INTEGRATION the VLSI journal 59 (2017) 98–108

Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

Optimizing scrubbing by netlist analysis for FPGA configuration bit classification and floorplanning

Bernhard Schmidt, Daniel Ziener*, Jürgen Teich, Christian Zöllner

Hardware/Software Co-Design, Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Cauerstrasse 11, 91058 Erlangen, Germany

ARTICLE INFO

Keywords:
Single Event Upsets
FPGA scrubbing
Configuration bit partitioning
Floorplanning
Fault injection

ABSTRACT

Existing scrubbing techniques for SEU mitigation on FPGAs do not guarantee an error-free operation after SEU recovering if the affected configuration bits do belong to feedback loops of the implemented circuits. In this paper, we a) provide a netlist-based circuit analysis technique to distinguish so-called *critical* configuration bits from *essential* bits in order to identify configuration bits which will need also state-restoring actions after a recovered SEU and which not. Furthermore, b) an alternative classification approach using fault injection is developed in order to compare both classification techniques. Moreover, c) we will propose a floorplanning approach for reducing the effective number of scrubbed frames and d), experimental results will give evidence that our optimization methodology not only allows to detect errors earlier but also to minimize the Mean-Time-To-Repair (MTTR) of a circuit considerably. In particular, we show that by using our approach, the MTTR for datapath-intensive circuits can be reduced by up to 48.5% in comparison to standard approaches.

1. Introduction

The application areas of Field Programmable Gate Arrays (FPGAs) have steadily grown over the last two decades. Initially, FPGAs were intended to replace CPLDs to implement simple glue logic functions, but today's SRAM-based FPGAs are able to host entire multi-core systems on a single chip. By now, FPGAs play a major role for implementing complex digital systems for telecommunication networks, software-defined radios, and computer vision. The success in these application areas makes FPGAs also interesting for new safety-critical applications and application fields, like in space missions or avionics, which are still in the hand of traditional ASICs. However, in these new operation sites, FPGAs have to deal with harsh environments, and the implemented systems are forced to guarantee a high reliability.

Today, most advanced SRAM-based FPGA devices such as the Xilinx Virtex-7 family offer up to 2 million logic cells. On the other hand, these huge numbers of FPGA resources must be configured which results in bitstream sizes of up to 55 MB [1]. In fact, this makes current FPGA devices to some of the largest SRAM chips available [2]. This development was only possible by advances in manufacturing process technology to ever-smaller scales. In general, the SRAM-cells of an SRAM chip are susceptible to effects of cosmic ray particles, which can lead to so called *Single Event Upsets* (SEUs), a bit-flip of the stored value in one SRAM-cell. Although, the charges stored in SRAM-cells decrease with every new technology generation, the susceptibility to SEU of one SRAM-cell has been lowered or has been held constant at least [3]. Nonetheless, since the number of SRAM-cells per FPGA-device has been increased significantly, the probability that an SEU occurs during operation of the FPGA device has been increased. This makes SEUs an important reliability issue, especially in harsh radiation-prone environments like space applications. Here, SEUs impose a well-established concern, but they receive increasing concern also for safety-critical terrestrial applications such as systems for medical, automotive, and power generation applications.

To prevent the accumulation of SEUs, well-known frame-based scrubbing techniques [4] can be successfully applied. Scrubbing allows the recovery from errors induced by SEUs by periodically or error-triggered frame-wise overwriting of configuration bits with the unfalsified values.

However, most of these approaches do not take into account that SEUs have varying impacts on the implemented logic. Even in large designs, only a minority of the configuration bits is used and, therefore, have influence on the implemented functionality. SEUs on *unused* configuration bits may obviously be ignored. However, SEUs on unused configuration bits might create some ghost circuits which might increase the FPGA power consumption, but have no influence on the functionality of the design. Moreover, SEUs on used configuration bits

* Corresponding author.
E-mail addresses: daniel.ziener@fau.de (D. Ziener), juergen.teich@fau.de (J. Teich).

http://dx.doi.org/10.1016/j.vlsi.2017.06.012
Received 30 April 2015; Received in revised form 8 May 2017; Accepted 26 June 2017
Available online 30 June 2017
0167-9260/

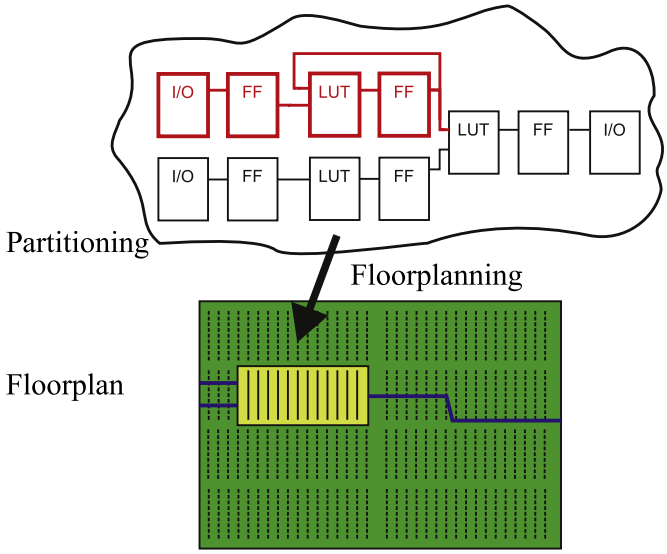

Fig. 1. Illustration of our two step approach. In the partitioning step, the primitive cells of the netlist, e.g., LUTs and flip-flops, and nets are categorized into *essential* (black) and *critical* (red) cells, nets respectively in order to identify and distinguish the associated *essential* and *critical* bits. In the floorplanning step, the primitive cells are placed and routed such to minimize the number of occupied configuration frames by using of special placement and routing constraints. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

may also have varying impacts on the circuit behavior and may demand different error-handling methods. In particular, not all SEU-induced errors may be corrected only by the application of scrubbing. Indeed, components of a circuit involved in feedback paths might still and infinitely cause a malfunction due to corrupted states even if scrubbing is applied to the involved configuration frames.

In this paper, we present an automatic partitioning approach which categorizes primitive cells and nets at logic level into *essential* and *critical* cells, respectively nets. Functional errors caused by SEUs on configuration bits of *essential*, but *non-critical* primitive cells and nets may be corrected by scrubbing without any further actions. For SEUs on configuration bits of *critical* primitive cells and nets, further actions, like resetting the registers, must be performed additionally to scrubbing. Some related work using the terms *sensitive* and *persistent* instead of *essential* and *critical*, e.g., [5]. In addition to this automatic netlist partitioning as illustrated in Fig. 1, we propose advanced floorplanning methods to reduce the overall number of frames which have to be scrubbed. As an important side effect, this also may lead to a reduction of the Mean-Time-To-Repair (MTTR) of a given system due to shorter scrubbing cycles. Furthermore, by knowing the classification of each configuration bit, the number of time-consuming state-restoring actions, like global resets of the circuit, are reduced, since not every SEU will demand such actions after scrubbing. A set of experiments will give evidence that the proposed technique may indeed reduce the downtime of a system considerably. Furthermore, an alternative classification approach by utilizing fault injection experiments is presented and a comparison to the introduced automatic partitioning approach is provided.

Circuits for the control path consist in general of more feedback loops compared to datapath-intensive circuits. In harsh environments, like space applications, the control-intensive part is mostly implemented into radiation hardened external microprocessors. Therefore, our focus is on FPGA-based datapath-intensive circuits which are common in digital signal processing applications.

This paper extends a preliminary publication [6] in the following major aspects: apart from the presentation of an automated netlist classification and floorplanning method, the design flow in Section 7 and an alternative fault injection classification approach in Section 8 are introduced, followed by a comparative study of the results of both classification techniques in Section 9.

The remainder of this paper is structured as follows: Section 2 discusses related work in the field. The problem statement and target applications are described in Section 3. In Sections 4 and 5, we provide formal definitions to describe the correlation between netlist primitive cells, respectively nets and the corresponding configuration bits. Furthermore, we introduce our definition of the MTTR. Our methodology is described in detail in Section 6, whereas the design flow to implement our efficient SEU scrubbing method is presented in Section 7. Section 8 describes the implementation of an alternative classification approach by using fault injection. In Section 9, we present experimental results and Section 10 concludes the paper.

2. Related work

Previous research has studied the impact of SEUs on SRAM FPGA devices[1] [8–10]. Many techniques have been proposed to provide highly reliable FPGA devices, e.g., *radiation-hardened FPGAs*,[2] to lower the effect of radiation-induced SEUs. However, radiation-hardened SRAM FPGAs typically have a lower density compared to non radiation-hardened FPGAs, and they only lower the probability of SEUs and do not completely avoid them. A comparison between a radiation-hardened FPGA (Virtex-5QV) and a standard FPGA (Kintex-7) was done in [15]. Even on radiation-hardened FPGAs, the SEU rate in a low-earth orbit can be up to 16 events per day for a Virtex-4QV [16] and up to 1.2 events per day for a Virtex-5QV [17]. Hence in space missions, SEU correction mechanisms become essential to avoid the accumulation of latent faults and ensure correct operation of an FPGA.

A wide variety of other SEU fault mitigation techniques for SRAM-based FPGAs have been proposed during the past years. These techniques can be categorized into module redundancy techniques, e.g., *Dual Module Redundancy* (DMR), *Triple Module Redundancy* (TMR) [18–20], *N Module Redundancy* (NMR), and techniques that use *scrubbing* of the FPGA configuration memory [21–23]. Also, the combination of both techniques has been shown to be able to increase the reliability of the FPGA modules significantly [24]. FPGA-based TMR approaches replicate a given module which shall be protected either statically [19] or dynamically [20]. However, TMR techniques are known to often cause an excessive and unacceptable overhead in terms of power consumption and area. Since the intensity of cosmic rays is not constant but may vary over several magnitudes depending on the solar activity, a worst-case radiation protection is far too expensive in most cases. A self-adaptive system is proposed in [17], which monitors the current SEU rate and exploits the opportunity of partial reconfiguration of FPGAs to implement redundancy such as TMR on demand. Furthermore, the possibility to partially reconfigure the FPGA could also be exploited to repair faulty modules (single TMR instances) through reconfiguration which may be identified by a majority voter [25]. Hereby, it is necessary that the modules do not include any feedback paths. Otherwise, costly state recovering actions must be applied.

Scrubbing techniques can also be categorized into *blind* and *non-blind* scrubbing. Blind scrubbing refreshes the configuration memory in a periodically manner without any error detection. Non-blind scrubbing, however, refreshes the configuration only once an SEU is detected. Commonly, the error detection of non-blind scrubbing is achieved by a frame-by-frame readback with Error Correction Code (ECC) checking [22]. In addition, a Cyclic Redundancy Check (CRC) over the complete array of frames is used for a fast detection of errors

[1] The configuration memory of Flash-based FPGAs like the FPGAs of the Microsemi ProAsic3 family is immune to SEUs. However, in general, Flash-based FPGAs have a significant lower logic density in comparison to SRAM-based FPGAs [7].

[2] Examples are Xilinx Virtex-4QV [11], Xilinx Virtex-5QV [12], Microsemi RTG4 [13], and the NanoXplore NXT-32000 [14].

in the configuration memory.

In [26,27], an approach called partial TMR is described in which TMR is selectively applied to a given design to keep the overhead small due to the triplication of combinatorial and sequential logic. To identify locations requiring triplication, the authors propose a netlist analysis. In our approach, also a netlist analysis is proposed, however, in order to identify the configuration bits of *critical* primitive instances and *critical* nets rather than places for module triplication. Finally, in [23], the authors propose to use placement constraints aligned to the frame boundaries to minimize the number of frames with *essential* bits which is similar to our minimization approach. However, they do not actively manipulate the routing. Moreover, the characterization of *essential* and *critical* bits is not considered in their work.

3. Problem statement

In general, SEUs in the configuration memory are considered as *soft errors*. A bit-flip in the configuration memory does not cause a permanent and non-recoverable defect, since it does not damage the SRAM cell itself, but corrupts its stored data. A typical manifestation of an SEU-related error is an altered LUT function or signal route. A circuit design may be simply corrected by overwriting the corresponding memory cell with the correct value. However, the configuration bits of an FPGA are typically written once at the FPGA boot-up time and are never changed or refreshed during operation. Therefore, without any SEU correction mechanism, the output errors induced by SEUs appear to be permanent. In this context, we do not consider SEUs, which change the state of the user-logic flip-flops directly, since they do not change the configuration of an FPGA.

As described in Section 2, SEUs can be corrected by *scrubbing* techniques. However, this takes some detection and correction time, in which the circuit might also produce and store wrong results in registers. If the circuit netlist is free of cycles, no further actions have to be carried out after scrubbing. However, if the circuit contains any feedback cycles, a recovery of the circuit after *scrubbing* can only be achieved by resetting all registers to an initial state or by replacing the register values by values saved at checkpoints during an uncorrupted circuit behavior. Nevertheless, the application has to tolerate such a behavior by invalidating corrupt output data. Streaming and packet-based applications, in which video, audio, or wireless communication data are processed, usually may tolerate such behavior as well as

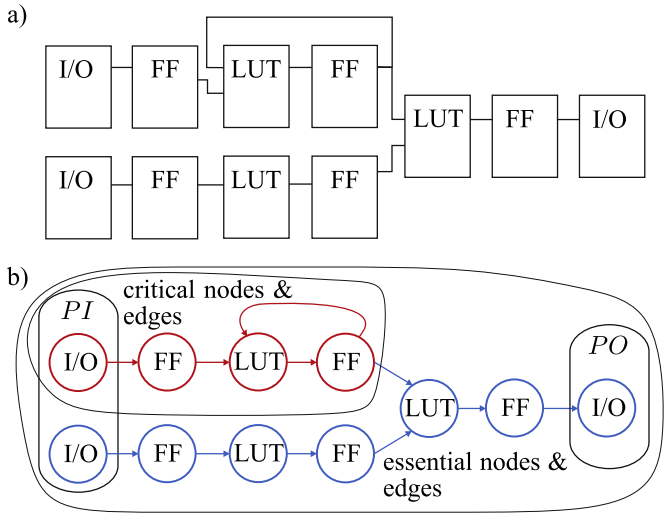

Fig. 2. Illustration of the logic-level netlist analysis to identify *critical* primitive instances and nets in the set of *essential* primitive instances and nets. The netlist in a) is converted to the directed graph $G(E, V)$ in b) with the primary input nodes PI and the primary output nodes PO. *Critical* primitive instances and nets are marked red. *Critical* primitive instances reside on either cycles or on feed-forward paths into cycles and each net which is fed into a *critical* primitive instance is itself *critical* [6].

partially corrupted outputs since data may also be corrupted by interference of wireless transmissions and, therefore, error correction schemes must always be present.

One example is the receiver chain of a communication satellite presented in [15]. A demodulator, including frequency conversion, filtering, and demapping of a Quadrature Phase-Shift Keying (QPSK) modulation is followed by a deinterleaver and a Low-Density-Parity-Check (LDPC). An erroneous output of this receiver chain might be tolerated for a short period of time due to implemented error handling in upper layers of the communication protocol. Note that control intensive designs might not tolerate such a behavior. Therefore, our approach focuses on data paths which are implemented in a stream-based manner. By combining our approach with TMR, also applications can be supported which do not tolerate this limitation.

One important reliability metric is the *Mean-Time-To-Repair* (MTTR). Consider a system model that tolerates corrupt outputs for short times and rollbacks to former saved checkpoints. In that case, the MTTR is the period of time in which the circuit may deliver corrupt output values. Therefore, a reduction of the MTTR will be of great benefit increasing the overall circuit reliability.

On the other hand, the MTTR is closely related to the *Mean-Time-To-Failure* (MTTF) since repair actions have to be taken only if a failure occurs. According to [24], the SEU-related MTTF depends on the SEU rate μ_{FPGA} of the FPGA's configuration memory and the probability that an SEU hits an *essential* bit. The SEU rate μ_{FPGA} depends on the FPGA device[3] and the environment.[4]

4. Definitions w.r.t. partitioning and floorplanning

In order to explain our MTTR minimization approach, we first introduce some formal definitions to clarify the dependency between configuration bits and different FPGA resources which is later used to describe our SEU-aware FPGA design flow in detail in Section 7.

During automatic partitioning, we analyze the logic-level netlist of a given digital circuit after logic synthesis. The resulting logic-level netlist is usually given in a structured text format and consists of instances of primitive cells, e.g., *LUTs*, *flip-flops*, *I/Os*, *block memories* (BRAM), *embedded multipliers* and *nets*. One illustration of an netlist is show in Fig. 2 a). The netlist can be equivalently described by a directed graph $G(V, E)$ with the set of nodes V and the set of edges E. The instances of primitive cells correspond to the nodes $v \in V$ and the nets correspond to edges $e \in E$ of the graph, respectively. As illustrated in Fig. 2b), nets with more than one sink are described by multiple directed edges between the source primitive cell and the corresponding sink primitive cells. Therefore, each edge $e = (v_i, v_j) \in E$ is directed from $v_i \in V$ to $v_j \in V$. Between any ordered pair of nodes, there exists at most one directed edge. A sequence of edges $p = (e_1, e_2, e_3, \ldots, e_n)$ with $e_i = (v_{i-1}, v_i)$ is called a path. p is called a cycle, if $v_0 = v_n$. We define the set K as the set of nodes $v \in V$ contained in at least one cycle. Furthermore, we call edge (v_j, v_i) out-edge for the node v_j and in-edge for node v_i. The set $E^-(v)$ contains all out-edges and the set $E^+(v)$ all in-edges of node v. If $E^+(v) = \{\}$, then v is called primary input node and $v \in PI$, and PI is called the set of primary input nodes. If $E^-(v) = \{\}$, then v is called primary output node and $v \in PO$, and PO denotes the set of primary output nodes.

Finally, as illustrated in Fig. 2b), all nodes and edges will be called

[3] The main factors of the SEU sensitivity of FPGA devices are the technology parameters (mainly V_{dd} and the structure size), the *die size*, and the *number of configuration bits* [24].

[4] The most important parameter of the environment is the *particle flux* that is environmental-dependent (ground or different orbits for space applications). A tool to estimate the SEU rate μ of semiconductor devices in *SEUs/device/s* for different orbits is CREME96 [28]. The MTTF differences between space and ground conditions for the same design and device can vary in the range of several magnitudes. Therefore, the reduction of the MTTR is utmost important, especially for space applications with high SEU rates of up to several SEUs per day, even on radiation-hardened FPGAs [16].

essential. Moreover, we define all paths p to be *critical* if for p holds that the first node v_0 is a primary input ($v_0 \in PI$) and the last node v_n belongs to a cycle ($v_n \in K$). All nodes and edges belonging to *critical* paths will be called *critical* as well. As a result, the set of *critical* nodes and edges forms a subset of the set of *essential* nodes and edges.

During the FPGA implementation process, the instances of the primitive cell types *lookup table*, *flip-flop*, as well as *carry logic cells* are first mapped into device-specific basic resource instances ρ of type $\tau_\rho = slice$ which are called *CLB* (configurable logic blocks) slices in case of Xilinx FPGAs. For example, a single Xilinx CLB slice consists of several lookup tables, flip-flops, carry logic, and multiplexers as well as corresponding nets E_{CLB} connecting them. Typically, a single instance ρ of type $\tau_\rho = slice$ is able to implement several nodes v as well as nets e between these nodes if the required routing resources are present and available. Furthermore, all embedded hard macros, like *multipliers*, *block memory*, *DCMs*, will also be mapped into corresponding device-specific basic resource instances ρ of types $\tau_\rho \in \{mult, bram, dcm, \ldots\}$. After placing the instances ρ, the unrouted nets $E_{con} = E \backslash E_{CLB}$ will be routed. Each routed net $e \in E_{con}$ typically occupies several instances ρ of type programmable interconnect points (PIPs) ($\tau_\rho = pip$) which are the basic instances of the interconnect network.

Technically, each instance ρ is configured by a set of configuration bits $B_\rho \subset B_{conf}$ with B_{conf} being the union of all configuration bits of the configuration memory of a given FPGA device. We define the mapping of the instance ρ to the set of configuration bits B_ρ by introducing the function $fmap_b$ such that

$$B_\rho = fmap_b(\rho). \tag{1}$$

In general, this mapping is proprietary and usually unknown to the circuit designer. In Xilinx FPGAs, the configuration memory is usually organized as a column-based array of equally sized frames which we unify in the set F. One frame $f_i \in F$ represents the smallest addressable segments of the configuration memory. For Xilinx Virtex-6 FPGAs, a frame has a height of one clock region and the width of one bit. We denote all bits which belong to one frame f_i as B_{fi}. We define

$$B_{\text{conf}} = \bigcup_{i=0}^{N_{\text{fr,total}}-1} B_{fi} \tag{2}$$

with $N_{\text{fr,total}}$ being the total number of configuration frames which depends on the size and type of the FPGA device. Now note that the set of used configuration bits B_ρ for implementing an instance ρ might be spread over several configuration frames f_i with

$$f_i \in frames(B_\rho) \quad \text{and} \quad i \in \{0, 1, \ldots, N_{\text{fr,total}} - 1\}, \tag{3}$$

whereat $frames(B_\rho)$ denotes the set of frames which contain configuration bits of instance ρ. For example, the bits which configure a Xilinx CLB slice are distributed over 36 subsequent frames for a Virtex-6 FPGA [29]. However, because of the vertical frame structure, one frame also spans over several CLB slices.

In this context, we define $N_{\text{fr,used}}$ to be the number of frames containing configuration bits used in a given design. Furthermore, the set of configuration bits B_{conf} can be partitioned into the set of *unused* bits B_u, a set of *essential* bits B_e used in a design and a set of *critical* configuration bits B_c as a subset of B_e with the distinction that *critical* bits belong to those instances ρ of a circuit that do occur in feedback paths:

$$B_{\text{conf}} = B_{\text{u}} \cup B_{\text{e}} \quad \text{and} \quad B_{\text{c}} \subseteq B_{\text{e}}. \tag{4}$$

Also, we let $n_{\text{e}} = |B_{\text{e}}|$, respectively $n_{\text{c}} = |B_{\text{c}}|$ denote the number of *essential* and *critical* bits.

5. Definitions w.r.t. reliability

Traditional scrubbing methods suffer from a long MTTR, since these techniques check and refresh the complete configuration memory frame-by-frame which can take up tens to hundreds of milliseconds [22]. Furthermore, by not knowing the criticality of a corrupted bit, the worst case has to be assumed and state-restoring actions, like a global reset or checkpoint restoring, have to be processed as well. This may further increases the MTTR, first by the time $t_{restore}$ needed to perform state-restoring actions and second by the time t_{lost} for reprocessing the lost data. According to [23], the MTTR can be therefore defined by

$$MTTR = MTTD + t_{\text{repair}} + t_{\text{restore}} + t_{\text{lost}}, \tag{5}$$

where MTTD denotes the *mean time to detect* an SEU and t_{repair} is the time to repair an SEU-corrupted frame. For *essential*, but *non-critical* configuration bits $b \in B_{\text{e}} \backslash B_{\text{c}}$, we need no state-restoring actions which means that t_{restore} and t_{lost} are zero. For continuous scrubbing methods, the MTTD can be defined as

$$MTTD = t_{\text{check}} \cdot \frac{1}{2} \cdot N_{\text{fr,used}}, \tag{6}$$

where $N_{\text{fr,used}}$ is the number of frames which have to be verified and t_{check} is the time to determine if one frame is corrupted or not.

6. Proposed approach

Now, in order to decrease the MTTR, we first distinguish and identify automatically *critical* bits from *essential* bits to reduce the number of state-restoring actions. Second, we try to minimize the number of frames $N_{\text{fr,used}}$ which have to be verified. We achieve the first goal by netlist analysis with subsequent partitioning of primitive cells v and nets e into *critical* and *non-critical* cells and nets (see Section 6.1). With the help of the Xilinx tool *bitgen*, we are able to determine the corresponding *critical* bits in a given bitfile (see Section 7). The great advantage of our method over previous fault-injection approaches like [30,5], is the automatic determination of *critical* bits without requiring any time-consuming bit-wise fault injection and complex verification techniques. Furthermore, after identifying the *critical* cells and nets at logic level, we manipulate the placement and routing to minimize $N_{\text{fr,used}}$. Verifying and correcting bits can only be done frame-wise by reading or writing whole frames. Therefore, the second goal, the reduction of the number of occupied frames $N_{\text{fr,used}}$, is achieved by manipulated floorplanning in such a way that a high frame utilization is achieved (see Section 6.2).

6.1. Analysis of logic-level netlist

The aim of the following netlist analysis is to identify exactly those instances of primitive cells and nets which may lead to a permanent corrupt state of the circuit even if scrubbing would be applied to the SEU-corrupted configuration memory. According to Section 4, these instances and nets are obviously located in the netlist in either feedback cycles or on the corresponding input paths to feedback cycles denoted as *critical* instances and nets since a malfunction of these instances or an error in the routing of these nets may lead to erroneous state being trapped in the corresponding cycle. An SEU occurring in any configuration bit of those components demands a special treatment after correction, like a circuit reset into a valid or fail-save state. Hence, we defined these configuration bits as *critical* configuration bits in Section 4. However, such state-restoring actions are not necessary in case an SEU hits a configuration bit belonging to a *non-critical essential* bit.

6.2. Minimization of the number of used frames $N_{\text{fr,used}}$

Typically, traditional scrubbing methods verify all $N_{\text{fr,total}}$ frames of the configuration memory, but since only $N_{\text{fr,used}}$ configuration frames contain *essential* bits, we suggest to read-back and verify only these $N_{\text{fr,used}}$ frames with $N_{\text{fr,used}} < N_{\text{fr,total}}$. However, if no constraints are used for resource placement and routing, many of the used frames may be low-utilized by *essential* bits and may be scattered all over the FPGA.

Therefore, by imposing allocation area constraints during the placement process, we first propose to align the used resources of the

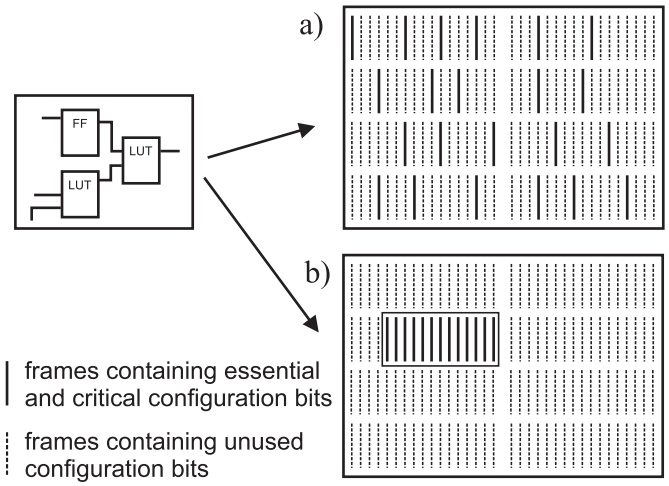

Fig. 3. Illustration: a) a standard placement and routing of a given circuit and b), our area constrained partitioning and floorplanning. By comparing a) and b), the number of occupied frames is reduced from $N_{\text{fr,used}} = 22$ to $N_{\text{fr,used}} = 13$ frames. Moreover in b), the cell instances and the routing are aligned to the frame boundaries and placed near to the I/O buffers located in the center of the FPGA [6].

FPGA device to frame boundaries as illustrated in Fig. 3 b). To determine the needed number of frames inside a rectangular allocation area, we use the number of utilized resources from the synthesis report. Furthermore, the nearest location to the used I/O pins is chosen for the constrained area in order to avoid long nets to the I/O pins. Such nets might elsewise generate a huge number of very low utilized additional frames, which must be also checked during scrubbing.

To avoid net routings leaving the allocated area and generating additional *essential* frames, we constrain also the routing. This can be done by blocking all routing resources outside the allocated area by using so-called *blocker macros*. Such macros can be generated automatically by a floorplanning tool such as *GoAhead* [31].

The result is a densely placed and routed design where almost all used frames are clustered together. By improving the utilization of frames, also the number of *essential* frames $N_{\text{fr,used}}$ is reduced. Another advantage is that the scrubbing may be executed on contiguous regions and without any holes in between. Such holes, which typically result from unconstrained placements, might generate additional scrubbing overhead when using blind scrubbing without readback. If the configuration memory cannot be subsequently written, the scrubber must include an additional configuration header which addresses the frame which should be written next [32].

7. Design flow

Our approach is applicable to many Xilinx SRAM-based FPGAs. The following proposed design flow, as illustrated in Fig. 4, is based on the Xilinx design tool *ISE 14.2*, on our own netlist analysis tool, and on the low-level FPGA design tool *GoAhead* [31] which is used for macro generation. Moreover, we use the *Xilinx LogiCORE Soft Error Mitigation (SEM) IP core* [33] to implement a scrubbing controller which can detect and correct SEUs in the configuration memory.

The entry point of the proposed design flow is the netlist after synthesis which consists of cell instances and nets given in the Xilinx proprietary *ngc* netlist format. This *ngc* file is converted into the EDIF file format and forwarded to the netlist analysis tool. In addition, the netlist is fed into the implementation stage of the standard Xilinx design flow.

The netlist analysis tool converts the netlist into the graph $G(V, E)$. From this graph, all *critical* cell instances and nets in the netlist as described in Section 6.1 are extracted. The resulting *critical* cells instances and nets are recorded into a Xilinx FPGA Editor *scr* script

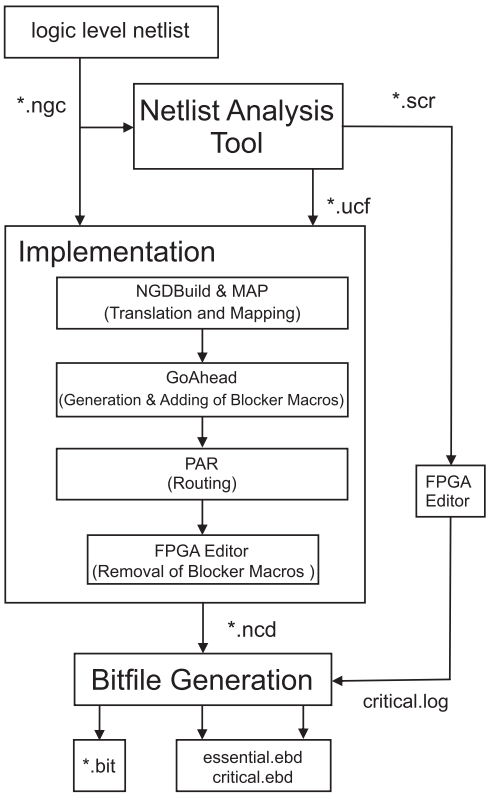

Fig. 4. Proposed design flow: The partitioning between *essential* and *critical* bits is done in the netlist analysis tool. Furthermore, the floorplanner to minimize the number of occupied frames is controlled by the *ucf* generated by the netlist analysis tool by defining area constraints for the cell instances and by so-called *blocker macros* generated and added by *GoAhead*. The information about the locations of the *essential* and *critical* bits are stored in the *ebd* files *essential.ebd* and *critical.ebd* which are generated by the Xilinx *bitgen* tool.

file.[5] Furthermore, for all cell instances ρ, placement constraints are generated and recorded into the Xilinx *User Constraint File* (*ucf*). These placement constraints are arranged such that the number of occupied frames $N_{\text{fr,used}}$ is minimized. This is done by aligning the targeted area for implementation of the circuit to the frame boundaries. Moreover, this area is placed next to the fixed I/O buffers in order to avoid long routing lines.

In the implementation stage, the *ncd* netlist is first translated and mapped to the specific resource instances ρ of the FPGA by using the Xilinx tools *ngdbuild* and *map*, respectively. Afterwards, blocker macros are generated and added to the design via the tool *GoAhead* in order to occupy the routing resources outside the specified area. Without blocking these resources, the Xilinx router might use these resources and generate additional low-utilized frames. After adding blocker macros, the design is placed and routed by the Xilinx *par* tool. Subsequently, the blocker macros are removed again by executing an FPGA-Editor script file, prepared during the generation of the blocker macros.

To identify *essential* configuration bits B_e and *critical* configuration bits B_c stemming from the corresponding *essential* and *critical* resource instances and nets, we use the Xilinx *bitgen* tool. A bitfile mask for the *essential* configuration bits B_e can be generated directly by *bitgen* using the option parameters-*g essential*. The resulting *ebd* file which stores for each configuration bit the information if a bit is *essential* or not, can be directly used by the Xilinx SEM IP core. To

[5] Exceptions are the nets of clock and reset signals which are ignored during our automatic netlist classification but are marked as *critical* nets in the Xilinx FPGA Editor *scr* script file. Furthermore, wildcard characters * and ? are selectively used to tolerate renaming of nets and cell instances during the MAP and PAR optimizations steps.

identify the *critical* bits, we may use the filter function of *bitgen* as described in the Xilinx Application Note 538 [34]. In this application note, so-called *prioritized essential bits* are introduced which are associated with one or more manually chosen hierarchical modules in the circuit description. To generate an *ebd* file which only consists of *critical* bits from the chosen modules, *bitgen* is executed using a filter file. This filter file consists of all resource instances ρ (slices and nets) which belong to the chosen modules. For the proposed approach, we use this filter file to store all *critical* slices and nets which are automatically identified by the method described in Section 6.1. To identify all *critical* resource instances ρ of type $\tau_\rho = slice$ from the *critical* cell instances $v \in V$, we use a script generated by our analysis tool which is executed on the routed netlist by the Xilinx FPGA Editor.

The results are two different *ebd* files, one for *critical* and one for *essential* bits. These *ebd* files or their containing information may be stored in external and may be used to implement a scrubber which verifies only used frames and finally, initiates state-restoring actions only if a fault on a *critical* bit has been detected.

8. Configuration bit classification through fault injection

In order to evaluate the results of our systematic approach for configuration bit classification in *essential* and *critical* bits, we implemented a fault injection approach (see Fig. 5). In the past, different fault injection approaches were successfully applied to emulate the effects of SEUs in the SRAM cells of the configuration memory [35–38,30,5]. The fault injection approach in [5] classifies the configuration bits into *sensitive* and *persistence* bits which is analogous to the *essential* and *critical* classification. Our fault injection approach is a straight-forward implementation and can be implemented on all modern Xilinx FPGAs without any need of off-chip hardware components. For the evaluation, we implemented the benchmark circuit in a constrained area of the FPGA and compared the results with a replicated reference circuit implemented in a distinguished area. Faults are injected with the help of a MicroBlaze softcore processor by reading back one configuration frame of the circuit under test over the *internal configuration access port* (ICAP), toggle a single bit, and a write back to the original location within the configuration memory. It is very important to spatially separate the circuit under test and the reference circuit into different clock regions which corresponds also to different frame addresses. Otherwise, the fault injection might affect also the reference design and, therefore, falsify the result. Both circuits are stimulated with the same input pattern, produced by a pseudo random number generator.

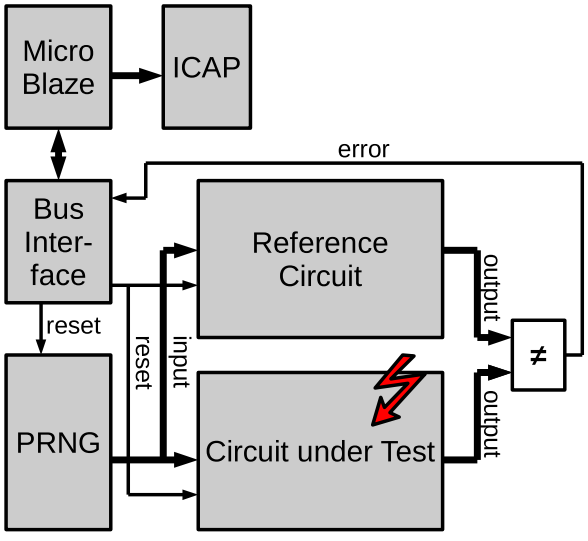

Fig. 5. The architecture of the FPGA-based fault injection approach. Faults are injected in the configuration of the circuit under test over the ICAP interface. The reference design and the circuit under test are stimulated with identical inputs from the pseudo random number generator (PRNG). A comparator compares the result of both circuits and signals an error to the fault injection software, running on the MicroBlaze.

In order to determine the *essential* and *critical* bits with fault injection, every configuration bit of the constrained area on which the circuit under test is implemented must be toggled with subsequent evaluation of the impact. This is done at follows: First, the circuit under test and the reference design is reset. The fault is injected in the corresponding configuration bit by reading back the corresponding frame, toggle the bit, and write it back into the configuration memory. After that, the circuit under test and the reference circuit process input data stemming from the pseudo random number generator for a specified time. The outputs of both circuits are compared and if a mismatch is discovered during the processing, the corresponding configuration bit is marked as *essential*. After that, the error-free frame is recovered in the configuration memory and the testing sequence is replayed without reseting the circuits by reseting only the pseudo random number generator. If an error occurs even on the now error-free configuration, the configuration bit to test is additionally marked as *critical*. Finally, both circuits are reset and the next configuration bit is evaluated. This is repeated until all configuration bits in the constrained area are tested. The corresponding test program which implements the described procedure is implemented on the MicroBlaze processor.

9. Experimental results

The design flow described in Section 7 was applied to a subset of 20 benchmark circuits from the MCNC suit obtained at [39]. Only circuits with flip-flops were considered. Additionally, we choose a *LMS equalizer*, a *floating-point unit* (FPU), an *AES 128-bit cipher* implementation and a *(204,188)-Reed-Solomon Decoder* from Opencores.org [40]. For our experiments, the circuits were implemented for the Virtex-6 *XC6VLX240T* FPGA. In the following, Section 9.1 shows our results regarding the floorplanning and Section 9.3 shows our results regarding the MTTR.

9.1. *Reducing the number of used frames* $N_{fr,used}$

In Table 1, the benchmark designs are characterized in terms of utilized LUTs, flip-flops, I/O buffers and the number of frames $N_{fr,FF}$ which contain the configuration bits of the utilized flip-flops.[6] Furthermore, the table shows the number of identified *essential* bits n_e, the number of identified *critical* bits n_c, the ratio n_c/n_e for the case that no additional placement and routing constraints are used. In Table 2, the numbers of occupied frames $N_{fr,used}$ are depicted for the case a) that no placement and no routing constraints are used, for the case b) that placement constraints and no routing constraints are used, and for the case c) that placement and routing constraints are used. The last two columns show the reductions of the number of used frames $N_{fr,used}$ when case c) is compared to case a) and case b), respectively.

The approach in [23] uses only placement constraints which corresponds to our case b). Therefore, the comparison between case b) and c) in the last column of Table 2 depicts our improvement over the approach in [23] by using additional routing constraints.

It can be seen that by comparing case c) to case a), the number of occupied frames can be reduced by up to 59% by using placement and routing constraints and by comparing c) to b), the number of occupied frames can be reduced by up to 23.7% just by using additional routing constraints. These gains directly affect the achievable MTTD as well, since according to Eq. (6), the MTTD is linearly dependent on the

[6] There are no *embedded multipliers* or BRAMs instantiated in the benchmark designs. While our automatic netlist analysis approach may apply to *embedded multipliers*, it does not apply to BRAMs since the associated content of the SRAM cells is altered during circuit operation and cannot be scrubbed.

Table 1
Benchmark circuits with used LUTs, flip-flops, IO Buffers, the number of frames $N_{fr,FF}$ containing configuration bits of user logic flip-flops, the number of *essential* and *critical* bits.

Circuit	LUTs	FFs	IO buffers	$N_{fr,FF}$	n_e	n_c	n_c/n_e
MCNC benchmark circuits							
bigkey	536	224	426	33	279,584	250,737	0.897
diffeq	555	248	31	15	205,813	205,813	1
elliptic	109	63	5	8	112,832	112,832	1
frisc	1889	822	136	23	529,562	527,779	0.997
s38417	2188	1263	135	37	502,052	411,022	0.819
s38584.1	1693	1140	342	41	465,790	386,888	0.831
tseng	478	221	174	19	216,002	196,154	0.908
Opencores benchmark circuits							
LMS equalizer	231	206	34	96	156,264	156,264	1
FPU	8007	553	110	171	1,713,480	1,680,509	0.981
AES 128-bit	10,450	10,769	389	316	2,387,020	2,372,907	0.994
(204,188)-RS decoder	3808	2735	21	183	883,437	858,757	0.972

Table 2
Benchmark circuits from Table 1 with the number of occupied frames without (a)) and with placement constraints (b)) and with placement and routing constraints (c)). The last two columns show the reductions of the number of used frames $N_{fr,used}$ when case c) is compared to case a) and case b), respectively.

Circuit	$N_{fr,used}$ for a)	$N_{fr,used}$ for b)	$N_{fr,used}$ for c)	$\Delta N_{fr,used}$ in (%) comparing c) and a)	$\Delta N_{fr,used}$ in (%) comparing c) and b)
MCNC benchmark circuits					
bigkey	4201	2393	1891	43	21.0
diffeq	1332	602	602	54.8	0.0
elliptic	912	395	395	56.7	0.0
frisc	1960	1292	1050	34.1	18.7
s38417	2027	1389	1389	31.5	0.0
s38584.1	3192	2270	1863	28.9	17.9
tseng	1924	1298	1054	32.5	18.8
Opencores benchmark circuits					
LMS equalizer	2051	1563	1120	23.8	28.3
FPU	4117	3520	3280	11.5	6.8
AES 128-bit	6822	6657	6335	2.4	4.8
(204,188)-RS decoder	4610	4073	3761	11.6	7.7

number of occupied frames. Nevertheless, the achievable gains may vary considerably from design to design and depends on the distribution of the used I/Os. Furthermore, our experiments show that either no (0%) (complete combinatorial circuit) or between 81.9–100% of the *essential* bits are also *critical* bits in the analyzed circuits. But nevertheless, in case of the circuit *s38417*, if an SEU occurs on a used configuration bit, there is still the probability of 18.1% that no state-restoring action has to be carried out after scrubbing.

Fig. 6 illustrates our floorplanning strategy for one of our benchmark designs. Fig. 6a) shows the placed, but unrouted design. The placement constraint is highlighted by the black frame which is aligned to the frames of the middle left clock region. It can be seen that many wires have to be routed from the I/O buffers of the left side to the I/O buffers in the middle. Therefore, as shown in Fig. 6b), blocker macros are generated with the tool *GoAhead* to force the wires to be routed through the middle left clock region. This is done to raise the frame utilization[7] in this area and to minimize the additional frames occupied by routing resources. Fig. 6c) shows the final optimized design without the removed blocker macros, in which all routing wires are mainly routed through the desired clock region. Note that placement and routing constraints have also an effect on the timing. The effect depends on the size of the constrained area, the circuit and the desired clock frequency. The decrease of the maximum clock frequency is usually below 10% (see also [41]).

Fig. 7 shows two diagrams of the frame utilization for the benchmark design *bigkey*. Fig. 7a) illustrates the frame utilization without using any special placement and routing constraints, and Fig. 7b) illustrates the frame utilization with placement and routing constraints. The frames in both diagrams are sorted in descending order from left to right regarding to the frame utilization with *essential* bits. It can be seen that the *essential* bits are grouped in less frames and that, in general, the utilization of the frames containing *essential* bits is higher when adding our placement and routing constraints.

9.2. Comparison to fault injection approach

The netlist configuration bit classification method is compared to the fault injection approach, described in Section 8. Hereby, we selected the three largest MCNC benchmark circuits (*frisc*, *s38417*, and *s38584.1*) for comparison (see Table 3). We count only the essential bits n_e and critical bits n_c for the constrained area in which the circuit under test resides. Therefore, the number of determined essential and critical bits of the netlist method n_e and n_c differ to the values in Table 1. The reason for the smaller number of essential bits in Table 3 might be the missing long connections to the I/O cells and the more dense packing.

The constrained area has the height of one clock region and the width of up to 8 CLB columns. This results in up to 746,496 configuration bits which have to be evaluated. For each configuration bit, the test sequence consisting of 1 million input vectors, generated by the pseudo random number generator has to be applied twice (see Section 8). The overall test time for one core is up to 20 h.

The values in Table 3 show that only a subset of the netlist method's *essential* bits is found by the fault injection method. The reason is that the *essential* bits determined by Xilinx *Bitgen* is a worst case estimation. For example, if only two inputs of a LUT are used, most bits are not used whereas *Bitgen* will classify all configuration bits belonging to this LUT as *essential*. Moreover, due to circuit inherent redundancy and false paths, a lot of errors are masked and have, therefore, no impact on the outputs. However, the one million input test vectors are far away to be enough to discover all possible configuration faults which lead to a circuit failure. Moreover, the number of found *critical* bits is much lesser than the number of found *essential* bits. Here, only a small fraction of the determined bits from the netlist method is identified. To understand this behavior, we have to emphasize that the netlist method categorize all bits belonging to feedback paths as

[7] The normalized frame utilization of one frame is defined the ratio of the number of *essential* bits to the total number of configurations bits in a frame.

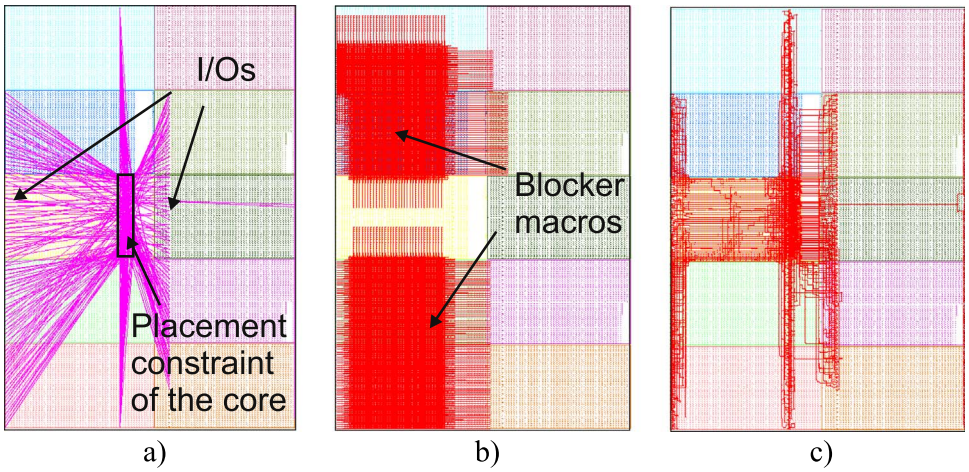

Fig. 6. Illustration of our floorplanning approach applied to one of our benchmark circuits. In a), the unrouted, but placed design is shown. The placement area is highlighted by the black box. In b), the blocker macros are shown which impose the routing constraints. In c), the placed and routed design is shown.

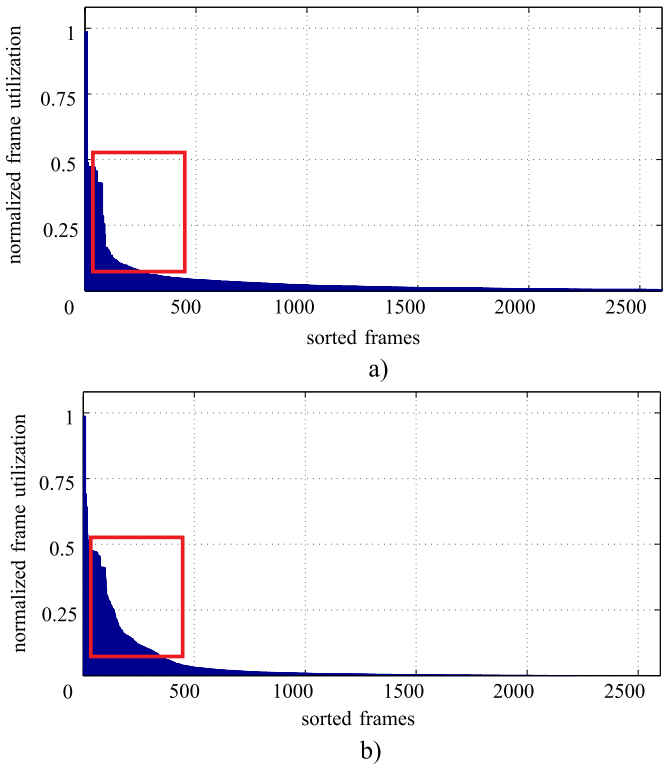

Fig. 7. The diagrams show the frame utilization of the benchmark circuit *bigkey* for the case that no special placement and routing techniques are used (a)) and for the case that our proposed placement and routing constraints are used (b)). The frames are sorted according to their frame utilization in terms of the number of *essential* bits in descending order. Due to the placement and routing constraints, the number of occupied frames $N_{fr,used}$ is reduced from 4201 to 1891 and, therefore, the frame utilization shown in b) is higher.

critical and *essential*. The experiment for determining the *essential* bits with the fault injection method shows, that only a small part of *essential* bits which were found by the netlist method has the ability to corrupt the output due to the above mentioned masking effects. However, we applied one million test vectors which means that the fault has plenty of time to corrupt at least one single bit in one output vector, but during the second run which identifies the *critical* bits, only the falsified states in the registers at the end of the first run has the ability to corrupt the outputs. It could also be the case that the state in the registers during switching from the first to second run was correct and an error on the outputs occurred much earlier during the first run. Even if the state is actually wrong, the falsified state might has no effect on the following states due to the above mentioned masking effects. This shows us that the results of the fault injection methods, especially for the *critical* bits, have to be handled with care.

On the other hand, all discovered *critical* and *essential* bits by fault injection are also included in the set of *essential* and *critical* bits of the netlist method. Therefore, the netlist method is surely an overestimation, but by using this method, we can be sure that we will find any configuration fault which leads to a failure. By avoiding the time-consuming tests of the fault injection method which must be applied after each new place and route iteration, the proposed netlist analysis is easy to use and produces very fast the corresponding result.

9.3. MTTR analysis

In order to calculate the MTTR, we assume a scrubbing system with checkpointing as described in [23]. Additionally, we assume that this system uses the SEM IP core [22]. All calculations rely on the duration times given in [22]. The SEM IP core can be directly used to support our classification approach of *critical* bits. However, the original SEM do not support selective scrubbing approach. To also support this feature, the core has to be modified. The original SEM IP core requires

Table 3
The results of the classification by fault injection method of three selected benchmark circuits. First, the numbers of essential bits n_e and critical bits n_c detected by the netlist and fault injection method are shown. Furthermore, the ratios between essential and critical bits detected by the fault injection and the netlist method are presented.

Circuit	netlist method n_e	netlist method n_c	fault injection method n_e	fault injection method n_c	fault injection vs. netlist method n_e	fault injection vs. netlist method n_c
MCNC benchmark circuits						
frisc	338,218	337,542	30,977	5215	9%	1.5%
s38417	311,584	255,810	17,677	457	6%	0.2%
s38584.1	313,059	261,091	160,008	0	51%	0%

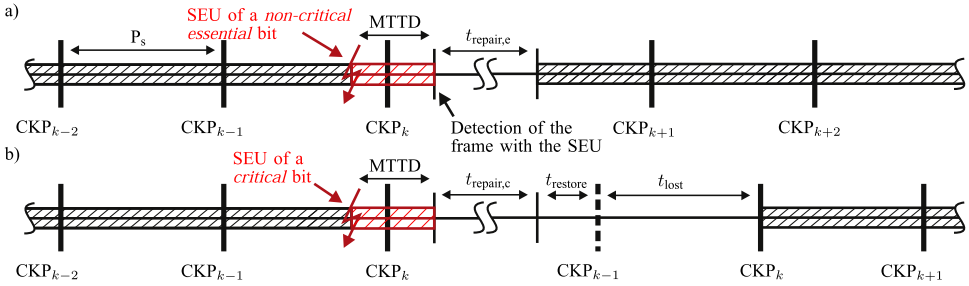

Fig. 8. Illustration of the adopted checkpointing system, in which the checkpoints CKP_k with the time index k are created in time intervals equal to the scan duration P_s. In a), a *non-critical essential* bit is corrupted by an SEU. The corresponding corrupted frame is detected on average after the MTTD. Afterwards, the corrupted *essential* bit is detected within the frame and repaired, which in total takes the time $t_{repair,e}$. In b), a *critical* bit is corrupted by an SEU. In this case, the system has to be rolled back to the checkpoint CKP_{k-1} after the corrupted frame has been detected (MTTD) and repaired ($t_{repair,c}$). The rollback to the checkpoint CKP_{k-1} takes the time $t_{restore}$. Furthermore, it takes the time $t_{lost} = P_s$ until the state of checkpoint CKP_k has been reached again [6].

459 LUTs and 383 flip-flops. Our experimental scrubber core with our SEM IP Core modifications requires 2706 LUTs and 2176 flip-flops.

As illustrated in Fig. 8, we assume that the occupied frames of the configuration memory are continuously scanned for SEUs and checkpoints CKP_k with the time index k are created in intervals equal to the scan duration P_s with $P_s = N_{fr,used} \cdot t_{check}$. If a *non-critical essential* bit is corrupted, as illustrated in Fig. 8a), the scrubbing system needs on averaged the time MTTD to detect the corrupted frame and repairs the corresponding bit with no further action, afterwards. The corresponding repair time $t_{repair,e}$ of an *essential* but *non-critical* bit includes the time to determine the corrupted bit within the corrupted frame and the time to write the repaired frame data into the configuration memory. However, if a *critical* bit is corrupted, as illustrated in Fig. 8b), the system state is rolled back to a valid checkpoint after the detection and correction of the corrupted bit. The system is rolled back to the checkpoint CKP_{k-1} under the assumption that the corrupted *critical* bit has been detected after the checkpoint CKP_k. Therefore, at least the two past checkpoints CKP_k and CKP_{k-1} have to be saved [23]. The repair time $t_{repair,c}$ of a *critical* bit is in general greater than $t_{repair,e}$ since $t_{repair,c}$ further includes the classification time to check if the corrupted bit is indeed a *critical* configuration bit, e.g., by evaluating an *ebd* file stored in external memory. In a frame, which contains *essential* bits, every SEU has to be corrected even if the SEU occurred on an unused bit since the frames are corrected by an evaluation of the ECC parity bits which can only correct one error per frame. Therefore, no accumulation of SEUs is allowed.[8] However, SEUs on unused bits do not contribute to the MTTR because the implemented circuit is not damaged. After a detected SEU on an *essential* bit, the criticality has to be determined (classification time). This step is always performed, however it contributes only to the MTTR if the SEU occurred on a *critical* bit, because after the correction of an *essential non-critical* bit, the circuit is again fully functional.[9] In order to set the registers into the state of the checkpoint CKP_{k-1}, the configuration bits of all used flip-flops have to be set to the stored register values. Therefore, $N_{fr,FF}$ frames have to be re-written after $t_{repair,c}$.

For our analysis, we assume that re-writing one frame to restore the state of a checkpoint requires the duration of two times t_{check} which on the one hand includes the overhead to write the frame with the mask bit to restore the corresponding flip-flops values and which on the other hand also includes the overhead to update the ECC parity bits. Therefore, we define $t_{restore} = 2 \cdot t_{check} \cdot N_{fr,FF}$. Furthermore, the rollback to the checkpoint CKP_{k-1} induce $t_{lost} = N_{fr,used} \cdot t_{check}$ which equals the time to reach the state of CKP_k. In case of a corrupted *essential* bit that is *non-critical*, we set $t_{restore}$ and t_{lost} to zero, since we assume that the system tolerates errors induced by a corrupted *essential* but *non-critical* bit and recovers after correction on its own. For our calculations of the MTTR, we propose the following equation:

$$MTTR = \left(\frac{n_e - n_c}{n_e}\right) \cdot (MTTD + t_{repair,e}) + \left(\frac{n_c}{n_e}\right) \cdot (MTTD + t_{repair,c} + t_{restore} + t_{lost}). \quad (7)$$

According to the product guide of the SEM IP core [22], we set the repair time of *essential* bits $t_{repair,e} = 490\,\mu s$ and the repair time of *critical* bits $t_{repair,c} = 1100\,\mu s$ which additionally includes $610\,\mu s$ for the classification. Furthermore, we obtain $t_{check} = 810\,ns$ which is determined by the frame size of 2592 bits, the ICAP interface word size of 32 bit and the ICAP frequency of 100 MHz.

In Table 4, we compare the MTTR of four different types of scrubbing controllers: a) *scrubbing without any classification*, b) *scrubbing with classification of unused and essential bits*, c) *scrubbing with classification of used, essential, and critical bits*, and d) *scrubbing with classification of used, essential, and critical bits and with placement and routing constraints*. For the type a) the scrubbing controller continuously scans the whole configuration memory consisting of $N_{fr,total} = 22261$ frames and initiates a reset with rollback after each error correction. Since we use a rather huge FPGA device for our small benchmark circuits, the application of this scrubber is not reasonable for our evaluation, but the MTTR is given for the sake of completeness.[10] A scrubbing controller of type b), scans only the occupied frames, but also initiates a reset with rollback even if the corrupted bit is *non-critical*. The scrubbing controller of type c) only scans the occupied frames and only initiates a reset in case a *critical* bit is corrupted. We finally suggest and compare a scrubbing controller of type d) which also uses classification but just scans a reduced set of frames due to the placement and routing constraints. In this analysis, the scrubbing controllers of type a) and b) treat every corrupted configuration bit as a corrupted *critical* configuration bit. Apparently, our proposed approach clearly outperforms the scrubbing controller types a) and b). If it is possible to reduce the number of occupied frames by using placement and routing constraints, then d) outperforms c) as well. In comparison to type b), which can be seen as a standard scrubbing controller, the savings regarding to the MTTR are up to 48.5% for our benchmark circuits.

Table 4 brings out that the savings regarding the MTTR heavily depends on the circuit. Furthermore, by analyzing complex circuits and circuits with a high utilization of the FPGA device, most of the *essential* bits will indeed be identified as *critical* bits. This is especially true for circuits which are not datapath-intensive.

[8] Our approach may be also applicable to multiple bit upsets as long as there is at most one bit-flip in one frame of the configuration memory.

[9] The *ebd* file with the *essential* bits is just needed to determine the frames which have to be scrubbed. Only the *ebd* with the *critical* bits is used for classification.

[10] The main part of these huge MTTRs are the terms MTTD and t_{lost} which corresponds directly to $N_{fr,total}$ and needs $MTTD + t_{lost} = 27137\mu s$. For the smallest available Virtex-6 FPGA, the XC6VLX75T, this time is $9234\mu s$.

Table 4
Comparison of the MTTR for the considered benchmark circuits using four different types of scrubbing controllers: a) *scrubbing without any classification*, b) *scrubbing with classification of unused and essential bits*, c) *scrubbing with classification of used, essential, and critical bits*, and d) *scrubbing with classification of used, essential, and critical bits and with placement and routing constraints*. The last three columns show the savings regarding the MTTR when type d) is compared to type a), type b), and type c).

Circuit	MTTR in [µs]				Δ MTTR in (%) comparing		
	for a)	for b)	for c)	for d)	d) and a)	d) and b)	d) and c)
MCNC benchmark circuits							
bigkey	27,590	6258	5838	3224	88.3	48.5	44.8
diffeq	27,561	2742	2741	1855	93.3	32.3	32.3
elliptic	27,550	2221	2221	1592	94.2	28.3	28.3
frisc	27,574	3519	3511	2408	91.3	31.6	31.4
s38417	27,597	3628	3203	2522	90.9	30.4	21.3
s38584.1	27,603	5045	4492	3059	88.9	39.3	31.9
tseng	27,567	3468	3266	2274	91.8	34.4	30.4
Opencores benchmark circuits							
LMS equalizer	27,693	3748	3748	2616	90.6	30.2	30.2
FPU	27,814	5769	6379	5294	81.0	17.0	15.9
AES 128-bit	28,049	9901	9861	9272	66.9	6.4	6.0
(204,188)-RS decoder	27,834	6998	6868	5856	79.0	16.3	14.7

10. Conclusions and future work

In this work, we present two new methods to enhance common scrubbing techniques for SEU mitigation: a) we apply a netlist analysis and the Xilinx tool *bitgen* to identify and distinguish *essential* and *critical* bits in the configuration memory and b) we use placement and routing constraints to align a given design to the frame boundaries in order to reduce the number of occupied frames that are affected by scrubbing. We integrated these two methods into one design flow, which is fully automated by script files. A comparison to an implemented alternative classification approach using fault injection is given which shows that through the proposed netlist analysis, time-consuming fault injection methods can be avoided to identify *essential* and *critical* configuration bits. Furthermore, due to differentiation between *unused*, *essential*, and *critical* bits, we can efficiently decide if an SEU in the configuration memory has to be corrected at all or if an SEU demands a reset after scrubbing. As was shown by experiments, the introduction of placement and routing constraints may not only help to minimize the number of occupied frames, but also may reduce the MTTR for specific designs by up to 48.5% in comparison to a standard scrubbing controller.

The proposed design flow and the two methods open many possibilities for future work. As seen in the experimental results, the number of *critical* bits dominates the *essential non-critical* bits on the most real world examples. However, error masking through exploitation of redundancy mechanisms and subsequently voting are not considered in our netlist analysis. This is also indicated by the comparison to the fault injection method which identify much lesser bits as *essential* and *critical*. By the sophisticated usage of methods such as DMR and TMR inside feedback loops, we are able to establish an error barrier to mask out an error by voting before it can affect the state of the circuit permanently. This means that former *critical* bits of these feedback loops might be transferred into *non-critical* bits which lowers the overall number of *critical* bits in a design. This might be reached by using the STAR framework [42]. For such designs we predict a higher benefit by using our proposed methodology to reduce the MTTR.

Furthermore, we did not consider any circuits which utilize BRAMs or which implements state-driven protocols to communicate with external peripherals, e.g., external memory. Both the interfaces to BRAMs and the interfaces to external peripherals may lead to feedback cycles which are not handled by our methodology so far. The consideration of memories will therefore extend the scope of our work even further.

For newer Xilinx FPGA families, like 7-series and Ultrascales, the Xilinx design tool suite *Vivado* has to be used. Our approach relies on the tool *GoAhead* [31] which uses the XDL interface of the *ISE* design tools. However, a new version is under development which utilizes the TCL interface of Vivado.

Acknowledgments

The work has been partially supported by EFRE funding from the Bavarian Ministry of Economic Affairs (Bayerisches Staatsministerium für Wirtschaft, Infrastruktur, Verkehr und Technologie) as a part of the “ESI Application Center” project.

References

[1] Xilinx Inc., 7 Series FPGAs Overview, 2012.
[2] D. Koch, J. Torresen, Advances and Trends in Dynamic Partial Run-time Reconfiguration, in: Dagstuhl 10281: Dynamically Reconfigurable Architectures, Schloss Dagstuhl, Germany, 2010, p. 6.
[3] Xilinx Inc., Device Reliability Report: Second Quarter 2013, Tech. rep., Xilinx Inc., 2013.
[4] C. Carmichael, M. Caffrey, A. Salazar, Correcting Single-Event Upsets Through Virtex Partial Configuration. Xilinx Corp., Tech. Rep., XAPP216 (v1.0), Tech. rep., Xilinx Inc., 2000.
[5] K. Morgan, M. Caffrey, P. Graham, E. Johnson, B. Pratt, M. Wirthlin, Seu-induced persistent error propagation in fpgas, IEEE Trans. Nucl. Sci. 52 (6) (2005) 2438–2445. http://dx.doi.org/10.1109/TNS.2005.860674.
[6] B. Schmidt, D. Ziener, J. Teich, Minimizing scrubbing effort through automatic netlist partitioning and floorplanning, in: Proceedings of the Reconfigurable Architectures Workshop (RAW), Phoenix, USA, 2014, pp. 299–304.
[7] Actel Inc., ProAsic3 Flash Family FPGAs Revision 13, 2013.
[8] C. Bernardeschi, L. Cassano, A. Domenici, Failure probability of SRAM-FPGA systems with stochastic activity networks, in: DDECS, 2011, pp. 293–296.
[9] N. Jing, J.-Y. Lee, Z. Feng, W. He, Z. Mao, S.-J. Wen, R. Wong, L. He, Quantitative SEU fault evaluation for SRAM-based FPGA architectures and synthesis algorithms, in: FPL, 2011, pp. 282–285.
[10] U. Legat, A. Biasizzo, F. Novak, Automated SEU fault emulation using partial FPGA reconfiguration, in: DDECS, 2010, pp. 24–27.
[11] Xilinx Inc., Space-Grade Virtex-4QV Family Overview, 2010.
[12] Xilinx Inc., Radiation-Hardened, Space-Grade Virtex-5QV FPGA Data Sheet, 2011.
[13] Microsemi Corp., PB0051: RTG4 FPGAs Product Brief, 2015.
[14] O. LEPAPE, NanoXplore NXT-32000 FPGA, in: SpacE FPGA Users Workshop, SEFUW, vol. 3, 2016.
[15] R. Glein, F. Rittner, A. Becher, D. Ziener, J. Frickel, J. Teich, A. Heuberger, Reliability of space-grade vs. COTS SRAM-based FPGA in N-modular redundancy, in: Proceedings of the 2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2015, pp. 1–8. ⟨http://dx.doi.org/10.1109/AHS.2015.7231159⟩.
[16] H. Quinn, P. Graham, K. Morgan, Z. Baker, M. Caffrey, D. Smith, R. Bell, On-orbit results for the Xilinx Virtex-4 FPGA, in: Proceedings of the Radiation Effects Data Workshop (REDW), 2012 IEEE, 2012, pp. 1–8.
[17] R. Glein, B. Schmidt, F. Ritter, J. Teich, D. Ziener, A self-adaptive SEU mitigation system for FPGAs with an internal block RAM radiation particle sensor, in: Proceedings of the Field-Programmable Custom Computing Machines (FCCM 2014), Bosten, USA, 2014.
[18] R.E. Lyons, W. Vanderkulk, The use of triple-modular redundancy to improve computer reliability, IBM J. Res. Dev. 6 (2) (1962) 200–209.
[19] L. Sterpone, M. Violante, A new analytical approach to estimate the effects of SEUs in TMR architectures implemented through SRAM-based FPGAs, IEEE Trans. Nucl. Sci. 52 (6) (2005) 2217–2223.
[20] J. Angermeier, D. Ziener, M. Glaß, J. Teich, Runtime stress-aware replica placement on reconfigurable devices under safety constraints, in: FPT, 2011.
[21] J. Heiner, B. Sellers, M. Wirthlin, J. Kalb, FPGA partial reconfiguration via configuration scrubbing, in: Proceedings of the International Conference on Field Programmable Logic and Applications, IEEE, 2009, pp. 99–104.
[22] Xilinx Inc., Product Guide: LogiCore IP Soft Error Mitigation Controller v3.4, 2012.
[23] A. Sari, M. Psarakis, Scrubbing-based seu mitigation approach for systems-on-programmable-chips, in: FPT, 2011, pp. 1–8.
[24] P. Ostler, M. Caffrey, D. Gibelyou, P. Graham, K. Morgan, B. Pratt, H. Quinn, M. Wirthlin, SRAM FPGA reliability analysis for harsh radiation environments, IEEE Trans. Nucl. Sci. 56 (6) (2009) 3519–3526.
[25] E. Cetin, O. Diessel, L. Gong, V. Lai, Towards bounded error recovery time in FPGA-based TMR circuits using dynamic partial reconfiguration, in: Field

Programmable Logic and Applications (FPL), 2013 23rd International Conference on, IEEE, 2013, pp. 1–4.
[26] B. Pratt, M. Caffrey, P. Graham, K. Morgan, M. Wirthlin, Improving FPGA design robustness with partial TMR, in: IRPS, 2006, pp. 226–232.
[27] B. Pratt, M. Caffrey, J. Carroll, P. Graham, K. Morgan, M. Wirthlin, Fine-grain seu mitigation for fpgas using partial tmr, IEEE Trans. Nucl. Sci. 55 (4) (2008) 2274–2280.
[28] A. Tylka, J. Adams Jr, P. Boberg, B. Brownstein, W. Dietrich, E. Flueckiger, E. Petersen, M. Shea, D. Smart, E. Smith, CREME96: a revision of the cosmic ray effects on micro-electronics code, IEEE Trans. Nucl. Sci. 44 (6) (1997) 2150–2160.
[29] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, B. Hutchings, M. Wirthlin, RapidSmith: A Library for Low-level Manipulation of Partially Placed-and-Routed FPGA Designs (Tech. rep.), Dept. of Elec. and Comp. Eng., Brigham Young University, 2012.
[30] J.-Y. Lee, C.-R. Chang, N. Jing, J. Su, S. Wen, R. Wong, L. He, Heterogeneous configuration memory scrubbing for soft error mitigation in FPGAs, in: ICFPT, 2012, pp. 23–28.
[31] C. Beckhoff, D. Koch, J. Torresen, Migrating static systems to partially reconfigurable systems on Spartan-6 FPGAs, in: IPDPSW PhD forum, 2011, pp. 212–219.
[32] Xilinx Inc., Virtex-II Platform FPGA User Guide, 2007.
[33] J. Hussein, G. Swift, Mitigating Single-Event Upsets (Tech. rep.), Xilinx Inc., 2012.
[34] R. Le, Application Note XAPP538: Soft Error Mitigation Using Prioritized Essential Bits (Tech. rep.), Xilinx Inc., 2012.
[35] F. Lima, C. Carmichael, J. Fabula, R. Padovani, R. Reis, A fault injection analysis of virtex fpga tmr design methodology, in: Radiation and its Effects on Components and Systems, 2001. 6th European Conference on, IEEE, 2001, pp. 275–282.
[36] M. Alderighi, F. Casini, S. D'Angelo, M. Mancini, S. Pastore, G.R. Sechi, Evaluation of single event upset mitigation schemes for sram based fpgas using the flipper fault injection platform, in: Defect and Fault-Tolerance in VLSI Systems, 2007. DFT'07. 22nd IEEE International Symposium on, IEEE, 2007, pp. 105–113.
[37] L. Sterpone, M. Violante, A new partial reconfiguration-based fault-injection system to evaluate seu effects in sram-based fpgas, IEEE Trans. Nucl. Sci. 54 (4) (2007) 965–970.
[38] A. Mohammadi, M. Ebrahimi, A. Ejlali, S.G. Miremadi, Scfit: a fpga-based fault injection technique for seu fault model, in: Proceedings of the Conference on Design, Automation and Test in Europe, EDA Consortium, 2012, pp. 586–589.
[39] K. Minkovich Kirill Minkovich's Home Page. ⟨http://cadlab.cs.ucla.edu/kirill/⟩ [link]. URL ⟨http://cadlab.cs.ucla.edu/~kirill/⟩.
[40] openCores. ⟨http://www.openCores.com⟩ [link]. URL ⟨http://www.openCores.com⟩.
[41] D. Koch, J. Torresen, Routing optimizations for component-based system design and partial run-time reconfiguration on FPGAs, in: Proceedings of the 2010 International Conference on Field-Programmable Technology, 2010, pp. 460–464. ⟨http://dx.doi.org/10.1109/FPT.2010.5681459⟩.
[42] L. Sterpone, M. Violante, A new analytical approach to estimate the effects of seus in tmr architectures implemented through sram-based fpgas, IEEE Trans. Nucl. Sci. 52 (6) (2005) 2217–2223.