# Algorithm Level Re-Computing --A Register Transfer Level Concurrent Error Detection Technique\*

Kaijie Wu and Ramesh Karri Department of Electrical and Computer Engineering Polytechnic University 6 Metrotech Center, Brooklyn NY 11201 kwu03@utopia.poly.edu ramesh@india.poly.edu

Abstract: In this paper we propose two algorithm-level time redundancy based Concurrent Error Detection (CED) schemes that exploit diversity in a Register Transfer (RT) level implementation. RT level diversity can be achieved either by changing the operation-to-operator allocation (allocation diversity) or by shifting the operands before re-computation (data diversity). By enabling a fault to affect the normal result and the re-computed result in two different ways, RT level diversity yields good CED capability with low area overhead. We used Synopsys Behavior Complier (BC) to implement the technique.

# 1. Introduction

Deep sub-micron VLSI circuits are susceptible to permanent and transient faults. Several techniques for Concurrent Error Detection (CED) recovery and correction have been proposed to target permanent and transient faults. Some of these CED techniques are based on time and hardware redundancy.

Patel and Fung [1,2] developed a logic level time redundancy technique called re-computing with shifted operands (RESO) to detect permanent faults. If an ALU performs a function f, and x is an input, then an error in the ALU is detected by comparing f(x) with right\_shift (f (left\_shift (x)). Error detection capability of RESO depends on the amount of shift. Minero et. al. developed a similar technique called pseudo-duplication [3]. Re-computing using duplication with comparison (REDWC) is another time redundancy technique [4]. [5] extends REDWC by increasing the number of partitions. Some logic level time redundancy techniques use alternating data by checking if  $\overline{f(x)} = f(\overline{x})$ , where x is the input and  $\overline{x}$  is its complement [6]. A CED

where x is the input and x is its complement [6]. A CED technique for array multipliers using bi-directional operations has been presented in [7]. The performance overhead of time redundancy based CED is about 100%.

Triple modular redundancy (TMR) is a hardware redundancy based error correction technique [8] and entails at least 200% hardware overhead. A time redundancy variant of TMR called RE-computing with Triplication With Voting (RETWV) trades-off area overhead for increased error detection/correction latency [9,10,11]. RETWV is similar to REDWC except that it partitions the operations and operators into thirds. RETWV has been applied to inner product units and convolvers [9], Newton-Raphson dividers [10] and Goldschmidt dividers [11]. Performance overhead of time redundancy based error correction is about 200%. Mitra and McCluskey [12] proposed a logic synthesis technique that uses diverse implementations of combinational circuits for CED. They presented a scheme [13] to choose between CED techniques using diversity as a metric.

Several RT level techniques for CED, recovery and correction have been proposed. Karri and Orailoglu [14] and Ravi et. al. [15] developed high-level synthesis algorithms targeting self-recovering data paths. Duplication and comparison of results at checkpoints was used in [14], while duplication and comparison of results as soon as they become available was used in [15]. Although these algorithms reduce the comparison overhead, they do not reduce the almost 100% hardware overhead of duplication. Karri and Iyer presented an RT level CED technique that uses the spare computation cycles and the spare data transfer cycles for CED [16]. Karri and Orailoglu [17] and Lakshminarayana et. al. [18] presented fault security based techniques that yield CED data paths with less than proportional increase in hardware. An RT level built-in self-repair using spare modules was proposed in [19].

In this paper we propose RT level time redundancy based CED techniques that exploit allocation and data diversities. By enabling a fault to affect the normal result and the recomputed result in two different ways, RT level diversity yields good CED capability with low area overhead. However, there is a time overhead. We will describe algorithm level recomputing in section 2, and discuss the underlying fault model in section 3. Then we will describe algorithm level recomputing using allocation diversity in section 4 and algorithm level re-computing using allocation diversity in section 5. CED capabilities of these schemes are analyzed and based on this analysis we propose additional improvements. Experimental results will be discussed in section 6. Finally, conclusions are given in section 7.

# 2. Algorithm level Re-Computing

Consider a Control Data Flow Graph (CDFG) with four additions and one multiplication shown in Figure 1. The schedule in Figure 1 (a) uses two adders, one multiplier and three clock cycles. It does not support CED. Figure 1 (b) shows hardware redundancy based CED technique. C denotes a comparison. Every large circle denotes a logic level fault tolerant operator. This kind of operator duplicates the original operation and carries it out in the same clock cycle as the normal operation. The implementation of the duplicate operation can be identical as the original design (straightforward duplication), or can be different (design diversity). It comes at the cost of double the hardware - four adders and two multipliers, though the computation time has not increased in this design. Figure 1 (c) shows a time redundancy based CED. Here two logic level fault tolerant adders and one logic level fault tolerant multiplier are used and each operation consumes two clock cycles. Although there

<sup>\*</sup> Supported by an NSF CAREER award CCR 996139

is no increase in the number of operators, the computation time increases by 100% to 6 clock cycles.



Figure 1: Scheduled CDFG of (a) basic design (b) fastest design with CED (c) design using logic fault tolerant operator (d) Algorithm level re-computing

Finally, in Figure 1 (d), the comparisons are moved out of the logic level operators to the end of the computation. First, the normal inputs are applied to the design. When the normal computation is finished, the results are saved to registers and the same inputs are applied to the design for re-computation. Following re-computation, the two results are compared with a mismatch suggesting an error. This technique is algorithm level re-computing and is the focus of this paper. By using algorithm level re-computing we do not have to check all the results like logic level CED techniques do; we can check the results periodically.

Algorithm level re-computing is a time redundancy based CED technique that uses two types of computations - the normal computation and re-computation. The normal computation is carried out on all input samples up to R<sup>th</sup> sample. After the R<sup>th</sup> input sample is processed by the normal computation, the result is stored. Then the R<sup>th</sup> result is re-computed and compared to the stored result with a mismatch indicating an error. R, called the checking ratio, is the ratio of the total number of results to the number of results that have been re-computed. Assuming that one iteration of a computation takes M clock cycles, a basic design without CED capability takes N×M clock cycles to process N input samples while the design using algorithm level re-computing takes (N+N/R)×M clock cycles. Hence the time overhead is 1/R ([(N+N/R)M – NM]/NM).

Checking ratio R can be used to trade-off performance overhead against detection latency and fault detection capability. The smaller the value of R, the more results will be re-computed and checked. If R is set to 1, all results are recomputed and checked. Minimum detection latency can be achieved while the time overhead is 100%. If R is set to 2 only half of the results will be re-computed and checked. Detection latency increases while the time overhead is reduced to 50%.

Different implementations of re-computation yield different fault detection capabilities. For example, straightforward duplication of operations in time can only detect transient faults. Permanent faults will be missed because for the same inputs, a hardware module with permanent faults will always produce the same faulty outputs. In this paper we propose allocation diversity and data diversity during re-computing to improve the CED capability.

# 3. RT level fault model

We will focus on transient and permanent stuck-at faults. Although the analysis in this paper is based on stuck-at-1 faults, the results extend to stuck-at-0 faults as well. We model the faults as offsets from the correct result. Consider the 4-bit array multiplier shown in Figure 2. The four adders enclosed in a dashed square form the  $3^{rd}$  bit slice since their sums will be accumulated into the  $3^{rd}$  result bit. Assuming that one of connections (for example, the thick line shown in Figure 2) is stuck-at-1, the faulty result output by the defective multiplier is offset from the correct result by  $2^3$  if the correct output of the thick line is 0. Table 1 summarizes all possible offsets due to one stuck-at-1 fault. Effects of stuck-at faults can be modeled as an offset from the correct result in other arithmetic operators such as adders and subtractors.



Figure 2: 4-bit Array Multiplier

Offset	Condition
2 <sup>i</sup>	If one Sum of i <sup>th</sup> adder slice or Carry of (i-1) <sup>th</sup>
	adder slice is stuck-at-1 and the original output is 0
0	If one Sum of i <sup>th</sup> adder slice or Carry of (i-1) <sup>th</sup>
	adder slice is stuck-at-1 and the original output is 1

Table 1: Possible offsets due to a stuck-at-1 fault of the array multiplier

## 4. Allocation Diversity

In allocation diversity, normal computation and recomputation use the same CDFG, identical RT level schedules and operators to compute results. However, operations in the CDFG that are carried out on an operator in the normal computation are carried out on a different operator during recomputation.



Figure 3: Allocation diversity based CED uses either (a) normal allocation for normal computation or (b) checking allocation for re-computation

Figure 3 shows two implementations of a CDFG that computes  $(a+b) \times (c+d) + (e+f) \times (g+h)$ . Corresponding operations in the normal and re-computations are allocated to different operators. For example, addition a+b is carried out on adder +1 in the normal computation and on adder +3 during re-computation. At the beginning of each computation the controller checks the checking ratio R to decide if it is a normal computation or a re-computation. If it is a normal computation the controller will select the normal allocation shown in Figure 3 (a). Otherwise, the controller will select the checking allocation shown in Figure 3 (b).

## 4.1 CED capability

Returning to example Figure 3, let us assume that only one operator has a stuck-at-1 fault. The faulty results obtained in the normal and re-computations due to this stuck-at-1 fault are summarized in Table 2.

Defective	Possible faulty results from			
module	Normal computation	Re-computation	fault?	
+3	$(a+b)(c+d)+(e+f+2^{i})(g+h)$	$(a+b+2^{i})(c+d)+(e+f)(g+h)$	No	
+4	$(a+b)(c+d)+(e+f)(g+h+2^{i})$	$(a+b)(c+d+2^{i})+(e+f)(g+h)$	No	
+1	$(a+b+2^{i})(c+d)+(e+f)(g+h)$	$(a+b)(c+d)+(e+f+2^{i})(g+h)$	No	
	$(a+b)(c+d)+(e+f)(g+h)+2^{i}$			
	$(a+b+2^{i})(c+d)+(e+f)(g+h)+2^{i}$			
+2	$(a+b)(c+d+2^{i})+(e+f)(g+h)$	$(a+b)(c+d)+(e+f)(g+h+2^{i})$	No	
		$(a+b)(c+d)+(e+f)(g+h)+2^{i}$		
		$((a+b)(c+d)+(e+f)(g+h+2^{i})+2^{i})$		
×1	$((a+b)(c+d)+2^{i})+(e+f)(g+h)$	$(a+b)(c+d)+((e+f)(g+h)+2^{i})$	Yes	
×2	$(a+b)(c+d)+((e+f)(g+h)+2^{i})$	$((a+b)(c+d)+2^{i})+(e+f)(g+h)$	Yes	

Table 2 Faulty results due to a single stuck-at-1 fault in one of the modules used in Figure 3

A stuck-at-1 fault in adder +3 translates into a faulty result of  $(a+b)(c+d)+(e+f+2^{i})(g+h)$  during normal computation and a faulty result of  $(a+b+2^{i})(c+d)+(e+f)(g+h)$  during recomputation. Since these two results differ by  $2^{i}((g+h)-(c+d))$ , the fault can be detected if  $g+h\neq c+d$ . Similarly, a stuck-at-1 fault in multiplier ×2 translates into a faulty result of  $(a+b)(c+d)+((e+f)(g+h)+2^{i})$  during normal computation and a faulty result of  $((a+b)(c+d)+2^{i})+(e+f)(g+h)$  during recomputation. In this case, since the two faulty results have the same offset, the fault may not be detected.

We will now compute the probability of missing a fault when allocation diversity based CED is used. Operations such as multiplication and exponentiation magnify the effects of faults at their inputs making them easy to detect. On the other hand, since operations such as additions and subtractions do not magnify the effects of faults at their inputs, we will focus on parts of CDFGs that have only additions/subtractions. We will consider stuck-at-1 faults in the analysis<sup>1</sup>. When a defective module is used several times, let  $P_{0i}$  be the probability that the expected result is 0 due to the i<sup>th</sup> use of the defective module. Similarly, let  $P_{1i}$  be the probability that the expected result is 1 due to the i<sup>th</sup> use of the defective module.

Consider the CDFG shown in Figure 4. This CDFG uses two adders and takes two clock cycles. It implements (a+b) +(c+d). Assuming the adder +2 (shaded in dark) is the defective module. After the first use of the faulty adder in clock cycle 1, let the probability of a correct result be P<sub>11</sub> and the probability of a wrong result be P<sub>01</sub>. Similarly, after the second use of the faulty adder in clock cycle 2 (assuming that the inputs to the second use are correct), let the probability of a correct result be P<sub>12</sub> and the probability of a wrong result be P<sub>02</sub>. Now we will derive probabilities for three cases: a single stuck-at-1 fault, stuck-at-1 faults in non-adjacent bit positions and stuckat-1 faults in adjacent bit positions.



Figure 4: Example CDFG

#### Case 1. A single stuck-at-1 fault

In Figure 4, every time the faulty adder +2 is used, it will possibly offset the correct result<sup>2</sup> by *X*. We will have the expected result (a+b)+(c+d) with probability  $P_{11} \times P_{12}$  and the wrong result (a+b)+ (c+d+X) with probability  $P_{01} \times P_{12}$ , the wrong result ((a+b)+(c+d))+X with probability  $P_{11} \times P_{02}$  and the wrong result (a+b)+(c+d+X)+X with probability  $P_{01} \times P_{02}$ . These probabilities for the general case when the defective module is used N times are summarized in Table 3.

Offset	Probability	Comments
0	$\prod_{i=1}^{N} P_{1i}$	All correct results are 1
Х	$\sum_{i=1}^N P_{0i} \prod_{\substack{j=1\\j\neq i}}^N P_{1j}$	All correct results except one are 1
$t \times X,$ $1 \le t \le N$	$\sum_{t \in N} \prod_{i \in t} P_{0i} \prod_{\substack{j \notin t \\ j \in N}} P_{1j}$	t of the correct results are 0
N×X	$\prod_{i=1}^{N} P_{0i}$	All the correct results are 0

Table 3: Probabilities. of different offsets due to a single stuck-at-1 fault

A fault will not be detected when the faulty offset from the normal computation is identical to the faulty offset from recomputation. If the defective module is used N times in the normal computation and M times in the re-computation, the probability  $P_t$  that the two results are offset by the same amount  $t \times X$  is:

$$P_{t} = \left(\sum_{t \in N} \prod_{i \in t} P_{0i} \prod_{\substack{j \in N \\ j \notin t}} P_{1j}\right) \left(\sum_{t \in M} \prod_{i \in t} P_{0i} \prod_{\substack{j \in M \\ j \notin t}} P_{1j}\right)$$

Hence, the probability  $P_u$  that a fault is not detected is:

$$P_{u} = \sum_{t=1}^{\min(N,M)} P_{t} = \sum_{t=1}^{\min(N,M)} \left[ \left( \sum_{t\in N} \prod_{i\in t} P_{0i} \prod_{\substack{j\in N \\ j\notin t}} P_{1j} \right) \left( \sum_{t\in M} \prod_{i\in t} P_{0i} \prod_{\substack{j\in M \\ j\notin t}} P_{1j} \right) \right]$$

Figure 5 plots  $P_u$  as a function of N and M assuming that  $P_{0i} = P_{1i} = 0.5$ , (i.e. in the correct output, 1's and 0's are equally likely). From this plot we can observe that the probability of detecting a fault is lowest when N $\approx$ M. Further, the probability of detecting a fault is highest when N >> M or M >> N.



Figure 5: The probability of missing a stuck-at-1 fault

<sup>&</sup>lt;sup>1</sup> A similar analysis can be carried out for stuck-at-0 faults

<sup>&</sup>lt;sup>2</sup> A result is correct if it is the expected result for the inputs even though these inputs may come from a faulty operation.

Error offset	Probability	Comments	
0	$\prod_{i=1}^{n} P_{1i}^{1} \prod_{i=1}^{n} P_{1i}^{2}$	All the results are correct and happen to be 1.	
$u \times X + v \times Y;$ $1 \le u, v \le N$	$\left(\sum_{u \in n} \prod_{i \in u} P_{0i}^{1} \prod_{\substack{j \in n \\ j \notin u}} P_{1j}^{1}\right) \left(\sum_{v \in n} \prod_{i \in v} P_{0i}^{2} \prod_{\substack{j \in n \\ j \notin v}} P_{1j}^{2}\right)\right)$	One fault affects the result u times while the other one affects the final results v times.	
$N \times X + N \times Y$	$\prod_{i=1}^n P_{0i}^1 \prod_{i=1}^n P_{0i}^2$	All correct results are 0.	

Table 4: Probability of different offsets due to two non-adjacent stuck-at-1 faults in adder +2 of Figure 4

$$P_{u} = \sum_{u=1}^{\min(n,m)} \sum_{\nu=1}^{\min(n,m)} \left( \sum_{u \in n} \prod_{i \in u} P_{0i}^{1} \prod_{\substack{j \in n \\ j \notin u}} P_{1j}^{1} \right) \left( \sum_{\nu \in n} \prod_{i \in \nu} P_{0i}^{2} \prod_{\substack{j \in n \\ j \notin \nu}} P_{1j}^{2} \right) \left( \sum_{u \in m} \prod_{i \in u} P_{0i}^{1} \prod_{\substack{j \in m \\ j \notin u}} P_{1j}^{1} \right) \left( \sum_{\nu \in m} \prod_{i \in \nu} P_{0i}^{2} \prod_{\substack{j \in m \\ j \notin \nu}} P_{1j}^{2} \right) \right)$$

Equation 1: Probability of missing two non-adjacent stuck-at-1 faults

# Case 2. Two stuck-at-1 faults in an operator

Let  $P_{0i}^1$  and  $P_{1i}^1$  be the above probabilities due to the first fault and  $P_{0i}^2$  and  $P_{1i}^2$  be the above probabilities due to the second fault. Also, let X be the error offset due to the first fault and Y be the error offset due to the second fault. Assuming that the first fault occurs in a more significant bit position and affects result u times while the second fault occurs in a less significant bit position and affects result v times, the possible final results are { $u \times X + v \times Y$ ;  $0 \le u, v \le N, M$ }.

# a) Stuck-at-1 faults in non-adjacent bit positions

If the faulty bit positions are so far apart (such that  $X > N \times Y$ ), that different (u,v) combinations yield different offsets, the probabilities of correct and faulty results when the defective module is used N times are summarized in Table 4.

The probability of missing a fault  $P_u$  can be calculated using Equation 1. And if we assume  $P_{0i}^1 = P_{1i}^1 = P_{0i}^2 = P_{1i}^2 = 0.5$ , Equation 1 can be simplified as follows:

$$P_{u} = \sum_{u=1}^{\min(n,m)} \sum_{v=1}^{\min(n,m)} {n \choose u} {n \choose v} {m \choose u} {m \choose v} (0.5)^{2(n+m)}$$

Figure 6(a) shows the probability of missing two non-adjacent stuck-at-1 faults.

## b) Stuck-at-1 faults in adjacent bit positions

Since the two faulty bit positions are adjacent,  $X = 2 \times Y$  and hence faults with different (u, v) combinations can produce identical faulty results. For example, a fault with combination (u=1,v=2) has the same effect on the final result as the faults with combinations (u=2,v=0) and (u=0,v=4). Figure 6 (b) shows the probability of missing two adjacent stuck-at-1 faults. Although faults will be missed with a higher probability compared to Case 2a, it is still better than Case 1. Once again the probability of detecting these faults is the highest when N >> M or M >> N.

Comparing Figure 6 (a, b) with Figure 5 shows that the probability of missing a fault decreases as the number of faults in the hardware increases. This is because as the number of faults increases the number of possible faulty results increases thereby reducing the possibility that these faulty results match.



Figure 6: The probability of missing two (a) non-adjacent and (b) adjacent stuck-at-1 faults

#### 4.2 Improving CED capability of allocation diversity

From the above analysis, the CED capability of allocation diversity can be improved by maximizing the difference between the number of times a defective module is used in the normal computation and the number of times it is used in the re-computation. It is not always possible to achieve this unevenness in the allocations for all hardware units in a design. Let us consider Figure 7 as an example. Since the design uses three adders, a single allocation cannot simultaneously maximize the usage difference for all three adders. An operation-to-operator allocation for the normal and the recomputations are shown in Figure 7 (a) and (b), respectively. This allocation minimizes the probability of missing the faults introduced by faulty adder +2 by maximizing the difference between the number of times it is used in the normal and recomputations. In Figure 7 (a) and (b), adder +2 is used once in normal computation and four times in re-computation vielding 0.125 probability of missing a single fault, 0.023 probability of missing two non-adjacent faults and 0.033 probability of missing two adjacent faults<sup>3</sup>. On the other hand, adder +1 is used 4 times in normal computation and 2 times in recomputation yielding a 0.219 probability of missing a single fault, 0.055 probability of missing two non-adjacent faults and 0.082 probability of missing adjacent faults. Finally, adder +3 is used 2 times in normal computation and 1 time in re-

<sup>&</sup>lt;sup>3</sup> Here we use same assumption as above that one bit output has equal likelihood to be 1 and 0.

computation yielding a 0.25 probability of missing a single fault, 0.125 probability of missing two faults and 0.141 probability of missing two adjacent faults.



Figure 7: An example CDFG using allocation diversity (a) normal allocation (b) checking allocation (c) Partitioned CDFG of normal allocation (d) Partitioned CDFG of checking allocation

Partitioning the CDFG into smaller sub-CDFGs and checking the intermediate results output by these sub-CDFGs can improve the CED capability of allocation diversity. Figure 7 (c) (d) show two-way partitioning of the CDFG of Figure 7 (a) (b). As shown in the figure, original CDFG has been divided into two sub-CDFGs A and B and an "×" denotes the intermediate results that are checked. For sub-CDFG A, adder + 1 and adder +2 are used in the normal computation while adder +2 and adder +3 are used in the re-computation. Allocation of sub-CDFG A simultaneously maximizes the usage differences of all three adders; adder +1 is used 4 times in normal computation and 0 times in re-computation, adder +2 is used once in normal computation and 4 times in the recomputation and adder +3 is used once in normal computation and 0 times in re-computation. Similarly allocation for sub-CDFG B maximizes the usage differences for adder +3 and adder +1. In all these cases, if a defective module is involved either in normal computation or re-computation but not both, the probability of missing a fault in it is 0.

### 5. Data diversity

In data diversity the normal computation is carried out on all input samples up to R<sup>th</sup>. After the R<sup>th</sup> input sample is processed by the normal computation, the result is stored in a register. Then the R<sup>th</sup> result is re-computed using shifted operands and compared to the stored result with a mismatch suggesting an error. We name this technique as algorithm level re-computing with shifted operands (ARESO). The RT level data path used in ARESO design is wider than the non-CED design. For example, a data path with original 32-bit wide will increase to 34-bit wide to support 2-bit shift.

# 5.1 CED capability

Logic level RESO and its error detection capabilities have been described in [1, 2]. In algorithm level re-computing, since intermediate results are not checked and a defective module can be used several times before checking the final results, the effect of a fault accumulates. ARESO requires more bits to be shifted to detect same faults as logic level RESO. If a defective adder that offsets a result by  $2^i$  is used twice, the possible offsets can be  $\{0, 2^i, 2 \times 2^i\}$  and ARESO with 1-bit shift is not guaranteed to detect this fault. We calculated the probabilities of missing fault(s) for data diversity using technique similar to that we used for allocation diversity.



Figure 8 The probabilities of missing (a) single stuck-at-1 fault (b) two non-adjacent stuck-at-1 faults (c) two adjacent stuck-at-1 faults by using data diversity

Figure 8 shows the probabilities of missing the single stuckat-1 fault, two non-adjacent stuck-at-1 faults and two adjacent stuck-at-1 faults by using data diversity. In these plots, X-axis stands for the number of times the defective module is used, while the Y-axis stands for the number of bits shifted in the data path. According to the plots, as the number of bits shifted increases, the probability of missing faults decreases. When only one bit is shifted and the defective module is used about 2 to 4 times, the detection probability is the worst. When two bits are shifted in the data path, the probabilities of missing these three types of faults are reduced.

# 5.2 Improving CED capability of data diversity

A straightforward approach to improve the CED capability of a data diversity data path is to shift more bits. However it entails hardware overhead. A second approach is to avoid using a unit less than 4 times. Feasibility of this depends on the number of operations in the CDFG and is not suitable for small CDFGs. Another approach is to partition the CDFG and check the outputs of all sub-CDFGs. If a defective module is used in more than one sub-CDFG, there will be a higher probability to detect the faults. In the CDFG shown in Figure 9 (a), adders 1, 2 and 3 are used 3, 2 and 2 times respectively. Assuming that data path supports 1-bit shift, the probabilities of missing a single stuck-at-1 fault, two non-adjacent stuck-at-1 faults and two adjacent stuck-at-1 faults are {0.14, 0.02, 0.06} for adder 1, and {0.13, 0.03, 0.07} for adders 2 and 3.



Figure 9 (a) Original CDFG (b) Partitioned CDFG

Figure 9 (b) partitions the original CDFG into two sub-CDFGs A and B. Outputs of both of them will be checked. Adder 1 is used once in sub-CDFG A and twice in sub-CDFG B, while adder 2 or 3 is used once in either sub-CDFG. The probability of missing faults can be calculated as:

$$P_{u} = P_{u \text{ of } A} \times P_{B \text{ is correct}} + P_{u \text{ of } B} \times P_{A \text{ is correct}} + P_{u \text{ of } A} \times P_{u \text{ of } B}$$

Using this equation, the probabilities of missing the three types of faults are  $\{0.016, 0, 0\}$  for adder 1 and  $\{0,0,0\}$  for adder 2 and 3.

# 6. Experimental Results

We used Synopsys Behavioral Compiler (BC) [20] to synthesize RT level designs with allocation diversity and data diversity. In this section we will show the results on three examples: Finite Impulse Response (FIR) filter, Windowed Filter and 8-point Discrete Cosine Transform (DCT). Although the experimental data and error detection probabilities are based on stuck-at-1 fault, the technique applies to stuck-at-0 faults as well.

# 6.1 FIR filter

A FIR filter implements:  $Out=In\times Coef(0) + \sum_{i=1..16}Coef(i)\times In(i)$  where In(i) are previous inputs and Coef(i) are constant coefficients. It accepts one input, produces one output and contains 17 multiplications and 16 additions. Our implementation uses three adders and four multipliers and takes 8 clock cycles for each computation.

Table 5 shows the results for the non-CED design, CED designs using allocation diversity and data diversity. The second and third rows show the number of operators used by these designs. The fourth row shows the area consumed in terms of unit cells while the fifth row shows the corresponding area overhead. Because the original design consumes very little hardware, all the proposed schemes involve a large overhead. Rows 6-8 show the probabilities of missing faults in three adders. We considered single stuck-at-1 fault, two nonadjacent stuck-at-1 faults and two adjacent stuck-at-1 faults and combined these three probabilities into one set. Since the four multipliers have similar RT level schedules, we reported the probability of missing faults in one of them in the last row. Allocation diversity using CDFG partitioning reduces the probabilities of missing a fault from around 0.3 to less than 0.04, while data diversity with CDFG partitioning reduces the probabilities of missing faults to almost 0.

	Non- CED	Allocation diversity		Data Diversity 2-bit ARESO	
		basic	partitioned CDFG	basic	partitioned CDFG
Adders	3	3	3	3	3
Multipliers	4	4	4	4	4
Area (unit cell)	9639	11417	12495	12027	12432
Area overhead		18%	30%	25%	29%
Prob. of missing	{1,1,1}	{0.16,0.03,0	{0,0,0}	{0.015, 0,	{0.002,
faults in adder +1		.05}		0.001}	0, 0}
Prob. of missing	{1,1,1}	{0.19,0.04	{0.03,0,0}	{0.024, 0,	{0,
faults in adder +2		0.08}		0.003}	0,0.002}
Prob. of missing	$\{1,1,1\}$	{0.24,0.06,0	{0.04,0,0}	{0.016, 0,	{0,0,0}
faults in adder +3		.11}		0.003}	
Prob. of missing	{1,1,1}	{0.27,0.07,0	{0,0,0}	{0.024, 0,	{0,0,
faults in a mult.		.12}		0.003}	0.002}

Table 5: Experimental results for FIR Filter

# 6.2 Windowed Filter

A windowed filter accepts one input, produces one output and implements  $Out = \sum_{i=0..14} Coef(i) \times [In(i)+In(29-i)]$  using 15 multiplications and 29 additions. Our implementation uses four adders, four multipliers and takes 9 clock cycles for each computation. Table 6 shows all the results. The meaning of each row is same as in Table 5. In this case, because original design consumes a large amount of hardware, area overheads consumed by proposed schemes are around 15%. Both schemes have a lower probability of missing faults in adders than in multipliers. The reason for this is that among the additions allocated to each adder, at least one of them is carried out prior to a multiplication and the effect of the fault(s) in adders is magnified by multiplication. By using CDFG partitioning, the probabilities of missing all possible faults are reduced to almost 0.

	Non-CED	Allocation diversity		Data Diversity 2-bits ARESO	
		basic	partitioned CDFG	basic	partitioned CDFG
Adders	4	4	4	4	4
Multipliers	4	4	4	4	4
Area (unit cell)	72293	80940	82577	81071	82257
Area overhead		12%	14%	12%	14%
Prob. of missing faults in adder +1	{1,1,1}	$\{0, 0, 0\}$	{0, 0, 0}	{0.004, 0, 0}	{0, 0, 0}
Prob. of missing faults in adder +2	{1,1,1}	{0, 0, 0}	{0, 0, 0}	{0, 0, 0}	{0, 0, 0}
Prob. of missing faults in adder +3	{1,1,1}	{0, 0, 0}	{0, 0, 0}	{0, 0, 0}	{0, 0, 0}
Prob. of missing faults in adder +4	{1,1,1}	$\{0, 0, 0\}$	{0, 0, 0}	$\{0, 0, 0\}$	{0, 0, 0}
Prob. of missing faults in mults ×1	{1,1,1}	{0.11, 0.01, 0.02}	{0, 0, 0}	{0.022, 0, 0.002}	{0.004, 0, 0.001}
Prob. of missing faults in mults ×2	{1,1,1}	{0.11, 0.01, 0.02}	{0, 0, 0}	{0.002} {0.015, 0.001, 0.007}	{0, 0, 0.004}
Prob. of missing faults in mults ×3	{1,1,1}	{0.27, 0.07, 0.11}	{0, 0, 0}	{0, 0.011, 0.013}	{0, 0, 0.006}
Prob. of missing faults in mult ×4	{1,1,1}	{0.27, 0.07, 0.11}	{0, 0, 0}	{0, 0.016, 0.023}	{0,0,0}

Table 6: Experimental results for Windowed Filter

## 6.3 A one-dimensional eight-point DCT

An eight points DCT design accepts 8 inputs and produces 8 outputs using 4 adders, 4 multipliers and 19 clock cycles for one computation. Table 7 summarizes the results. In this design, each of the outputs corresponds to a independent sub CDFG. Since in algorithm level re-computing we check all outputs, straightforward allocation diversity achieves 0 probability of missing fault(s).

	Non-CED	Allocation diversity	Data Diversity (2-bit ARESO)
Adders	4	4	4
Multipliers	4	4	4
Total area	42168	48682	53962
Area overhead		15%	28%
Prob. of missing	{1,1,1}	$\{0, 0, 0\}$	$\{0, 0, 0\}$
faults in one operator			

Table 7: Experimental results for DCT

#### 7. Conclusions

We proposed two algorithm level re-computing CED schemes using allocation diversity and data diversity. In allocation diversity the operation-to-operator allocation used in the normal computation is different from the one used in re-

computation. In data diversity operands are shifted before the re-computation. These techniques entail about 10-30% area overhead depending on the size of the original design. Although in some designs these techniques provide good CED capability, they do not do as well in other designs. For such designs partitioning the CDFG and checking some intermediate results increases the CED capability. The area overhead for this enhancement is only slightly larger than that for the basic techniques.

# 8. Reference

- J.H. Patel, L.Y. Fung, "Concurrent Error Detection in ALUs by Recomputing with Shifted Operands," *IEEE Transaction on Computer*, Vol. C.31, No.7, pp. 589 – 595, Jul. 1982.
- J.H. Patel, L. Fung, "Concurrent Error Detection in Multiply and Divide Arrays," *IEEE Transactions on Computer*, Vol. c32, No. 4, pp. 417-422, Apr. 1983.
- 3. R. H. Minero, A.J. Anello, R.G. Furey, L.R Palounek, "Checking by Pseuduplication," *US3660646*, May 1972.
- B.W. Johnson, J.H. Aylor, H.H. Hana, "Efficient Use of Time and Hardware Redundancy for Concurrent Error Detection in a 32-bit VLSI Adder," *IEEE Journal of Solid-State-Circuits*, pp. 208-215, Feb. 1988.
- T.H. Chen, L.G. Chen, Y.S. Chang, "Design of Concurrent Error-Detectable VLSI-Based Array Dividers," *Proceedings of IEEE International Conference on Computer Design*, pp. 72-75, Oct. 1992
- D.A. Reynolds, G. Metze, "Fault Detection Capabilities of Alternating Logic," *IEEE Transactions on Computers*, Vol. C27, No.12, pp. 1093-1098, Dec. 1978
- T.H. Chen, Y.P. Lee, L.G. Chen, "Concurrent Error Detection in Array Multipliers by BIDO," *Proceedings of IEE Computers and Digital Techniques*. Vol. 142, No.6, pp. 425 – 430, Nov. 1995.
- 8. B.W. Johnson, "Design and Analysis of Fault-Tolerant Digital Systems," *Addison-Wesley*, 1989.
- 9. E. Swartzlander, Y.M. Hsu, "Efficient Time Redundancy for Error Correcting Inner-Product Units and Convolvers," *Proceedings of IEEE International workshop on defect and fault tolerance in VLSI systems*, pp. 198-206, Nov. 1995.
- 10. W.L. Gallagher, E.E. Swartzlander, "Fault Tolerant

Newton-Raphson Dividers using Time Shared TMR," *Proceedings of IEEE International Symposium on defect and fault tolerance in VLSI systems*, pp. 240 – 248, Nov. 1996.

- W.L. Gallagher, E.E. Swartzlander, "Error-Correcting Goldschmidt Dividers Using Time Shared TMR," *Proceedings of IEEE International Symposium on defect and fault tolerance in VLSI systems*, pp. 224 – 232, Nov. 1998.
- S. Mitra, E.J. McCluskey, "Combinational Logic Synthesis for Diversity in Duplex System", *Proceedings* of *IEEE International Test Conference*, pp. 179-188, Oct. 2000.
- S. Mitra, E.J. McCluskey, "Which Concurrent Error Detection Scheme to Choose", *Proceedings of IEEE International Test Conference*, pp. 985-994, Oct. 2000.
- R. Karri, A. Orailoglu, "Scheduling with Rollback Constraints in High-level Synthesis of Self-Recovering ASICs," *Proceedings of Fault Tolerant Computing*, pp. 519-526, Jul. 1992
- S.S. Ravi, R. Narasimhan, D.J. Rosekrantz, "Efficient Algorithms for Analyzing and Synthesizing Fault-Tolerant Datapaths," *Proceedings of IEEE International workshop* on defect and fault tolerance in VLSI systems, pp. 81 – 89, Nov. 1995.
- R. Karri, B. Iyer, "Introspection: A register Transfer Level Technique for Concurrent Error Detection and Diagnosis," ACM Transactions on Design Automation of Electronic Systems, vol. 7, no.1, Jan. 2002.
- R. Karri, A. Orailoglu, "Time-constrained scheduling during high-level synthesis of fault-secure VLSI digital signal processors," *IEEE Transactions on Reliability*, pp. 404 – 412, Sep. 1996.
- G. Lakshminarayana, A. Raghunathan, N.K. Jha, "Behavioral Synthesis of Fault Secure Controller/Datapaths using Aliasing Probability Analysis," *Proceedings of Fault Tolerant Computing*, pp. 336–345, Jun. 1996.
- L.M. Guerra, M.M. Potkonjak, J.M. Rabaey, "High level synthesis techniques for efficient built-in-self-repair," *Proceedings of IEEE International Workshop on Defect* and Fault Tolerance in VLSI Systems, pp. 41–48, 1993.
- 20. <u>http://www.synopsys.com/</u>