Autonomous Soft-error Tolerance of FPGA Configuration Bits

ANUP DAS, National University of Singapore
SHYAMSUNDAR VENKATARAMAN, National University of Singapore
AKASH KUMAR, National University of Singapore

Field Programmable Gate Arrays (FPGAs) are increasingly susceptible to radiation-induced single event upsets (SEUs). These upsets are predominant in space environment, however, with increasing use of Static RAM (SRAM) in modern FPGAs, these SEUs are gaining prominence even in terrestrial environment. SEUs can flip SRAM bits of FPGA, potentially altering the functionality of the implemented design. This has motivated FPGA designers to investigate on techniques to protect the FPGA configuration bits against such inadvertent bit flips (soft-error). Traditionally, triple modular redundancy (TMR) is used to protect the FPGA bit flips. Increasing design complexity and limited battery life motivate for alternative approaches for soft-error tolerance . In this paper, we propose a technique to improve autonomous fault-masking capabilities of a design by maximizing the number of zeros or ones in lookup tables (LUTs). The technique analyzes critical configuration bits and utilizes spare resources (XOR gates and carry chains) of FPGAs to selectively manipulate the logic implemented in LUT using two operations – LUT restructuring and LUT decomposition. We implemented the proposed approach for Xilinx Virtex-6 FPGAs and validated the same with a wide set of designs from MCNC, IWLS 2005 and ITC99 benchmark suites. Results demonstrate that the proposed logic restructuring maximizes logic 0 (or 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. Furthermore, the logic decomposition algorithm provides incremental fault-tolerance capabilities and achieves an additional 5% fault-masking with an average 7% increase in slice usage.

The complete methodology is implemented into a tool for Xilinx FPGA and is made available online for the benefit of the research community. The algorithms are lightweight and the whole design flow (including Xilinx Place and Route) was completed in 75 minutes for the largest benchmark in the set.

Categories and Subject Descriptors: B.7.0 [**Hardware**]: Programmable Logic Elements; B.7.0 [**Hardware**]: System Level Fault-Tolerance; B.7.0 [**Hardware**]: Transient Errors and Upsets; B.7.0 [**Hardware**]: Circuit Optimization

General Terms: FPGA, Fault-tolerance

Additional Key Words and Phrases: FPGAs; soft-errors; SRAM Configuration bits; LUTs

1. INTRODUCTION

Radiation-induced single event upsets (SEUs) in the configuration memory are the major cause of faults in static RAM (SRAM)-based field programmable gate arrays (FPGAs). These SEUs cause bit flips (also known as soft-errors), potentially altering an implemented logic and rendering the device useless, unless the affected bits are re-programmed. These errors are predominant in space environment, however, the increasing use of SRAM cells in modern FPGAs (constituting around 90% of all sequential logic elements on a device) is leading to a growing prominence of soft-errors even in terrestrial environment. This has attracted significant attention in industry as well as in academia to investigate on fault-tolerance of FPGA configuration bits.

DOI: http://dx.doi.org/10.1145/0000000.0000000

Traditionally, soft-errors are tolerated using triple modular redundancy (TMR) [Kastensmidt et al. 2005]. However, growing design complexity and limited battery life are restricting the use of TMR to only safety-critical design components. Towards this end, periodic reprogramming of FPGA (known as scrubbing) has also been studied. Although this technique has negligible area overhead, it is often associated with high reconfiguration delay limiting the speed of operation. Furthermore, errors due to SEUs cannot be prevented, only the effect can be mitigated using scrubbing. Some of the low overhead solutions to the aforementioned problem include fault masking [Srinivasan et al. 2004; Lee et al. 2010b; Lee et al. 2010a; Feng et al. 2009; Huang et al. 2011; Cong and Minkovich 2010] and information redundancy [Lima et al. 2003].

1.1. Scope of the work

Given an FPGA device and a design to be mapped on the same, we propose a technique to analyze the critical configuration bits and maximize the number of zeros (or ones) of lookup tables (LUTs) by logic restructuring, to maximize the autonomous fault-masking capability of the design implemented on the FPGA. Furthermore, a technique is proposed to provide incremental fault-tolerance by decomposing a given LUT into component LUTs.

1.2. Contributions

Following are our key contributions[1].

- Maximizing zeros or ones of LUT configuration bits through LUT restructuring.
- Controlled decomposition of LUTs for higher granularity of fault-tolerance.
- Considering critical configuration bits to minimize area overhead.
- A generic technique for combinatorial and sequential circuits.
- A design tool for Xilinx FPGA incorporating the decomposition and restructuring of LUT. The tool is made online for the benefit of the research community.

The proposed techniques of LUT restructuring and decomposition are incorporated in the tradition FPGA design flow and validated extensively with a diverse set of benchmarks from MCNC, IWLS and ITC99 benchmark suite on Virtex-6 FPGA board from Xilinx. Results 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.

A tool has been generated for Xilinx-based FPGA and is tested on Virtex-6 FPGA boards. The complete fault-tolerant bitstream generation takes 75 minutes (including the conventional synthesis and, place and route step) for the largest benchmark in the set of benchmarks considered. In addition to this tool, an easy-to-use GUI tool is currently under development for Windows and Linux operating system.

1.3. Paper organization

The remainder of this paper is organized as follows. The related works are discussed in Section 2 followed by a brief overview of the mainstream FPGA architecture and fault masking of LUT in Section 3. The design flow is introduced in Section 4 with description of the two key components – LUT decomposition and LUT restructuring.

[1]A subset of these contributions appeared in a recent work from the same authors [Das et al. 2013]

Experimental results are discussed next in Section 6. Finally, the paper is concluded in Section 7 along with scope for future enhancements.

2. RELATED WORKS

Fault masking of FPGAs have attracted significant attention in recent years as a light-weight fault-tolerant approach for LUT configurations bit. A technique is proposed in [Lee et al. 2010b] to AND or OR the dual outputs of modern FPGAs 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, however, not modified to maximize the number of zeros or ones in the LUTs. This technique achieves high fault-masking with small area overhead. As shown in Section 6, the technique proposed in this paper modifies the logic implemented in an LUT to maximize the number of zeros or ones in LUT to improve fault-masking by 20% using the spare FPGA resources.

Another technique is proposed in [Feng et al. 2009] to maximize the identical configuration bits for complementary inputs of an LUT, 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 [Lee et al. 2010a] where faults in SRAM bits are masked by decomposing an 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.

3. FPGA ARCHITECTURE AND AUTONOMOUS FAULT MASKING OF LUT

Xilinx Virtex-6 FPGA devices consist of 6-input LUTs. Each 6-input LUT is internally composed of two 5-input LUTs as shown in Figure 3(a). This is the architecture for other mainstream FPGAs from vendors such as Altera. The two outputs (designated as o_5 and o_6) of a 6-input LUT can be used individually to implement two different 5-input functions in the two component LUTs. The 6-input LUT can also implement one 6-input function, in which case the o_5 output is unused. 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. An 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 an 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 bit-flip to 0 (BF0) and bit-flip to 1 (BF1) and therefore total number of BF0 faults and BF1 faults are same and equal to 2^r). The BF0 (and respectively BF1) faults for logic-0 (and logic-1) entries are benign. The total number of faults which can impact the circuit behavior is therefore 2^r. If the two outputs of the component LUTs are ANDed (respectively ORed), all BF1 faults of logic-0 entries (respectively BF0 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

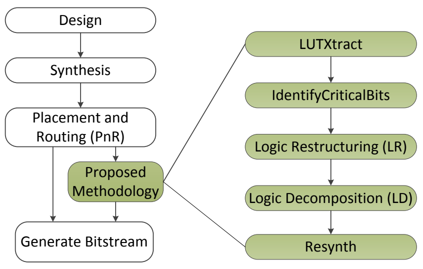


Fig. 1: Autonomous fault-tolerance design flow

possible for the LUT is given by

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

4. DESIGN FLOW

Figure 1 illustrates the proposed methodology as a part of the FPGA-based design implementation flow. The conventional flow adopted by most FPGA vendors 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 during the LUT mapping phase of the *Placement and Routing* step. The LUT extraction is performed in the *LUTXtract* block of the proposed design flow. This is then followed by the identification of the critical (essential) bits. This is performed using Xilinx *Prioritized Essential Bits* and is indicated in the methodology as *IdentifyCriticalBits*. Following this step, are the two operations – logic restructuring (*LR*) and logic decomposition (*LD*). The effectiveness of the two operations are evaluated in Section 6. 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 introduced in the design flow are introduced next.

4.1. LUT extraction

LUT extraction is widely studied in literature [Ziener et al. 2006]. In this work we use *RapidSmith* tool [Lavin et al. 2011] to perform 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 AN LUT) *The support of an 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 AN LUT) *The composition of an LUT is a tuple consisting of the indexed content of an 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

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]
```

Algorithm 2 Criticality aware LUT modification

```
Input: $LUTDB, PEB$
Output: $LUTDB_m$
1: for all $lut \in LUTDB$ do
2:     $lut_t = lut$ and $CritList = PEB(lut_t)$ // Get the essential bits for the corresponding LUT
3:     $[n_0\ \ n_1] = GetZerosOnes(CritList)$ // Calculate number of zeros and ones for the $CritList$
4:     for all $a_i \in composition(lut_t) \setminus CritList$ do
5:         if $n_0 > n_1$ then $a_i = 0$ else $a_i = 1$
6:     end for
7:     $LUTDB_m.push(lut_t)$
8: end for
```

content of the LUT[2] at location *bin2dec(A)*, where *bin2dec* routine converts 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*) [Beckhoff et al. 2011]. 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* [Lavin et al. 2011] tool to generate the *support* and *composition*. These are then stored in the $LUTDB$ database for use in the subsequent steps.

4.2. Identify Critical Bits

The next step in the design flow is the identification of the critical bits of an LUT. Several definition exist in literature for the identification of critical bits. In [Ferron et al. 2009], the authors classified the LUT bits as *critical*, *transparent* and *suspect* bits. In this context, critical bits are defined as those which have the highest impact of SEUs. In [Feng et al. 2009], the *critical* bits are defined as the bits which have the highest fault-masking effectiveness. Finally, in the Xilinx Virtex-6 FPGAs, *prioritized essential* bits [Patterson et al. 2008] are defined as bits causing functional failure, if change state. According to [Patterson et al. 2008], only a subset of the LUT configuration bits belong to this category. In our earlier work [Das et al. 2013], restructuring is performed on an LUT to maximize the number of zeros or ones based on the logic masking effectiveness within the LUT. In this work we incorporate the restructuring of LUT considering the Xilinx *prioritized essential* bits (PEB). For this, an LUT modification algorithm is proposed shown as pseudo-code in Algorithm 2. For each LUT in $LUTDB$, the algorithm first identifies the essential bits from the *prioritized essential* bit file generated using Xilinx ISE. The number of ones and zeros amongst the essential bits are counted individually. If there are more ones than zeros, each non-essential bit is converted to logic one. Similarly, if the number of zeros is more than the number of ones, each non-essential bit is converted into logic zero. It is to be noted that changing the non-essential bit do not affect the logic functionality [Patterson et al. 2008]. However, this simplifies the logic restructuring and decomposition step. An example is provided to demonstrate this. Let the composition of a 3-LUT implementing a two input AND logic be $[1\ 0\ 0\ 0\ 1\ 1\ 1\ 0]$. The lower 4 bits of the LUT are non-essential.

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

Clearly, there are four zeros and four ones, implying that the fault-masking capability of this LUT without any modification is 50%. Due to this low default fault-masking capability of the LUT, the the amount of extra logic required (in terms of XOR gates or extran LUTs) to increase its fault-masking to a satisfactory level (say, 75%) using logic restructuring and decomposition is high. To overcome this limitation and to reduce the area overhead, the composition of the LUT is modified according to the composition of the essential bits. The four essential bits of the LUT are $[1\ 0\ 0\ 0]$ and therefore there are three zeros and one ones (more zeros than ones). Hence, the LUT is modified as $[1\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$. This LUT has a default-fault masking of 87.5% and therefore, depending on the fault-masking threshold, the next steps of the flow (i.e., logic restructuring and decomposition) may be skipped. This reduces the overhead of our approach.

4.3. Restructuring of LUT

The restructuring of an 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 an LUT is defined as the set of indices 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 $\{0, 2, 4, 6\}$ and 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 n-LUT is 2^{n-1}.

Similarly, the *1-sensitivity* of an 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 an 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 contents 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 an *XNOR* gate, the original function can be easily recovered[3]. Specifically, if f be the original output of an 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 $f = \overline{f' \oplus i}$.

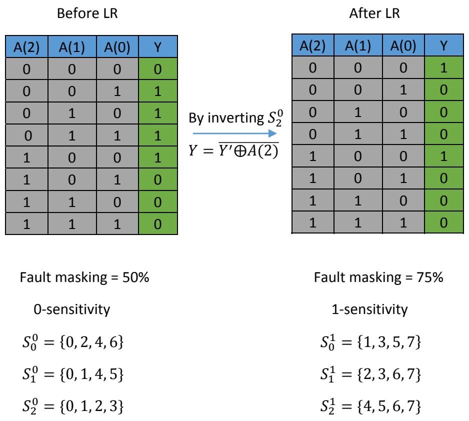
Algorithm 3 provides the pseudo-code for the logic restructuring technique. For each support of the LUT, the *0/1 sensitivity* are determined and the fault-masking is calculated. As output, a support is determined along with its sensitivity type.

Figure 2 illustrates an example of the logic restructuring. The two tables shown represent the *support* and LUT contents before and after the logic restructuring. When the LR algorithm is run on the example, it determines a suitable *support* and *sensitivity*, which when restructured, would provide the highest fault-masking. For this example, restructuring *support* S_2^0 increases the fault-masking from 50% to 75%. The original function can then be obtained back by a simple boolean transformation as discussed earlier in this section.

[3]The Xilinx FPGA slices contain spare XOR gates which is used for this purpose.

Algorithm 3 LUT restructuring

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



A(2)	A(1)	A(0)	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

A(2)	A(1)	A(0)	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Fig. 2: Logic restructuring of LUT

4.4. Decomposition of LUT

Synthesis of optimal boolean logic is a well studied research topic for FPGA technology mapping [Mishchenko et al. 2001; Mishchenko et al. 2003; Sasao and Matsuura 2004]. 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.

DEFINITION 4. (DECOMPOSABILITY OF LUT) *Let $f(X)$ be a function implemented in an 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 AN LUT) *The* min set *of an LUT is the set of indices for which the LUT contents are logic* 1.

The *min set* is given by $ms = \{i | a_i = 1, \forall 1 \leq i \leq m\}$, 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, $\forall 1 \leq i \leq s$) sharing the minterms.*

Mathematically, this can be expressed as $ms = \cup_{i=1}^{s} c_i$. The *cut* can be overlapping (common elements in *cut sets*) or non-overlapping.

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.

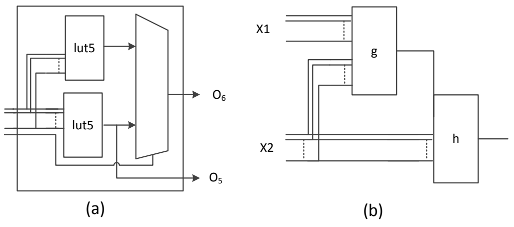


Fig. 3: Logic decomposition of LUT

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.

LEMMA 1. *The decomposition of an LUT is equivalent to a order 2 cut of its min set.*

The following notations are defined.

f	n-input function implemented in an 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

Using the notations defined, $f = \varphi_1 + \varphi_2$ and $n_1, n_2 \leq n$. Thus, 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 (b)). 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.

$$f_1 = LUT(\varphi_1) \text{ and } f = LUT(\varphi_2') \quad \text{where } \varphi_2' = f_1 + \varphi_2$$

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.

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

The optimization problem is formulated as follows:

$$\textbf{maximize } FM \textbf{ subject to } n_1 \leq n;\ n_2 < n;\ ms(f) = c_1 \cup c_2 \tag{3}$$

Solution approach

A heuristic is proposed here to solve the above optimization problem. The vector V_{min} holds the minterms of the function f with two copies of each minterm to allow overlapping of the *min sets* c_1 and c_2. A second vector (V_{assign}) is defined to indicate the *min set* (c_1 or c_2) corresponding to the minterms in V_{min}.

$$V_{min} = \langle t_1, t_2, \cdots, t_l, t_1, t_2, \cdots, t_l \rangle; V_{assign} = \langle u_1, u_2, \cdots, u_{2l} \rangle \quad \text{where } u_i \in [1, 2] \tag{4}$$

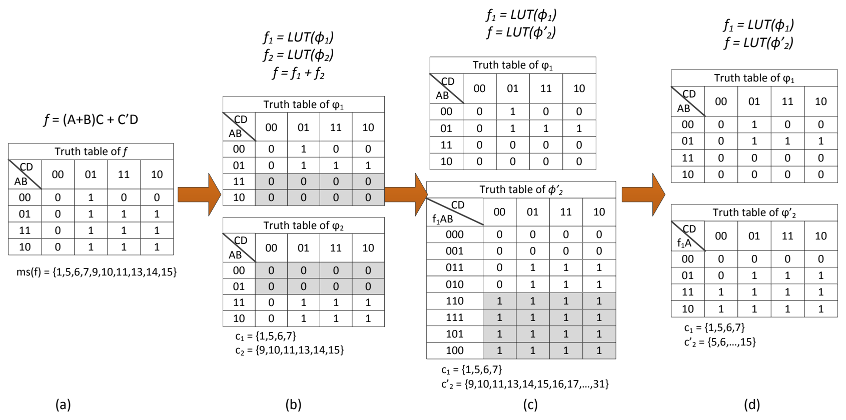
Algorithm 4 LUT decomposition

```
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:     $V_{assign}(i) = 1, \forall 1 \le i \le 2l$; $fm_{best} = calculateFaultMasking(V_{assign})$; $V_{best} = V_{assign}$
 5:     while $numIter < maxIter$ do
 6:       for $i = 1$ to $2l$ do
 7:         $[fm_1\ \varphi_1\ \varphi_2'] = calculateFaultMasking(V_{assign})$; $V_{assign}(i) = [V_{assign}(i)\ (+)\ 1]$
 8:         $[fm\ \varphi_1\ \varphi_2'] = calculateFaultMasking(V_{assign})$
 9:         if $fm < fm_1$ then $V_{assign}(i) = [V_{assign}(i)\ (-)\ 1]$; $fm = fm_1$
10:       end for
11:       $numIter + +$
12:       if $fm > fm_{best}$ then $fm_{best} = fm$; $V_{best} = V_{assign}$
13:     end while
14:     $[fm\ \varphi_1\ \varphi_2'] = calculateFaultMasking(V_{best})$; $[lut_1 lut_2] = convertToLUT(\varphi_1, \varphi_2')$
15:     $LUTDB_f.push(lut_1, lut_2)$
16:   else
17:     $LUTDB_f.push(lut)$
18:   end if
19: end for
```

Algorithm 4 reports the pseudo-code for the proposed heuristic. The algorithm inputs $LUTDB$ (generated using Algorithm 1) and a user defined parameter (T) indicating 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 3 (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 7). The set assignment is changed and the value is recalculated (line 8). The assignment is retained if this value is greater than the previously calculated one, otherwise the move is discarded (line 9). The $(+)$ and $(-)$ are modulo-2 addition and subtraction respectively. If the fault masking obtained is greater than the best value obtained thus far, the best values are updated (line 12). To enable the algorithm search for the global maximum, 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 termination of the algorithm and solution quality.

An essential component of Algorithm 4 is the *calculateFaultMasking()* routine, which is provided as pseudo-code in Algorithm 5. 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 truth table according to the *Quine McCluskey* algorithm (lines 4-5). If the number of inputs satisfy the constraints in Equation 3, the fault masking is calculated and returned.

An example is provided to better understand 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) serves as one of the inputs of the second LUT. The second LUT of Figure 4(c) indicates this. Finally, Figure 4(d) plots the result after optimization of the second LUT of Figure 4(c) using *Quine McClusky* algorithm.



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

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}

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

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

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

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

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

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

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

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

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

Fig. 4: Example of LUT decomposition

Algorithm 5 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 $V_{assign}(i) = 1, 1 \leq i \leq 2l\}$ and $c_2 = V_{min} \setminus c_1$; Determine $n_2$
2: $c'_2 = \{(2^{n_2}+1), (2^{n_2}+2), \cdots, 2^{(n_2+1)}\} \cup c_2$
3: $tt_1 = formTruthTabl(c_1)$; $tt_2 = formTruthTabl(c'_2)$
4: $[n_1 \ \ \varphi_1] = QuineMcCluskey(tt_1)$ and $[n'_2 \ \ \varphi'_2] = QuineMcCluskey(tt_2)$
5: if $n_1 \leq n$ and $n'_2 \leq n$ then compute $FM$ according to Equation 2 else $FM = 0$
6: Return $[FM \ \ \varphi_1 \ \ \varphi'_2]$
```

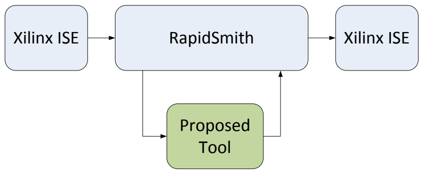


Fig. 5: Proposed tool as part of the design flow

4.5. 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 [Lee et al. 2010b], the authors proposed to merge the masking logic for an 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 an LUT, then the *carry chain* logic implemented is given by the equation

$$C_{out} = C_{in}.O_5 \ + \ C_{in}.O_6 \ + \ O_5.O_6 \tag{5}$$

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

5. TOOL IMPLEMENTATION

We have implemented the proposed methodology as a tool. Figure 5 envisages the proposed tool as a part of the FPGA design flow. The other tools required are also highlighted in the same figure. The tool flow is implemented in C++ and Matlab. This tool is made available online for the benefit of the research community [LDL 2013]. The complete fault-tolerant bitstream generation takes 75 minutes (including the conventional synthesis and place and route step) for the largest benchmark in the set of benchmarks considered. In addition to this tool, an easy-to-use GUI tools is currently under development for Windows and Linux operating system.

6. 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.

6.1. 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. Let N denote the number of LUTs used in a given design. The complexity of Algorithm 3 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 3 is given by

$$O(C_3) = O(N * 2 * n) = O(N * n) \quad (6)$$

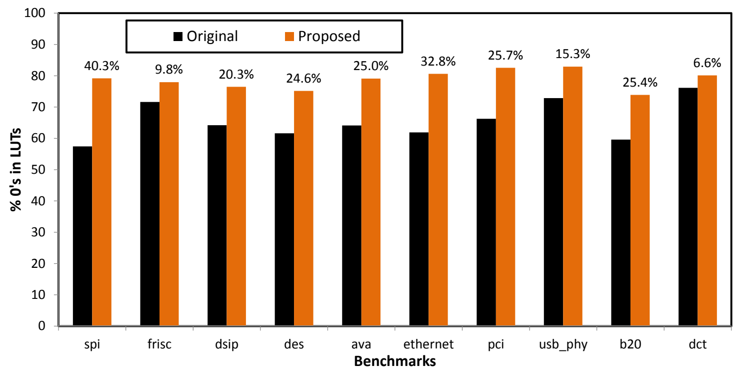


Fig. 6: Maximizing of logic 0 of LUTs

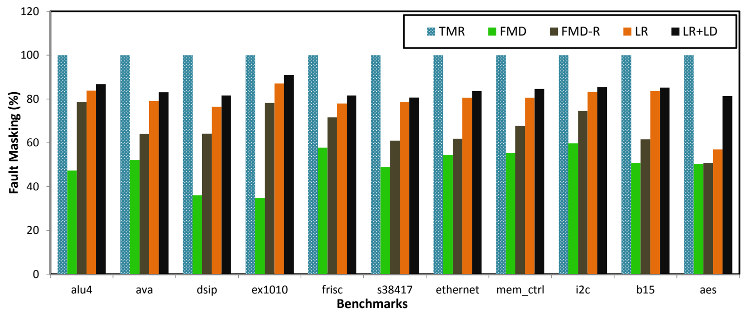


Fig. 7: Fault masking of different techniques

The complexity of Algorithm 4 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 $O(C_5)$, the worst-case complexity of Algorithm 4 is given by $O(C_4) = N * maxIter * 2l * O(C_5)$. The complexity of Algorithm 5 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 [Safaei and Beigy 2007].

6.2. 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%.

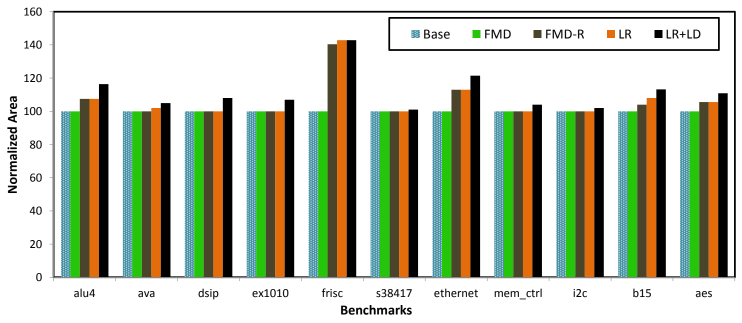


Fig. 8: Area utilization of different techniques

6.3. 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 [Kastensmidt et al. 2005] (referred to as *TMR* in the figure), the AND-OR masking-based fault-tolerance technique of [Lee et al. 2010b] (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* are also reported in this figure. Since the techniques in [Lee et al. 2010a] and [Feng et al. 2009] are based on fault-masking of entire circuit instead of individual LUTs, they are not included for comparison here. These techniques are compared with the proposed technique in terms of circuit-wise fault-masking in Subsection 6.6.

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* with a threshold set to 0.7) 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.

6.4. Bit Criticality aware LUT modification

Table II reports the percentage use of the XOR gates needed to recover back the logic after an LUT restructuring for the initial methodology [Das et al. 2013] and the one

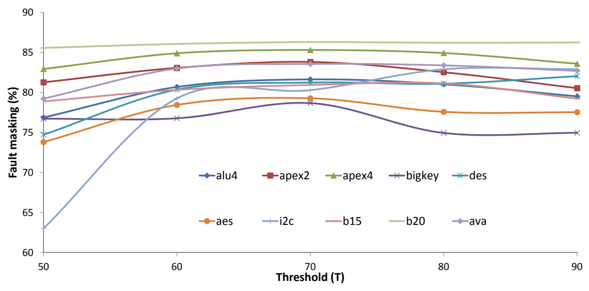


Fig. 9: Fault-masking for different threshold

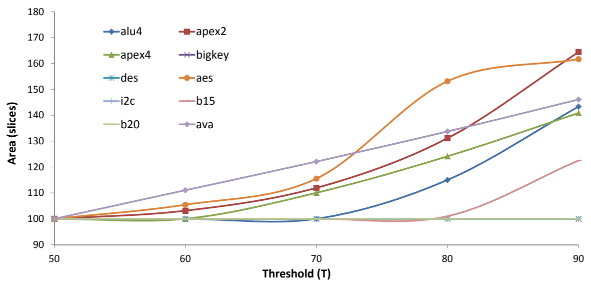


Fig. 10: Area for different threshold

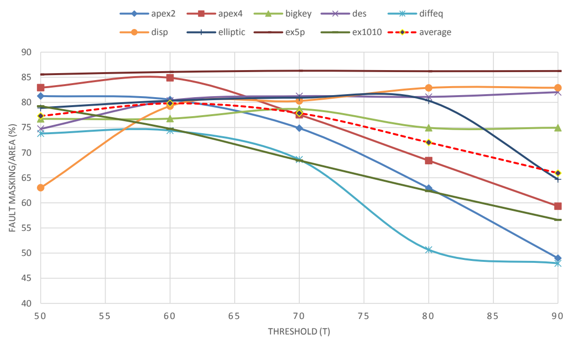


Fig. 11: Fault masking per unit area for different threshold

proposed in this paper. Ten benchmarks are considered for comparison. As can be seen from the table, for smaller design (column 1), the reduction of XOR gates is negligible (maximum 0.2%). However, for larger designs (column 4), considering the essential bits leads to a reduction of upto 1% in the number of XOR gates. As a future modification, the criticality determination using the technique of [Ferron et al. 2009; Feng et al. 2009] can be considered.

6.5. 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

Table II: Usage (%) of XOR Gates

Benchmark	*[Das et al. 2013]*	*Proposed*	Benchmark	*[Das et al. 2013]*	*Proposed*
alu4	82.2	82.1	bigkey	66.2	65.7
apex4	88.7	88.6	disp	55.5	55.4
elliptic	92.5	92.5	s38417	81.0	80.6
exp5p	79.5	79.3	s38584	60.0	59.1
pdc	74.1	74.1	frisc	75.8	74.9

Table III: Fault-rate (%) of combinatorial benchmarks

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

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 fault masking desired with affordable area overhead.

6.6. Fault-masking of entire circuits

Table III 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 [Lee et al. 2010a] 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 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.

6.7. Algorithm overhead

Table IV 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. A point to be noted here is that, by introducing the extra XOR gate and extran LUT, the flop to flop delay is increased. This has an impact on the maximum frequency supported for the design. In this version of the work, the extran LUT or XOR gate in proximity of the original LUT were used to minimize the routing delay. However, it is anticipated that in large design it might be

Table IV: Execution time (in secs) of algorithms

Benchmark	*PnR*	*Alg 1*	*Alg 3*	*Alg 4*	*Total*	Benchmark	*PnR*	*Alg 1*	*Alg 3*	*Alg 4*	*Total*
wb_dma	624.0	14.5	58.0	386.4	1082.8	bigkey	484.0	11.2	45.0	299.7	839.9
tseng	431.2	10.0	40.0	267.0	748.3	apex4	494.4	11.5	45.9	306.1	857.9
pci	1311.2	30.4	121.8	811.9	2275.3	apex2	564.8	13.1	52.5	349.7	980.1
ethernet	2584.0	60.0	242.5	1616.2	4502.7	alu4	409.6	9.5	38.0	253.6	710.8
elliptic	106.4	2.5	9.9	65.9	184.6	aes	460.0	10.7	42.7	3284.8	3798.2

possible to have no free LUTs or XOR gates in the vicinity of the LUT to be modified. In this case, a trade-off analysis can be done between increased fault-masking and increased circuit delay. This is reserved for future work.

7. CONCLUSIONS

In this paper we proposed a technique to maximize the fault-masking capabilities of an LUT using logic decomposition and bit criticality aware restructuring. The proposed methodology is validated experimentally with benchmarks from a wide range of benchmark suites on Xilinx Virtex-6 FPGA. Results demonstrate that 85% of the faults in an LUT can be masked with only 7% increase in slice usage. The complete methodology is implemented into a tool for Xilinx FPGA and is made available online for the benefit of the research community.

REFERENCES

2013. LUT-RD: LUT Restructuring and Decomposition, http://wiki.nus.edu.sg/display/mpsoc/Documents.

BECKHOFF, C., KOCH, D., AND TORRESEN, J. 2011. The Xilinx Design Language (XDL): Tutorial and use cases. In *International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*.

CONG, J. AND MINKOVICH, K. 2010. LUT-based FPGA technology mapping for reliability. In *ACM Design Automation Conference (DAC)*.

DAS, A., VENKATARAMAN, S., AND KUMAR, A. 2013. Improving autonomous soft-error tolerance of fpga through lut configuration bit manipulation. In *International Conference on Field Programmable Logic and Applications (FPL)*. 1–8.

FENG, Z., HU, Y., HE, L., AND MAJUMDAR, R. 2009. IPR: in-place reconfiguration for FPGA fault tolerance. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*.

FERRON, J. B., ANGHEL, L., LEVEUGLE, R., BOCQUILLON, A., MILLER, F., AND MANTELET, G. 2009. A methodology and tool for predictive analysis of configuration bit criticality in sram-based fpgas: Experimental results. In *International Conference on Signals, Circuits and Systems (SCS)*. 1–6.

HUANG, K., HU, Y., LI, X., HUA, G., LIU, H., AND LIU, B. 2011. Exploiting free lut entries to mitigate soft errors in sram-based fpgas. In *IEEE Asian Test Symposium (ATS)*.

KASTENSMIDT, F., STERPONE, L., CARRO, L., AND REORDA, M. 2005. On the optimal design of triple modular redundancy logic for SRAM-based FPGAs. In *IEEE Conference on Design, Automation and Test in Europe (DATE)*.

LAVIN, C., PADILLA, M., LAMPRECHT, J., LUNDRIGAN, P., NELSON, B., AND HUTCHINGS, B. 2011. RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*.

LEE, J.-Y., FENG, Z., AND HE, L. 2010a. In-place decomposition for robustness in FPGA. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*.

LEE, J.-Y., HU, Y., MAJUMDAR, R., HE, L., AND LI, M. 2010b. Fault-tolerant resynthesis with dual-output LUTs. In *IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*.

LIMA, F., CARRO, L., AND REIS, R. 2003. Designing fault tolerant systems into SRAM-based FPGAs. In *ACM Design Automation Conference (DAC)*.

MISHCHENKO, A., STEINBACH, B., AND PERKOWSKI, M. 2001. An algorithm for bi-decomposition of logic functions. In *ACM Design Automation Conference (DAC)*.

MISHCHENKO, A., WANG, X., AND KAM, T. 2003. A new-enhanced constructive decomposition and mapping algorithm. In *ACM Design Automation Conference (DAC)*.

PATTERSON, C. D., SUNDARARAJAN, P., BLODGET, B. J., AND MCMILLAN, S. P. 2008. Method and system for identifying essential configuration bits. US Patent 7,406,673.

SAFAEI, J. AND BEIGY, H. 2007. Quine-mccluskey classification. In *IEEE/ACS International Conference on Computer Systems and Applications*.

SASAO, T. AND MATSUURA, M. 2004. A method to decompose multiple-output logic functions. In *ACM Design Automation Conference (DAC)*.

SRINIVASAN, S., GAYASEN, A., VIJAYKRISHNAN, N., KANDEMIR, M., XIE, Y., AND IRWIN, M. 2004. Improving Soft-error Tolerance of FPGA Configuration Bits. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*.

ZIENER, D., ASSMUS, S., AND TEICH, J. 2006. Identifying FPGA IP-Cores Based on Lookup Table Content Analysis. In *International Conference on Field Programmable Logic and Applications (FPL)*. 1–6.