Fine-Grained Module-Based Error Recovery in FPGA-Based TMR Systems

ZHUORAN ZHAO, NGUYEN T. H. NGUYEN, DIMITRIS AGIAKATSIKAS, GANGHEE LEE, EDIZ CETIN†, and OLIVER DIESSEL, UNSW Sydney and †Macquarie University

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 between subcomponents, such as in interconnection nets, which are not recovered by MER. This article addresses this problem by proposing a fine-grained module-based error recovery technique, which can localize and correct errors that classic MER fails to do without additional system hardware. We evaluate our proposal via fault-injection campaigns on three types of circuits implemented in Xilinx 7-Series devices. With respect to scrubbing, we observed reductions in the mean time to repair configuration memory errors of between 48.5% and 89.4%, while reductions in energy used recovering from configuration memory errors were estimated at between 77.4% and 96.1%. These improvements result in higher reliability for systems employing TMR with fine-grained reconfiguration than equivalent systems relying on scrubbing for configuration error recovery.

CCS Concepts: • **Computer systems organization → System on a chip**; **Reliability**; *Redundancy*;

Additional Key Words and Phrases: SRAM FPGA, radiation-induced errors, configuration memory errors, partial reconfiguration, dynamic reconfiguration, reliability, mean time to recover, recovery energy

ACM Reference format:
Zhuoran Zhao, Nguyen T. H. Nguyen, Dimitris Agiakatsikas, Ganghee Lee, Ediz Cetin, and Oliver Diessel. 2018. Fine-Grained Module-Based Error Recovery in FPGA-Based TMR Systems. *ACM Trans. Reconfigurable Technol. Syst.* 11, 1, Article 4 (January 2018), 23 pages.
https://doi.org/10.1145/3173549

1 INTRODUCTION

SRAM-based FPGAs utilize SRAM cells to both implement a desired digital circuit and to store its state while operating. Therefore, Single Event Upsets (SEUs) in SRAM-based FPGA designs affect both their user memory space, such as block RAM (BRAM) contents and flip-flop (FF) state, as well as the configuration memory space that determines the functionality of the design. This

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

Authors' addresses: Z. Zhao, School of Computer Science & Engineering, UNSW Sydney NSW 2052, Australia, (Current address) Room 706, Zhongguancun International Innovation Building, Haidian District, Beijing 100080, China; email: zhuoran.zhao@outlook.com; N. T. H. Nguyen, D. Agiakatsikas, G. Lee, and O. Diessel, School of Computer Science & Engineering, UNSW Sydney NSW 2052, Australia; emails: {h.nguyentran, d.agiakatsikas, ganghee.lee, o.diessel}@unsw.edu.au; E. Cetin, School of Engineering, Macquarie University NSW 2109, Australia; email: ediz.cetin@mq.edu.au.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 ACM 1936-7406/2018/01-ART4 $15.00
https://doi.org/10.1145/3173549

article is concerned with methods to ensure reliable operation of user circuits in high-radiation environments. In particular, we focus on the timely and efficient recovery from configuration memory errors due to SEUs. This article considers approaches to recovering from the most prevalent (single-bit) errors.

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 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 corrupted memory frames. On the other hand, module-based error recovery (TMR-MER) reconfigures the frames of a TMR module when an error in its configuration memory is detected. While scrubbing occurs periodically, whether errors are present, TMR-MER relies on the repeated detection of an error by the same TMR voter to trigger a reconfiguration of the module presenting the error [6]. 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 Reference [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. Due to the distributed nature of RCN resources, [2] resorted to scrubbing the 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 it.

The work described in this article aims to address the considerable latency and energy used scrubbing the device when components outside the modules of the triplicated components, such as the inter-component nets and the RCN, are affected by configuration memory errors. Our contributions are as follows:

(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; and (c) periodic scrubbing of the device as a fault prevention strategy.

The article is organized as follows: Section 2 provides background to our work. Section 3 gives an overview of the TMR circuit model and the effect of errors in different sub-components of the model and proposes a repair strategy that reduces the total number of configuration frames and, thus, the time and energy required to recover an SEU in the system. Section 4 explains how the fine-grained dynamic partial reconfiguration employed by us is implemented. Section 5 describes our experimental evaluation and details our findings while concluding remarks are given in Section 6.

This article builds on and extends our earlier work [22] by providing more detailed explanations of the methods we use to implement fine-grained module-based dynamic reconfiguration and extending the evaluation of the proposed approach to cover several more complex designs, including three High Level Synthesis– (HLS) generated circuits and one real nine-component system. While the extended evaluation demonstrates that the initial premise of Reference [22] can be realized more generally, it also highlights the potential to improve the performance and usability of the proposed method.

2 BACKGROUND

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 periodically at frame level, detecting and correcting bit-errors in frames. The period between scrub cycles and the long scrub latency lead to a relatively high Mean Time To Repair (MTTR), and since scrubbing occurs obliviously, irrespective of error occurrence, TMR-S also suffers a recovery energy penalty relative to TMR-MER. To reduce the need for data retrieval from off-chip memory, 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 a complete reconfiguration of the device is required [19]. To reduce the energy consumed by the oblivious and periodic nature of scrubbing in TMR-S, a scrub cycle can be triggered via a dedicated error transmission network (referred to as an RCN in TMR-MER). The overall MTTR can also be reduced by prioritizing the frames scanned for different error signatures generated by the network [13].

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 in the modules that make up the TMR components, while errors in the majority voters, the interconnecting nets between modules and voters, the RCN wires, as well as the single points of failure, are almost always neglected. Furthermore, the TMR-MER technique relies on the vendor's partial reconfiguration flow [20] to generate partial bitstreams, but this flow is not able to generate efficient partial bitstreams for recovering non-block-oriented designs. Hence it is not possible for the flow to generate efficient partial bitstreams for the nets between the TMR modules.

This article takes a significant step towards solving the above-mentioned drawbacks of TMR-MER. A repair strategy that enhances the localization of single errors within fully triplicated systems is presented. Inspired by the fact that scrubbing is able to rewrite a frame without resorting to the Partial Reconfiguration Flow [20], a fine-grained reconfiguration approach is proposed to determine configuration frame sets pertaining to sub-components after a system is implemented. To save the cost of storing partial bitstreams, we describe and evaluate a dynamic bitstream composition method that retrieves the desired frame data from the full bitstream at runtime.

3 CONFIGURATION ERROR DETECTION, CLASSIFICATION, AND CORRECTION

3.1 System Architecture

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

There are at least three possible approaches to partitioning and constraining the placement and routing, i.e., floorplanning, the system's TMR components so that they can be partially reconfigured to recover from SEUs. In the most commonly reported approach, only SEUs in the circuit

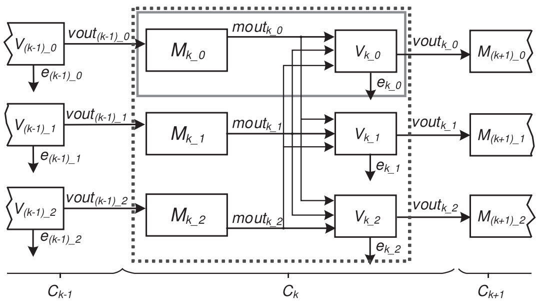


Fig. 1. System model showing three interconnected TMR components, C_{k-1}, C_k, and C_{k+1}.

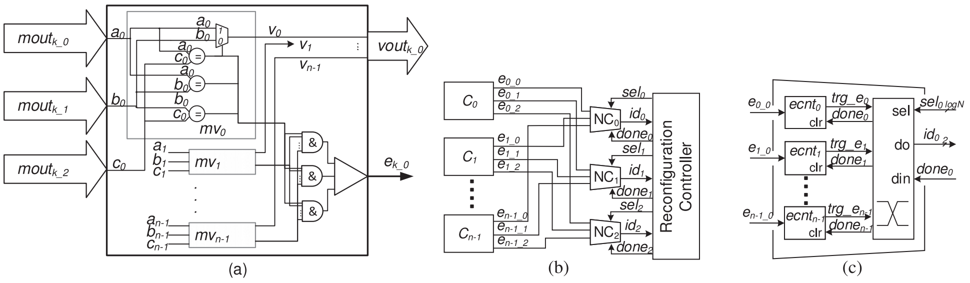


Fig. 2. Block diagrams of (a) voter V_{k_0}, (b) point-to-point RCN, and (c) elaborated NC$_0$ logic.

modules themselves (indicated as $M_{k\_0,1,2}$ in Figure 1) are placed and routed in reconfigurable regions, or *pblocks*, as Xilinx refers to them. This approach leaves the voters and interconnecting nets to be recovered by other means—perhaps via selective scrubbing, as proposed in Reference [1], or via fine-grained modular reconfiguration, as studied in this article. A second partitioning approach creates three pblocks for each TMR component with each pblock including the circuit module, its corresponding voter, and some of the interconnecting nets of the component, as illustrated by the grey box surrounding M_{k_0} and V_{k_0}. This approach allows both the 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 partitioning alternative includes all of the sub-components of a triplicated component in one pblock, as illustrated by the dashed box surrounding C_k. The three circuit modules of the TMR component, its nets, and the voters, but not the voter outputs, which must cross component boundaries, are then recovered by reconfiguring the one pblock. However, recovery in this case takes 3 times as long as for the second approach.

The voter blocks in our model, as depicted in Figure 2(a), are enhanced, as suggested in Reference [8], to identify the module whose output differs from the majority. The voters therefore not only protect the output of the user circuit but also detect 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 $i, j \in \{0, 1, 2\}$.

The error reports from the triplicated voter blocks are aggregated by a central RC through three identical point-to-point RCNs [2]. The RC thus receives triplicated error reports from each component. Figure 2(b) illustrates the triplicated RCNs. The RC manages the triplicated Network Controllers (NCs) to check the error state of a particular component. The RCNs are synchronous, and all NCs operate in lock step and check the voter of each component sequentially.

The NC aggregates error signals and distinguishes among transient errors in the user circuit and "permanent" errors that are likely due to configuration memory corruption. Figure 2(c) 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 [6]. 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 counter.

3.2 Persistent Errors

Permanent errors reported by the counters in the NCs may, as indicated, be 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.

Configuration memory errors in different parts of a component, as shown in Figures 1 and 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 whether 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}, then 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, then all three voters, or just two of them, may assert an error. Errors in the part 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 [3]. If an error is present in the switch matrix used by any two of $mout_{k\_0,1,2}$, then two voters may report different modules to be in error.
- If an error is present in one of the triplicated RCNs, then that RCN may raise or mask errors for a random module in some component.

It is evident that errors in the various sub-components of C_{k-1} and C_k or in RCN$_j$ result in different behaviours 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 error signatures are as follows: Type-I, three identical error reports ($id_{0,1,2} = M_{k\_j}$); Type-II, one error report ($id_i = M_{k\_j}$); or Type-III, all others, including three false-negative error reports and two or three (possibly differing) reports.

If the error signature is of Type-I and the reconfiguration of the module indicated by the RCNs fails to clear the error such that the same error signature is present after the recovery operation,

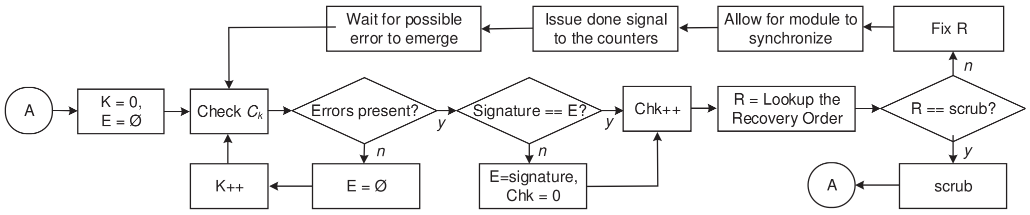

A
K = 0, E = Ø
Check C_k
Errors present?
y
n
Signature == E?
y
n
Chk++
R = Lookup the Recovery Order
R == scrub?
n
y
Fix R
Allow for module to synchronize
Issue done signal to the counters
Wait for possible error to emerge
K++
E = Ø
E=signature, Chk = 0
A
scrub

Fig. 3. Repair flow based on the number of consecutive observations of a particular error signature.

Table 1. Sequence in Which Sub-components Are Recovered

Error Signature		**Number of Checks (Chk)**					
		1	2	3	4	5	6
Type-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
Type-II	$id_i = M_{k\_j}$	V_{k_i}	$mout_k$	NC_i	e_{k_i}	scrub	
Type-III	others	$mout_k$	scrub				

then the error is deemed to be "persistent." It should then be suspected that the error may be present 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}) could have an error. For error signatures of Type-III, the only possible cause is an error in $mout_k$.

Other possible RCN outputs, such as a sequence of error reports that changes after each modular reconfiguration, are likely caused by an accumulation of SEUs or multi-bit upsets. Two or more errors may then be present in different sub-components. We do not consider these effects in this article, but all could be dealt with by triggering a complete scrub of the device when they appear.

3.3 Repair Strategy

We propose a repair strategy to guide the design of an RC that is capable of efficiently detecting and repairing persistent errors in systems using fine-grained dynamic reconfiguration. The repair strategy includes a repair flow and a recovery sequence as determined by the flow.

3.3.1 Repair Flow. Figure 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 1). After the dynamic reconfiguration of the sub-component has been performed, the RC waits for the component to be resynchronized [6] 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 on 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 [2, 6]. The RC checks whether C_k is still affected by errors, and if the same error signature is detected, Chk is incremented 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 A.

3.3.2 Recovery Sequence. Table 1 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 reconfiguring the block components first results in higher repair efficiency. For Type-I signatures, M_{k_j} is to be recovered if deemed persistent (Section 3.2) 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 Type-II error signatures, V_{k_i} and $mout_k$ are recovered first, followed by NC_i and the dedicated routing for e_{k_i}. For signatures of Type-III, it is only worthwhile recovering $mout_k$ before resorting to a complete scrub of the device.

3.3.3 Triggered Scrubbing. Systems that solely rely on scrubbing to correct configuration memory errors usually perform a scrub periodically as they do not usually monitor component 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 3.1, it is feasible to trigger a scrub cycle when a configuration error is detected. In this case, the error correction flow of Figure 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.

4 FINE-GRAINED DYNAMIC RECONFIGURATION

Conventional TMR-MER SoCs (e.g., References [4, 6, 16]) rely on the vendor's partial reconfiguration flow [20] to generate partial bitstreams for the component modules. These partial bitstreams are commonly stored in external memory with a lookup table being used by the RC to index them [6, 16]. When a permanent error is detected, the RC fetches the indexed file from memory and writes it to the Internal Configuration Access Port (ICAP). However, as indicated in Section 2, 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. In the architecture described in Section 3.1, the RCN and the interconnecting nets between voters and modules are not amenable to modular reconfiguration using the partial reconfiguration flow, since these nets are spatially distributed and cannot readily be contained within rectangular bounding boxes.

In this section, we propose a fine-grained dynamic partial reconfiguration approach (FDPR), which 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 method for identifying the sets of frames pertaining to the sub-components, such as the modules and their outputs, voters, voter outputs, RCN nets, and network controllers, and is applied after the design is floorplanned and implemented using the standard flow. We also describe a bitstream composition method that enables retrieving the desired sets of frame data from the full bitstream at runtime.

4.1 Major Columns in 7-Series FPGAs

We illustrate our approach using a Xilinx Artix-7 device, which is similar to other 7-Series Xilinx FPGAs, such as the Zynq All-Programmable SoCs. Programmable resources on a 7-Series device, such as Configurable Logic Blocks (CLBs), Digital Signal Processing slices (DSPs), BRAMs, and Input/Output Buffers (IOBs), are tiled into major columns [18, 20]. The major columns also include the switch matrices that provide access to the general routing matrix of the FPGA.

A simplified diagram of the layout of an Artix-7 device is shown in Figure 4. The chip comprises several configuration rows, which correspond to the clock regions of the device and are indexed in ascending order from the centre of the device towards both its top and bottom edges. Figure 4 depicts a section of the upper-left region of the device. The major columns span the configuration

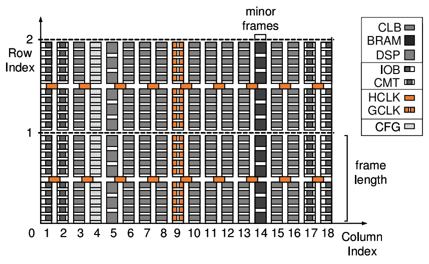


Fig. 4. Simplified Artix-7 layout showing two configuration rows and resources distributed across columns.

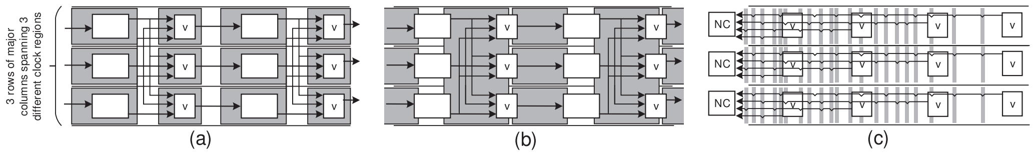


Fig. 5. One possible floorplan of the system architecture: (a) Logic, (b) Intra-component nets, and (c) RCN.

rows and are indexed in ascending order from left to right. A major column is configured via a contiguous set of configuration frames that are indexed by a minor address from left to right. The number of contiguous frames needed to configure a major column depends on the type of resources it contains. Clock signals are propagated from a central column of the device to local clock buffers in a clock region via a dedicated clock network; pairs of major columns share a local clock buffer.

FDPR can be applied to other Xilinx device families with appropriate adjustments for the frame size, the number of frames used by each type of resource column, the distribution of resources across the device, and the number of configuration rows. These concepts also translate to dynamically reconfigurable devices of other vendors with a similar configuration architecture.

4.2 Floorplanning

Floorplanning enables resource allocation within specified regions, known as pblocks, 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 a pblock. 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, configuration memory errors affecting these resources can be recovered by just reconfiguring the frames of that region.

Figure 5 illustrates one possible floorplan for the system architecture outlined in Section 3.1. The circuit spans three rows of major columns. Modules and voters are constrained to be placed in different major columns as shown in Figure 5(a). Figure 5(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. Our experiments show that voter outputs and module outputs are routed around block boundaries in the gaps between the closest module and voter. Figure 5(c) shows the major columns relating to the triplicated RCN. The NCs are also isolated and constrained to different rows of major columns. 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,

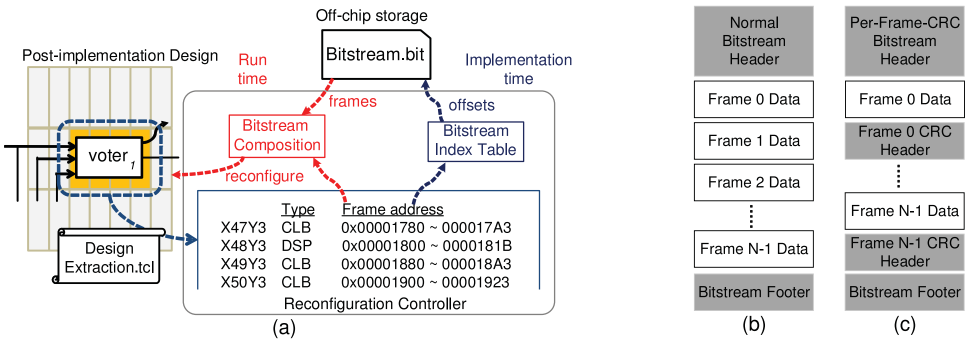


Fig. 6. (a) Bitstream composition involves extracting a bitstream index at implementation time and composing a partial bitstream at runtime. (b) Full bitstream without and (c) with CRC checking enabled.

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. While rare, it is possible for single SEUs to cause bridging between the nets of multiple sub-components if these components are not isolated from each other using a design method such as the Xilinx Isolation Design Flow [21] or RoRA [15]. An ICAP-based RCN [2], which eliminates the need for the designer to implement the RCN by relying on built-in FPGA infrastructure, can instead be used for this purpose.

4.3 Dynamic Repair Bitstream Composition

Bitstream composition involves extracting a frame offset index for the on-chip location of sub-components after the design has been implemented. The frame index is subsequently used at runtime to retrieve the configuration of a sub-component from off-chip storage when it is to be reconfigured, as shown in Figure 6(a).

4.3.1 Design Extraction. The detailed routing of nets that interconnect various 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 Point (PIP) the net uses, the switch matrices (SMs) 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 SMs used. Configuration memory errors in a net can thus also be recovered by simply reconfiguring the configuration frames of the major columns containing the switches it uses.

The Xilinx Vivado CAD tool is able to generate two types of a full bitstream without and with CRC per frame enabled. In the typical full bitstream file (see Figure 6(b)), the frame address starts from 0 and is automatically incremented at the end of each frame's data segment. On the other hand, a full bitstream for critical systems usually has single-frame CRC checking enabled (see Figure 6(c)), which includes frame headers that are inserted after every frame of data. The frame headers contain the CRC check sequence and the address of the previous frame.

Knowing the number of words per frame and per CRC header, we can extract frame addresses and index them according to their byte offset in the full bitstream file. Using this information together with the range of frame addresses spanned by a sub-component, we can extract the byte offsets of the frame data for each sub-component that needs to be reconfigured at runtime.

4.3.2 Dynamic Partial Reconfiguration. When the RC receives an error signature that indicates a sub-component is in error, it invokes a recovery procedure that follows the recovery sequence described in Section 3.3.2. The recovery procedure involves partially reconfiguring the sub-components indicated in the sequence and re-checking the error signature until either the error has been corrected or the device has been completely scrubbed.

The partial bitstream used to reconfigure a sub-component is dynamically formed and written to the ICAP at runtime. For logic block sub-components, which are generally formed from contiguous columns of configuration data, the RC starts by issuing, to the ICAP, header commands for a write operation, which include the number of words that are to be retrieved from off-chip memory. It then specifies the byte offset of the first frame before issuing a DMA transfer that will load the frame data from off-chip memory and write the data to the ICAP. Finally, the RC issues a pad frame and footer commands to end the write operation. For the net sub-components, for which it is more likely that the major columns that need to be reconfigured are scattered across the device, the RC needs to form the partial bitstream from several frame sets, with each set involving an indexed DMA operation into the full bitstream stored off-chip.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the recovery latency, the 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 in between the modules, with on-demand scrubbing of the complete device, which is triggered when SEUs are detected, and with periodic device scrubbing.

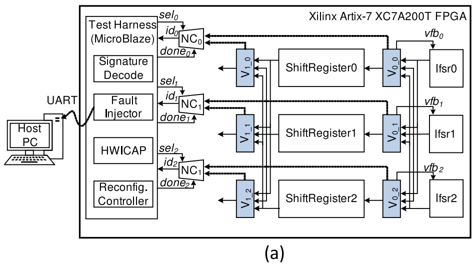
We have evaluated our proposal on three different types of circuits. In the following two subsections (5.1 to 5.2), we first describe our method, report results, and detail our findings on a small, hand-crafted two-component system [22]. We then more briefly report results and summarize our findings using the same methodology for three HLS benchmarks of small to medium complexity in Section 5.3 and on a larger hand-crafted satellite design in Section 5.4.

For each circuit, we report the utilization and recovery times of its logic block and net sub-components before presenting the results of fault-injection experiments. Fault injection was used to estimate the distribution of error signatures of Types-I to -III, and we would observe if the circuits were subjected to real SEUs. Given the observed error signature and knowledge of which sub-component was targeted by each injected fault, we were able to determine which sub-components would have been reconfigured, and in which order, according to our proposed recovery sequence. From these data, we were able to determine the error sensitivity of each sub-component and the average latency for each recovery strategy considered.

In TMR systems that employ configuration memory error recovery such as scrubbing or module-based error recovery, a TMR component fails when a second or third module suffers errors before the errors that have affected the first module are recovered. The rate at which errors are detected and recovered therefore affects the reliability of the system directly, especially in environments with high error rates. Given that we are only modifying the error recovery scheme, only the recovery rates differ. We therefore use MTTR to compare our proposed error recovery scheme with conventional on-demand scrubbing and module-based error recovery approaches; lower MTTR implies a higher recovery rate, which in turn implies higher reliability and availability.

5.1 Experimental Setup

5.1.1 Test Circuit. Figure 7(a) depicts a block diagram of the first test system implemented on a Nexys-4 Video board using Vivado 2015.4. This board includes a Xilinx Artix-7 XC7A200T device, which is clocked at 100MHz. The test circuit comprises two TMR components representing both cyclic state-machine logic and acyclic datapath logic [6]. To represent a typical state machine, we chose a 64-bit Linear-Feedback Shift Register (LFSR). The LFSR serves as a random test vector



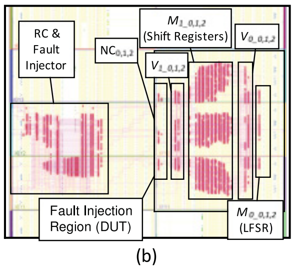


Fig. 7. (a) Artix-7 200T configuration with a MicroBlaze-based test harness and reconfiguration controller on the left, and the design under test, which includes all logic blocks and their interconnections, to its right. (b) Floorplan of the test harness and the test circuit in three clock regions on the right side of the device.

generator for the Design Under Test (DUT) and emulates the upstream voters and nets illustrated in Figure 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 SR comprised 8 stages of 64-bit registers with a variety of arithmetic operations mapped into the 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 DUT.

We floorplanned the test circuit according to the guidelines provided in Section 4.2. 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 4.3.

5.1.2 Test Harness. We used a MicroBlaze processor to implement the test harness, which included a fault injector, an error signature decoder, and an RC program running the FDPR of Section 4.3.

We measured the time to reconfigure each sub-component and used the same hardware setup to time a complete blind scrub of the device. The frames to be scrubbed were also extracted from the full bitstream as described in Section 4.3.

Figure 7(b) 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 100μs (10,000 clock cycles) 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 RC reported the sensitive bit location and the error signature to the host. Thereafter, the faulty bit was corrected by reversing

Table 2. Block Sub-component Area and Recovery Time

Sub-component	LUTs	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
$NC_{0,1,2}$	43	34	2	72	2.3

Table 3. Net Sub-component Area and 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

the injected bit flip and waiting another 100μs to allow the correct data to flush through the test circuit.

5.2 Results

5.2.1 Fine-grained Dynamic Reconfiguration of Sub-components. Tables 2 and 3 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, we found that our platform needed on average 32.7μs to transfer a frame from off-chip flash memory to the ICAP. The transfer time varies a little depending on whether contiguous or non-contiguous frames are retrieved, but this is a reasonably good result considering the constraints imposed by the board architecture (SPI flash) and the use of MicroBlaze and AXI-HWICAP. It should be noted that the design did not include DSPs or BRAMs.

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 432ms, which corresponds to a sustained transfer rate of one frame every 23.6μs. 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 bespoke scrubber could be expected to take considerably less than 432ms to scrub the device.

On the tested circuit, the worst-case repair time using our proposed repair strategy was 135ms, 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 one-third of the scrub cycle latency, which represents a substantial reduction in the repair time. At most,

Table 4. Fault Injection Results

Error Signature	Number of Checks (Chk)						Sub-component Reconfiguration Sequence
	1	2	3	4	5	6	
Type-I$_0$	4,882	959	N/A	627	N/A	0	$M_{0\_0,1,2} \rightarrow V_{0\_0,1,2} \rightarrow mout_0$
Type-II$_0$	15,862	6,623	159	44	0		$V_{0\_0,1,2} \rightarrow mout_0 \rightarrow \mathrm{NC}_{0,1,2} \rightarrow e_{0\_0,1,2}$
Type-III$_0$	3,656	0					$mout_0$
Type-I$_1$	633,901	4,788	2,038	748	23	0	$M_{1\_0,1,2} \rightarrow V_{0\_0,1,2} \rightarrow vout_{0\_0,1,2} \rightarrow mout_1 \rightarrow mout_0$
Type-II$_1$	18,248	7,282	115	0	0		$V_{1\_0,1,2} \rightarrow mout_1 \rightarrow \mathrm{NC}_{0,1,2}$
Type-III$_1$	4,364	0					$mout_1$

we found we had to reconfigure 4,112 frames, which is less than one-fourth of those needed to perform a scrub.

5.2.2 Fault Injection Results. Table 4 reports on the fault-injection experiment. The subscripts indicate the error signatures for the LFSR and SR components (0 and 1), respectively. The table reports for each error signature (Type-I$_0$—Type-III$_1$) how many reconfigurations of the sub-components were triggered (Number of Checks) according to our repair strategy for the error reports received. The table lists the order in which sub-components would have been reconfigured to clear errors.

As can be seen from Table 4, 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, 680,913 errors were reported for 16,134,144 fault injections. For error signatures of Type-I, the number of reports for the LFSR and SR modules differ greatly, because their utilizations differ substantially (Table 2). For error signatures of Type-II and Type-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 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 and take more time to reconfigure (Table 3).

5.2.3 Error Sensitivity and Recovery Time. Table 5 summarizes the number of critical bits we found in the test circuit via the fault-injection experiment and the average recovery times for these sub-components. We have compared the recovery time for sub-components using the fine-grained DPR approach to MER proposed in this article (MER/FDPR) with, on the one hand, a more coarse-grained approach that makes use of scrubbing to recover from errors that occur outside the reconfigurable modules (MER/Scrub) [2] and, on the other hand, by scrubbing whenever any error is detected in the system (Triggered Scrub).

We determined the number of critical bits found in each sub-component from the number of errors that were reported for each check (Chk) as listed in Table 4. For example, since 4,882 error reports were made at Chk = 1 for signature Type-I$_0$, but only 959 were recorded for Chk = 2, we deduce that 4,882 - 959 = 3,923 reconfigurations of component C_0, i.e., one of the modules $M_{0\_0,1,2}$, were needed to clear errors affecting this sub-component.

The average recovery times using MER/FDPR were obtained by weighting the time to recover all components in each reconfiguration sequence by the number of error occurrences for each

Table 5. Sub-component Critical Bits Summary Using Proposed Recovery Sequence

Sub-component	Critical Bits Found	Average Recovery Latency (ms)		
		MER/FDPR	MER/Scrub	Triggered Scrub
$M_{0\_0,1,2}$	3,923	2.0	2.0	216.0
$mout_{0\_0,1,2}$	10,770	41.5	216.2	216.0
$V_{0\_0,1,2}$	12,321	7.4	221.1	216.0
$vout_{0\_0,1,2}$	1,290	39.0	238.6	216.0
$e_{0\_0,1,2}$	44	61.2	216.0	216.0
$M_{1\_0,1,2}$	629,113	22.6	22.6	216.0
$mout_{1\_0,1,2}$	12,256	59.5	217.3	216.0
$V_{1\_0,1,2}$	10,966	2.3	216.0	216.0
$\mathrm{NC}_{0,1,2}$	230	52.4	216.0	216.0

Table 6. Frames, MTTR, and Energy Results

	MER/FDPR	MER/Scrub	Triggered Scrub
Frames	718	1,963	18,300
MTTR (ms)	23	36	216
Energy (mJ)	0.38	1.05	9.79

signature. For example, while sub-component e_0 was only reconfigured 44 times in total, sub-components V_0, $mout_0$, and $\mathrm{NC}_{0,1,2}$ were reconfigured before e_0 for a total reconfiguration time of 61.2 ms/event, as can be calculated from Tables 2 and 3. Similarly, sub-component $mout_0$ was reconfigured 3,656 times for a recovery time of 39.7ms/event as a result of error signatures of Type-III$_0$, but it was also reconfigured 627 times for a recovery time of 44ms/event due to signatures of Type-I$_0$, 6,464 times for 42ms/event due to signatures of Type-II$_0$, and 23 times for 134.5ms/event due to signatures of Type-I$_1$, resulting in an average recovery time of 41.5ms for $mout_0$.

The average recovery time using MER/Scrub is also determined by weighting individual recovery times by the number of occurrences. Sub-components $M_{0\_0,1,2}$ and $M_{1\_0,1,2}$ are recovered by MER when signatures of Type-I$_0$ and Type-I$_1$ are detected. However, all other sub-components in the recovery sequences for error signatures of Type-I are recovered by scrubbing after the modules $M_{0\_0,1,2}$ and $M_{1\_0,1,2}$ are reconfigured in a first attempt to clear the error. For error signatures of Type-II and Type-III, all errors in the sub-modules are recovered by scrubbing. We adopt the convention that determines that the average time to recover from an error via scrubbing is halfway through the scrub. Thus, since a scrub cycle takes 432ms on our system, the average time to recover from an error via scrubbing is 216ms. Triggered scrubs occur whenever an error is detected in our system using the voters, RCNs and NCs.

Table 6 compares the average number of frames reconfigured per error, the average MTTR, and the energy expended to repair the error assuming each frame write consumes 535nJ on average [17]. While triggered scrubbing needs to scrub 18,300 frames, MER/FDPR only needs to reconfigure 718 frames on average (weighted). While these results are application and device dependent, they are representative of the gains that are possible. In this experiment, the weighted average MTTR for MER/FDPR was only 10% of that for scrubbing. Of the methods we studied, MER/FDPR is therefore also the most energy efficient. Since energy consumed recovering from configuration memory errors is assumed to be proportional to the number of frames that are rewritten, in this 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

Table 7. TLegUp Block Utilization and Recovery Time

Design	MMULT						SATD						GSM					
Sub-Logic	LUTs	FFs	DSPs	BRAMs	Frames	Recovery Time (μs)	LUTs	FFs	DSPs	BRAMs	Frames	Recovery Time (μs)	LUTs	FFs	DSPs	BRAMs	Frames	Recovery Time (μs)
$M_{0\_0,1,2}$	500	352	3	2	200	754	2,346	2,404	0	0	416	1,564	3,784	2,976	37	2	688	2,588
$V_{0\_0,1,2}$	239	2	0	0	72	270	821	2	0	0	136	512	994	2	0	0	144	540
$M_{1\_0,1,2}$	41	71	0	0	72	270	1,300	569	0	0	208	782	1,851	1,321	14	2	482	1,812
$V_{1\_0,1,2}$	175	2	0	0	64	242	84	2	0	0	72	270	426	2	0	0	72	270
$NC_{0,1,2}$	27	22	0	0	72	270	27	22	0	0	128	484	27	22	0	0	64	242

energy if the period between scrubs is small enough to result in needless scrubs, or, alternatively, it will compromise MTTR if the period between scrubs is increased to avoid unnecessary scrubs. With all other factors being equal, the reliability of FPGA-based TMR systems depends only on the MTTR from configuration memory errors [2, 11]. We therefore deduce that systems employing MER/FDPR have higher reliability than those relying on MER/Scrub or triggered scrubbing [22]. However, this assessment assumes that all circuits have the same size; in particular, we assume that the same voter type, RCN, and RC are used to trigger error recovery and/or report on affected components.

5.3 TLegUp Experiments

The LFSR/SR experiment is hand-crafted and relatively small in scale. The TLegUp experiments were developed to find out whether the methods described in this article apply to larger circuits and, in particular, to circuits that are not hand-crafted.

TLegUp [10] extends the well-known LegUp HLS framework [5] to enable the automatic synthesis and implementation of TMR-protected circuits from C-language specifications. Designers specify the behaviour of the circuit they wish to have implemented as an algorithm written in C, which is then transformed by TLegUp into a partitioned and triplicated Register Transfer Level hardware description in Verilog with feed-forward voters inserted at the partition (component) boundaries. The resulting circuit description can then be implemented using Xilinx CAD tools.

We targeted three small- to medium-scale HLS benchmarks, *mmult*, *satd*, and *gsm* [12], and implemented these on a Xilinx Zynq 7020 device hosted on a Digilent ZedBoard with Vivado 2016.1. The small-scale Bambu HLS benchmark *mmult* multiplies two 32-bit integer square matrices of order 20, while the medium-scale DWARV benchmark *satd* finds the distance between pixels in a pair of video frames, and CHStone's larger *gsm* benchmark includes a linear predictive encoder to compress audio signals.

TLegUp was configured to partition all benchmarks into two TMR components to obtain a system architecture similar to that described in Section 5.1 and because limitations with TLegUp at the time of experimentation prevented us from exploring higher degrees of partitioning. The main circuit modules were manually floorplanned before placement and routing. The utilization of the modules and their sizes in configuration frames and the time to recover each module using FDPR are recorded in Tables 7 and 8.

To compare the results of the TLegUp benchmarks with each other and with those of the LFSR/SR experiment, we have used a standard naming convention: $M_{0\_0,1,2}$ and $M_{1\_0,1,2}$ refer to the triplicated modules of components C_0 and C_1; $V_{0\_0,1,2}$ and $V_{1\_0,1,2}$ refer to the voters on each component; $NC_{0,1,2}$ refers to the network controllers, here arbitrating between point-to-point RCN nets denoted $e_{0\_0,1,2}$ and $e_{1\_0,1,2}$ coming from the corresponding voters; module outputs $mout_0$ and $mout_1$ connecting to the voters; and inter-component voter outputs $vout_{0\_0,1,2}$ and $vout_{1\_0,1,2}$. Note

Table 8. TLegUp Net Utilization and Recovery Time

Design	**MMULT**				**SATD**				**GSM**			
Sub-Net	**PIPs**	**SMs**	**Frames**	**Recovery Time (μs)**	**PIPs**	**SMs**	**Frames**	**Recovery Time (μs)**	**PIPs**	**SMs**	**Frames**	**Recovery Time (μs)**
$vout_{0\_0,1,2}$ (*)	1,374	157	417	1,572	3,549	382	798	2,999	5,213	831	1,291	4,848
$mout_0$	21,828	871	1,332	5,013	93,467	2,723	2,958	11,124	113,325	4,437	4,192	15,755
$e_{0\_0,1,2}$ (*)	35	13	364	1,367	32	12	315	1,182	30	9	273	1,026
$vout_{1\_0,1,2}$ (*)	397	95	360	1,356	14	8	175	656	904	277	1,142	4,292
$mout_1$	8,062	761	1,496	5,632	6,546	484	1,930	7,247	24,141	2,158	4,200	15,783
$e_{1\_0,1,2}$ (*)	32	16	412	1,549	28	9	233	874	29	15	386	1,450

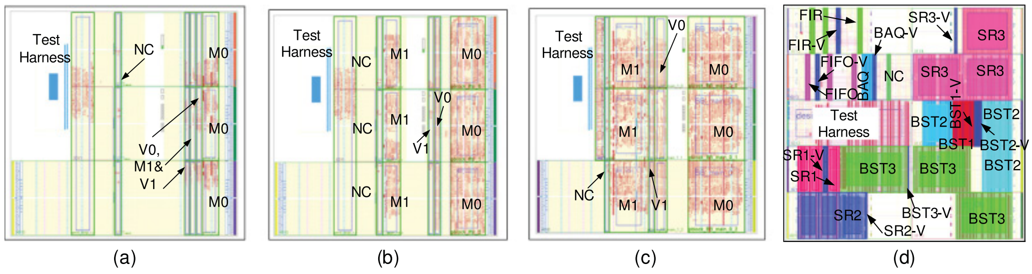


Fig. 8. Floorplans of (a) *satd*, (b) *mmult*, (c) *gsm*, and (d) RUSH designs.

that the results for nets labelled with (*) have been averaged over the three copies of the triplicated net.

We observe from Table 7 that the TLegUp circuits have considerably larger voter blocks than the small, hand-crafted LFSR/SR circuit, since many more edges in the data-flow graphs of the TLegUp circuits are cut when they are partitioned into two components. Consequently, the $mout_0$ and $mout_1$ nets are considerably more congested and their recovery times are considerably higher.

Table 9 reports the result of injecting faults into every configuration bit of the design under test, which spans 4,182 frames on the Zynq 7020 device for *mmult* and *satd* and 5,238 frames for *gsm* (see Figure 8(a)–(c) for an illustration of the circuit layouts). We report the counts (Chk) for each error signature type emanating from both components in each design. The table first shows the sequence in which modules are recovered to eliminate the error. Thereafter, as for Table 4, we record the number of times each error has occurred.

Similarly to Section 5.2.3, we derived Table 10 from Table 9. Table 10 records the number of critical bits we found in each module of the circuit and the average time to recover from errors in these sub-components using MER/FDPR and MER/Scrub. The average recovery time using Triggered scrubbing is 12,656μs (half the scrub period) in every case.

We observed that the TLegUp circuits are less sensitive to errors (at most 2/3×) than the small, hand-crafted LFSR/SR circuit despite *gsm* having up to 2.4× greater utilization. However, we recorded several unrecognized errors, which had to be dealt with by scrubbing as per our repair strategy, and believe that these events are likely due to common mode failures in shared interconnection resources [21]. The cost of recovering from these errors is substantial: In *gsm* the recovery latency is as high as that of two scrub cycles. We note that the occurrence of Type-I errors appears to be related to logic utilization, whereas Type-II and Type-III events are due to voter size and routing complexity. In these circuits, the $mout_0$ and $mout_1$ components contribute markedly to the sensitivities and recovery latencies of the circuits. It is possible that reconfiguration of these

Table 9. TLegUp Error Injection Results

Design	Error Signature	Number of Checks (Chk)						
		1	2	3	4	5	6	7
Recovery Sequence	Type-I$_0$	$M_{0\_0,1,2}$	$V_{0\_0,1,2}$	$V_{1\_0,1,2}$	$vout_{1\_0,1,2}$	$mout_0$	$mout_1$	Scrub
	Type-II$_0$	$V_{0\_0,1,2}$	$mout_0$	NC$_{0,1,2}$	$e_{0\_0,1,2}$	Scrub		
	Type-III$_0$	$mout_0$	Scrub					
	Type-I$_1$	$M_{1\_0,1,2}$	$V_{1\_0,1,2}$	$V_{0\_0,1,2}$	$vout_{0\_0,1,2}$	$mout_1$	$mout_0$	Scrub
	Type-II$_1$	$V_{1\_0,1,2}$	$mout_1$	NC$_{0,1,2}$	$e_{1\_0,1,2}$	Scrub		
	Type-III$_1$	$mout_1$	Scrub					
MMULT	Type-I$_0$	7,923	1,038	884	498	282	15	0
	Type-II$_0$	14,965	7,595	103	23	1		
	Type-III$_0$	3,775	0					
	Type-I$_1$	5,322	1,275	1,225	383	6	0	0
	Type-II$_1$	9,135	5,080	82	6	0		
	Type-III$_1$	1,519	0					
SATD	Type-I$_0$	35,117	3,402	1,907	1,708	1,158	28	28
	Type-II$_0$	112,132	62,955	115	0	0		
	Type-III$_0$	30,615	0					
	Type-I$_1$	1,560	270	262	253	119	102	69
	Type-II$_1$	1,987	963	74	7	0		
	Type-III$_1$	329	2					
GSM	Type-I$_0$	243,075	55,782	44,921	37,358	20,352	118	10
	Type-II$_0$	133,025	57,669	0	0	0		
	Type-III$_0$	28,890	0					
	Type-I$_1$	59,001	3,315	2,166	1,452	287	58	51
	Type-II$_1$	27,887	15,768	0	0	0		
	Type-III$_1$	5,714	0					

Table 10. TLegUp Sensitivity and Recovery Time

Design	MMULT			SATD			GSM		
Sub-component	**Crit. Bits**	**Avg. Recovery Time (μs)**		**Crit. Bits**	**Avg. Recovery Time (μs)**		**Crit. Bits**	**Avg. Recovery Time (μs)**	
Recovery Method		**MER/ FDPR**	**MER/ Scrub**		**MER/ FDPR**	**MER/ Scrub**		**MER/ FDPR**	**MER/ Scrub**
$M_{0\_0,1,2}$	6,885	754	754	31,715	1,564	1,564	187,293	2,588	2,588
$mout_0$	11,534	5,248	12,673	94,618	11,504	12,675	106,800	17,517	13,146
$V_{0\_0,1,2}$	8,366	335	12,697	50,681	558	12,702	86,931	880	12,994
$vout_{0\_0,1,2}$	377	2,374	12,926	134	4,646	13,438	1,165	7,363	14,468
$e_{0\_0,1,2}$	22	6,954	12,656	0	N/A	N/A	0	N/A	N/A
$M_{1\_0,1,2}$	4,047	270	270	1,290	782	782	55,686	1,812	1,812
$mout_1$	6,538	5,837	12,658	1,233	7,503	12,667	21,819	16,171	12,688
$V_{1\_0,1,2}$	4,491	333	12,724	1,231	611	12,914	20,831	1,506	13,696
$vout_{1\_0,1,2}$	216	2,605	13,410	550	3,071	14,220	17,006	7,608	15,244
$e_{1\_0,1,2}$	6	7,555	12,656	7	8,738	12,656	0	N/A	N/A
NC$_{0,1,2}$	156	5,841	12,656	182	10,604	12,656	0	N/A	N/A
Unrecognized	1	19,368	12,656	99	34,892	13,643	61	51,297	14,595
Total	42,639			181,740			497,592		

Table 11. TLegUp Recovery Time and Energy

Design	MMULT			SATD			GSM		
Recovery Method	**MER/ FDPR**	**MER/ Scrub**	**Triggered Scrub**	**MER/ FDPR**	**MER/ Scrub**	**Triggered Scrub**	**MER/ FDPR**	**MER/ Scrub**	**Triggered Scrub**
Frames	697	5,766	7,692	1,736	6,377	7,692	1,636	4,329	7,692
MTTR (μs)	2,624	9,585	12,656	6,521	10,667	12,656	6,146	7,955	12,656
Energy (uJ)	373	3,085	4,115	929	3,412	4,115	875	2,316	4,115

sub-components may have masked errors that actually occurred in the RCN nets and the NCs whose resource usage overlapped with major columns of the *mout* nets.

As in Section 5.2.3, Table 11 summarizes the average number of frames that were reconfigured to correct a configuration memory error, the MTTR from configuration errors, and the reconfiguration energy using MER/FDPR, MER/Scrub, and Triggered scrubbing.

Across the TLegUp circuits, we observed that MER/FDPR has better recovery latency and lower energy cost than MER/Scrub and either triggered or periodic scrubbing. However, MER/FDPR presents implementation difficulties, since certain elements of the design, such as voter inputs and outputs, need to be manually identified, and thus the firmware for the RC cannot be automatically generated. We note that some major columns of a design may be overwritten multiple times to recover the system from some rare errors—inefficiencies therefore remain in the approach. This leads us to consider the potential to benefit from modifying our approach—to reduce the high costs of recovering relatively insensitive components such as RCN nets and voter outputs at the ends of our repair sequences, especially when the cost to recover dense *mout* sub-components is high, we should scrub at an earlier point in the sequence, perhaps after the sub-component blocks have been reconfigured, to cut off the long tail in our average recovery latencies.

5.4 RUSH Experiment

The LFSR/SR and TLegUp experiments involved small- to medium-scale designs comprising two components each. To test the MER/FDPR approach on a larger circuit, we also experimented with applying the approach to recover from errors in a nine-component system implemented on a Xilinx Artix-7 XC7A200T FPGA.

The test circuit (see Figure 8(d)) is one of two configurations deployed on the RUSH platform [2, 7], which we developed to test the reliability of SRAM-based FPGA circuits in the thermosphere as part of the European Commission's QB50 Project [14]. The configuration consists of nine components that are representative of digital sub-systems for satellites. These include a 21-tap 16-bit iterative (single MAC) Finite Impulse Response (FIR) filter; an 8-to-3-bit Block Adaptive Quantizer (BAQ); an 8,096-word deep 32-bit FIFO; three 32-bit Shift Registers (SR1-3), having different lengths and a variety of combinational functions between each stage; and three 32-bit Binary Search Trees (BST1-3) of different heights, which include a variety of combinational functions at each node. There are no dependencies between the components; thus each component acts as C_0, comprising modules $M_{0\_0,1,2}$, and is directly connected via its voters $V_{0\_0,1,2}$ and error nets $e_{0\_0,1,2}$ to a network controller $NC_{0,1,2}$ and a MicroBlaze processor acting as the reconfiguration controller. The system was manually floorplanned and implemented using Vivado 2016.1.

Tables 12 and 13 report the resource utilization, frame occupancy, and partial reconfiguration times (recovery times) for the RUSH circuit modules and nets, which include the connections ($mout_0$) between the components and their voters. As before, nets labelled with (*) report average utilizations over three individual nets. Note that the routing complexity of the RUSH *mout* nets is far lower than that of the TLegUp circuits, since only necessary voters are inserted into a hand-partitioned circuit. The data widths being checked are smaller as well.

Table 12. RUSH Block Utilization and Recovery Time

Comp	Sub-Logic	LUTs	FFs	DSPs	BRAMs	Frames	Recovery Time (μs)
FIR	$M_{0\_0,1,2}$	29	22	1	0	64	1,009
	$V_{0\_0,1,2}$	498	213	0	0	144	2,244
FIFO	$M_{0\_0,1,2}$	64	71	0	8	64	1,009
	$V_{0\_0,1,2}$	231	102	0	0	144	2,244
BAQ	$M_{0\_0,1,2}$	196	196	0	2	72	1,122
	$V_{0\_0,1,2}$	333	144	0	0	72	1,122
SR3	$M_{0\_0,1,2}$	5,318	10,626	31	0	796	12,451
	$V_{0\_0,1,2}$	48	30	0	0	72	1,122
BST2	$M_{0\_0,1,2}$	3,767	6,189	31	0	480	7,515
	$V_{0\_0,1,2}$	48	30	0	0	64	1,009
BST1	$M_{0\_0,1,2}$	1,384	2,510	0	0	144	2,244
	$V_{0\_0,1,2}$	48	30	0	0	72	1,122
BST3	$M_{0\_0,1,2}$	9,117	12,205	31	0	1,228	19,183
	$V_{0\_0,1,2}$	48	30	0	0	66	1,070
SR2	$M_{0\_0,1,2}$	2,632	5,280	10	0	208	3,253
	$V_{0\_0,1,2}$	48	30	0	0	64	1,009
SR1	$M_{0\_0,1,2}$	1,608	3,264	0	0	410	6,454
	$V_{0\_0,1,2}$	48	30	0	0	72	1,122
RCN	$\mathrm{NC}_{0,1,2}$	104	92	0	0	72	1,122

Table 13. RUSH Net Utilization and Recovery Time

Comp	Sub-Net	PIPs	SMs	Frames	Recovery Time (μs)
FIR	$mout_0$	8,484	596	752	11,777
	$e_{0\_0,1,2}$ (*)	43	18	443	6,918
FIFO	$mout_0$	3,862	301	602	9,481
	$e_{0\_0,1,2}$ (*)	36	15	377	5,907
BAQ	$mout_0$	6,492	510	352	5,497
	$e_{0\_0,1,2}$ (*)	28	8	144	2,244
SR3	$mout_0$	621	107	936	14,586
	$e_{0\_0,1,2}$ (*)	41	25	699	10,958
BST2	$mout_0$	598	103	904	14,134
	$e_{0\_0,1,2}$ (*)	53	33	812	12,716
BST1	$mout_0$	527	59	360	5,610
	$e_{0\_0,1,2}$ (*)	54	33	885	13,835
BST3	$mout_0$	728	249	1,566	24,515
	$e_{0\_0,1,2}$ (*)	39	20	292	4,562
SR2	$mout_0$	545	53	228	3,588
	$e_{0\_0,1,2}$ (*)	54	36	704	11,029
SR1	$mout_0$	656	103	552	8,637
	$e_{0\_0,1,2}$ (*)	53	35	503	7,853

Table 14. RUSH Error Injection Results

Component	Error Signature	Number of Checks (Chk) 1	2	3	4	5
Recovery Sequence	Type-I Type-II Type-III	$M_{0\_0,1,2}$ $V_{0\_0,1,2}$ $mout_0$	$V_{0\_0,1,2}$ $mout_0$ Scrub	$mout_0$ $NC_{0,1,2}$	Scrub $e_{0\_0,1,2}$	Scrub
FIR	Type-I Type-II Type-III	646 287 29	77 58 2	46 37	25 34	33
FIFO	Type-I Type-II Type-III	332 80 27	80 5 0	71 2	52 0	0
BAQ	Type-I Type-II Type-III	1,205 156 7	24 36 0	6 2	2 1	0
SR3	Type-I Type-II Type-III	19,462 17 7	33 4 1	33 4	18 1	0
BST2	Type-I Type-II Type-III	12,719 16 3	109 4 0	107 2	89 0	0
BST1	Type-I Type-II Type-III	5,000 18 0	75 11 0	73 1	67 1	1
BST3	Type-I Type-II Type-III	41,663 14 6	91 0 0	89 0	20 0	0
SR2	Type-I Type-II Type-III	9,277 21 2	16 6 0	15 6	4 3	1
SR1	Type-I Type-II Type-III	5,406 25 6	8 10 1	7 8	5 4	0

Table 14 reports the result of injecting 1,000,000 random errors into the RUSH design under test, which spans 17,330 frames of the Artix-7 200T device. Random testing was undertaken due to the expected two-month turnaround time needed to perform exhaustive fault-injection testing on the RUSH platform. Like in Table 9, we report the counts (Chk) for each error signature emanating from each component of the design. We first indicate the sequence in which modules are recovered to eliminate the error and then record the number of times each error occurred.

We derived Table 15 from Table 14, as previously explained. The table records the number of critical bits we found in each sub-component of the circuit and the average time to recover from errors using MER/FDPR, MER/Scrub, and Triggered scrubbing. Entries labelled with (**) report the total number of errors found for the net and the average recovery latency for each error.

We observed less than 10% critical bits in response to 1,000,000 randomly injected configuration errors, which is consistent with Xilinx's estimates of the distribution of critical bits within a

Table 15. RUSH Sensitivity and Recovery Time

Comp	Sub-Comp	Critical Bits	Average Latency (μs)		
			MER/ FDPR	MER/ Scrub	Triggered Scrub
FIR	$M_{0\_0,1,2}$	569	1,009	1,009	131,357
	$V_{0\_0,1,2}$	260	2,364	131,477	131,357
	$mout_0$	69	13,450	131,664	131,357
	$e_{0\_0,1,2}$ (**)	1	23,332	131,357	131,357
FIFO	$M_{0\_0,1,2}$	252	1,009	1,009	131,357
	$V_{0\_0,1,2}$	84	2,352	131,465	131,357
	$mout_0$	49	8,931	131,748	131,357
	$e_{0\_0,1,2}$ (**)	0	0	0	0
BAQ	$M_{0\_0,1,2}$	1,181	1,122	1,122	131,357
	$V_{0\_0,1,2}$	138	1,268	131,503	131,357
	$mout_0$	45	6,544	131,457	131,357
	$e_{0\_0,1,2}$ (**)	1	9,985	131,357	131,357
SR3	$M_{0\_0,1,2}$	19,429	12,451	12,451	131,357
	$V_{0\_0,1,2}$	13	1,122	131,357	131,357
	$mout_0$	21	24,281	140,251	131,357
	$e_{0\_0,1,2}$ (**)	1	29,168	131,357	131,357
BST2	$M_{0\_0,1,2}$	12,610	7,515	7,515	131,357
	$V_{0\_0,1,2}$	14	2,083	132,431	131,357
	$mout_0$	23	20,893	137,238	131,357
	$e_{0\_0,1,2}$ (**)	0	0	0	0
BST1	$M_{0\_0,1,2}$	4,925	2,244	2,244	131,357
	$V_{0\_0,1,2}$	9	1,621	131,856	131,357
	$mout_0$	16	7,574	132,199	131,357
	$e_{0\_0,1,2}$ (**)	0	0	0	0
BST3	$M_{0\_0,1,2}$	41,572	19,183	19,183	131,357
	$V_{0\_0,1,2}$	16	3,468	133,755	131,357
	$mout_0$	75	43,148	149,005	131,357
	$e_{0\_0,1,2}$ (**)	0	0	0	0
SR2	$M_{0\_0,1,2}$	9,261	3,253	3,253	131,357
	$V_{0\_0,1,2}$	16	1,212	131,560	131,357
	$mout_0$	13	7,194	134,110	131,357
	$e_{0\_0,1,2}$ (**)	2	14,691	131,357	131,357
SR1	$M_{0\_0,1,2}$	5,398	6,454	6,454	131,357
	$V_{0\_0,1,2}$	16	1,525	131,760	131,357
	$mout_0$	9	10,570	132,791	131,357
	$e_{0\_0,1,2}$ (**)	4	20,192	131,357	131,357
Others	$\mathrm{NC}_{0,1,2}$	18	12,365	131,357	131,357
	Unrecognized	321	148,230	134,556	131,357
Total		96,431			

bitstream. We again observed some (0.3%) unrecognized errors and suspect these are caused by errors in the input vector logic or the global control logic. In contrast to the TLegUp results, the cost of recovering from unrecognized errors in the RUSH circuit is not excessive: on average, just 13% more than for triggered scrubbing. Not surprisingly, Type-I errors are most prevalent due to the majority of frames being dedicated to logic block components.

Table 16. RUSH Recovery Time and Energy

	MER/FDPR	MER/Scrub	Triggered Scrub
Frames	874	1,041	18,300
MTTR (μs)	13,586	15,969	131,357
Energy (uJ)	468	557	9,791

Table 16 summarizes the average number of frames that were reconfigured to correct a configuration memory error, the MTTR from configuration errors, and the reconfiguration energy using MER/FDPR, MER/Scrub, and Triggered scrubbing.

For RUSH, a larger, albeit hand-crafted design, we found that MER/FDPR also outperforms MER/Scrub and scrubbing in terms of recovery latency and recovery energy. The MTTR of MER/FDPR was found to be 14.9% lower than for MER/Scrub and 89.7% lower than for Triggered Scrubbing. The energy devoted to recovering from configuration memory errors in MER/FDPR was 16.0% lower than for MER/Scrub and 95.2% lower than for Triggered Scrubbing. We can expect even greater margins were MER/FDPR pitted against periodic scrubbing.

6 CONCLUSION

We have proposed a fine-grained module-based error detection and dynamic reconfiguration scheme that can detect and recover from configuration memory errors in SRAM FPGA-based TMR circuits more rapidly, using less energy, and more reliably than scrubbing-based methods.

We evaluated our method by injecting faults into three types of circuits: a small hand-crafted system, three HLS benchmarks, and a large hand-crafted design. The results consistently demonstrate superior recovery times and reduced energy cost to recover from errors with our approach. We recorded reductions in MTTR with respect to triggered scrubbing of between 48.5% for a medium-scale HLS circuit and 89.4% for a small, hand-crafted circuit. Reductions in reconfiguration energy with respect to triggered scrubbing ranged between 77.4 and 96.1% on the same circuits, respectively. More significant reductions compared with periodic scrubbing can be expected. These results are of importance in high-radiation environments, such as in GEO satellites, where designers are increasingly wanting to utilize SRAM FPGAs in fault-tolerant and time-critical applications.

Our method allows systems to recover from configuration errors that occur in the nets between TMR components by reconfiguring just those frames that include the critical bits of the affected nets. In so doing, the method only rarely relies on a scrub cycle to recover from those errors, which traditional module-based recovery schemes cannot access. Our method does not require any additional hardware components or partial bitstreams. Instead, it relies on post-implementation techniques to identify the addresses, within the full bitstream, of the configuration frames that relate to the logic blocks and inter-block nets of the design. These addresses are used at runtime to dynamically compose partial bitstreams to reconfigure the sub-components in a predetermined order according to the error signature presented by the TMR voters.

Our analysis has also found opportunities to further tune the recovery sequence to optimize the recovery latency and energy cost. We intend to explore this problem further in the future. We also plan to investigate automated approaches to extracting, from a full bitstream, the sub-component frame information given information about the design hierarchy. While our methods, as presented, will successfully cope with some accumulated and multi-bit errors, they would need to be modified to guarantee all such errors are detected and corrected. Assessing our method with accumulated errors and adjusting for these is thus of interest to us, as is testing our approach using radiation-beam experiments.

REFERENCES

[1] Dimitris Agiakatsikas, Ediz Cetin, and Oliver Diessel. 2016. FMER: A hybrid configuration memory error recovery scheme for highly reliable FPGA SoCs. In *FPL*. 1–4.

[2] Dimitris Agiakatsikas, Nguyen T. H. Nguyen, Zhuoran Zhao, Tong Wu, Ediz Cetin, Oliver Diessel, and Lingkan Gong. 2016. Reconfiguration control networks for TMR systems with module-based recovery. In *FCCM*. 88–91.

[3] Ghazanfar Asadi and Mehdi B. Tahoori. 2005. Soft error rate estimation and mitigation for SRAM-based FPGAs. In *FPGA*. 149–160.

[4] Cristiana Bolchini, Antonio Miele, and Chiara Sandionigi. 2011. A novel design methodology for implementing reliability-aware systems on SRAM-based FPGAs. *IEEE Trans. Comput.* 60, 12 (2011), 1744–1758.

[5] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *FPGA*. 33–36.

[6] Ediz Cetin, Oliver Diessel, Lingkan Gong, and Victor Lai. 2013. Towards bounded error recovery time in FPGA-based TMR circuits using dynamic partial reconfiguration. In *FPL*. 1–4.

[7] Ediz Cetin, Oliver Diessel, Tuo Li, Jude A. Ambrose, Thomas Fisk, Sri Parameswaran, and Andrew G. Dempster. 2016. Overview and investigation of SEU detection and recovery approaches for FPGA-based heterogeneous systems. In *FPGAs and Parallel Architectures for Aerospace Applications*. Springer, 33–46.

[8] Sergio D'Angelo, Cecilia Metra, Sandro Pastore, A. Pogutz, and Giacomo R. Sechi. 1998. Fault-tolerant voting mechanism and recovery scheme for TMR FPGA-based systems. In *DFT*. 233–240.

[9] Jonathan M. Johnson and Michael J. Wirthlin. 2010. Voter insertion algorithms for FPGA designs using triple modular redundancy. In *FPGA*. 249–258.

[10] Ganghee Lee, Dimitris Agiakatsikas, Tong Wu, Ediz Cetin, and Oliver Diessel. 2017. TLegUp: A TMR code generation tool for SRAM-based FPGA applications using HLS. In *FCCM*. 1–4.

[11] Daniel McMurtrey, Keith S . Morgan, Brian Pratt, and Michael J Wirthlin. 2008. *Estimating TMR Reliability on FPGAs Using Markov Models*. Technical Report. Brigham Young University. Retrieved from http://scholarsarchive.byu.edu/facpub/149.

[12] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. Comput.-Aid. Des. Integr. Circuits Syst.* 35, 10 (2016), 1591–1604.

[13] Gabriel Luca Nazar, Leonardo Pereira Santos, and Luigi Carro. 2015. Fine-grained fast field-programmable gate array scrubbing. *IEEE Trans. VLSI Syst.* 23, 5 (2015), 893–904.

[14] QB50 Project. 2009. Homepage. Retrieved June 6, 2017 from https://www.qb50.eu.

[15] Luca Sterpone, Matteo Sonza Reorda, and Massimo Violante. 2005. RoRA: A reliability-oriented place and route algorithm for SRAM-based FPGAs. In *PRIME*, Vol. 1. IEEE, 173–176.

[16] Martin Straka, Jan Kastil, Zdenek Kotasek, and Lukas Miculka. 2013. Fault tolerant system design and SEU injection based testing. *Microprocess Microsy* 37, 2 (2013), 155–173.

[17] Jorge Tonfat, Fernanda Kastensmidt, and Ricardo Reis. 2015. Analyzing the effectiveness of a frame-level redundancy scrubbing technique for SRAM-based FPGAs. *IEEE Trans. Nucl. Sci.* 62, 6 (Dec. 2015), 3080–3087.

[18] Xilinx Inc. 2013. *UG470: 7 Series FPGAs Configuration User Guide*. Retrieved from https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf.

[19] Xilinx Inc. 2015. *PG036: Product Guide - Soft Error Mitigation Controller (v4.1)*. Retrieved from https://www.xilinx.com/support/documentation/ip_documentation/sem/v4_1/pg036_sem.pdf.

[20] Xilinx Inc. 2015. *UG909: Vivado Design Suite User Guide—Partial Reconfiguration*. Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug909-vivado-partial-reconfiguration.pdf.

[21] Xilinx Inc.2016. *XAPP1222: Isolation Design Flow for Xilinx 7 Series FPGAs or Zynq-7000 AP SoCs (Vivado Tools)*. Retrieved from https://www.xilinx.com/support/documentation/application_notes/xapp1222-idf-for-7s-or-zynq-vivado.pdf.

[22] Zhuoran Zhao, Dimitris Agiakatsikas, Nguyen T. H. Nguyen, Ediz Cetin, and Oliver Diessel. 2016. Fine-grained module-based error recovery in FPGA-based TMR systems. In *FPT*. 101–108.

Received June 2017; revised October 2017; accepted December 2017