Improving Autonomous Soft-error Tolerance of FPGA through LUT Configuration Bit Manipulation

Anup Das, Shyamsundar Venkataraman and Akash Kumar
National University of Singapore, Singapore
Email: {akdas, shyam, akash}@nus.edu.sg

***Abstract*—Soft-errors in LUT configuration bits of FPGAs can alter the functionality of an implemented design, rendering it useless, unless re-programmed. This paper proposes a technique to improve autonomous fault-masking capabilities of a design by maximizing the number of zeros or ones in LUTs. The technique utilizes spare resources (XOR gates and carry chain) of FPGA devices to selectively manipulate LUT contents using two operations – LUT restructuring and LUT decomposition. Experiments conducted with a wide set of benchmarks from MCNC, IWLS 2005 and ITC99 benchmark suite on Xilinx Virtex 6 FPGA board demonstrate that the proposed methodology maximizes logic 0/1 of LUTs by an average 20% achieving 80% fault-masking with no area overhead. The fault-rate of the entire design is reduced by 60% on average as compared to the existing techniques. Further, an additional 5% fault-masking can be achieved with a 7% increase in slice usage.**

I. Introduction

Field-Programmable Gate Arrays (FPGAs) are emerging as an attractive alternative to Application Specific Integrated Circuits (ASICs) due to faster turnaround time, low cost and programming flexibility. Static Random-Access Memory (SRAM) is the most prevalent memory technology used in FPGAs (e.g. Xiling, Altera). For most of the modern SRAM-based FPGAs, SRAM cells constitute 90% of all the logic elements on the device. An inadvertent change in value of one or more of these SRAM cells due to single event upsets (SEUs)[1] can potentially alter the functionality of an implemented design. Such errors manifest as permanent faults until the SEU affected bit(s) are re-written. The increasing concern of SEUs in FPGAs has attracted significant attention in recent years. Two popular solutions to this problem are hardware redundancy e.g. Triple Modular Redundancy (TMR) [1] and configuration scrubbing [2] [3]. However, both these techniques are associated with high overhead (area and power for the former and reconfigurable delay for the latter). Some of the low overhead solutions to the aforementioned problem include fault masking [4]–[9] and information redundancy [10].

The technique proposed in this paper involves logic manipulation for autonomous fault-tolerance and therefore techniques [5]–[9] are discussed in more detail. A technique is proposed in [5] where dual outputs of modern FPGAs are ANDed/ORed depending on the logic masking effectiveness (ANDing for LUT with more zeros and ORing for LUTs with more ones). The logic implemented in the LUTs are not modified to maximize the number of zeros or ones in the LUTs. The proposed technique is shown to achieve high fault-masking with small area overhead. As shown in Section IV, the technique proposed in this paper modifies the LUT implemented logic and improves fault-masking by 20% using the spare resources on FPGA devices.

[1]SEUs are caused by alpha and neutron particles striking the device.

Another technique proposed in [7] maximizes the identical configuration bits for complementary inputs of a LUT, thereby reducing the propagation of faults seen at a pair of complementary inputs. The technique preserves the functionality and the topology of the LUT network (*in-place*) while maximizing the fault masking. This technique reduces the relative fault rate by 48% and increases the Mean Time To Failure (MTTF) by 1.94 times with no area overhead. An in-place decomposition technique is proposed in [6] where faults in SRAM bits are masked by decomposing a LUT logic into 2 smaller LUT logic functions using the dual output feature of modern FPGAs. The decomposed functions are then combined back to the initial logic using unused carry-chains within a logic block. This technique improves MTTF of Xilinx Virtex 5 FPGAs by 1.43 times. One limitation of these two techniques is that they are limited to combinatorial circuits only.

Contributions: This paper proposes a fault-masking technique for autonomous fault-tolerance of the LUTs of SRAM-based FPGAs. Key contributions in this respect are the following.

- Maximization of zeros and ones of LUT configuration bits through LUT restructuring
- Controlled decomposition of LUTs for higher granularity of fault-tolerance
- A generic technique for combinatorial and sequential circuits

Experiments conducted with a diverse set of benchmarks from MCNC, IWLS and ITC99 benchmark suite on Virtex 6 FPGA board from Xilinx demonstrate that the proposed technique maximizes the number of zeros or ones in LUT by an average 20%. Fault-masking of 80% is achieved for the entire set of benchmarks which is 22% better as compared to the state-of-art techniques. Further, fault-masking can be increased by another 5% with 7% increase in the number of slices. Monte Carlo simulations with randomly injected faults show that the proposed technique tolerates 60% more faults on average for the entire design for all the benchmarks considered.

The rest of the paper is organized as follows. A brief overview of the FPGA architecture and the fault masking of LUT is provided in Section II. The design flow is introduced in Section III with a brief overview of the two key components

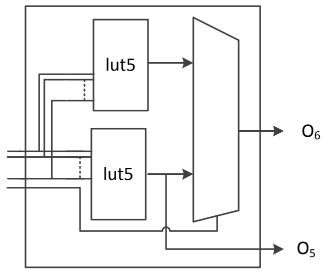


Fig. 1. Dual output feature of modern FPGA LUT

– LUT decomposition and LUT restructuring. Experimental setup and results are discussed next in Section IV. Finally, conclusions are drawn in Section V along with scope for future enhancements.

II. Autonomous Fault Masking of LUT

Xilinx Virtex 6 FPGA devices consist of 6-input LUTs. Each 6-LUT internally consists of two 5-LUTs as shown in Figure 1. LUTs of FPGAs from other vendors such as Altera also resembles this structure. The two outputs (o_5 and o_6) of a 6-LUT can be used individually to implement two different 5-input functions in the two component LUTs. The block can also implement one 6-input function in which case the o_5 output is unusable. If not all inputs of the LUT are used to implement a function, one of the component LUTs remains unused. Specifically, if the used inputs of an n-LUT is less than n, the number of unused entries in the LUT is at least 2^{n-1}. This has motivated researchers to focus on free LUT entries to provide autonomous fault-tolerance. A LUT is said to be autonomous fault-tolerant if it is able to tolerate faults without system or user intervention.

Let the number of used inputs of a LUT be r, where $r < n$. If the same content is duplicated in the two component LUTs of an n-LUT and the two outputs are ANDed, any $0 \rightarrow 1$ faults in the 2^r used entries can be tolerated. In a similar manner, if the two outputs are ORed, any $1 \rightarrow 0$ faults can be tolerated. If n_0 and n_1 denotes the number of zeros and ones respectively in the used entries then $n_0 + n_1 = 2^r$. The total number of faults possible in the entries is $2 * 2^r = 2^{r+1}$ (every entry can have a stuck-at 0 (SA0) and stuck-at 1 (SA1) fault and therefore total number of SA0 faults and SA1 faults are same and equal to 2^r). The SA0 (and respectively SA1) faults for logic-0 (and logic-1) entries are benign. The total number of faults which can impact the circuit behaviour is therefore 2^r. If the two outputs of the component LUTs are ANDed (respectively ORed), all SA1 faults of logic-0 entries (respectively SA0 faults of logic-1 entries) can be tolerated. The total faults tolerated is therefore n_0 (ANDing) or n_1 (ORing). Assuming the possibility of ANDing or ORing, the maximum fault masking possible for the LUT is given by

$$FM = \frac{\max(n_0, n_1)}{2^r} \tag{1}$$

III. Design Flow

Figure 2 shows the FPGA-based design implementation flow. The conventional flow adopted by most FPGA vendors

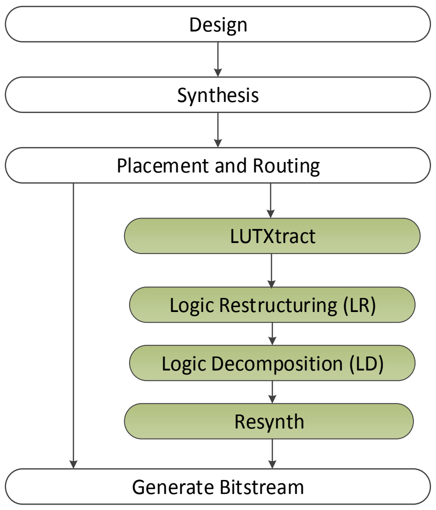


Fig. 2. Autonomous fault-tolerance design flow

are marked with the white boxes in the figure. The boxes in gray are the steps introduced for autonomous fault-tolerance. The first step towards this is the extraction of the LUT and its contents from the *place and route* netlist. For Xilinx based design flow, this information is available in the netlist circuit description (*ncd*) file generated in the LUT mapping part of the *Placement and Routing* step. The LUT extraction is performed in the *LUTXtract* block of the proposed design flow. Following this step, are the two operations – logic restructuring (*LR*) and logic decomposition (*LD*). The effectiveness of the two operations are evaluated in Section IV. Finally, the *Resynth* block modifies the gate netlist by making necessary connections with the carry chain and spare xor gates and prepares it for bitstream generation. The components of the design flow are introduced next.

A. LUT extraction

The LUT extraction step is provided as pseudo-code in Algorithm 1. The algorithm takes a placed and routed *ncd* file and generates a database of LUTs consisting of the following information – *support* and *composition*. These are defined as follows.

Definition 1: (Support of a LUT) *The support of a LUT is the set of used inputs of the LUT.*

As an example, if a 6-LUT (with inputs $A[5:0]$) is used to implement a function $y = (A[0] \oplus A[1])A[2]$, the *support* is the set $\{A[0], A[1], A[2]\}$.

The *support* of a logic function is the same as the support of the LUT used to implement the function.

Definition 2: (Composition of a LUT) *The composition of a LUT is a tuple consisting of the indexed content of a LUT.*

The *composition* of an n-LUT is represented as $\langle a_0, a_1, \cdots a_{m-1} \rangle$, where $m = 2^n$ and $a_i \in [0, 1]$. If the input to the LUT is denoted by $A[(n-1)$ downto $0]$, then a_i is the content of the LUT[2] at location *bin2dec(A)*, where *bin2dec* routine converts

[2]Content of a LUT is determined by the logic function it implements.

Algorithm 1 LUT extraction

```
Input: Netlist circuit description (ncd) file
Output: $LUTDB$
1: xdl = ncd2xdl(ncd)
2: [support composition] = RapidSmith(xdl)
3: $LUTDB$ = [support composition]
```

a binary number to its equivalent decimal.

The first step in Algorithm 1 is the conversion of the *ncd* file to Xilinx Description Language (*xdl*) [11]. This is a proprietary format of Xilinx consisting of clear-text representation of the implemented design allowing designers to get access to a very low-level description of the FPGA's internal state. The *ncd2xdl()* routine provided in the Virtex 6 tool chain is used to convert the same. The *xdl* file is then input to *RapidSmith* [12] tool to generate the *support* and *composition*. These are then stored in the $LUTDB$ database for use in the subsequent steps.

B. Restructuring of LUT

The restructuring of a LUT involves selective inversion of some entries of the LUT to maximize the number of zeros or ones. The following definitions are provided for the problem formulation.

Definition 3: (0-SENSITIVITY OF A SUPPORT) *The* 0-sensitivity *of a support of a LUT is defined as the set of indices in the LUT for which the value of the support is logic 0.*

If the positions (indices) of a 3-LUT with inputs $A[2:0]$ is the set $\{0,1,2,\cdots,7\}$, then *0-sensitivity* of $A(0)$ is the set $\{0,2,4,6\}$, that for $A(1)$ and $A(2)$ are the sets $\{0,1,4,5\}$ and $\{0,1,2,3\}$ respectively. It is not difficult to see that the cardinality of the *0-sensitivity* of any support of a LUT is 2^{n-1}, where n is the total number of supports of the LUT.

Similarly, the *1-sensitivity* of a LUT support can also be defined. The *0,1 sensitivity* of a support i is denoted by S_i^0 and S_i^1 respectively.

The proposed logic restructuring technique involves determining a support of a LUT and the corresponding sensitivity such that, logic inversion of the content of the LUT at the positions specified in the sensitivity list maximizes the number of zeros or ones in the LUT. Continuing with the same example as above, the *1-sensitivity* of the three inputs $A(0)$, $A(1)$ and $A(2)$ are respectively $\{1,3,5,7\}$, $\{2,3,6,7\}$ and $\{4,5,6,7\}$. The content of LUT at positions specified by each of the 6 sets (*0-sensitivity* and *1-sensitivity* of the three inputs) are inverted one at a time and the fault-masking is determined. The set that gives the highest fault-masking is recorded for the LUT.

Clearly, selectively inverting the LUT content leads to a different implemented functionality than original. However, by using *XOR* or a *XNOR* gate, the original function can be easily recovered. Specifically, if f be the original output of a LUT (i.e. implemented by the tool) and f' be the output of the LUT after inverting the LUT content of S_i^1, then, $f = f' \oplus i$. Instead, if S_i^0 is used, then $f = \overline{f' \oplus i}$.

Algorithm 2 provides the pseudo-code for the logic restructuring technique. For each support of the LUT, the *0/1 sensitivity* are determined and the fault-masking is calculated.

Algorithm 2 LUT restructuring

```
Input: $LUTDB$, $T$
Output: $LUTDB_n$
1: for all $lut \in LUTDB$ do
2:   compute $FM$ of $lut$ according to Equation 1
3:   $FM_{best} = FM$, $sup_{best} = \emptyset$, $sen_{best} = \emptyset$, $lut_{best} = lut$
4:   for all $i \in support(lut)$ do
5:     for all $j \in [0,1]$ do
6:       generate $S_i^j$
7:       $\forall k \in S_i^j, lut(k) = \overline{lut(k)}$
8:       compute $FM$ of $lut$
9:       if $FM > FM_{best}$ then
10:        $FM_{best} = FM$, $sup_{best} = i$, $sen_{best} = j$
11:        $lut_{best} = lut$
12:      end if
13:    end for
14:  end for
15:  $LUTDB_n.push(lut_{best})$
16: end for
```

At the output of the algorithm, a support is determined along with its sensitivity type.

C. Decomposition of LUT

Synthesis of optimal boolean logic is a well studied research topic for FPGA technology mapping [13]–[15]. One of the fundamental operations in logic synthesis is to minimize circuit routing complexity by logic decomposition. This involves breaking down a large boolean function into smaller components, keeping the functionality unchanged. The following definitions are provided.

Definitions and lemmas

Definition 4: (DECOMPOSABILITY OF LUT) *Let $f(X)$ be a function implemented in a LUT. The LUT can be decomposed and represented as $f(X) = h(g(X_1, X_2), X_2)$ where $X = X_1 \cup X_2$.*

Figure 3 shows the decomposition of the logic function f.

Definition 5: (MIN SET OF A LUT) *The* min set *of a LUT is the set of indices for which the LUT contents are logic* 1.

The *min set* of a LUT is given by

$$ms = \{i | a_i = 1,\ \forall 1 \le i \le m\} \tag{2}$$

where m is the number of LUT entries.

Definition 6: (CUT OF A MIN SET) *The* cut *of a* min set *is defined as the decomposition of the* min set *into s smaller sets (c_i, $\forall 1 \le i \le s$) sharing the minterms.*

Mathematically, this can be expressed as

$$ms = \cup_{i=1}^{s} c_i \tag{3}$$

The *cut* can be overlapping (common elements in *cut sets*) or non-overlapping (otherwise).

Definition 7: (ORDER OF A CUT) *The* order *of a* cut *is defined as the maximum number of* cut sets *formed from the decomposition of the corresponding* min set.

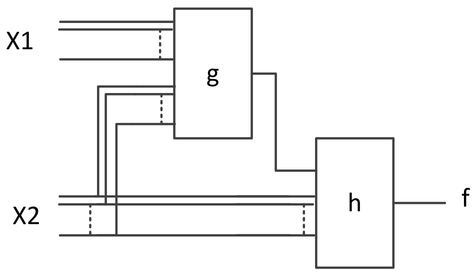


Fig. 3. Logic decomposition of LUT

Clearly, *cut* of order 1 is same as the *min set*. For this research, the *order* of a *cut* is restricted to 2 (i.e. $s = 2$).

With the above definitions, the following lemma can be stated. The proof is omitted for space limitations.

Lemma 1: The decomposition of a LUT is equivalent to a cut of order 2 of the corresponding min set.

Notations used in problem formulation

The following notations are defined.

f	n-input function implemented in a LUT
l	total number of minterms of f
$ms(f)$	$\langle t_1, t_2, \cdots, t_l \rangle$ = *min set* of f
c_1, c_2	cut sets of $ms(f)$ with a *cut* of order 2
φ_i	logic function represented by c_i
n_i	support of φ_i

Problem formulation

With the notations defined, it can be concluded that $f = \varphi_1 + \varphi_2$ and $n_1, n_2 \leq n$. Three LUTs are required to implement f (one LUT each to implement φ_1 and φ_2 respectively and one LUT to implement the OR-operation). However, with a simple modification, the same can be represented using two LUTs (as shown in Figure 3). Here, the first LUT implements φ_1 while the second implements φ_2 and the OR-functionality. Denoting φ_2' as the functionality of the second LUT, the following Equation holds trivially.

$$\begin{aligned} f_1 &= LUT(\varphi_1) \\ f &= LUT(\varphi_2') \\ &\text{where } \varphi_2' = f_1 + \varphi_2 \end{aligned} \tag{4}$$

Since the second LUT requires one additional input (output of the LUT implementing φ_1), the *support* of the second LUT is $n_2 + 1$ where n_2 is the *support* of φ_2.

The *min set* of *LUT*(φ_1) is the set c_1. The *min set* of *LUT*(φ_2') is calculated as follows. The total entries of the truth table of φ_2' is 2^{n_2+1}. Half of these entries have $f_1 = 1$ (since f_1 is an input to the function φ_2'). Further, for $f_1 = 1$, the function $\varphi_2'(= f_1 + \varphi_2)$ assumes logic-1. Thus the *min set* of φ_2' is $c_2' = \{(2^{n_2}+1), (2^{n_2}+2), \cdots, 2^{(n_2+1)}\} \cup c_2$

Assuming the LUT faults are independent and identically distributed, the joint fault masking of the two LUTs is calculated according to Equation 1 as shown below.

$$FM = \frac{max(|c_1|, 2^{n_1} - |c_1|)}{2^{n_1}} + \frac{max(|c_2'|, 2^{n_2+1} - |c_2'|)}{2^{n_2+1}} \tag{5}$$

Algorithm 3 LUT decomposition

```
Input: $LUTDB_n$, $T$
Output: $LUTDB_f$
1: for all $lut \in LUTDB_n$ do
2:     compute $FM_{lut}$
3:     if $FM_{lut} < T$ then
4:         $V_{assign}(i) = 1,\ \forall 1 \leq i \leq 2l$
5:         $fm_{best} = calculateFaultMasking(V_{assign})$
6:         $V_{best} = V_{assign}$
7:         while $numIter < maxIter$ do
8:             for $i = 1$ to $2l$ do
9:                 $[fm_1\ \varphi_1\ \varphi_2'] = calculateFaultMasking(V_{assign})$
10:                $V_{assign}(i) = [V_{assign}(i)\ (+)\ 1]$
11:                $[fm\ \varphi_1\ \varphi_2'] = calculateFaultMasking(V_{assign})$
12:                if $fm < fm_1$ then
13:                    $V_{assign}(i) = [V_{assign}(i)\ (-)\ 1]$
14:                    $fm = fm_1$
15:                end if
16:            end for
17:            $numIter++$
18:            if $fm > fm_{best}$ then
19:                $fm_{best} = fm;\ V_{best} = V_{assign}$
20:            end if
21:            /* randomly assign the minterms to a set
22:        end while
23:        $[fm\ \varphi_1\ \varphi_2'] = calculateFaultMasking(V_{best})$
24:        $[lut_1 lut_2] = convertToLUT(\varphi_1, \varphi_2')$
25:        $LUTDB_f.push(lut_1, lut_2)$
26:    else
27:        $LUTDB_f.push(lut)$
28:    end if
29: end for
```

The optimization problem is formulated as follows:

$$\begin{aligned} \text{maximize } & FM \\ \text{subject to } & n_1 \leq n \\ & n_2 < n \\ & ms(f) = c_1 \cup c_2 \end{aligned} \tag{6}$$

Solution approach

The optimization problem defined in Equation 6 is *quasi-convex*. A heuristic is proposed here to solve the same. A vector (V_{min}) is defined to hold the minterms of the function f. Each minterm in entered twice in the vector (V_{min}) to allow overlapping of the *min sets* c_1 and c_2. A second vector (V_{assign}) is defined of the same size as V_{min}. Each element, $V_{assign}(i)$ denotes the *min sets* (c_1 or c_2) to which the minterm $V_{min}(i)$ is assigned.

$$\begin{aligned} V_{min} &= \langle t_1, t_2, \cdots, t_l, t_1, t_2, \cdots, t_l \rangle \\ V_{assign} &= \langle u_1, u_2, \cdots, u_{2l} \rangle \\ &\text{where } u_i \in [1, 2] \end{aligned} \tag{7}$$

The pseudo-code for the proposed heuristic is shown in Algorithm 3. The algorithm takes $LUTDB$ (generated using

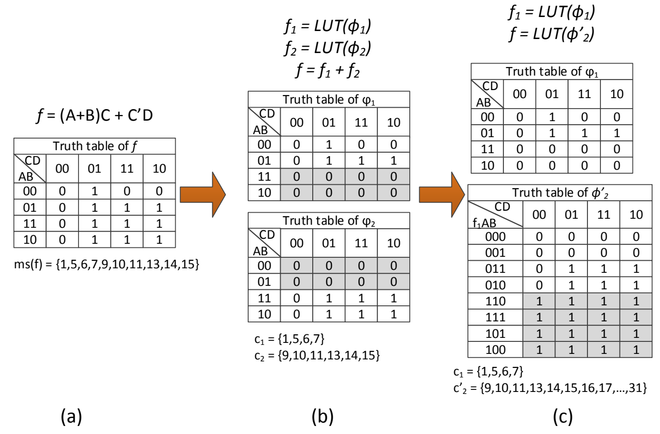
Algorithm 4 calculateFaultMasking(): calculate the fault masking

```
Input: Minterm vector $V_{min}$ and assignment vector $V_{assign}$
Output: Fault masking $FM$, logic functions $\varphi_1$, $\varphi'_2$
 1: $c_1 = \{V_{min}(i)|$ such that $V_{assign}(i) = 1, 1 \leq i \leq 2l\}$
 2: $c_2 = V_{min} \setminus c_1$;  Determine $n_2$
 3: $c'_2 = \{(2^{n_2}+1), (2^{n_2}+2), \cdots, 2^{(n_2+1)}\} \cup c_2$
 4: $tt_1 = formTruthTabl(c_1)$;  $tt_2 = formTruthTabl(c'_2)$
 5: $[n_1 \;\; \varphi_1] = QuineMcCluskey(tt_1)$
 6: $[n'_2 \;\; \varphi'_2] = QuineMcCluskey(tt_2)$
 7: if $n_1 \leq n$ and $n'_2 \leq n$ then
 8:     compute $FM$ according to Equation 5
 9: else
10:     $FM = 0$
11: end if
12: Return $[FM \;\; \varphi_1 \;\; \varphi'_2]$
```

Algorithm 1) and a user defined parameter (T) signifying the fault masking threshold. For every LUT of the $LUTDB$, the fault masking is computed using Equation 1 (line 2). If this is higher than the threshold (T), no decomposition is performed on the LUT. If the fault masking is less than the threshold, LUT decomposition is performed to maximize FM according to Equation 6 (lines 4-15). The first step towards this is the assignment of a set for all the minterms in V_{min} (line 4). For each of the minterms, the fault masking is computed using the *calculateFaultMasking()* routine (line 9). The set assignment is changed (line 10) and the value is recalculated (line 11). The assignment is retained if this value is greater than the previously calculated one, otherwise the move is discarded (lines 12-15). The $(+)$ and $(-)$ are modulo-2 addition and subtraction respectively. If the fault masking obtained is greater than the best value obtained so far, the best values are updated (line 19). To enable the algorithm search for the global maxima, minterms are randomly assigned to different sets and the steps are repeated. This is continued for $maxIter$ number of iterations, where $maxIter$ is a user defined parameter governing the algorithm execution time and solution quality.

An essential component of Algorithm 3 is the *calculate-FaultMasking()* routine, which is provided as pseudo-code in Algorithm 4. The algorithm takes the minterm vector V_{min} and the assignment vector V_{assign}. The minterms are partitioned into two sets c_1, c_2 according to the assignment. The corresponding truth tables are generated with minterms in c_1 and c_2 respectively. The next step is the minimization of each of the truth tables according to the *Quine McCluskey* algorithm (lines 4-5). if the number of inputs satisfy the constraints in Equation 6, the fault masking is calculated and returned, else 0 is returned.

An example is provided to better understanding of the proposed LUT decomposition algorithm. Figure 4(a) plots the truth table of the function $f = (A + B)C + C'D$. The corresponding min set (*ms*) is indicated. Figure 4(b) plots the one possible *cut* of *ms*. Here $ms = c_1 \cup c_2$ and $c_1 \cap c_2 = \emptyset$. Figure 4(c) represents the implementation of Figure 3 where the f_1 output of the first LUT (implementing the function φ_1)


$f = (A+B)C + C'D$

Truth table of f

CD \ AB	00	01	11	10
00	0	1	0	0
01	0	1	1	1
11	0	1	1	1
10	0	1	1	1

ms(f) = {1,5,6,7,9,10,11,13,14,15}

(a)

$f_1 = LUT(\phi_1)$
$f_2 = LUT(\phi_2)$
$f = f_1 + f_2$

Truth table of φ_1

CD \ AB	00	01	11	10
00	0	1	0	0
01	0	1	1	1
11	0	0	0	0
10	0	0	0	0

Truth table of φ_2

CD \ AB	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	1	1	1
10	0	1	1	1

$c_1 = \{1,5,6,7\}$
$c_2 = \{9,10,11,13,14,15\}$

(b)

$f_1 = LUT(\phi_1)$
$f = LUT(\phi'_2)$

Truth table of φ_1

CD \ AB	00	01	11	10
00	0	1	0	0
01	0	1	1	1
11	0	0	0	0
10	0	0	0	0

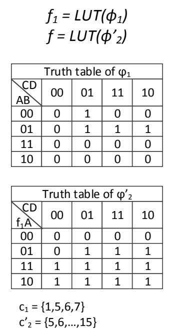
Truth table of ϕ'_2

CD \ f_1AB	00	01	11	10
000	0	0	0	0
001	0	0	0	0
011	0	1	1	1
010	0	1	1	1
110	1	1	1	1
111	1	1	1	1
101	1	1	1	1
100	1	1	1	1

$c_1 = \{1,5,6,7\}$
$c'_2 = \{9,10,11,13,14,15,16,17,...,31\}$

(c)

Fig. 4. Example of LUT decomposition


$f_1 = LUT(\phi_1)$
$f = LUT(\phi'_2)$

Truth table of φ_1

CD \ AB	00	01	11	10
00	0	1	0	0
01	0	1	1	1
11	0	0	0	0
10	0	0	0	0

Truth table of φ'_2

CD \ f_1A	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	1	1	1	1
10	1	1	1	1

$c_1 = \{1,5,6,7\}$
$c'_2 = \{5,6,...,15\}$

Fig. 5. LUT optimization using Quine McClusky algorithm

serves as one of the inputs of the second LUT. The second LUT of Figure 4(c) indicates this. Finally, Figure 5 plots the result after optimization of the second LUT of Figure 4(c) using *Quine McClusky* algorithm.

D. LUT re-synthesis

The LUT restructuring step of the flow involves implementing the AND and OR masking for each LUT of the implemented design. In [5], the authors proposed to merge the masking logic for a LUT in the LUT of its fanout. This can lead to a reduction of the number of usable inputs of the fanout LUT. To avoid this problem, this paper proposes to use the *carry chain* logic of the Virtex 6 FPGA. If o_5 and o_6 are the dual-outputs of a LUT, then the *carry chain* logic implemented is given by the equation

$$C_{out} = C_{in}.O_5 \;+\; C_{in}.O_6 \;+\; O_5.O_6 \qquad (8)$$

Clearly, setting $C_{in} = 1$, results in ORing of O_5 and O_6, while setting it to 0, results in ANDing.

The objective of the LUT re-synthesis step is to determine the value of c_{in} to maximize the logic masking effectiveness. In other words, for each LUT, if the number of zeros is more than the number of ones, c_{in} is set to 0 to mask $0 \rightarrow 1$ faults. Similarly, for LUTs with more number of ones, c_{in} is set to 1 to mask $1 \rightarrow 0$ faults.

TABLE I
SLICE AND LUT USAGE OF BENCHMARKS CONSIDERED

Suites	Benchmarks	Used slices	Used LUTs	% Free LUTs	Suites	Benchmarks	Used slices	Used LUTs	% Free LUTs
MCNC	alu4	178	512	28	Opencores	aes	184	573	22
	apex2	252	706	30		ethernet	1168	3179	32
	apex4	198	618	22		i2c	80	200	37.5
	bigkey	374	605	60		mem ctrl	503	1171	42
	clma	4	7	56		pci	755	1695	44
	des	366	564	61.5		spi	202	564	30.2
	diffeq	227	526	42		tv80	577	1724	25.3
	disp	555	683	69.2		usb phy	78	102	67.3
	elliptic	61	133	45.5		vga lcd	132	251	52.5
	exp5p	68	107	60.7		wb dma	386	779	49.5
	ex1010	205	612	25.3	ITC99	b5	61	155	36.5
	frisc	550	1905	13.4		b15	647	1877	27.4
	misex3	236	500	47.03		b20	588	2049	13
	pdc	138	276	50		b22	896	3165	11.7
	s298	9	23	36.1	UMass RCG	ava	1035	2611	37
	s38417	1235	2168	56.1		dct	8	15	53
	s38584	1259	1944	61.4	VPR	mkSMAdapter	415	1064	36
	seq	220	739	16		sha	400	1457	9
	spla	199	449	43.6		steriovision0	1990	3099	61
	tseng	208	539	35.2		or1200	855	2333	31.7

IV. RESULTS

The proposed algorithms are implemented in Matlab running on 2.1 GHz Intel Core i5 PC with 8GB memory running Windows. The benchmarks used for analysis and the slice usage of each benchmark are reported in Table I. All benchmarks are synthesized, placed and routed using Xilinx ISE 13.1 with area minimization as the optimization strategy. The target FPGA used for all experiments is Xilinx Virtex 6 where each configuration logic block (*CLB*) consists of two slices with each slice consisting of four 6-LUTs.

As can be seen from the Table I, on average 40% of LUTs in the *used* slices are unoccupied. This clearly motivates to exploit the unused resources for fault-tolerance.

A. Complexity analysis of proposed algorithms

There are three algorithms proposed in this work. However, Algorithm 1 is tool dependent and not much insight is available on the exact complexity. This section therefore estimates the complexity of Algorithms 2, 3 and 4.

Let N denote the number of LUTs used in a given design. The complexity of Algorithm 2 is computed as follows. For each LUT, the *0/1 sensitivity* is generated for all the support. Fault masking is then computed after inversion of the LUT bits corresponding to the sensitivity list. Assuming, n-LUT, the worst case complexity of Algorithm 2 is given by

$$O(C_2) = O(N * 2 * n) = O(N * n) \quad (9)$$

The complexity of Algorithm 3 is computed as follows. For each LUT with fault masking less than T, lines 7-25 are executed. The complexity of this section is dependent on the complexity of the $calculateFaultMasking()$ routine. Denoting this as $O(C_4)$, the worst-case complexity of Algorithm 3 is given by

$$O(C_3) = N * maxIter * 2l * O(C_4) \quad (10)$$

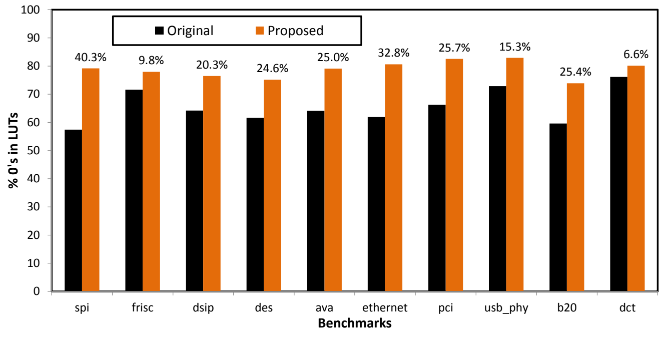


Fig. 6. Maximizing of logic 0 of LUTs

The complexity of Algorithm 4 is dependent on the complexity of *Quine-McClusky* algorithm. This is known to be NP-complete hard and a greedy heuristic is proposed to solve the same [16].

B. Maximization of logic 0 in LUTs

Figure 6 plots the average distribution of logic 0's in the LUTs of some of the benchmarks after applying the proposed technique (indicated by the bars titled *Proposed*). For comparison, the distribution of 0's in the LUTs after place and route (in the original flow) is indicated with the bars titled *Original*. Results in the figure can be interpreted as follows. The LUTs in the benchmark *spi* have on average 57% logic 0 (and 43% of logic 1) after place and route stage. Post logic restructuring and decomposition, the LUTs have on average 80% logic 0 i.e. 40% increase in the number of 0's per LUT for *spi*. Similarly, the results for other benchmarks can be interpreted. The numbers quoted on the bars titled *Proposed* indicates the percentage increase as compared to the original content. Although not explicitly shown here, on average for all 40 benchmarks considered, the proposed technique improves number of 0's by 20%.

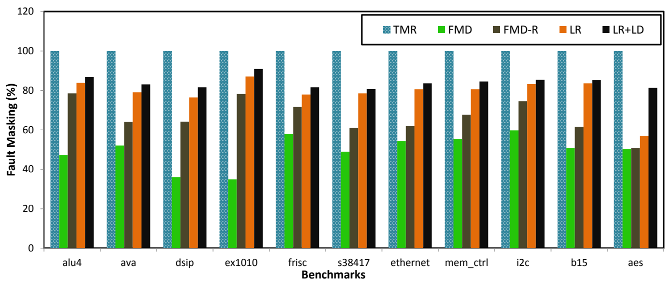


Fig. 7. Fault masking of different techniques

C. Fault-masking of LUT

Figure 7 plots the percentage fault masking of LUTs achieved using the proposed technique in comparison with the TMR based technique of [1] (referred to as *TMR* in the figure), the AND-OR masking based fault-tolerance technique of [5] (referred as *FMD*) and the restructuring-based variant of the same (referred as *FMD-R*). The technique proposed in this paper is referred as *LR+LD* (based on logic restructuring and decomposition). Additionally, results after logic restructuring *LR* is also reported in this figure. Since the techniques in [6] and [7] are based on fault-masking of entire circuit instead of individual LUT, they are not included for comparison here. These techniques are compared with the proposed technique in terms of circuit-wise fault-masking in Subsection IV-E.

As can be seen from the figure, *TMR*-based technique achieves the highest fault masking of all the techniques. This is due to the triplication of LUT contents. A point to note here is that, the fault-masking achieved by *TMR* is computed based on LUT contents only. The voting logic is not included in the computation. Although, TMR achieves 100% fault-masking, this is associated with high area and power penalties. The proposed *LR+LD* achieves highest fault masking of all the techniques. On average for all the benchmarks considered, *LR+LD* achieves fault-masking of 85% which is 60% and 22% better with respect to *FMD* and *FMD-R* respectively. The fault-masking achieved using *LR* is average 80% for all benchmarks. However, for some circuits such as *aes*, the fault masking of *LR* is not significantly high ($\approx 57\%$). Performing logic decomposition (*LD*[3]) on the same improves LUT fault-masking to 82%. From these results, it can be concluded that while *LR* achieves good fault-masking for most circuits, a combination of the two (*LR+LD*) guarantees to provide more than 80% fault-masking for all circuits.

Figure 8 plots the area overhead of the proposed fault-tolerant techniques in comparison with the existing techniques for the same set of benchmarks. The area overhead is measured in terms of slices used. The area of the base design (without incorporating fault-tolerant techniques) is normalized to 100 slices. As can be see from the figure, *FMD* achieves minimum area overhead. However, only 50% faults are masked as shown in Figure 7. The area overhead for *FMD-R* and *LR* are respectively 2% and 4%. The proposed *LR+LD* technique has an area overhead of 7% on average for all the benchmarks considered.

[3]The threshold for *LD* is set to 0.7.

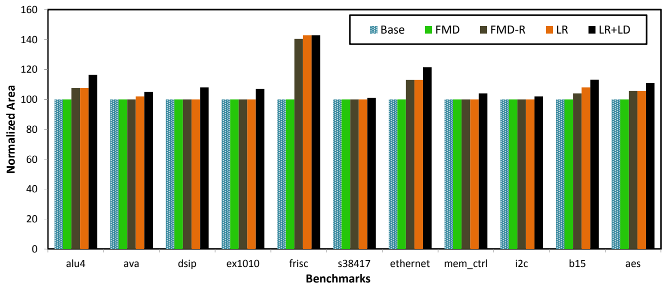


Fig. 8. Area utilization of different techniques

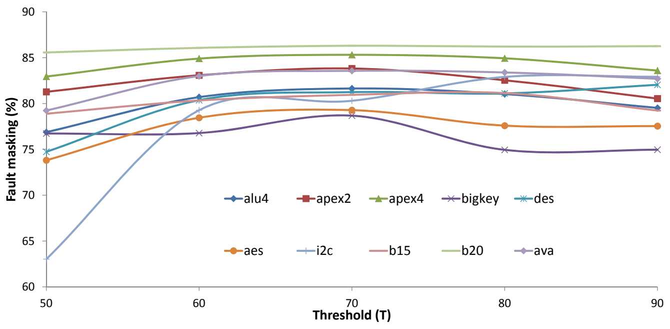


Fig. 9. Fault-masking for different threshold

D. Performance with varying fault-threshold

Figures 9 and 10 plots the fault masking and the area of the proposed *LD* technique for varying threshold (T). From Figure 9, we can see that the best fault masking for most benchmarks is achieved when the threshold is set at 70. Moreover, at this threshold, the area overhead is only 7% more on average as compared to a design with no fault masking. However, if optimum area and fault masking is required, it can be seen from Figure 11, that a threshold of 60 would give the best fault masking for the least area. Since the optimum threshold varies with each design, it is left to the user to tune the threshold according to the amount of fault masking needed and the area overhead tolerable.

E. Fault-masking of entire circuits

Table II reports the circuit-wise (full chip) fault-rate obtained by Monte Carlo simulations with 50K input vectors. Faults are injected randomly into the circuit. The fault-rate is measured by the number of observable faults. A fault is observable if the observed primary output of the circuit differs from the reference output. Otherwise, the fault is considered to be masked in the circuit. The fault-rate of proposed *LR+LD* is compared with the *FMD* technique and the in-place decomposition technique of [6] referred as *IPD*. Our technique can be used for fault masking of both combinatorial and sequential circuits since the faults are masked individually for each used LUT. However, *IPD* uses an end-to-end fault masking technique that currently only works for combinatorial circuits. Due to this, only combinatorial circuit benchmarks are included for comparison.

There are few trends to note from the table. Firstly, the fault rate for entire circuit are generally lower than those obtained per LUT (refer Figure 7). The circuit-wise fault-masking is

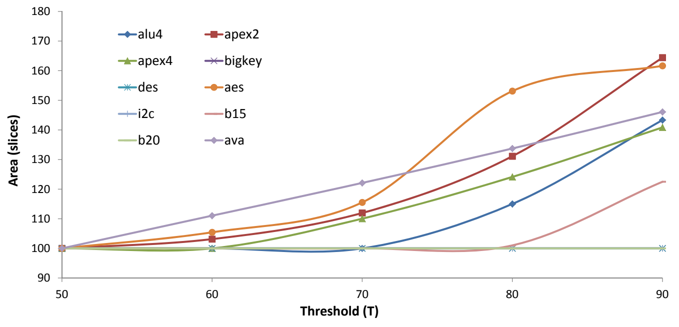


Fig. 10. Area for different threshold

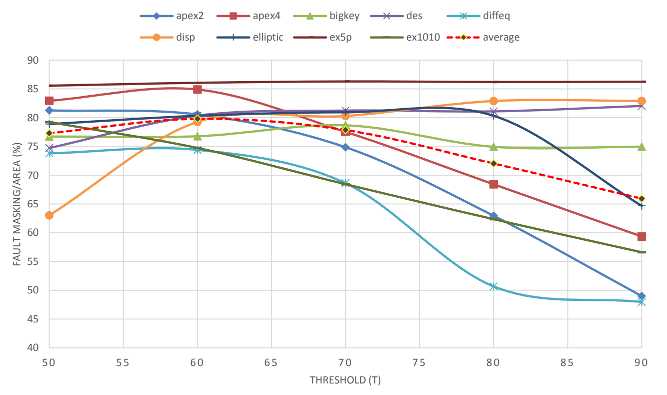


Fig. 11. Fault masking/area for different threshold

measured from primary inputs to primary outputs with some of the LUT bits getting masked in the subsequent LUT. Secondly, the proposed *LR+LD* reduces the fault-rate significantly achieving 68% and 60% lower fault-rate as compared to *FMD* and *IPD* respectively.

F. Algorithm runtime

Table III reports the execution time of the different algorithms proposed in this paper in comparison with the time taken by the *synthesis* and *place and route* steps of the conventional flow using Xilinx ISE 13.1.

V. Conclusions

This paper proposes a technique to maximize the fault-masking capabilities of a LUT using logic decomposition and restructuring. Experiments conducted with benchmarks from a wide range of benchmark suites on Xilinx Virtex 6 FPGA board demonstrate that 85% of the faults in a LUT can be masked with only 7% increase in slice usage. An open source tool release is planned to help researchers world-wide to benefit from our work and easily implement and test their techniques with various benchmarks and compare with state of the art techniques.

TABLE II
Fault-rate (%) of combinatorial benchmarks

Benchmark	*FMD*	*IPD*	*LR+LD*
alu4	0.33	0.27	0.23
apex2	0.26	0.21	0.17
apex4	1.10	0.99	0.13
des	1.41	1.27	0.65
ex1010	1.05	0.72	0.08
exp5p	0.62	0.52	0.07
misex3	0.49	0.38	0.29
pdc	0.83	0.63	0.2
seq	0.56	0.45	0.1
spla	1.05	0.82	0.12

TABLE III
Execution time (in secs) of algorithms

Benchmark	*Placement and Routing*	*Alg 1*	*Alg 2*	*Alg 3*	*Total*
wb_dma	624.0	14.5	58.0	386.4	1082.8
tseng	431.2	10.0	40.0	267.0	748.3
pci	1311.2	30.4	121.8	811.9	2275.3
ethernet	2584.0	60.0	242.5	1616.2	4502.7
elliptic	106.4	2.5	9.9	65.9	184.6
bigkey	484.0	11.2	45.0	299.7	839.9
apex4	494.4	11.5	45.9	306.1	857.9
apex2	564.8	13.1	52.5	349.7	980.1
alu4	409.6	9.5	38.0	253.6	710.8
aes	460.0	10.7	42.7	3284.8	3798.2
Mean	747.0	17.3	69.6	764.1	1598.1

References

[1] F. Kastensmidt, L. Sterpone, L. Carro, and M. Reorda, “On the optimal design of triple modular redundancy logic for SRAM-based FPGAs,” in *IEEE Conference on Design, Automation and Test in Europe (DATE)*, 2005.

[2] C. Carmichael, M. Caffrey, and A. Salazar, “Correcting single-event upsets through Virtex partial configuration,” *Xilinx Corporation*, 2000.

[3] C. Bolchini, A. Miele, and C. Sandionigi, “A Novel Design Methodology for Implementing Reliability-Aware Systems on SRAM-Based FPGAs,” *IEEE Transactions on Computers*, 2011.

[4] S. Srinivasan, A. Gayasen, N. Vijaykrishnan, M. Kandemir, Y. Xie, and M. Irwin, “Improving Soft-error Tolerance of FPGA Configuration Bits,” in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2004.

[5] J.-Y. Lee, Y. Hu, R. Majumdar, L. He, and M. Li, “Fault-tolerant resynthesis with dual-output LUTs,” in *IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010.

[6] J.-Y. Lee, Z. Feng, and L. He, “In-place decomposition for robustness in FPGA,” in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*.

[7] Z. Feng, Y. Hu, L. He, and R. Majumdar, “IPR: in-place reconfiguration for FPGA fault tolerance,” in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2009.

[8] K. Huang, Y. Hu, X. Li, G. Hua, H. Liu, and B. Liu, “Exploiting free lut entries to mitigate soft errors in sram-based fpgas,” in *IEEE Asian Test Symposium (ATS)*, 2011.

[9] J. Cong and K. Minkovich, “LUT-based FPGA technology mapping for reliability,” in *ACM Design Automation Conference (DAC)*, 2010.

[10] F. Lima, L. Carro, and R. Reis, “Designing fault tolerant systems into SRAM-based FPGAs,” in *ACM Design Automation Conference (DAC)*, 2003.

[11] C. Beckhoff, D. Koch, and J. Torresen, “The Xilinx Design Language (XDL): Tutorial and use cases,” in *International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2011.

[12] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, “RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs,” in *International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[13] A. Mishchenko, B. Steinbach, and M. Perkowski, “An algorithm for bi-decomposition of logic functions,” in *ACM Design Automation Conference (DAC)*, 2001.

[14] A. Mishchenko, X. Wang, and T. Kam, “A new-enhanced constructive decomposition and mapping algorithm,” in *ACM Design Automation Conference (DAC)*, 2003.

[15] T. Sasao and M. Matsuura, “A method to decompose multiple-output logic functions,” in *ACM Design Automation Conference (DAC)*, 2004.

[16] J. Safaei and H. Beigy, “Quine-mccluskey classification,” in *IEEE/ACS International Conference on Computer Systems and Applications*, 2007.