# Fine-grained Module-based Error Recovery in FPGA-based TMR Systems

Zhuoran Zhao*, Dimitris Agiakatsikas*, Nguyen T. H. Nguyen*, Ediz Cetin†, and Oliver Diessel*
*School of Computer Science and Engineering, UNSW Australia
†School of Electrical Engineering and Telecommunications, UNSW Australia

***Abstract*—Space processing applications deployed on SRAM-based Field Programmable Gate Arrays (FPGAs) are vulnerable to radiation-induced Single Event Upsets (SEUs). Compared with the well-known SEU mitigation solution — Triple Modular Redundancy (TMR) with configuration memory scrubbing — TMR with module-based error recovery (MER) is notably more energy efficient and responsive in repairing soft-errors in the system. Unfortunately, TMR-MER systems also need to resort to scrubbing when errors occur in sub-components, such as nets, which are not recovered by MER. This paper addresses this problem by proposing a fine-grained module-based error recovery technique that without additional system hardware can localize and correct errors that classic MER fails to do. We evaluate our proposal via a fault-injection campaign on a Xilinx Artix-7 application circuit and compare the reliability, the error correction latency and the energy cost of repairing errors, of our proposal with those of a conventional MER approach and with periodic and on-demand blind scrubbing. We find the reliability of our proposal to be the highest and the energy expenditure to be the lowest amongst those methods considered.**

## I. Introduction

Reliable space-based digital systems implemented using Commercial Off-The-Shelf (COTS) SRAM-based FPGAs and programmable System-on-Chips (SoCs) commonly rely on Triple Modular Redundancy (TMR) to mask the effects of radiation-induced Single Event Upsets (SEUs) in the application circuits. The considerable amount of configuration memory in these devices is also susceptible to radiation-induced corruption. Two approaches have emerged to deal with this problem. Configuration memory scrubbing (TMR-S) periodically scans the entire device and corrects configuration memory errors by rewriting the memory frames containing them. Module-based error recovery (TMR-MER), on the other hand, reconfigures the frames of a TMR module when an error in its configuration memory is detected. While scrubbing occurs periodically, whether or not errors are present, TMR-MER relies upon the repeated detection of an error by the same TMR voter to trigger a reconfiguration of the module presenting the error [1]. Both methods utilize a controller to operate. However, MER also requires a Reconfiguration Control Network (RCN) to relay error requests from the voters in the system to the Reconfiguration Controller (RC) [2].

In [2], it was found that FPGA SoCs that rely on TMR-MER have lower reliability than those relying on TMR-S unless the RCN is also triplicated and corrected when configuration memory errors become evident in it. Due to the distributed nature of RCN resources, Agiakatsikas *et al.* [2] resorted to scrubbing the whole device when an RCN was affected by configuration memory errors. Although error recovery using scrubbing is slow and energy is wasted checking/reconfiguring frames that are not in error, it was found that scrub operations were only occasionally needed since the triplicated RCN had a relatively low susceptibility to errors due to the comparatively few resources utilized by this system component.

The work described in this paper aims to address the considerable latency and energy used scrubbing the device when the RCN is affected by configuration memory errors. Our contributions are:

1) To localize configuration memory errors more precisely than has previously been reported in the literature and to explain how the response to error signals should be prioritized;
2) To describe a fine-grained method for dynamically reconfiguring sub-components that are suspected of containing configuration memory errors; and
3) To compare the reliability, latency and energy cost of correcting configuration memory errors using the proposed approach with (a) TMR-MER with complete scrubbing of the device when errors are detected outside the TMR modules, (b) on-demand scrubbing of the device when an SEU is detected in the system, and (c) periodic scrubbing of the device as a fault prevention strategy.

The paper is organized as follows: Section II provides background to our work. Section III gives an overview of the TMR circuit model, the effect of errors in different sub-components of the model, and a proposal for a repair strategy that reduces the total number of configuration frames required to recover an SEU in the system, which results in reduced recovery time and energy consumption. Section IV explains how the fine-grained dynamic partial reconfiguration approach relied upon by the proposed approach is implemented. Section V describes our experimental evaluation and details our findings while concluding remarks are given in Section VI.

## II. Background

COTS FPGAs are being considered as an alternative to radiation-hardened devices in Low Earth Orbits (LEO), and, if proven to be sufficiently reliable, may even have applications in Geosynchronous satellites. The main reason for this acceptance is that they have been proven to be able to tolerate destructive radiation effects — Total Ionization Dose (TID) and Single Event Latchup (SEL) [3], which can lead to device failure. Radiation-hardened devices are designed with high immunity to these two effects even in high earth orbits e.g. Geostationary Equatorial Orbit (GEO). Since the radiation level of LEO is hundreds of times lower than that of GEO, and there are hardly

This research was supported under the Australian Research Council's Linkage (LP140100328) and Discovery (DP150103866) Projects funding schemes.

978-1-5090-5602-6/16/$31.00 © 2016 IEEE

any heavy ions that could result in SELs, COTS FPGAs can provide a safe operating platform at this orbital altitude [4]. Secondly, just like radiation-hardened devices that have built-in protection against Single Event Transient (SET) and SEU [5], COTS FPGAs can be strengthened with TMR-S or TMR-MER to mitigate malfunctions caused by SETs and SEUs [3].

Both TMR-MER and TMR-S rely on Dynamic Partial Reconfiguration (DPR) to correct SEUs in the configuration memory. TMR-S can be implemented by simply overwriting all of the configuration frames, or reading and comparing these frames with a golden copy, to replace any frame that is found to be in error. TMR-S operates at frame-level, detecting and correcting faulty frames periodically. To reduce the need for data retrieval, single frame errors can be detected and corrected using Error Correcting Codes embedded in the frame. This approach can be complemented with a device level Cyclic Redundancy Check (CRC) to determine whether or not a complete reconfiguration of the device is required [6]. To reduce the energy consumed by the periodic scrubbing of TMR-S, a scrub cycle can be triggered via a dedicated error transmission network (referred to as a Reconfiguration Control Network (RCN) in TMR-MER). The overall mean-time-to-repair (MTTR) can be further reduced by prioritizing the frames scanned for different error signatures generated by the network [7].

Until now, as it has been described in the literature, the TMR-MER technique is not as robust as TMR-S, since the method can only detect and correct errors inside the TMR modules, while errors inside the majority voters, affecting the interconnecting nets between voters and modules, and the RCNs are almost always neglected. Furthermore, the TMR-MER technique relies upon the vendor's partial reconfiguration flow [8] to generate partial bitstreams, but this flow is not able to generate partial bitstreams for non-block-oriented designs. It is therefore not possible for the flow to generate partial bitstreams for the nets between TMR modules.

This paper takes a significant step towards solving the above-mentioned drawbacks of TMR-MER. In this paper, a repair strategy is used to enhance the localization of single errors within fully-triplicated systems. Inspired by the fact that scrubbing is able to rewrite a frame without resorting to the Partial Reconfiguration Flow [8], a fine-grained reconfiguration approach is applied to determine configuration frame sets pertaining to either block or non-block sub-components after a system is implemented. To save on the cost of storing partial bitstream, we describe and demonstrate a dynamic bitstream composition method that retrieves arbitrary frame data from the full bitstream at run time. These ideas have, to our knowledge, not previously been reported in the literature.

## III. Configuration Memory Error Detection, Classification and Correction

### A. System Architecture

Our system model (Fig. 1) assumes the user circuit comprises $n$ TMR components. In our system model the voters and their input and output nets are triplicated in order to maximize reliability. Each set of triplicated modules, voters and interconnecting nets comprise a single component ($C_k$, $k \in \{0, 1, ...n-1\}$). Fig. 1 illustrates $C_k$ and its interconnections with $C_{k-1}$ and $C_{k+1}$. We assume the modules are acyclic, as discussed in [9], and that cyclic components are implemented by allowing voter outputs to re-enter a component as module inputs. While Fig. 1 illustrates a linear sequence of TMR components, in general there may be several immediate predecessors and immediate successors of $C_k$.

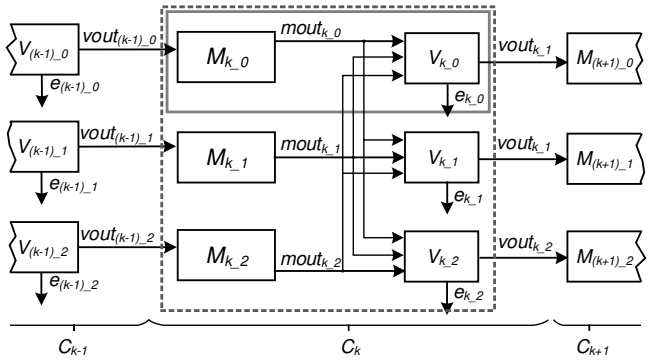


Fig. 1. TMR model

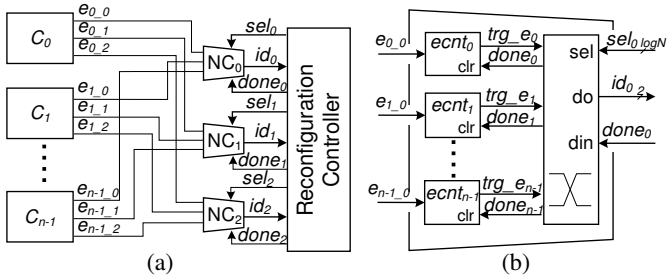


Fig. 2. (a) Star-type RCN, (b) Elaborated NC logic

Each component consists of three identical circuit replicas, referred to as modules, and additional sub-components, including voters and nets (between circuit modules and voters, between voters and downstream components, and between voters and a reconfiguration controller). There are at least three possible approaches to partitioning the component into reconfigurable modules that can be independently recovered when they are found to be in error. Most commonly in the literature, only SEUs in the circuit modules themselves (indicated as $M_{k\_0,1,2}$ in Fig. 1) are localized and corrected. This approach leaves the voters and interconnecting nets to be recovered by other means - perhaps via scrubbing, as proposed in [2], or via fine-grained modular reconfiguration, as studied in this paper. Another approach includes the corresponding voter and some of the interconnecting nets, as illustrated by the grey box surrounding $M_{k\_0}$ and $V_{k\_0}$. This approach allows both sub-components and some of the interconnecting nets to be recovered by modular reconfiguration, but does not provide a means for recovering all of the nets comprising $mout_{k\_0,1,2}$. A third alternative therefore includes all sub-components of a triplicated component in the reconfigurable partition, as illustrated by the dashed box surrounding $C_k$. This approach can be used to cover all the nets except for the voter outputs, which must cross partition boundaries, but suffers a significantly larger reconfiguration delay, which translates to increased Mean Time To Repair (MTTR).

The voter blocks in our model are enhanced to identify the module whose output differs from the majority based on [10]. We have implemented voter logic that not only protects the output of the user circuit, but also detects which module output ($mout_{k\_j}$) is incorrect in the minority and raises a 2-bit error report ($e_{k\_i}$) identifying that module, where $i, j \in \{0, 1, 2\}$. Values 00, 01 and 10 identify the erroneous module ($M_{k\_0,1,2}$), while the value 11 indicates the absence of an error in $C_k$.

The error reports from the triplicated voter blocks are aggregated by a central Reconfiguration Controller (RC) through three identical star-type Reconfiguration Control Networks

(RCNs). The RC thus receives triplicated error reports from each component. Fig. 2(a) illustrates the triplicated RCNs. The RC manages the triplicated Network Controllers (NCs) in order to check the error state of a particular component. The RCNs are synchronous and all NCs operate in lock step to check the voter of each component in turn according to the direction of the RC.

The NC aggregates error signals and distinguishes between transient errors in the user circuit and "permanent" errors that are likely due to configuration memory corruption. Fig. 2(b) elaborates the architecture of an NC. Each of the error signals inputs to a saturating up-down counter. A permanent error is associated with repeated error signals that are expected to saturate the counter and trigger an error report $trg\_e_i$ to the RC [1]. Transient errors will result in both up- and downward counts and will therefore not saturate the counter. A switch selects between the error triggers of the individual counters.

The RC controls the $sel$ signal to the NC switch to select the output of the desired error counter, which triggers the partial reconfiguration of the indicated module. The RC then retrieves the frame data of the indicated module from external memory and writes these to the configuration memory space. When the reconfiguration is complete, the $done$ signal is asserted to clear the corresponding error counter.

### B. Persistent Errors

Permanent errors reported by the counters in the NCs may be, as indicated, due to configuration memory errors present in one of a component's modules. But a substantial number of reported errors are caused by permanent errors present in the voters, the intra-component nets connecting modules and voters, the nets connecting voters to downstream components, as well as in the wires of the RCNs and the NCs themselves. In this paper, we study the effect of configuration memory SEUs on TMR-MER systems and propose a remediation scheme that enhances the reliability of TMR-MER that is more responsive, and saves considerable energy with respect to equivalent systems employing scrubbing.

Configuration memory errors in different parts of a component, as shown in Figs. 1 & 2, result in a range of error symptoms. The error reports received by the RC due to errors in the various sub-components are as follows:

- An error in the configuration of $M_{k\_j}$ may cause $mout_{k\_j}$ to differ from $mout_{k\_i}$, $i \neq j$. In this case all three error outputs for $C_k$ should report $M_{k\_j}$ to be in error.
- Errors in the net $vout_{(k-1)\_j}$ or the majority voter logic of $V_{(k-1)\_j}$ can cause the input for $M_{k\_j}$ to be incorrect. The three RCNs will then report that $M_{k\_j}$ is incorrect but cannot correctly determine that the error lies with the logic of the upstream voter or with its corresponding outgoing nets.
- If an error is present in the minority voter logic of $V_{k\_i}$, the voter may assert that an error is present in $C_k$ while the other two voters report no error. The RC thus only receives one error report for $C_k$ instead of three.
- An error present in the branch of $mout_{k\_j}$ can cause the connected voters to incorrectly indicate the presence or absence of an error in $M_{k\_j}$. If the error is present in the main trunk of the net, all three voters, or just two of them, may assert an error. Errors in the region of $mout_{k\_j}$ where it branches may also cause two different voters to indicate that different modules are in error. This is because a single error in a switch matrix can affect more than two nets [11]. If an error is present in the switch matrix used by any two of $mout_{k\_0,1,2}$, two voters may report different modules to be in error.
- If an error is present in one of the triplicated RCNs, that RCN may raise or mask errors for a random module in some component.

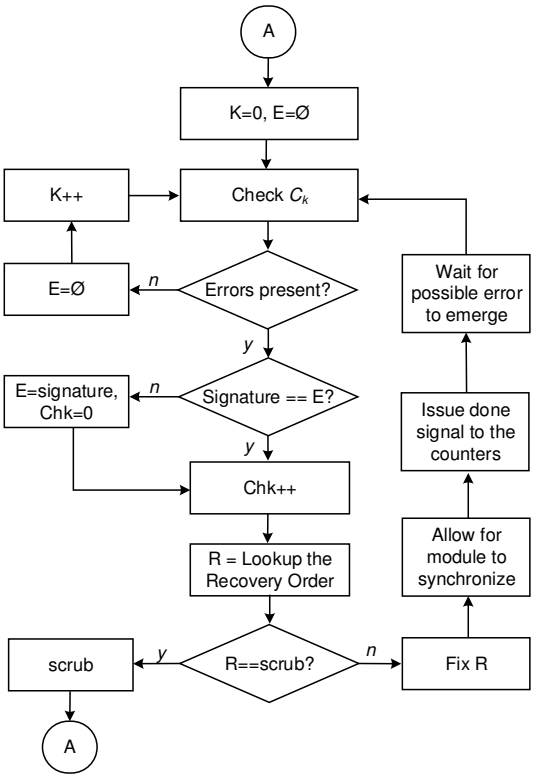


Fig. 3. Repair flow

It is evident that errors in the various sub-components of $C_{k-1}$ and $C_k$ or in RCN$_j$ result in different behaviors at the outputs of the triplicated RCNs which we refer to as error signatures. A single error may cause one of three different types of error signature to be observed for $C_k$. These are: (i) three identical error reports ($id_{0,1,2}$ = $M_{k\_j}$), (ii) one error report ($id_i$ = $M_{k\_j}$), and (iii) all others.

If the error signature is of type (i) and the reconfiguration of the module indicated by the RCNs fails to clear the error, the same error signature will present after the recovery operation, in which case the error is deemed to be persistent. It should then be suspected that the error is in $V_{(k-1)\_j}$, $vout_{(k-1)\_j}$, $mout_k$ or $mout_{(k-1)}$. For error signatures of type (ii), besides $mout_k$, $V_{k\_i}$ and the RCN logic (NC$_i$ & $e_{k\_i}$) are suspects. For error signatures of types (iii), the only suspect is $mout_k$.

Other possible RCN outputs are caused by an accumulation of SEUs. Two or more errors may then be present in different sub-components. We do not consider these effects in this paper, but all could be dealt with by triggering a complete scrub of the device when they appear.

### C. Repair Strategy

We propose a repair strategy to guide the design of an RC that is capable of detecting and repairing persistent errors in systems using fine-grained dynamic reconfiguration. The repair

TABLE I. RECOVERY SEQUENCE

| Error Signature | Number of Checks (Chk) | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| i. $id_{0,1,2} = M_{k\_j}$ | $M_{k\_j}$ | $V_{(k-1)\_j}$ | $vout_{(k-1)\_j}$ | $mout_k$ | $mout_{(k-1)}$ | scrub |
| ii. $id_i = M_{k\_j}$ | $V_{k\_i}$ | $mout_k$ | $\text{NC}_i$ | $e_{k\_i}$ | scrub | |
| iii. others | $mout_k$ | scrub | | | | |

strategy includes a repair flow and a recovery sequence as determined by the flow.

*1) Repair Flow:* Fig. 3 depicts the proposed flow for recovering from persistent errors in the system. An error check and correction cycle commences at entry point A, when the component number K and the error signature E are cleared. The first component is checked after initialization. K is incremented while the RCN reports no errors for the component currently being checked. When an error is reported for $C_k$ the error signature is compared with the previously recorded signature E and the check index Chk is incremented if so. The error signature and check index indicate which sub-component R is to be reconfigured (Table I). After the dynamic reconfiguration of the sub-component has been performed, the RC waits for the component to be resynchronized [1] and issues a done signal to the RCN to reset the error counters for $C_k$. After the error counters have been cleared, the RC waits for a period of time to allow any residual error to once again manifest itself by saturating the error counters. This period depends upon the saturation level of the counter and the latencies of component $C_k$ and the RCN. These wait times are of the order of a few µs [1], [2]. The RC checks whether $C_k$ is still affected by errors, and if the same error signature is decoded, Chk is incremented in order to recover the next sub-component indicated by the recovery sequence. If errors persist after all suspect sub-components have been reconfigured, a scrub is performed to sweep away all accumulated errors, and the flow is re-initialized by returning to entry point A.

*2) Recovery Sequence:* Table I lists the repair order we use with the flow. The recovery sequence is based on the suspects identified for each error signature. For each type of signature, the priority is to recover logic blocks first, followed by the component nets. The main reason for this preference is that the programmable bits are more densely located in the logic blocks than in the nets, while the nets are more vulnerable at their source and destination terminals, which are always included in the logic blocks. Reconfiguring the block components first results in higher repair efficiency. For signature (i), $M_{k\_j}$ is to be recovered first. If the recovery of the module does not clear the error, the error is deemed persistent and $V_{(k-1)\_j}$ is recovered next, followed by the three component nets $vout_{(k-1)\_j}$, $mout_k$ and $mout_{(k-1)}$. For signature (ii), since the utilization of routing and switch resources for dedicated nets and NC-related logic is far lower than for the logic sub-components, the recovery sequence commences with logic blocks first and follows up with recovery of the RCN sub-components: $V_{k\_i}$ and $mout_k$ are recovered first, followed by $\text{NC}_i$ and the dedicated routing for $e_{k\_i}$. For signature (iii), it is only worthwhile recovering $mout_k$ before resorting to a complete scrub of the device.

### D. Triggered Scrubbing

Systems that solely rely on scrubbing to correct configuration memory errors usually perform a scrub periodically as they do not usually monitor system outputs to determine when configuration memory errors may be present. However, when the outputs of system components are monitored using the architecture outlined in Section III-A, it is feasible to trigger a scrub cycle when a configuration error is detected. In this case, the error correction flow of Fig. 3 can be adapted to perform a scrub whenever an error is detected, and to restart the flow after the scrub is finished. In this case the prioritized recovery sequence is ignored.

## IV. FINE-GRAINED DYNAMIC RECONFIGURATION

Conventional TMR-MER SoCs (e.g. [1], [12], [13]) rely on the vendor's partial reconfiguration flow [8] to generate partial bitstreams for the component modules. These partial bitstreams are stored in external memory and a lookup table is created for the RC to index these files [1], [13]. When a permanent error is detected, the RC fetches the indexed file from memory and writes it to the ICAP. However, as indicated in Section II, the flow is not flexible enough for robust and efficient SEU recovery in fault-tolerant systems. This is mainly because the original intention of the flow was to create a partial region to allow for dynamic hardware changes. Partition pins have to be placed on the boundary of the region, which result in extra delay on the interface signals as well as slower compilation and longer design cycles. Most importantly, in the architecture described in Section III-A, the RCN and the interconnecting nets between voters and modules are not amenable to modular reconfiguration using the partial reconfiguration flow.

In this section, we propose a fine-grained dynamic partial reconfiguration approach (FDPR). Our approach uses the standard FPGA project flow while overcoming the drawbacks of the vendor's partial reconfiguration flow when applied to fault-tolerant systems. The approach includes a design method for identifying the sets of frames pertaining to the sub-components, such as the module outputs $mout_k$, voter outputs $vout_{k\_j}$, RCN nets $e_{k\_i}$, voters and network controllers, which needs to be applied after the design is floorplanned and implemented using the standard flow. We also describe a bitstream composition method that enables retrieving arbitrary sets of frame data from a full bitstream at run time.

### A. Major Columns on 7-Series FPGAs

We describe the proposed FDPR approach for Xilinx 7-series devices. Programmable resources on a 7-Series device are tiled in major columns [8] [14]. A major column is a column of resources in a single clock region, of which there are several in a device. It also includes a column of switch matrices that provides access to the general routing matrix. A major column is configured via a contiguous set of configuration frames, each of which forms a bit slice of the configuration memory for the major column. The size of the contiguous set of frames used to configure a major column depends upon the type of resources it contains. Every pair of major columns shares a local clock buffer.

### B. Configuration Memory Boundaries

An FPGA application circuit is typically implemented using a number of CLBs. Interconnecting signals are routed via switch matrices to their destination CLBs. Floorplanning enables resource allocation within a specified region, which may be as

narrow as a single major column. The placer can be instructed to only place the logic of part of a design (a logic block) within such a specified region. The router can also be constrained to only use the switch matrices within the region for the internal routing of the logic block. When a logic block and its internal routing are constrained in this way, the design can be partially reconfigured to recover from configuration memory errors that affect the block by just reconfiguring the configuration frames of that region. If the design is synchronous, then local clock buffers are also used, and the other major column that shares the clock buffer must also be reconfigured.

The detailed routing of nets that interconnect different logic blocks can be retrieved from the implementation database of the design. The information stored in the database includes the name of any entry or exit Programmable Interconnect Pin (PIP) the net uses, the switch matrices that own these PIPs, and the segments of the general routing matrix forming the net. The major columns that implement the net can be extracted from the names of the switch matrices used. Configuration memory errors in a net can thus also be recovered by simply reconfiguring the configuration frames of the major columns it uses.

### C. Floorplanning

Fig. 4 illustrates one possible floorplan for the proposed TMR architecture outlined in Section III-A. The circuit spans three rows of major columns. Modules and voters are constrained to be placed in different major columns as shown in Fig. 4(a). Fig. 4(b) presents the major columns relating to the interconnection nets between the modules and the voters. As the output nets from the triplicated modules ($mout_{k\_0,1,2}$) are intertwined, we treat them as a single sub-component for recovery purposes. The router automatically routes voter outputs and module outputs in the gaps between the closest module and voter. Fig. 4(c) shows the major columns relating to the triplicated RCN. The NCs are also isolated and constrained to different rows of major column. The NCs connect to the voters via error signals. The signal sinks correspond to switch matrices. The major columns that contain these switch matrices will be recovered if the RCN signals are determined to be in error. The routes of the triplicated RCN are isolated because they are implemented in different major column rows. However, in large-scale systems, the RCN connects voters from different regions of the device. It may then become impossible to isolate the RCN replicas from each other in this way. Although few single errors would cause more than two RCN replicas to fail, a better design method is then needed to avoid the use of PIPs that will bridge two nets if corrupted.

To improve area usage, small modules can be implemented in a single clock region using a similar layout. However, in this case, each copy should be provided with its own clock signal to ensure ongoing protection if an SEU disables a clock buffer.

### D. Dynamic Repair Bitstream Composition

A full bitstream is written when the device boots. This bitstream comprises a bitstream header and configuration frame data. The bitstream header is the command sequence that is loaded before the frame data are written. After the commands are issued to the device fabric, the data for the first frame, at frame address 0, are written. The frame address register auto-increments and the data of the second frame are written straight after the last word of the first frame.

Usually a full bitstream is difficult to index. However, compared with a typical full bitstream, the full bitstream for critical systems usually have single-frame CRC checking enabled.

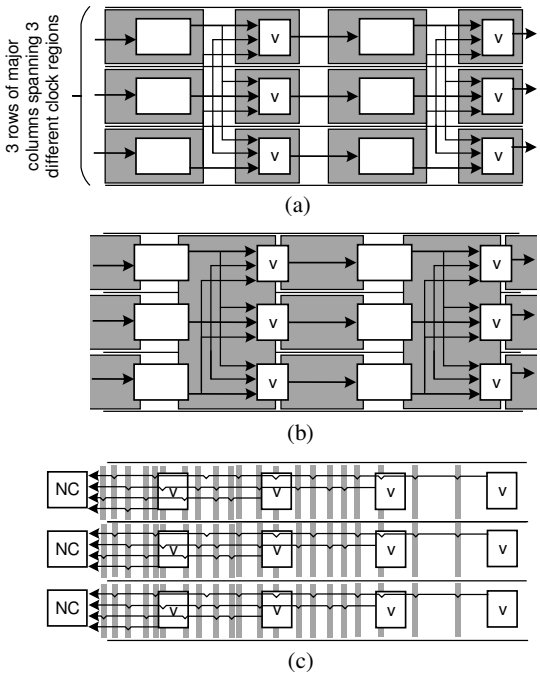


Fig. 4. One possible floorplan (a) Logic, (b) Intra-component nets, (c) RCN

Frame headers are then inserted after every single frame of data. The header contains the CRC check sequence as well as the address of the previous frame. We use this information to extract frame addresses and to index them according to their byte offset in the file. When the bitstream is loaded into external memory, we can then extract the data for arbitrary frames according to their byte offsets.

Storing the frame index potentially necessitates a large context memory for the RC. To minimize the amount of memory required, we only index the address of the first frame for each major column that is needed for the recovery strategy. The range of frame addresses for the sub-components can be calculated off-line and stored with the index. The frame write operation is done as for the full bitstream with single frame CRC check enabled.

## V. Experimental Evaluation

In this section, we evaluate the recovery latency, energy used in correcting configuration memory, and the reliability of the proposed approach, and compare the results with those obtained for MER when scrubbing is used to recover from errors that occur outside the TMR modules, with on-demand scrubbing that is triggered by SEUs located anywhere in the system, and with periodic scrubbing. We evaluate the fine-grained dynamic reconfiguration method described in Section IV via an exhaustive fault injection experiment on a typical hardware implementation of the system model of Section III-A, based on a Nexys-4 Video board implemented using Vivado 2015.4. The board includes a Xilinx Artix-7 XC7A200T device which is clocked at 100MHz.

The purpose of the fault injection experiment was to find the critical bits of the test circuit, to log their locations, and to record the error signatures they gave rise to. The results were used to validate the persistent error effects described in Section III-B, to evaluate the soft-error vulnerability of the sub-components in the test circuit, and to evaluate the performance of the proposed recovery method.

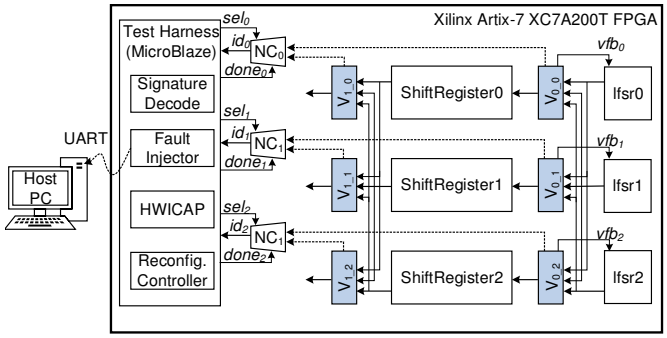


Fig. 5. Experimental setup

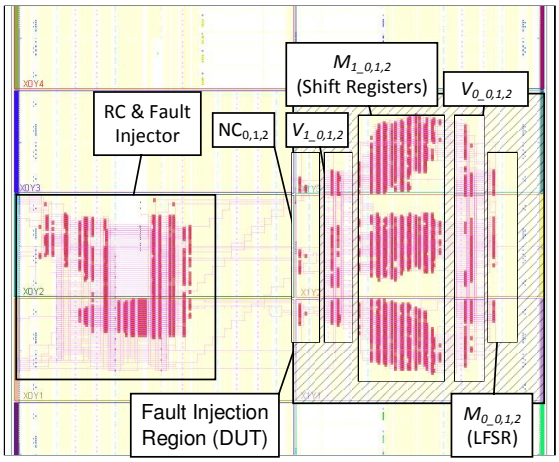


Fig. 6. Floorplan of the test harness, and test circuit in three clock regions on the right-hand side of the device

### A. Experimental Setup

*1) Test Circuit:* Fig. 5 depicts a block diagram of the test system implemented on the Nexys-4 Video board. The test circuit comprises two TMR components representing both cyclic state-machine logic and acyclic datapath logic [1]. To represent a typical state machine, we chose a 64-bit Linear-Feedback Shift Register (LFSR). The LFSR serves as a random test vector generator for the DUT and emulates the upstream voters and nets illustrated in Fig. 1. The acyclic datapath logic was represented by a Shift Register (SR) module in which all logic paths travelled without feedback from the input to the output of the module. The module comprised 8 stages of 64-bit registers with a variety of arithmetic operations mapped into the look-up tables (LUTs) of each stage. In our experimental setup, the SR module processed the data generated by the LFSR module.

The LFSR modules, their voters and module outputs comprise $C_0$ of the design. As there is feedback from the LFSR voter back to the LFSR input, the LFSR voter also behaves as an upstream voter with respect to the LFSR component. The SR modules, its voters, and their connection to the RCN form $C_1$ of the Design Under Test (DUT).

We floorplanned the test circuit according to the guidelines provided in Section IV-C. The DUT contained 26 sub-components in total. These included three identical LFSR modules ($M_{0\_0,1,2}$) and three SR modules ($M_{1\_0,1,2}$), six voters ($V_{0,1\_0,1,2}$), triplicated NCs ($\mathrm{NC}_{0,1,2}$), two sets of nets connecting the modules to their voters ($mout_{0,1}$), three voter output signals ($vout_{0\_0,1,2}$), and the dedicated error signal for each voter ($e_{0,1\_0,1,2}$). The voter output of $C_1$ was not evaluated because it did not connect to any downstream logic in this design. The frame sets for these sub-components were extracted using the method described in Section IV-B.

TABLE II. BLOCK SUB-COMPONENT AREA & RECOVERY TIME

| Sub-component | LUT | Flip-Flops | Major Columns | Frames | Recovery Time (ms) |
|---|---|---|---|---|---|
| $M_{0\_0,1,2}$ | 8 | 64 | 2 | 64 | 2.0 |
| $V_{0,1\_0,1,2}$ | 153 | 66 | 2 | 72 | 2.3 |
| $M_{1\_0,1,2}$ | 2,548 | 512 | 20 | 712 | 22.6 |
| $\mathrm{NC}_{0,1,2}$ | 43 | 34 | 2 | 72 | 2.3 |

TABLE III. NET SUB-COMPONENT AREA & RECOVERY TIME

| Sub-component | PIPs | Switch Matrices | Major Columns | Frames | Recovery Time (ms) |
|---|---|---|---|---|---|
| $mout_0$ | 10,690 | 632 | 35 | 1,188 | 39.7 |
| $mout_1$ | 10,358 | 637 | 50 | 1,712 | 55.8 |
| $vout_{0\_0}$ | 3,077 | 161 | 12 | 416 | 13.7 |
| $vout_{0\_1}$ | 3,050 | 143 | 13 | 452 | 14.8 |
| $vout_{0\_2}$ | 3,131 | 172 | 12 | 416 | 13.7 |
| $e_{0\_0}$ | 41 | 11 | 11 | 380 | 12.3 |
| $e_{0\_1}$ | 41 | 18 | 18 | 624 | 20.3 |
| $e_{0\_2}$ | 41 | 19 | 16 | 552 | 18.0 |
| $e_{1\_0}$ | 22 | 7 | 4 | 136 | 4.6 |
| $e_{1\_1}$ | 23 | 7 | 5 | 172 | 5.7 |
| $e_{1\_2}$ | 22 | 4 | 3 | 100 | 3.4 |

*2) Test Harness:* The test harness implemented a fault injector, an error signature decoder, and an RC program running FDPR of Section IV-D.

Since the full bitstream was stored in an SPI flash memory from which the device booted, we implemented a flash controller to access this memory. The controller allowed a maximum read throughput of 25 Mbyte/s through quad SPI protocol running at 50MHz. We stored the byte offset of the first frame of each major column in an array. The total program memory needed for the array was 530 entries, corresponding to the $5\times106$ major columns available in the device. The list of the major columns used by the sub-components was also stored in program memory, the size of which depended on the number of major columns they used. When the reconfiguration of a sub-component was triggered, the data for the frames of the corresponding major columns were fetched from the flash memory and transferred to the ICAP with the required frame headers using a DMA engine.

We measured the reconfiguration latency of different sub-components and used the same hardware setup to measure the latency of a complete blind scrub of the device. The frames to be scrubbed were also extracted from the full bitstream as described in Section IV-D.

Fig. 6 shows the layout of the test system on the device. The test circuit was implemented in the three shaded clock regions depicted on the right side of the device — we injected faults (configuration bit flips) into this region using dynamic partial reconfiguration. The MicroBlaze was implemented in the two central clock regions on the left side of the device. This region was not subjected to fault injection. In total, we injected 16,134,144 faults into 4,992 frames of the DUT, thereby injecting a fault into every configuration bit of the DUT.

After an error was injected, the RC was programmed to wait for 1 µs in order to let the error emerge. Only then did it check the output of the RCNs. For this DUT, a bit was deemed not to be sensitive if the RCN did not report an error. Otherwise, the

TABLE IV. FAULT INJECTION RESULTS

| Sig | Number of Checks (Chk) | | | | | | Sub-component Reconfiguration Sequence |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | |
| $i_0$ | 4,882 | 959 | N/A | 627 | N/A | 0 | $M_{0\_0,1,2} \rightarrow V_{0\_0,1,2} \rightarrow mout_0$ |
| $ii_0$ | 15,862 | 6,623 | 159 | 44 | 0 | | $V_{0\_0,1,2} \rightarrow mout_0 \rightarrow \text{NC}_{0,1,2} \rightarrow e_{0\_0,1,2}$ |
| $iii_0$ | 3,656 | 0 | | | | | $mout_0$ |
| $i_1$ | 633,901 | 4,788 | 2,038 | 748 | 23 | 0 | $M_{1\_0,1,2} \rightarrow V_{0\_0,1,2} \rightarrow vout_{0\_0,1,2} \rightarrow mout_1 \rightarrow mout_0$ |
| $ii_1$ | 18,248 | 7,282 | 115 | 0 | 0 | | $V_{1\_0,1,2} \rightarrow mout_1 \rightarrow \text{NC}_{0,1,2}$ |
| $iii_1$ | 4,364 | 0 | | | | | $mout_1$ |

TABLE V. SUB-COMPONENT CRITICAL BITS SUMMARY USING PROPOSED RECOVERY SEQUENCE

| Sub-component | Critical Bits Found | Average Recovery Latency (ms) | | |
|---|---|---|---|---|
| | | MER/ FDPR | MER/ Scrub | Triggered Scrub |
| $M_{0\_0,1,2}$ | 1,307 | 2.0 | 2.0 | 216.0 |
| $mout_{0\_0,1,2}$ | 3,590 | 41.5 | 216.2 | 216.0 |
| $V_{0\_0,1,2}$ | 4,107 | 7.4 | 221.1 | 216.0 |
| $vout_{0\_0,1,2}$ | 430 | 38.9 | 238.6 | 216.0 |
| $e_{0\_0,1,2}$ | 15 | 48.3 | 216.0 | 216.0 |
| $M_{1\_0,1,2}$ | 211,300 | 22.6 | 22.6 | 216.0 |
| $mout_{1\_0,1,2}$ | 4,058 | 59.9 | 216.8 | 216.0 |
| $V_{1\_0,1,2}$ | 3,655 | 2.3 | 216.0 | 216.0 |
| $\text{NC}_{0,1,2}$ | 77 | 52.4 | 216.0 | 216.0 |

RC reported the sensitive bit location and the error signature to the host. Thereafter, the faulty bit was corrected by reversing the injected bit flip. Between each injection cycle, the RC waited another 1 µs to allow the correct data to flush through the test circuit.

The host PC used the bit locations reported as causing errors to determine which sub-component of the design was affected and, together with the reported error signature, determined which sub-components would have been reconfigured to clear the error according to the repair strategy described in Section III-C. The total number of frames that would consequently have been reconfigured and the reconfiguration latency that would have been incurred to recover the observed errors were logged against each error report. Note, therefore, that for this experiment the repair strategy was not actually running on the board. Instead, the host simulated the strategy based on the error location and signatures contained in the error reports it received and assuming that the error would have persisted until the sub-component containing the error, as identified by the error location, would have been reconfigured.

### B. Results

*1) Fine-grained Dynamic Reconfiguration of Subcomponents:* Tables II & III report the utilization, number of major columns and number of configuration frames for each of the sub-components in the design, as well as the reconfiguration times that we measured using the proposed fine-grained dynamic bitstream composition method. Using this method, on our platform we found that we could sustain a transfer rate of one frame every 32 us. This is a reasonably good result considering the constraints imposed by the board architecture (SPI flash) and the use of Microblaze and AXI-HWICAP.

While the LFSR modules are relatively small, since they are synchronous, we need to reconfigure the neighboring major columns to recover from any errors affecting their clock buffers, as explained in Section IV-B. In our design, the outputs of the LFSR voters were registered to separate the feedback path from the module output. For efficiency's sake, we included the feedback path in the $mout_0$ sub-component.

On our platform, we found that to perform a blind scrub of the device we had to overwrite 18,300 frames in total. The latency for a scrub cycle was 432 ms, which corresponds to a sustained transfer rate of one frame every 24 us. This performance is limited not just by the board and circuit architecture, but also by the need in our test to retrieve each frame individually from the indexed complete bitstream. It should be noted that the use of a purpose-designed scrubber could be expected to use considerably less than 432 ms to scrub the device.

On the tested circuit, the worst-case repair time using our proposed repair strategy was 135 ms, which involved the reconfiguration of one of the SR modules, its upstream voter and voter output, the nets between the SR modules and their voters ($mout_1$), and those of the LFSR modules and their voters ($mout_0$ of the upstream component). This maximum repair time is approximately 1/3 of the scrub cycle latency, which represents a substantial reduction in the repair time. At most, we found we had to reconfigure just over 4,136 frames, which is less than 1/4 of those needed to perform a scrub.

*2) Fault Injection Results:* Table IV reports on the exhaustive fault injection experiment. The subscripts indicate the signatures for the LFSR and SR components (0 & 1) respectively. The table reports for each error signature ($i_0$ - $iii_1$) how many reconfigurations of the sub-components were triggered (Number of Checks) according to our repair strategy. The sub-components reconfigured for each error signature and their priority, as proposed by the repair sequence, is also listed.

As can be seen from Table IV, a complete scrub of the device was not required to recover any emulated SEU during our fault injection experiment. We therefore conclude that the repair strategy is effective at quickly recovering from single errors within the test circuit, and that in the worst case, the strategy is able to recover from errors substantially faster and using considerably less energy than a scrub cycle.

In total, 680,931 errors were reported for 16,134,144 fault injections. For error signature (i), the number of reports for the LFSR and SR modules differ greatly because their utilizations differ greatly. For error signatures (ii) & (iii), both components show the same trend on the number of checks per signature. It is clear that voters and the interconnecting nets between voters and modules are much more prone to errors than the RCNs. Since the LFSR component is further away from the NC, its routing net utilization is greater than for the SR component, and thus, its RCN nets present more errors. However, since the reconfiguration frames for these sub-components are different, the average number of frames recovered for these sub-components differ.

### C. Frames, MTTR & Energy

Table VI compares the average number of frames reconfigured per error, the average recovery time and the energy expended to repair the error assuming each frame write consumes 535 nJ [15]. The proposed fine-grained DPR approach to MER is listed as MER/FDPR and its performance is compared with,

TABLE VI. FRAMES, MTTR AND ENERGY RESULTS

| | MER/FDPR | MER/Scrub | Triggered Scrub |
|---|---|---|---|
| **Frames** | 695 | 1,950 | 18,300 |
| **MTTR (ms)** | 22 | 36 | 216 |
| **Energy (mJ)** | 0.37 | 1.04 | 9.79 |

on the one hand, a more conventional approach to MER in which errors that occur outside the reconfigurable modules are recovered by scrubbing the device (MER/Scrub) [2], and on the other, by triggering a scrub whenever any error is detected in the system (Triggered Scrub). In each case, the test system is used to trigger reconfigurations or scrubs as outlined in Section III. While triggered scrubbing needs to scrub 18,300 frames, MER/FDPR only needs to reconfigure 695 frames on average. While these results are application and device dependent, they are representative of the gains that are possible. In our case, the MTTR for MER/FDPR was only 10% of that for scrubbing. MER/FDPR is also the most energy efficient approach of those we have studied. As energy consumption is assumed to be proportional to the number of frames that are rewritten, in our study we found the fine-grained DPR approach was 2.8× more efficient than MER/Scrub and 26× more efficient than triggered scrubbing. A periodic scrubbing approach is likely to expend more energy or compromise MTTR if the period between scrubs is reduced.

### D. Reliability

Table V summarizes the critical bits we found in the test circuit via the fault-injection experiment and the average recovery time for these sub-components using MER/FDPR, MER/Scrub and Triggered Scrub. The average recovery times for sub-components using MER/FDPR and MER/Scrub were derived from Table IV, which was used to calculate the number of times an error was found in each sub-component, and to identify which sub-components were reconfigured according to the reconfiguration sequence. The average recovery time was used as input to a Markov reliability model derived from [16], [2]. Fig. 7 plots the resulting reliability of the test circuit using the proposed MER/FDPR, the more conventional MER/Scrub, Triggered Scrub and periodic scrubbing (Periodic Scrub) in LEO and GEO over a 4-year mission. Furthermore, we have zoomed in to certain parts of the reliability plots to show how the MER/FDPR performance compares with MER/Scrub. Note that the Periodic Scrub plot needs the same recovery time as Triggered Scrub, but that it excludes the RCN sub-components ($NC_{0,1,2}$ and $e_{0\_0,1,2}$), as these are not needed for this approach. In our analysis, we have assumed high bit failure rates using the peak 5-min CREME96 model [17] with 2.54 mm aluminum shielding for these two orbits. Thus, in LEO, we assume 8.41E-12 upset/bit·s, while in GEO, this figure is 2.66E-10 upset/bit·s. We find that both MER/FDPR and MER/Scrub approaches can be expected to be more reliable than scrubbing because of the significant reduction in the MTTR of critical system sub-components.

## VI. CONCLUSION

In this paper, we have proposed a fine-grained module-based error detection and dynamic reconfiguration scheme that can detect and recover any error present in TMR-based SoCs without additional hardware or resorting to scrubbing. We evaluated our method by injecting faults into a typical test circuit. The result shows that with the proposed method, we can achieve a slight improvement in reliability over the more conventional module-based configuration memory error recovery scheme that triggers a scrub cycle when an error repeats

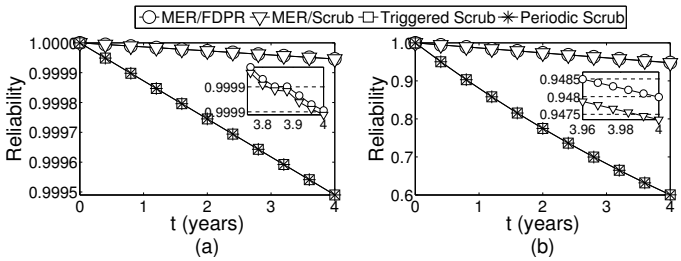


Fig. 7. Reliability results: (a) $\lambda_{bit} = 8.41E-12$ (LEO), (b) $\lambda_{bit} = 2.66E-10$ (GEO)

or the triplicated reconfiguration control networks disagree [2], but it is also much more energy efficient — in our case we found an improvement of almost two-thirds in the energy consumption. The advantages of the proposed approach in terms of energy efficiency and reliability over triggered and periodic scrubbing are even greater. Finally, it should be noted that our fine-grained reconfiguration method applies to Xilinx-4 to 7-series devices, but may not port directly to the UltraScale family.

## REFERENCES


[1] E. Cetin, O. Diessel, L. Gong, and V. Lai, "Towards bounded error recovery time in FPGA-based TMR circuits using dynamic partial reconfiguration," in *FPL*, 2013, pp. 1–4.

[2] D. Agiakatsikas, N. T. Nguyen, Z. Zhao, T. Wu, E. Cetin, O. Diessel, and L. Gong, "Reconfiguration Control Networks for TMR Systems with Module-based Recovery," in *FCCM*, 2016, pp. 88–91.

[3] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam, "Mitigation of Radiation Effects in SRAM-Based FPGAs for Space Applications," *CSUR*, vol. 47, no. 2, p. 37, 2015.

[4] D. M. Hiemstra and V. Kirischian, "Single Event Upset Characterization of the Kintex-7 Field Programmable Gate Array Using Proton Irradiation," in *REDW*, 2014, pp. 1–4.

[5] G. Swift, C. Carmichael, G. Allen, G. Madias, E. Miller, R. Monreal *et al.*, "Compendium of XRTC radiation results on all single-event effects observed in the Virtex-5QV," *MAPLD*, pp. 1–33, 2011.

[6] *PG036: Product Guide - Soft Error Mitigation Controller (v4.1)*, Xilinx Inc., 2015.

[7] G. L. Nazar, L. Pereira Santos, and L. Carro, "Fine-Grained Fast Field-Programmable Gate Array Scrubbing," *IEEE Trans. on VLSI Systems*, vol. 23, no. 5, pp. 893–904, 2015.

[8] *UG909: Vivado Design Suit User Guide - Partial Reconfiguration*, Xilinx Inc., 2015.

[9] J. M. Johnson and M. J. Wirthlin, "Voter insertion algorithms for FPGA designs using triple modular redundancy," in *FPGA*, 2010, pp. 249–258.

[10] S. D'Angelo, C. Metra, S. Pastore, A. Pogutz, and G. R. Sechi, "Fault-tolerant voting mechanism and recovery scheme for TMR FPGA-based systems," in *DFT*, 1998, pp. 233–240.

[11] G. Asadi and M. B. Tahoori, "Soft error rate estimation and mitigation for SRAM-based FPGAs," in *FPGA*, 2005, pp. 149–160.

[12] C. Bolchini, A. Miele, and C. Sandionigi, "A novel design methodology for implementing reliability-aware systems on SRAM-based FPGAs," *IEEE Trans. on Computers*, vol. 60, no. 12, pp. 1744–1758, 2011.

[13] M. Straka, J. Kastil, Z. Kotasek, and L. Miculka, "Fault tolerant system design and SEU injection based testing," *Microprocessors and Microsystems*, vol. 37, no. 2, pp. 155–173, 2013.

[14] *UG470: 7 Series FPGAs Configuration User Guide*, Xilinx Inc., 2013.

[15] J. Tonfat, F. Kastensmidt, and R. Reis, "Analyzing the effectiveness of a frame-level redundancy scrubbing technique for SRAM-based FPGAs," *IEEE Trans. on Nuclear Science*, vol. 62, no. 6, pp. 3080–3087, Dec 2015.

[16] D. McMurtrey, K. S. Morgan, B. Pratt, and M. J. Wirthlin, "Estimating TMR reliability on FPGAs using Markov models," 2008. [Online]. Available: http://scholarsarchive.byu.edu/facpub/149

[17] A. Tylka, J. Adams, P. Boberg, B. Brownstein, W. Dietrich, E. Flueckiger, E. Petersen, M. Shea, D. Smart, and E. Smith, "CREME96: A revision of the cosmic ray effects on micro-electronics code," *IEEE Trans. on Nuclear Science*, vol. 44, no. 6, pp. 2150–2160, Dec 1997.