-tolerant processor when programmed to implement the application g_i can tolerate all single and multiple unit failures from the set U_i. In general, the resulting processor is tolerant to all single unit failures in the set $U_1 \cup U_2 ... \cup U_N$. In fact, it can tolerate every single function unit failure if $U_1 \cup U_2 ... \cup U_N$ turns out to be the set of all functional units in the design.

Based on these observations we present an algorithm (outlined in fig. 8) to synthesize multifunctional processors whose fault-tolerance is maximized.

```
G = { $g_i$ | bundle of applications}
H = { $h_i$ | hardware units available}
$A_i$ = {hardware units allocated to graph $g_i$}
I($h_i$) = idleness count of functional unit type $i$
I($h_i$) is initialized to 0
Fault-TolerantProcessorSynthesis(G)
{
  /* Derive a feasible solution using ASPP synthesis */
  1:  H ← Find_Feasible_Allocation(G);
  /* Resynthesize to maximize fault-tolerance */
  2:  foreach($g_i \in$ G) {
   3:  $A_i \leftarrow \phi$;
   4:  repeat {
    5:  $h \leftarrow h_k$ s.t.  I($h_k$) is $\max_{h_i \in H - A_i}$ I($h_i$);
    6:  $A_i \leftarrow A_i \cup \{h\}$;
   7:  } until no more hardware required for $g_i$
   /* end fault-tolerant hardware allocation for $g_i$ */
   8:  foreach($h_i \in H - A_i$) I($h_i$)++;
   9:  Assign_and_Schedule($g_i, A_i$);
  } }
```

Figure 8: Synthesis Algorithm to Maximize Fault-Tolerance

6.1 Incorporation of Fault-Tolerance Constraints

Initially, an ASPP allocation, assignment and scheduling step (step 1) is carried out. This determines the exact hardware allocation for the entire design and for each application in the design. Starting with this definitive hardware allocation, the ***fault-tolerance constraints*** can be incorporated by reallocating the available hardware units as shown in steps 2 thro' 9 in figure 8. A ***Least Frequently Used (LFU)*** hardware reallocation heuristic is used to maximize the fault-tolerance. Towards this end, all the hardware units are sorted based on the number of applications that they have already been allocated to. This is captured by the idleness count I. The idleness count is then used to determine the ***least frequently used*** unit of a hardware type and allocates it to the candidate application CDFG. This ***LFU*** strategy matches an application CDFG with those units that have not yet been used by previously synthesized applications. Faults in units that have not been allocated to an application are tolerated by that application. Using the resulting fault-tolerant hardware allocation, the data path required for implementing the multifunctional processor is synthesized (step 9). Observe that the amount of fault-tolerance is limited by the number of hardware units which in turn is determined by the throughput constraints of the implemented applications.

6.2 Towards Maximizing the Fault-Tolerance via HBIR

Table 9 summarizes the results of eight multifunctionality based fault-tolerant processors obtained using the synthesis system. Column 2 shows the applications that are implemented as a multifunctional fault-tolerant processor. The throughput constraints of applications are shown in the next column. In column 7 the modules that are not used by an application are listed. Prefixes A, S, and M stand for adder, subtractor and multiplier, respectively. Failures in any of these modules can be tolerated by programming the processor to implement this application. The following three columns show the average utilization of hardware units. The average utilization for an application is computed by considering the number of fault free modules that are used when the processor is programmed to implement it. For example, in design # 1 the ADAP application is invoked when either subtractor S0 or subtractor S1 is faulty. Consequently, the subtractor hardware utilization for the ADAP is calculated assuming that either S0 or S1 is faulty. Finally, the area of the synthesized processor and the area overhead vis-a-vis the area of the largest dedicated ASIC design are summarized in the last column. On an average the area overhead of the proposed technique is 17.9 %.

Each of the synthesized processors can tolerate all single unit failures. Also, in contrast

#	CDFG	t	#units			tolerates faults in	utilization			area (mm^2)
			+	-	*		+	-	*	
1	PR1	15	2	2	3		43	43	48	27.29 (14.9%)
	ADAP	9	2	0	3	S0,S1	50	0	37	
	WDF5	13	1	2	1	M1,M2,A1	31	38	20	
	GM1M	15	1	1	2	S1,M0,A0	40	13	18	
2	DIF	12	2	3	4	A1	71	33	41	38.61 (20.7%)
	WLET	15	3	1	2	S1,S2,M2,M3	42	24	27	
	LDLP	6	2	2	1	S0,M0,M1,M3,A1	20	24	20	
	IIR7	10	2	2	2	S2,M1,M2,A0	26	26	29	
3	DIT	14	3	2	3		57	42	33	31.24 (25.1%)
	8IIR	23	1	0	2	A1,A2,S0,S1,M2	28	0	25	
	CASC	11	3	0	2	S0,S1,M1	48	0	46	
	FIR20	17	2	0	2	S0,S1,M0,A0	44	0	22	

Table 8: Maximizing the ASPP Fault-Tolerance via HBIR

to the general purpose processors [10], the multifuntional fault-tolerant processors have high resource utilizations.

In table 9 the first three designs are truly multifunctional in that all applications in a bundle are distinct. The next two designs are partially multifuntional in that some of the applications have been used twice. For example in design #4, there are three different applications (namely, WLET, WDF9 and ADAP) and two versions of the WDF9 application. Both these versions have identical latencies but different hardware allocations. Consequently, these two versions tolerate faults in different functional units. Finally, designs 6, 7 and 8 implement multiple versions of a single application (with different latencies). Consequently, these fault-tolerant designs yield ***gracefully degradable*** implementations of a single algorithm. For example, in design #7, the performance degrades by six clock cycles in the presence of any single module failure. In design #8, if any single module from the set {A1, A0, S1, S0, M1, M0} fails the performance degrades by 5 clock cycles while if any single module from the set {A2, A3, M2, M3} fails the performance degradation is only 2 clock cycles. The synthesized data path and layout for the multifuntional fault-tolerant design #2 are shown in figure 9.

#	CDFG	t	#units			tolerates	utilization			area
			+	-	*	faults in	+	-	*	(mm^2)
	WLET	15	3	1	2		42	46	46	
4	WDF9	9	2	1	1	M1,A2	42	44	42	21.36
	WDF9	9	2	1	1	M0,A1	42	44	42	(16.4%)
	ADAP	10	2	0	2	A0,S0	42	0	50	
	ARAI	10	4	4	3		40	35	43	
	WDF7	12	2	4	2	M1,A0,A1	20	29	26	30.30
5	WDF7	12	2	4	2	M0,A2,A3	20	29	26	(17.3%)
	IIR8	13	2	0	2	S1,S2,S3	34	0	49	
						S1,M2				
						A2,A3				
		22	4	0	4		63		68	
		27	3	0	3	A3,M3	60		65	16.79
6	DIR	27	3	0	3	A2,M2	60	-	65	(8.3%)
		27	3	0	3	A1,M1	60		65	
		27	3	0	3	A0,M0	60		65	
		19	4	3	3		39	59	52	
		25	3	2	2	A3,S2,M2	33	53	47	16.05
7	MCM	25	3	2	2	A2,S2,M2	33	53	47	(13.2%)
		25	3	2	2	A1,S0,M0	33	53	47	
		25	3	2	2	A0,S2,M2	33	53	47	
		11	4	2	4		29	59	50	
		13	3	2	3	A3,M3	29	50	49	22.32
8	WANG	13	3	2	3	A2,M2	29	50	49	(27.3%)
		16	3	1	3	A1,S1,M1	22	54	38	
		16	3	1	3	A0,S0,M0	22	54	38	

Table 9: Fault-Tolerant Synthesis

Tolerance to Multiple Functional Unit Failures: In addition to tolerating all single functional unit faults, these designs can tolerate some multiple functional unit failures as well. For example, design #1 can tolerate 33.3 % of all double functional unit failures, and 5.7 % of all triple functional unit failures. This additional fault-tolerance is summarized in Table 10. Furthermore, the improvements in yield and productivity for selected fault-tolerant designs are calculated using Stapper's method [13, 3]. On an average a factor greater than 2 improvement in yield and productivity were obtained.

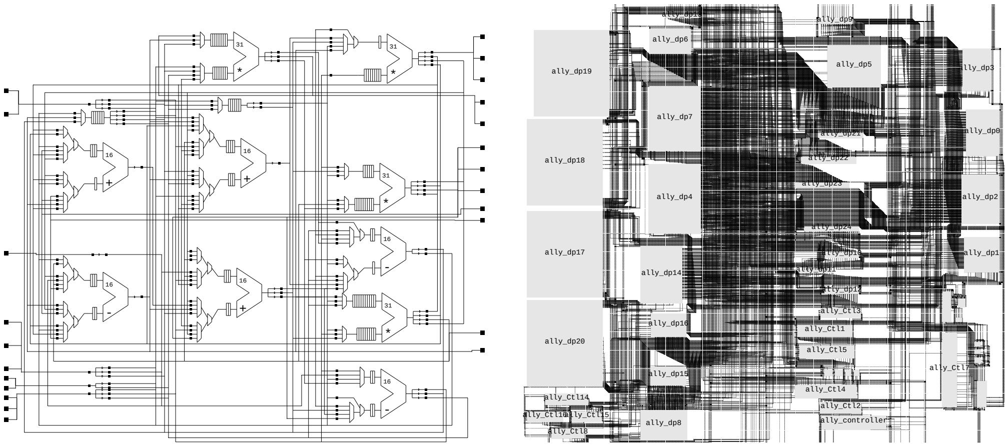


Figure 9: (a) Data path and (b) Layout of the fault-tolerant processor corresponding to design # 2 in table 2

	k-unit failure tolerance			Yield	Productivity
example	$k=1$	$k=2$	$k=3$	Gain	Gain
1	100.0%	33.3%	5.7%	2.31	2.01
2	100.0%	46.6%	15.0%	3.06	2.54
7	100.0%	24.4%	3.3%	2.47	2.18
8	100.0%	17.8%	1.7%	2.23	1.75

Table 10: Multiple Unit Failure Tolerance and Yield Enhancement

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 systemm has been used to synthesize numerous industrial strenght HBIR processors that are k-unit tolerant. This is one of the first computer aided systems that can synthesize designs that can tolerate multiple functional unit failures. The reported system results in very low area overheads for tolerating single functional unit failures and modest overheads for tolerating multiple functional unit failures. We introduced an approach which uses the flexibility provided by multiple functionalities of application specific programmable processors to develop an effective fault-tolerance technique for yield improvement. New synthesis algorithms and software implementations are used to demonstrate significant yield improvements on numerous designs.

References

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

[2] G. De Micheli, Synthesis and Optimization of Digital Circuits, McGraw Hill, New York, NY, 1994.

[3] 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.

[4] B. Iyer, R. Karri, I. Koren, "Phantom Redundancy: A High-Level Synthesis Approach for Manufacturability," ICCAD 95, pp. 658-661, 1995.

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

[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] W.R. Moore, "A review of fault-tolerant techniques for the enhancement of integrated circuit yield," Proc IEEE, Vol. 74, No. 5, pp. 684-698, 1986.

[10] D.A. Patterson, J.L. Henessy, Computer Architecture: A Quantitative Approach, Morgan Kaufmann Publishers, San Mateo, CA, 1989.

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

[12] D.P. Siewiorek, R.S. Swartz, Reliable Computer Systems: Design and Evaluation, 2nd edition, Digital Press, Burlington, MA.

[13] C. H. Stapper, "A New Statistical Approach for Fault-Tolerant VLSI Systems," FTCS 22, pp. 356-365, Boston, MA, 1992.

[14] D. M. Blough, F. J. Kurdahi, and S. Y. Ohm, "Optimal Recovery Point Insertion For High Level Synthesis of Recoverable Microarchitectures," *Proceedings of the 25th Fault Tolerant Computing Symposium*, 1995.

[15] G. De Micheli, Synthesis and Optimization of Digital Circuits, McG raw Hill, New York, NY, 1994.

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

[17] M. R. Garey, D. S. Johnson, "Computer and Intractability: A Guide to the theory of NP-Completeness", W. H. Freeman and Co., New York.

[18] R. Karri and A. Orailoğlu "Synthesis of Optimal Self-Recovering Microarchitectures," *Proceedings of 23rd International Symposium on Fault-Tolerant Computing*, June 1993.

[19] 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.

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