MAXIMIZING THE FAULT-TOLERANCE OF APPLICATION SPECIFIC PROGRAMMABLE SIGNAL PROCESSORS [1]

Kyosun Kim, Ramesh Karri
Department of ECE
University of Massachusetts
Amherst, MA 01003
{kkim,karri}@ecs.umass.edu

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

Abstract – As witnessed by their recent rapid market growth, application specific programmable processors provide an attractive alternative to both fully programmable and fully custom hardware platforms. ASPP are data paths which provide efficient implementation for any of k functional specifications assuming that only one will be executed at any given time. In this research we combine the flexibility provided by multiple functionalities with judicious operation-to-application allocation to maximize the permanent fault-tolerance of such ASPP designs. The approach and the synthesis algorithms are demonstrated on a number of signal processing applications.

1. INTRODUCTION

The market which supplies integrated circuits (ICs) to the application specific consumer and industrial electronics, communication, and computer products has been historically sharply divided into two groups. The first group consists of general purpose platforms, such as microprocessors, DSP and video processors, and microcontrollers. The second group form ASIC chips realized using either full custom or gate-array technologies. The advantages and disadvantages of two groups are also sharply divided. While the former group most often provides lower turn-around time and higher flexibility, the later has a number of advantages due to its lower cost in mass production, higher level of achievable performances, and lower power.

In the last 10 years field-programmable gate arrays (FPGA) have become a popular design implementation medium [13]. While numerous synthesis techniques targeting FPGAs have been developed [14], synthesis methods for ASPPs have been addressed only tangentially. However, the market trends clearly indicate a rapidly growing need for efficient synthesis technologies of ASPP designs. Almost all major semiconductor companies, and in particular all ASIC manufacturers, offer numerous ASPP designs [3, 15].

[1] This research was partially supported by SAMSUNG Electronics, Co., Ltd.

-7803-3134-6/96 $5.00 © 1996 IEEE

1.1 A MOTIVATING EXAMPLE

Multifunctionality-based fault-tolerance is illustrated using the five example control data flow graphs (CDFGs) shown in figures 1(a)-(e). Assume that (i) all operations finish in a single cycle, (ii) the algorithms have identical word lengths and (iii) each algorithm is implemented in three clock cycles.

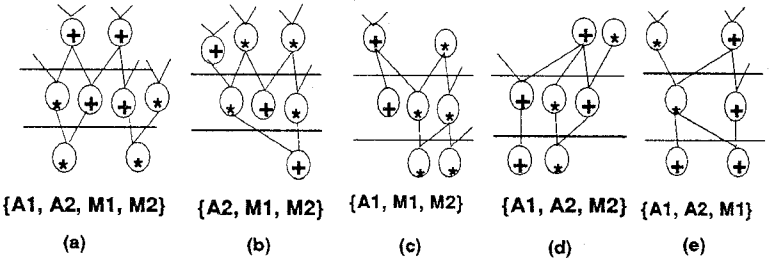

Figure 1: Multifunctionality based Fault-Tolerance

If implemented as a dedicated ASIC, the CDFG shown in figure 1(a) requires two adders and two multipliers. On the other hand, the CDFGs shown in figures 1(b,c) require one adder and two multipliers each. Similarly the CDFGs shown in figures 1(d,e) require one multiplier and two adders each. However, these dedicated ASICs are not tolerant to any fabrication time or in-operation functional unit failures. Consider an alternate approach wherein these five algorithms are implemented on a single processor that can be **programmed to run any one of these algorithms at any given time.** The resulting multifunctional, programmable processor requires two adders and two multipliers. Consider the hardware allocations shown in figure 1 and the fabrication time IC harvesting strategy summarized in table 1.

- If all modules are operational, program the processor to the CDFG in figure 1(a).
- If adder A1 is faulty, program the processor to the CDFG in figure 1(b).
- Similarly, in the presence of a faulty A2, M1 or M2 program the processor to the CDFG 1(c), 1(d), and 1(e) respectively.

Faulty Module	DFG Used	Hardw Allocation		% Utilization	
		Add	Mult	Add	Mult
–	(a)	A1, A2	M1, M2	66.7	66.7
A1	(b)	A2	M1, M2	100	66.7
A2	(c)	A1	M1, M2	66.7	83.3
M1	(d)	A1, A2	M2	66.7	100
M2	(e)	A1, A2	M1	66.7	66.7

Table 1: A Fault-Tolerant Allocation and Reconfiguration Strategy

The synthesized fault-tolerant processor can be used to harvest good ICs even

when one or more modules are found to be faulty at the time of fabrication. This is mainly due to:

- implementing multiple applications on a processor and
- judiciously assigning functional units to individual applications ensuring in the process that at least one application is operational in the presence of a faulty module.

Returning to table 1, it can be seen that the hardware utilization is very high when compared to 1-3 % reported for general purpose programmable processors [10] on page 75.

The rest of the paper is organized in the following way. We first briefly survey the related work. Next we will discuss the hardware and fault models. In section 4 we formulate the ASPP fault-tolerance maximization problem and describe the allocation algorithms that implement it. Finally, in section 5 we present experimental results.

2. RELATED RESEARCH

Behavioral synthesis traditionally has been addressing synthesis and optimization of a single application for sampling rate, area, and more recently power and test hardware overhead minimization [8, 2]. Recently, a few efforts have been reported on behavioral synthesis techniques for fault tolerant design. Karri and Orailoglu [6] 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. 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 and PLAs [9, 12].

3. PRELIMINARIES

Computational, Timing and Hardware 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]. This model also dominates current research and development efforts in behavioral and system synthesis. Each application is defined by a control data flow graph (CDFG) and the set of timing constraints, most commonly throughput requirements.

Modern datapath designs, 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 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 fault-tolerance 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 no bus merging, so there exists a dedicated bus for the output of each unit connecting the output to the inputs of other units. 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.

4. MAXIMIZING THE FAULT-TOLERANCE OF ASPPs

The synthesis problem for maximizing fault-tolerance of ASPPs 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 design is maximized.

We solve this problem in two steps. Initially, an ASPP allocation, assignment and scheduling step is carried out. Briefly, the algorithm starts with an initial estimate of the hardware allocation determined as follows: Let h_j^i be the minimum bound on the necessary amount of hardware of each type j for the i^{th} application. For each hardware type j and for each application i, relaxation based scheduling techniques[11] are used to derive an estimate of h_j^i. For a bundle of applications G, a global min bound $h_j = \max_{i \in G} h_j^i$ is then used as the initial allocation for the j^{th} hardware type. This is because there will be **at least** one application in the bundle that requires at least these many hardware units of type j. The candidate applications are then synthesized one after the other with the ordering determined by criticality metrics such as the minimum hardware requirement of the applications and the scheduling difficulty of nodes in the applications.

4.1. FAULT-TOLERANCE CONSTRAINTS

The previous step 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** are incorporated by real-

locating and reassigning the available hardware units. For a given hardware type, the functional units used in implementing the ASPP will be greater than or equal to those required by the implemented applications. Consequently, once the overall functional unit requirements are known, the hardware allocations for each of the implemented applications can be chosen to maximize the fault-tolerance of the ASPP. This is illustrated in figure 2. $A_1, A_2, ..., A_m$ are the m applications implemented in the ASPP. $H_1^1, H_2^1, ..., H_{n1}^1$ are the all possible hardware allocations for application A_1. Similarly, $H_1^m, H_2^m, ..., H_{nm}^m$ are the possible hardware allocations for application A_m.

Appl'n #	Hardware Allocation	Faulty unit combinations tolerated								
		1-unit				..	k-unit			
		C_1^1	C_2^1	...	C_n^1	..	C_1^k	C_2^k	...	C_n^k
A_1	H_1^1	x	x		x			x		
	H_2^1		x	x			x	x		x
	..									
	H_{n1}^1		x	x			x			
A_2	H_1^2	x	x				x		x	
	H_2^2			x	x	x		x		
	..									
	H_{n2}^2	x		x	x	x				x
.....										
A_m	H_1^m			x			x	x		
	H_2^m		x		x	x		x	x	
	..									
	H_{nm}^m		x							

Figure 2: Application-to-faulty-unit matching problem

If the hardware allocation H_1^1 is used to implement application A_1 then three 1-unit faults (C_1^1, C_2^1, C_n^1) and one k-unit fault (C_n^k) can be tolerated. The fault-tolerance maximization problem is to **choose one hardware allocation for each application so that the number of distinct faulty unit combinations that are tolerated is maximized.** The fault-tolerant allocation algorithm is outlined in figure 3. The fault tolerance of the ASPP is maximized incrementally, starting from one-unit fault-tolerance ($k = 1$) and going upto the maximum achievable k-unit fault-tolerance ($k = MaxK$). $MaxK$ is obtained from the hardware units used in the ASPP and the hardware units required for each of the applications. For each value of k, we `Generate_An_Allocation()` by selecting an allocation for each application. Following which we `Evaluate()` by counting the number of distinct k-faulty unit combinations covered by it. If the number of distinct k-faulty unit combinations covered by the candidate allocation is better than that of the best allocation(s), it becomes the best allocation. However, if the candidate allocation covers the same number of distinct k-faulty unit combinations as

```
MaximizeASPPFaultTolerance( )
{
for (k ← 1; k ≤ MaxK; k++) {
  while ((Candidate_Allocation ← Generate_An_Allocation()) ≠ φ) {
   NumCovers ← Evaluate(Candidate_Allocation);
   UpdateBestAllocation(NumCovers, Candidate_Allocation);
} } }
```

Figure 3: Incremental Fault-Tolerant Allocation

the best allocation(s), it is added to this list of best allocation(s). This list of best allocations is then used as the starting point for improving the k+1-unit fault tolerance. Such an incremental strategy is justified because k-unit failure is more probable than a k+1-unit failure.

4.2 LFU HARDWARE ALLOCATION

Extensive experimentation has revealed that even when the solution space was exhaustively searched for fault-tolerance maximization, the resulting designs were mostly one-unit fault-tolerant. The % of two-unit and three-unit faults that were tolerated were very limited. This is because, for the given hardware, the available redundancy is low. Based on these observations, we have developed a polynomial time allocation algorithm that maximizes one unit fault-tolerance. Let U_i be the set of hardware units that are not used in implementing the application A_i . Then the fault-tolerant ASPP when programmed to implement the application A_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.

A **Least Frequently Used (LFU)** hardware allocation strategy shown in figure 4 is used to maximize the tolerance to single and multiple unit failures. As part of the LFU strategy, all the hardware instances 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 allocates those units that have not already been used to the candidate application. Faults in units that have not been allocated to an application are tolerated by that application.

```
H = { h_i | hardware units available}
H^i = {hardware units allocated to application A_i}
I(h_i) = idleness count of functional unit type i
I(h_i) is initialized to 0
Fault-TolerantHardwareAllocation()
{
  1: foreach A_i {
   2: H^i ← φ;
   3: repeat {
    4: h ← h_k s.t. I(h_k) is max_{h_i ∈ H−H^i} I(h_i);
    5: H^i ← H^i ∪ {h};
   6: } until no more hardware required for A_i
   /* end fault-tolerant hardware allocation for A_i */
   7: foreach(h_i ∈ H − H^i) I(h_i)++;
  }
}
```

Figure 4: Least Frequently Used Fault-Tolerant Allocation

5. EXPERIMENTAL RESULTS

The fault-tolerant hardware allocation techniques proposed in this paper were validated on a large set of DSP, video, control and communication applications. 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). Synthesis modules for hardware mapping and layout generation from HYPER high level synthesis system[11] were used to complete the synthesis trajectory.

Table 2 summarizes the results of eight fault-tolerant ASPPs obtained using the synthesis system. Column 2 shows the applications that are implemented as a fault-tolerant ASPP 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 # 2 the IIR7 application is invoked when subtractor S2 is faulty. Consequently, the subtractor hardware utilization for the IIR7 is calculated assuming that S2 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. In contrast

#	CDFG	t	#units			tolerates faults in	utilization			area (mm^2)
			+	-	*		+	-	*	
	PR1	9	4	4	4		36	36	61	
1	ADAPT	9	2	0	3	A0,A1,S0,S1,S2 S3,M0	25	0	28	
	WDF5	13	2	1	1	A0,A2,S0,S1,S2 M0,M1,M2	12	19	12	37.83 (13.68%+ 1.80%)
	GM1M	15	2	1	1	A1,A3,S0,S1,S3 M1,M2,M3	15	5	12	
	DIF	11	3	3	3		51	36	60	31.05
2	IIR7	10	2	2	2	A0,S2,M0	23	23	33	(17.64%+
	LDILP	6	2	2	1	A1,S0,M0,M2	22	22	22	4.83%)
	WAVELET	15	2	1	1	A2,S1,S2,M1,M2	42	15	31	
	DIT	11	5	2	3		43	54	42	
3	8IIR	23	1	0	2	A1,A2,A3,A4,S0 S1,M2	14	0	23	33.45 (24.35%+
	CASCADE	11	3	0	2	A0,A4,S0,S1,M1	29	0	39	1.54%)
	FIR20	17	4	0	1	A0,S0,S1,M0,M2	23	0	19	
	ADAPT	10	2	0	2	S0	45	0	50	20.27
4	WAVELET	14	2	1	2		68	50	50	(15.50%+
	WDF9	9	2	1	1	M1	50	44	28	2.62%)
	WDF9	9	2	1	1	M0	50	44	28	
	ARAI	9	4	6	3		44	27	48	
5	IIR8	13	1	0	2	A2,A3,S0,S1,S2 S3,S4,S5,M2	31	0	46	33.80 (16.12%+
	WDF7	12	3	0	2	A0,A1,S4,S5,M1	17	19	22	6.92%)
	WDF7	12	3	0	2	A0,A1,S2,S3,M0	17	19	22	
		22	4	0	4		63	-	68	
		27	3	0	3	A3,M3	52	-	55	16.86
6	DIR	27	3	0	3	A2,M2	52	-	55	(4.9%+
		27	3	0	3	A1,M1	52	-	55	4.58%)
		27	3	0	3	A0,M0	52	-	55	
		19	3	4	3		52	45	52	
		25	2	3	2	A0,S0,M0	40	34	40	15.76
7	MCM	25	2	3	2	A0,S1,M1	40	34	40	(7.34%+
		25	2	3	2	A1,S2,M0	40	34	40	6.04%)
		25	2	3	2	A2,S3,M2	40	34	40	

Table 2: Fault-Tolerant ASPP Synthesis: Results

to the general purpose processors [10], the multifunctional fault-tolerant processors have high resource utilizations. The first term within paranthesis in the the last column shows the % overhead of a non-fault-tolerant ASPP. The second term shows the % overhead of rendering these ASPPs fault-tolerant. The area overhead of rendering a basic ASPP fault-tolerant is less than 7%. In table 2 the first three designs are truly multifunctional in that all applications in an ASPP are distinct. The next two ASPPs are partially multifuntional in that some of the applications have been used twice. For example in design #4, there are three different applications (namely, WAVELET, WDF9 and ADAPT) 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 and 7 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. The synthesized data path and layout for the multifuntional fault-tolerant design #2 are shown in figure 5.

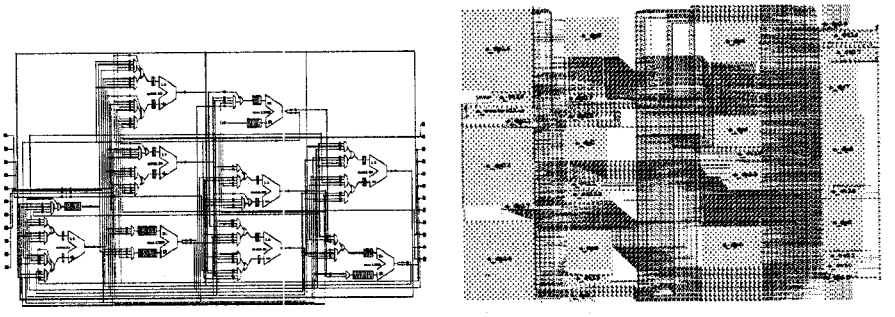

Figure 5: (a) Data path and (b) Layout of the fault-tolerant processor corresponding to design # 2 in table 2

Evalauting the Fault-Tolerance: The fault-tolerance capabilities of the designs synthesized using the two algorithms are summarized in table 3. Except for design #4 all others can tolerate all single functional unit faults. In addition, these designs can tolerate some multiple functional unit failures as well. For example, design #1 can tolerate 84.8% of all double functional unit failures, and 58.6 % of all triple functional unit failures and so on.

6. CONCLUSIONS

We introduced two fault tolerant allocation algorithms during ASPP synthesis. These approaches use the flexibility provided by multiple functionalities of application specific programmable processors to develop an effective fault-tolerance technique. The area overhead of fault-tolerance is less than 7%.

ex	k-unit failure tolerance							
	$k=1$		$k=2$		$k=3$		$k=4$	
	algo 1	lfu	algo 1	lfu	algo 1	lfu	algo 1	lfu
1	100.0%	100.0%	84.8%	81.8%	58.6%	54.5%	33.9%	32.1%
2	100.0%	100.0%	52.8%	50.0%	17.9%	17.9%	4.8%	4.8%
3	100.0%	100.0%	73.3%	66.7%	43.3%	37.5%	21.4%	19.0%
4	60.0%	60.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
5	100.0%	100.0%	67.9%	62.8%	36.4%	35.0%	19.0%	18.9%
6	100.0%	100.0%	14.3%	14.3%	0.0%	0.0%	0.0%	0.0%
7	100.0%	100.0%	26.6%	24.4%	0.0%	3.3%	0.0%	0.0%

Table 3: Multiple Unit Failure Tolerance

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] Fujitsu Microelectronics Inc: "AFP MB86975 Data Sheet", August 1987.

[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 95, pp. 658-661, 1995.

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

[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] J. Rose, A. El Gamal, A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays", Proc. IEEE, Vol. 81, No. 7, pp. 1013-1029, 1993.

[14] A. El Gamal, J. Rose, A. Sangiovanni-Vincentelli, "Synthesis Me thods for Field Programmable Gate Arrays", Proc. IEEE, Vol. 81, No. 7, pp. 1013-1029, 1993.

[15] C.C. Stearns, D. A. Luthi, P.A. Ruetz, P.H. Ang: "A Reconfigurable 64-tap Transversal Filter", IEEE CICC pp. 8.8.1-8.8.4, 1988.