Exploiting Idle cycles for Algorithm Level Re-Computing*

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

Although algorithm level re-computing techniques can trade-off the detection capability of Concurrent Error Detection (CED) vs. time overhead, it results in 100% time overhead when the strongest CED capability is achieved. Using the idle cycles in the data path to do the re-computation can reduce this time overhead. However dependencies between operations prevent the re-computation from fully utilizing the idle cycles. Deliberately breaking some of these data dependencies can further reduce the time overhead associated with algorithm level re-computing.

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.

Antola and his colleagues presented a register transfer (RT) level CED technique in which normal and re-computation data paths share hardware resources when 'aliasing' probability is not harmed [5,6]. This technique reduces the hardware overhead associated with the hardware redundancy based CED. They also provided a semi CED technique that uses the idle cycles in the data path to implement the re-computation [7]. However the idle cycles in an iteration of normal computation may not be enough to implement the re-computation. A re-computation may span several successive iterations of normal computation. Karri and Iyer presented a RT level CED technique that uses the spare computation cycles and the spare data transfer cycles for CED [12]. Karri and Orailoğlu [13] and Lakshminarayana et. al. [14] presented fault security based techniques that yield CED data paths with less than proportional increase in hardware.

Blough et. al. presented an algorithm to find the optimal checkpoints in a roll back based system [8]. The algorithm either searches for the shortest length recovery point path by a given maximum number of registers, or searches for the lowest cost recovery point by given the maximum recovery points delay. Karri and Orailoğlu [10] and Ravi et. al. [11] also developed high-level synthesis algorithms targeting self-recovering data paths. Duplication and comparison of results at checkpoints was used in [10], while duplication and comparison of results as soon as they become available was used in [11]. Hamilton and Orailoğlu presented a roll forward based transient fault recovery technique by grouping operations in a scheduled and check-pointed Control Data Flow Graph (CDFG) of an algorithm into non-overlapping sub CDFGs called strings [1]. Upon detecting a fault the faulty string is recomputed in parallel with the computation of the subsequent strings. This roll forward technique reduces the performance overhead associated with recovery.

Hamilton and Orailoğlu presented a mirror hamming code to identify the faulty unit [2]. The track of a string includes the units that carry the operations in this string. To be able to identify the faulty unit, any two units must be differentiated such that they are used exclusively in at least two different tracks. By using the mirror hamming code to allocate the operations, any faulty unit can be identified.

An RT level built-in self-repair technique using spare modules was proposed in [15]. Chan and Orailoğlu presented a methodology to re-configure the data path upon detecting a faulty unit [3]. An L/U block matrix is constructed such that columns of the matrix indicate the hardware units while rows indicate clock cycles. If the i^{th} hardware unit is busy at j^{th} clock cycle, the cell (i,j) indicates an operation. The cells above the diagonal construct the U block while the cells below the diagonal construct the L block. If the i^{th} unit is diagnosed as faulty, operation in the U block that is originally bound to H_j, $j<i$, will be bound to H_{j-1}, while the operations of L block that is originally bound to H_j, $j>i$ will be rebound to H_{j+1}. Orailoğlu extended this technique to register operations as well [4]. Iyer, Karri and Koren presented an area-efficient technique for fabrication-time re-configurability called phantom redundancy. Phantom redundancy adds extra interconnect so as to render the resulting micro architecture reconfigurable in the presence of any (single) functional unit failure [9].

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^{th} sample. After the R^{th} input sample is processed by the normal computation, the result is stored. Then the R^{th} result is re-computed and compared to the stored result

*Supported by an NSF CAREER award CCR 996139

with a mismatch indicating an error. Different implementations of re-computation yield different fault detection capabilities. Straightforward duplication of operations in time will miss permanent faults because for the same inputs a hardware module with permanent faults will always produce the same faulty outputs. Karri and Wu proposed two algorithm level re-computing techniques that exploit RT level diversities (data diversity and allocation diversity) [16]. 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. 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. The checking ratio R is the ratio of the total number of results to the number of results that have been re-computed. R can be used to trade-off fault detection capability against time overhead associated with CED. When R is set to 1, the strongest CED capability is achieved at the cost of 100% time overhead. In this paper we propose a new RT level time redundancy based CED technique that exploits the idle cycles in the data path. Deliberately breaking the data dependence of re-computation and using the idle cycles in the normal computation data path can reduce the time overhead associated with algorithm level re-computing. We will discuss the underlying fault model in section 2. Then we will describe algorithm level re-computing using idle cycles in section 3. CED capability of the proposed scheme is analyzed. Experimental results will be discussed in section 4 and faults injection study is given in 5. Finally, conclusions are presented in section 6.

2. 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 a fault as offset from the correct result. Consider the 4-bit array multiplier shown in Figure 1.

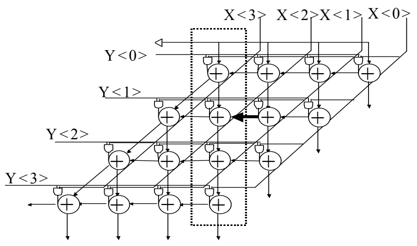

Figure 1: 4-bit Array Multiplier

The four full adders enclosed in a dashed rectangle form the 3rd bit slice since their sums will be accumulated into the 3rd result bit. Assuming that one of connections (for example, the thick line shown in Figure 1) 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 a 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.

Offset	Condition
2^i	If one Sum of i^{th} adder slice or Carry of $(i-1)^{th}$ adder slice is stuck-at-1 and the original output is 0
0	If one Sum of i^{th} adder slice or Carry of $(i-1)^{th}$ 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

3. Algorithm Level Re-Computing using Idle Cycles

Consider a CDFG representation of an algorithm with three additions (+1, +2, +3) and two multiplications (×1, ×2) shown in Figure 2 (a). Two adders (A1 and A2), two multipliers (M1 and M2) and three clock cycles (C1, C2 and C3) are used. All operators are not used in every clock cycle. For example, Out of the six available add computation cycles (two adders × three clock cycles) only half of them are being used. Data dependencies exist between operations ×1 and +1, ×2 and +1, +1 and +2, +1 and +3.

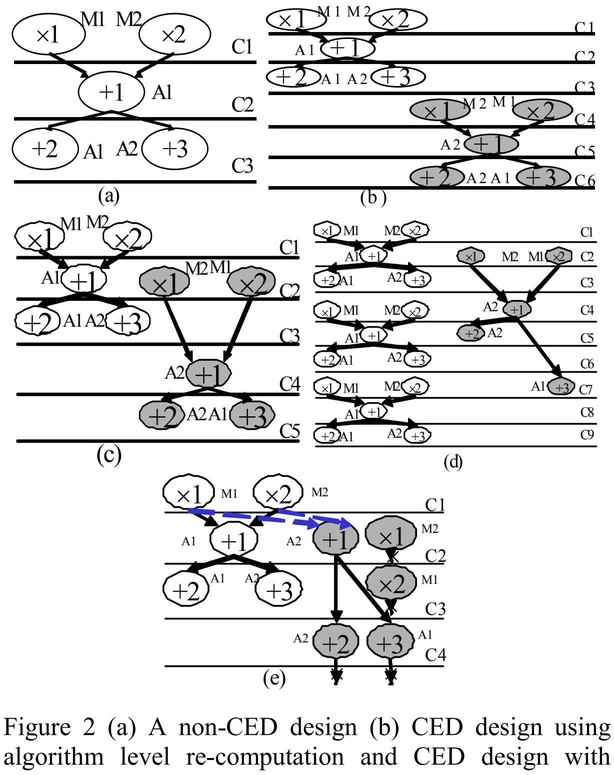

Figure 2 (a) A non-CED design (b) CED design using algorithm level re-computation and CED design with normal and re-computations using (c) the same CDFG (d) the semi CED and (e) different CDFGs

In algorithm level re-computing, the re-computation (the part of CDFG constituted by shaded operations) uses the same CDFG and identical RT level schedule as the normal computation, and starts only after normal computation finishes as shown in Figure 2 (b). It uses 6 clock cycles and when checking ratio is 1 it entails 100% time overhead. The proposed technique uses the idle cycles in the normal computation data path to implement re-computation as shown in Figure 2 (c). It uses the same CDFG but a different RT level schedule when comparing to normal computation. No extra operator is used. Meanwhile, operations in the CDFG that are carried out on an operator in the normal computation are carried out on different operators during re-computation. The re-computation starts from clock cycle C2 and ends at clock cycle C5 resulting in 67% time overhead. The dependence between two multiplications and addition +1 prevents re-computation from using the idle add computation cycle in clock cycle C2. The semi CED technique [7] also has this problem. As shown in Figure 2 (d), the data dependence between additions 1 and 2 forces the re-computation to span three iterations of normal computation.

By breaking these dependencies we can further reduce the time overhead associated with CED. Figure 2 (e) shows the idea. In this CED design, the re-computation of addition +1 receives operands from normal computation data path and is carried in clock cycle C2. The results of multiplications ×1, ×2 and additions +2, +3 will be checked respectively with a mismatch indicating an error. The re-computation does not use extra operator and starts from clock cycle C2 and ends at clock cycle C4 thereby reducing the time overhead to 33%.

The procedure of generating the CDFG used for re-computation is listed as follows:

1) First, we duplicate the CDFG used for normal computation and use the idle cycles in the normal computation for re-computation as shown in Figure 2 (c). If the time overhead of this CED design satisfies the user constraint, we stop. Otherwise, go to the next step.
2) Next, we identify the idle cycles in the newly constructed CED design and the As Soon As Possible (ASAP) schedule for each operation. Since we only need to consider the data dependencies in the normal computation data path, the ASAP schedule of an operation in re-computation is the clock cycle in which its corresponding normal computation is carried. For example in Figure 2 (e) the ASAP clock cycle of the re-computation of addition +1 is C2.
3) Next, we identify the first idle cycle for each operation by choosing the first available idle cycle between its ASAP schedule and its current schedule. Select the operation that has the maximum difference between its current schedule and its first idle cycle and partition the CDFG used for re-computation by breaking the data edges between the selected operation and its predecessors. After partitioning, the selected operation will receive operands from normal computation. Meanwhile the outputs of its original predecessors (i.e. in re-computation) will be checked.
4) Next, schedule the selected operation at its first idle cycle and reschedule the CDFG of re-computation. If the time overhead is satisfying, we stop. Otherwise, go to the second step.

In algorithm level re-computing, we either change the operation-to-operator allocations or shift inputs before re-computation. However, as we analyzed in [16], faults may be missed even when RT level diversity is exploited. Return to the allocation diversity technique shown in Figure 2 (b), the probability of missing a single stuck-at fault in multipliers M1 and M2 is 0.25. This probability is even higher when checking ratio is larger than 1. On the other hand, to be able to use the idle cycles in the data path of normal computation; the idle cycles based algorithm level re-computing deliberately breaks some data edges, partitions the CDFG into parts and checks the results of all the sub-CDFGs. A defective operator that is used in multiple sub-CDFGs is checked several times thereby inherently increasing the probability of detecting the faults in it. The probability of missing a single stuck-at fault in any of the operators shown in Figure 2 (e) is 0.

4. Experimental Results

We used Synopsys Behavioral Compiler (BC) [17] to synthesize RT level designs using idle cycles. In this section we will show the results on two examples: Finite Impulse Response (FIR) filter and Windowed Filter. 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. Meanwhile, we also implemented these two examples using allocation diversity and semi CED [7] and compared the area and performance overheads of the new technique.

4.1 FIR filter

A FIR filter implements: Out=In×Coef(0) + $\sum_{i=1..16}$Coef(i)×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.

Table 2 shows the results for the non-CED design and CED design using allocation diversity, idle cycles and the semi CED in column 2, 3, 4 and 5 respectively. The second and third rows show the number of operators used by these designs. In this example, all the CED techniques use the same number operators as non-CED design. 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, the proposed scheme involves 53% area overhead while the semi CED design consumes 74.3% area overhead. The number of clock cycles and the corresponding time overhead are listed in row 6 and 7 respectively. Comparing to the 8 clock cycles used by non-CED design, the CED design using allocation diversity uses 16 clock cycles and results in 100% time overhead while the CED design using idle cycles uses 12 clock cycles and results in 50% time overhead. In the semi CED design, the re-computation takes three iterations of normal computation translating to 24 (3×8) clock cycles. However since in semi CED technique the re-computation does not interrupt normal computation, the time overhead associated with this technique is 0. The last row shows the probabilities of missing faults in one of the operators. Since different designs use different operation-to-operator allocations, here only the worst missing probability among all the operators is shown. We considered single stuck-at-1 fault, two non-adjacent stuck-at-1 faults and two adjacent stuck-at-1 faults and combined the probabilities of missing these faults into one set. The probabilities of missing faults in CED design using allocation diversity and semi CED design are similar, because both of them only check the final results. The CED design using idle cycles partitions the CDFG of re-computation into 7 sub-CDFGs, hence the probability of missing fault(s) in any operator is almost 0.

	Non-CED	CED		
		A.D	I.C	SEMI
Adders	3	3	3	3
Multipliers	4	4	4	4
Area (unit cell)	9568	11417	14649	16676
Area overhead	--	19.3%	53%	74.3%
Time (clock cycle)	8	16	12	24
Time overhead	--	100%	50%	0
Prob. of missing faults in one op.	{1,1,1}	{0.27, 0.07, 0.12}	{0,0,0}	{0.27, 0.07, 0.12}

Table 2: Experimental results for FIR Filter

4.2 Windowed Filter

A windowed filter accepts one input, produces one output and implements Out = $\sum_{i=0..29}$ Coef(i)×In(i) using 30 multiplications and 29 additions. Table 3 shows the results for all designs. The meaning of each row is same as in Table 2. The non-CED design uses four adders, four multipliers and takes 11 clock cycles. In this case, while the CED designs using allocation diversity and idle cycles do not use extra operator, the semi CED design uses one more multiplier and one more adder and its re-computation spans two iterations of normal computation. The area overheads of three CED designs are 23%, 25.5% and 57.1%, while the time overhead of them are 100%, 54.5% and 0, respectively. Meanwhile, since the CDFG of re-computation in the CED design using idle cycles is partitioned into 8 parts, the CED design achieves almost 0 probability to miss faults in any operator, while the design using allocation diversity has {0.18, 0.04, 0.09} to miss the three kinds of faults and the design using semi CED has about {0.20, 0.04, 0.09}.

	Non-CED	CED		
		A.D	I.C	SEMI
Adders	4	4	4	5
Multipliers	4	4	4	5
Area (unit cell)	76635	94290	96204	120370
Area overhead	--	23%	25.5%	57.1%
Time (clock cycle)	11	22	17	22
Time overhead	--	100%	54.5%	0
Prob. of missing faults in one op.	{1,1,1}	{0.18, 0.04, 0.09}	{0, 0, 0}	{0.20, 0.04, 0.09}

Table 3: Experimental results for Windowed Filter

5. Fault injection study

We used the Windowed Filter example for the transient and permanent faults injection study. To simulate the injection of stuck-at-1 fault, we appended every operator with an OR gate. One input of the OR gate is the output of the operator while the other input controls fault injection. The output of the operator is stuck-at-1 if this input is 1. A stuck-at-0 fault can be similarly injected by appending an AND gate to the operator and applying 0 to one of the AND inputs. We implemented the fault injection study as a C++ program.

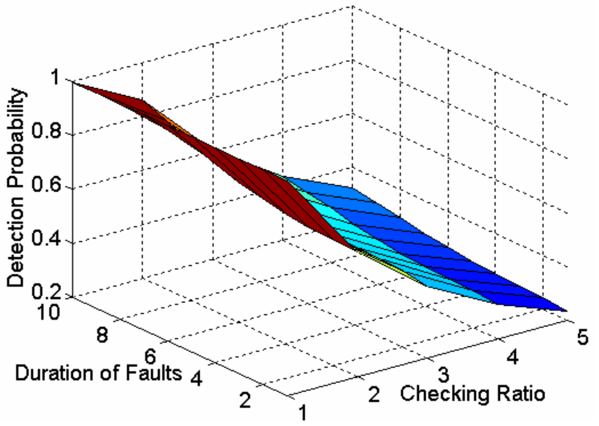

Figure 3 Transient faults detection probabilities of CED design using idle cycles

First we injected transient faults and computed how many of these faults are detected and how many faults of them are missed. There are two reasons why a fault is missed: The proposed CED technique inherently misses some faults as we analyzed in section 3. A fault may also be missed when the time between re-computations is larger than the duration of the fault. Figure 3 plots the probability of detecting transient faults. X-axis shows the duration of the transient fault and Y-axis shows the checking ratio of the CED design. From the simulation, transient faults with longer duration are easier to detect and the designs with smaller checking ratios have stronger CED capability.

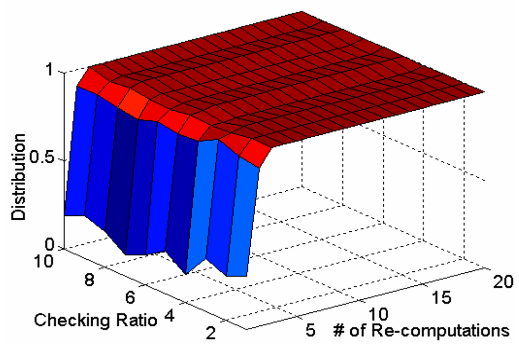

Figure 4. The probability distributions of the number of re-computation of CED design using idle cycles

Next we injected permanent faults and studied the average number of re-computations necessary to detect a permanent fault. Figure 4 plots the probability distributions of the number of re-computations versus checking ratio. X-axis shows the different checking ratios while the Y-axis shows the number of re-computations. As shown in the figure, 2-3 re-computations are able to detect 99% injected permanent faults.

6. Conclusions

We proposed a new RT level time redundancy based CED technique that exploits the idle cycles in the data path. According to the experiment results, the CED design using idle cycles has a strong CED capability and 50% - 60% time overhead. Meanwhile, it consumes less hardware than semi CED technique. The proposed CED technique is applicable to the designs that have strong data dependence and a few idle clock cycles in their data paths.

7. Reference

1. S.N. Hamilton and A. Orailoğlu, 'Transient and Intermittent Fault Recovery without Rollback'. *Proceedings of IEEE International symposium on defect and fault tolerance in VLSI systems*, pp. 252-260, 1998.
2. S.N. Hamilton and A. Orailoğlu, 'Microarchitectural Synthesis of ICs with Embedded Concurrent Fault Isolation'. *Proceedings of 27th Annual International Symposium on Fault-Tolerant Computing*, pp. 329-338, 1997.
3. W. Chan and A. Orailoğlu. "High-Level Synthesis of Gracefully Degradable ASICs," *Proceedings of European Design and Test Conference*, pp. 50-54, 1996.
4. A. Orailoğlu. 'Graceful Degration in Synthesis of VLSI Ics,' *Proceedings of IEEE International symposium on defect and fault tolerance in VLSI systems*, pp. 301-311, 1998.
5. A. Antola, V. Piuri, M. Sami, 'Optimizing high-Level Synthesis for Self-Checking Arithmetic Circuits,' *Proceedings of IEEE International symposium on defect and fault tolerance in VLSI systems*, pp. 268-276, 1996.
6. A. Antola, V. Piuri, M. Sami, 'High-level Synthesis of Data Paths with Concurrent Error Detection,' *Proceedings of IEEE International symposium on defect and fault tolerance in VLSI systems*, pp. 292-300, 1998.
7. A. Antola, F. Ferrandi, V. Piuri, M. Sami, 'Semiconcurrent Error Detection in Data Paths'. *IEEE Transactions on Computer*, Vol.50, No. 5, pp. 449-465, May, 2001.
8. D.M. Blough, F.J. Kurdahi, S.Y. Ohm, 'Optimal Algorithms for Recovery Point Insertion in Recoverable Microarchitectures,' *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 16, No. 9, Sept, 1997
9. B. Iyer, R. Karri, I. Koren. "Phantom redundancy: a high-level synthesis approach for manufacturability", *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 658-661, Nov. 1995.
10. 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
11. 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.
12. 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. 6, no.4, Oct. 2001.
13. 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.
14. 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.
15. 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.
16. K. Wu, R. Karri, "Algorithm Level Re-Computing -- A Register Transfer Level Concurrent Error Detection Technique," *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, Nov. 2001.
17. http://www.synopsys.com