Context-Aware Resources Placement for SRAM-based FPGA to minimize Checkpoint/Recovery overhead

Sahraoui Fouad*, Fakhreddine Ghaffari*, Mohamed El Amine Benkhelifa* and Bertrand Granado‡
*ETIS, CNRS UMR 8051, ENSEA, Université Cergy-Pontoise; FRANCE
‡ LIP6, UPMC, CNRS UMR 7606; FRANCE

***Abstract*—Existing SRAM-based Field Programmable Gate Arrays (FPGAs) are very sensitive to Single Event Effects (SEE) phenomena in harsh environments. To protect applications running on SRAM-based FPGAs from SEE, those applications mainly relay on resources redundancy approaches, which involve significant resources overhead. New proposed fault mitigation approaches use Partial Dynamic Reconfiguration to overcome such huge overhead of redundancy methods. In [1] a Backward Error Recovery (BER) approach based on Partial Dynamic Reconfiguration (PDR) is proposed. Nevertheless, such approach suffers greatly from time latency issue. In this paper, we introduce a new context-aware resources placement strategy to minimize the time overhead induced by the BER fault mitigation approach. Both of checkpoint and recovery overhead are evaluated with and without our context-aware resources placement strategy. A reduction of up to 71% of context frame is reported.**

Keywords: Fault Tolerance, SRAM-based FPGA, Reliability, Resources Placement, Backward Error Recovery.

I. Introduction

Over the past years, SRAM-based Field Programmable Gate Arrays (FPGA) showed a significant progress on advancing from prototyping platforms to execution platforms. This progress mainly came from their attractive features, such as high resources availability, high speed of execution and complete/partial reconfiguration capability. The use of SRAM technology to store the configuration data, called bitstream, was also a key feature for this advance. Bitstream configures functional elements such as Look-Up Table (LUT), Flip-Flop (FF), Block RAM (BRAM), Digital Signal Processor (DSP) and the routing wires/matrices that link those elements together. Despite this progress, SRAM-based FPGAs are still very sensitive to fault occurrences and this can be a limitation to their widespread use in harsh environments and critical-safety domains such as aerospace and avionics. This weakness can be especially important at configuration memory which represents more than 80% of the total memory inside FPGA (the rest 20% is BRAMs and FFs), and can lead to erroneous executions and wrong behaviors of the system.

When integrated circuits, such as FPGAs, are exposed to cosmic rays or energetic particles, a certain perturbations of their electrical behavior may occur. Those perturbations are known as Single Event Effects (SEE) and can lead to different fault models [2]. The most likely fault models in FPGAs are Single-Event Upsets (SEUs) and Multiple-Bit Upsets (MBUs).

To cope with this weakness, many Fault Tolerant (FT) methods have been proposed to enhance the reliability of FPGA-based systems [3]. A great number of those methods are based on redundancy, such as Triple Modular Redundancy (TMR) or Duplication with Comparison (DWC). When redundancy methods are used, more resources are needed to mask or detect faults. The overhead varies from 200% to 400% compared to the initial system [4], in addition to the overhead introduced by voters needed to validate intermediate/final results.

Research works in [5]–[7] tend to reduce this added overhead by finding a compromise between performance of the hardened system and the resource overhead based on other approaches. However, because redundancy methods mask only faults, they suffer from some weakness such as faults accumulation in replicas [4], MBUs affecting multiple replicas at the same time or faults occurring in voters [8].

Redundancy-alternative methods try to take advantage of Partial Dynamic Reconfiguration (PDR) feature, like Configuration Scrubbing [9], where a golden bitstream is periodically written to the SRAM memory to eliminate any eventual bit upset occurrence. Scrubbing approach is generally combined with Error Detection and Correction Codes (EDAC), the latter can provide a solution to minimize read/write overhead by localizing and correcting only the erroneous part of the bitstream.

Although those alternative methods enhance the application reliability on SRAM-based FPGA, they are inappropriate to a certain type of applications where both bitstream and context must be corrected to avoid fault propagation, such applications can be for example Evolutionary Algorithms (EA). Formally, for any application which is more effective and provides better results upon time execution (generation/iteration) needs to be protected with methods that take into account its behavior of evolution. Recent researches [10]–[13] show that FPGAs-based platforms are becoming more and more interesting to be used as execution platforms for those types of applications within uncertain and harsh environments.

In the aim to protect those types of applications against transient faults occurring into configuration layer of SRAM-based FPGA, we propose the use of Backward Error Recovery (BER) as a fault mitigation mechanism [1]. We continually verify the correctness of system configuration and save several

978-1-4799-5944-0/14/$31.00

checkpoints with minimal context. Upon fault detection, we use hardware context restoration to go back to an error-free state (checkpoint) to limit computation loss and faults accumulation/propagation.

The rest of this paper is organized as follows: Section II gives an overview of PDR-based methods proposed in the literature to mitigate transient faults on FPGAs. In Section III, we present our fault mitigation approach based on Backward Error Recovery (BER) and Partial Dynamic Reconfiguration. Section IV details our new Context-Aware Resources Placement algorithm (CARP) which reduces the number of context frame and so minimizes the overhead induced by checkpoint/recovery. Section V gives results on both reliability controller and CARP algorithm. Finally, section VI summarizes our contribution and discusses future works.

II. Related Works

Mitigation of transient faults by hardware redundancy have been improved in numerous research works. New approaches are proposed to provide a reasonable trade-off between the resources utilization, the circuit operating frequency, the power consumption and the reliability improvement.

In [14], mitigation by hardware redundancy is only applied on parts of the circuit that are classified as persistent (i.e. part of the circuit that causes functional errors when upsets occur on it), while the rest of the circuit is only protected by scrubbing. However, determining persistent bits can be time consuming and unreliable [15]. Another approach presented in [16] introduces an extra step in the classical design flow, where after a successful place-and-route of the design, an enumeration of not-fully occupied LUTs is conducted on the netlist. These LUTs are replaced by a more robust and functionally equivalent LUTs to reduce fault effects and their propagation. Despite the low resources overheard and path delay overhead reported in this approach, it can only protect LUT elements and needs another mitigation method for routing elements.

Adaptive approaches based on environmental information are also investigated. In [17], a reliability controller based on PDR-feature is presented. At run-time, the controller determines the number of present replicas following the current orbital position changes to provide different levels of faults mitigation. With this adaptive approach, more resources are available to execute other non-critical tasks while the system is at low radiation orbit. A hard condition imposed by this method is that protected system must perform significant change of its altitude position (variation of failure rate).

As mentioned previously, hardware redundancy methods mitigate faults by masking them with majority votes of intermediate/final results and still need correction methods to eliminate those faults accumulated into the configuration memory. A well-known approach to conduct such a correction is using Configuration Scrubbing. Depending on the access port used to read/write configuration memory, scrubbing can be performed externally or internally [18]. Moreover, it can be performed periodically or only after an error is detected.

Taking into account that, the configuration layer of an SRAM-based FPGA is partitioned into frames, where a frame is the smallest portion of the configuration memory that can be read or written. Information redundancy approaches are used to protect those frames by appending additional parity bits. In [19], an embedded self-test core is presented. It uses parity bits in conjunction with DPR feature to continually monitor the status of the configuration layer and corrects bit-flips when they occur. However, because of the use of extended hamming code (EDAC) [19], only single error correction and double errors detection (SECDED) are achieved. Moreover, results reported by [18] show clearly that relying only on SECDED code is not sufficient to mitigate faults, even worse it can lead to the introduction of errors upon false detection of upsets. Authors in [20] propose a fully hardware internal scrubber and they evaluate its performances against two soft-core-based scrubbers. Nevertheless, this solution also relays on EDAC.

Heterogeneous configuration memory scrubbing is proposed in [21] and is compared to classical homogeneous scrubbing approaches like [18], [22]. It shows an improvement in Mean-Time-To-Failure (MTTF) when scrubbing is focused only on critical parts of the mapped application into the configuration memory. However scrubbing is still limited by the fact that it only cleans configuration memory from faults and can leave eventual errors in results which already propagated between the moment that the fault occurs and the moment it is detected and removed.

III. Backward Error Recovery Using Partial Dynamic Reconfiguration

An application implemented on FPGA is assumed to be a set of tasks, where each one is represented by a number of operations applied on a predetermined block of input data to produce a block of processed data. The task can be a hardware module (IP: Intellectual Property) which is placed on Enhanced Reliability Region (ERR) and executed directly by the FPGA [1] or it can be a soft-core processor running an application and is executed on the FPGA inside an ERR (here the soft-core processor is considered as a special IP).
We adopt a firm real-time model, where a task T_i is characterized by a work that needs C_i unit of time to be accomplished before a deadline D_i. The task T_i has I_i bits of input data and produce O_i bits of output processed data.

A task is represented, in addition, by:

- n_{clb_i} the number of configurable logic blocs (CLB) needed to implement task T_i,
- $nf_{critical_i}$ the number of critical frame that contains at least one critical bit (a critical bit is a bit which if its initial value changes an error occurs in the corresponding task [23]),
- $nf_{context_i}$ the number of context frame (a context frame contains one or more bits that store FFs or LUTRAMs contents of the corresponding task).

The fault tolerance approach proposed in this work is divided into three components : Fault Detection (FD), Hardware

Checkpoint (HC) and Hardware Recovery (HR). Each component uses PDR-feature to achieve its operations. To protect a task against transient faults, the task is placed into an Enhanced Reliability Region (ERR). A Reliability Controller (RC) is added to the system and it is hosted into a static region [1]. An ERR is a normal partial dynamic region protected by the RC. However this latter can be assigned to protect multiple ERRs by performing the following operations on each one separately. Figure 1 presents an example of multiples ERRs on a Virtex-5 FPGA.

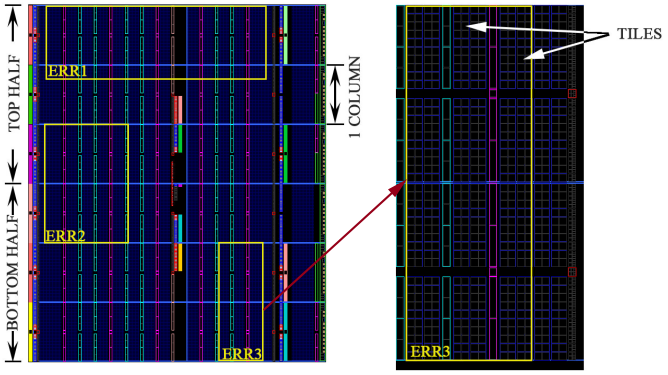


Fig. 1. Enhanced Reliability Regions illustration on a Virtex-5 SX50T layout.

A. Fault Detection (FD)

As stated before, using Error Detection and Correction Codes, only one fault can be corrected and up to two faults can be detected per frame. In our reliability controller, we use only the detection capability of the available EDAC in each frame to activate the Recovery action. We avoid its correction capability due to the potential risk of fault introduction [18]. Each frame of the ERR contains a parity bits calculated at the bitstream generation step. At run-time, those bits are used together with a hardwired circuit *"FRAME_ECC"* to check the correctness of the data stored in that frame at each read cycle. To solve the problem of odd number of faults (> 2) reported in [18], the granularity of our detection is set to a frame, which means that a detection of at least one faulty bit in a frame makes the hole frame completely faulty and induces a recovery action of all the corresponding ERR. The fault detection is performed continuously in multitasking with the tasks execution until a fault is detected or a hardware checkpoint is required, hence no delay is introduced in the task workload.

However, the latency of FD influences the checkpoint frequency and location. A worst scenario of fault detection latency would be a fault occurring on the previous frame which was reported as correct. This latency is estimated by:

$$Latency_{detection} = t_{frame_{scan}} \times nf_{scan}, \quad (1)$$

where $t_{frame_{scan}} = t_{frame_{read}} + t_{frame_{check}}$ is the time required to readback a frame from configuration memory and to verify the parity bits, and nf_{scan} is the number of frames to verify before reaching the faulty frame.

B. Hardware Checkpoint (HC)

Backward Error Recovery (BER) tends to return a task to a free-fault state when a fault is detected. To always satisfy a free-fault state, BER manages the creation and restoration of task context. This state creation is called checkpoint and can be periodic or sporadic depending on the FT strategy adopted. Because of the heterogeneous nature of FPGA-based application, BER must be applied on both software and hardware tasks. A unified way to create checkpoints on FPGAs is the use of PDR [24], the context of each task is retrieved from the configuration layer of the FPGA by performing a readback capture of the corresponding ERR.
A readback capture is a normal readback of bitstream [25] using PDR-feature preceded by a capture sequence (GCAPTURE) of FFs/LUTRAMs contents belonging to the target ERR. This distributed information is localized on a set of frames, which we call *context frames*, the number of those frames is given by $nf_{context}$ and can be computed from the produced files after the bitstream generation (Xilinx file with extension .ll) [26]. $nf_{context} \leq nf_{total}$ is always true, where nf_{total} represents the total number of frames that configures a task's ERR.

Hardware checkpoint is also performed by the RC in multitasking with the execution of tasks and with a frequency equal to f_c. No preemption of tasks is conducted when the capture sequence of FFs or LUTRAMs is triggered nor the readback of frames. However, the time required by the RC to perform one checkpoint of a task T_i is given by the following equation:

$$T_{checkpoint_i} = t_{frame_{read}} \times nf_{context_i}. \quad (2)$$

The $t_{frame_{read}}$ is heavily correlated to the implementation of the reconfiguration controller and the provided input clock [27].

C. Hardware Recovery (HR)

Following a fault detection, RC must halt the execution of the corresponding ERR and perform a hardware recovery using partial dynamic reconfiguration. The HR is a set of two actions, first a rewrite of task's context frames from the last checkpoint and the correct frame which was detected as faulty is performed, then a restore sequence (GRESTORE) is activated to re-initialize the content of FFs and LUTRAMs to the previously saved context of the task.

Unlike previous operations, the execution of HR induces a time overhead for the recovered task, this overhead $T_{recovry_i}$ is expressed by:

$$T_{recovry_i} = t_{frame_{write}} \times (nf_{context_i} + 1), \quad (3)$$

where $t_{frame_{write}}$ is the required time to write one frame to the configuration layer by the reconfiguration controller. Therefore, the amount of lost computation $C_{lost(i)}$ due to the recovery action in the case of periodic checkpoint is given by:

$$C_{lost(i)} = C_i \times f_c. \quad (4)$$

Figure 2 depicts two scenarios of a task execution with our reliability controller: (a) shows a normal execution where no fault is detected by FD and no overhead introduced and (b) shows the hardware recovery in action upon a fault detection. The execution of the task is stopped during the HR execution. From figure 2, it is clear that a mutual exclusion exists between

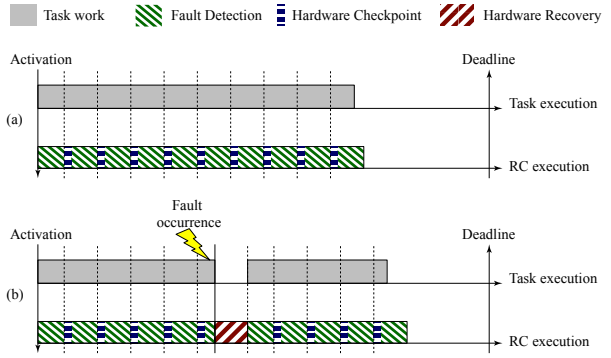


Fig. 2. Execution scenario of a task with reliability controller (a) without and (b) with fault occurrence.

the fault detection and the checkpoint creation/restoration. This exclusion is due to the fact that only one Internal Configuration Access Port (ICAP) can be used at the same time, even if there are two instances in modern FPGAs [25]. The longer HC/HR takes time to create/restore a checkpoint the longer fault detection is inactive. This means that the probability to have fault occurring and been undetected will grow proportionally with the interval of FD inactivity. In the next section, we propose an original design approach to minimize this interval of inactivity by introducing a new resources placement strategy.

IV. Context Aware Resources Placement (CARP)

Generally, the design-flow applied to generate a system-on-chip for SRAM-based FPGA is composed of three steps: synthesis, map and place-and-route. A synthesizer takes a hardware description of a circuit and translates it into an RTL design. After that, a mapper transforms this RTL design into a gate-level model (which is composed of a set of tiles). Finally, a placement algorithm is executed to assign each tile to a location on the FPGA matrix and it is followed by a routing algorithm to connect them together. However, it is not always simple to integrate new heuristics for one of those steps. This is one of the reasons for the introduction of RapidSmith framework [28] in our work, which is a set of tools and APIs dedicated to create new CADs tools for Xilinx FPGAs.

It is clear that the distribution of used frames is correlated to the placement of tiles onto the FPGA. However available placement algorithms do not take into account distribution of context frames usage as a criteria of placement. Other heuristics consider the maximum clock frequency or the low power usage as criteria. So by using RapidSmith, we were able to integrate context frame size as a new criteria when placing tiles of a given design. We were also able to compare bitstreams resulting from different placed designs and thus finding a relationship between the number of CLB column used to host the ERR and the number of context frames needed for its configuration. For example, we noticed that FFs on a Virtex-5 family hosted on the same CLB column shares two frames to store their content values on the configuration layer. Using such a type of correlation as criteria, the algorithm described in Algorithm 1, as a C-like pseudo-code, takes a technology-mapped description of the design and produces a placed technology-mapped description while minimizing the number of CLB columns.

Context-Aware Resources Placement:
Data: unplaced design: D; bounding box of EER: box
Result: placed design: D'
$L_{Shape} \leftarrow ShapesTiles(D)$;
$L_{Context} \leftarrow ContextTiles(D)$;
$L_{Logic} \leftarrow LogicTiles(D)$;
for $i \in \{$ *Shape, Context, Logic*$\}$ **do**
  for *tile* $\in L_i$ **do**
    currentColumn $\leftarrow$ getFirstColumn(box);
    while *tile not placed* **do**
      if *currentColumn can hold tile* **then**
        Place tile on currentColumn;
      else
        currentColumn $\leftarrow$ getNextColumn(box);
      end
    end
  end
end
check unplacedTiles(D);

Algorithm 1: Context-Aware Resources Placement algorithm flow

First, the tiles are divided into three lists : *shape, context* and *logic*. A shape is formed of two or more tiles which must be placed in a certain manner otherwise the routing operation will automatically fail. An example of a placed design is given in figure 3 where the shape tiles are depicted. Context tiles are the ones that contain a used FFs/LUTRAMs, which represent locations of task context. Logic tile are neither shape tile nor context tile, for example it can be a tile that only implements a Boolean function.

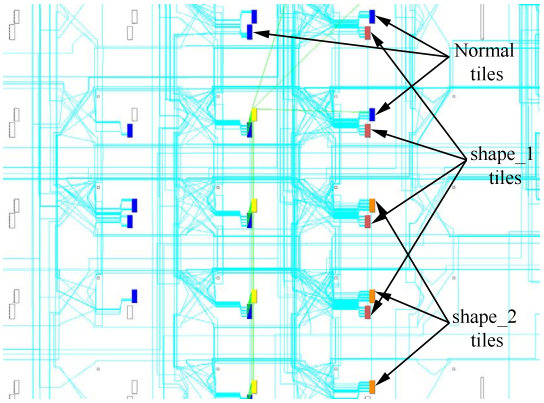


Fig. 3. Example of shapes in an FPGA matrix.

We point out that both functions "$ContextTiles$" and "$LogicTiles$", first construct the corresponding lists of tiles and then apply an optimization sort based on tiles' connectivity

to minimize the risk of an un-routed design.
Once the construction of lists is achieved, we iterate over each tile of each list and search for a free place on the available CLB column in the bounding box of the considered ERR. We begin by placing the shapes list first because of their placement constraints, then contexts list and finally logics list.

Each time a tile is successfully placed, the "currentColumn" variable is reinitialized to the first column of the bounding box. This allows minimization of total number of CLB's column when the placement algorithm finishes.

V. Experimental results

Our experiments have been conducted on ML506 board which incorporates Virtex-5 FPGA. The detection, checkpoint and recovery were implemented on a MicroBlaze-based SoC platform with a fully 32-bits hardware implementation of a reconfiguration controller. This later use the ICAP at a frequency of 100Mhz. The time needed for a read/write operation of one frame is: $t_{frame_{read}} = t_{frame_{write}} = 1.8\mu s$.

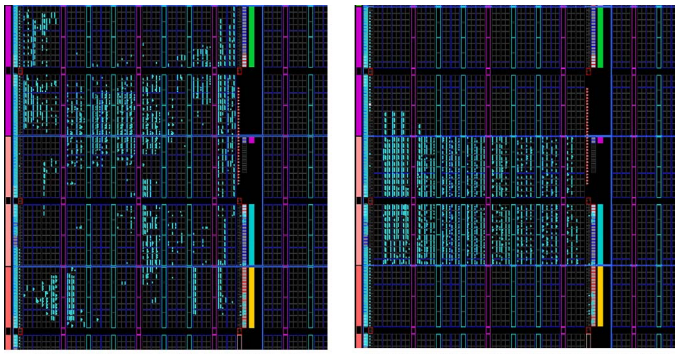
Further, to reduce the overhead of checkpoint/recovery, we evaluated our proposed context-aware resources placement strategy on the ITC99 Benchmark (b01,b02,...) and an AES algorithm on a PicoBlaze soft-core (PicoAES), both context size and tile usage are listed on Table I for each benchmarks.

TABLE I
Context size and tiles usage of considered Benchmarks.

Circuit	Context size (bits)	#of shapes	#of shape tile	#of context tile	#of logic tile
b01	8	0	0	2	1
b02	3	0	0	1	1
b03	31	0	0	10	2
b04	67	2	4	18	6
b05	36	5	14	17	35
b06	3	0	0	1	3
b07	43	2	4	14	7
b08	9	0	0	4	3
b09	28	0	0	10	0
b10	20	0	0	11	0
b11	34	1	3	15	8
b12	141	0	0	65	55
b13	43	1	3	14	4
b14	179	14	88	68	162
b15	417	19	142	161	359
b17	1323	57	414	538	699
b18	2803	128	887	1203	2031
b19	5579	263	1816	2322	3129
b20	431	31	194	191	328
b21	432	33	207	179	320
b22	617	49	305	274	547
PicoAES	4374	12	22	84	26

Using RapidSmith, the classical design-flow described early is executed until the place and route step, where the NCD file of the design is converted to an XDL file [29] and placement of tiles is performed through RapidSmith API. Finally, the placed XDL is converted back to an NCD file and the design is routed and the bitstream is generated.

We implemented the algorithm 1 as a Java application that take an XDL description of the design and produces a placed netlist with an optimal number of context frame. To evaluate the gain of our placement strategy, we compared its results with the placement produced by Xilinx Place and Route (PAR) tool. Figure 4 gives resources placement of b15 benchmark with and without CARP and Table II summarizes the results obtained for the number of context frames and both number of critical frames and critical bits.



(1) XILINX PLACEMENT (PAR) (2) CONTEXT AWARE RESOURCES PLACEMENT (CARP)

Fig. 4. Resources Placement for b15.

TABLE II
Summary of implementation results for Xilinx (PAR) and Context-aware resources placement (CARP).

	Xilinx (PAR)			Context-aware resources (CARP)		
Circuit	#of context frame	#of critical frame	#of critical bit	#of context frame	#of critical frame	#of critical bit
b01	1	75	2053	2	229	2913
b02	2	73	942	2	163	1506
b03	8	236	7692	2	211	9399
b04	8	222	17483	2	240	21693
b05	8	878	32273	2	550	39571
b06	2	73	2120	2	132	3334
b07	4	121	14511	2	326	17390
b08	2	83	5112	2	458	7646
b09	4	99	5729	2	175	6500
b10	2	104	7754	2	260	10380
b11	6	169	16931	2	598	20331
b13	4	128	10248	2	579	14365
b12	21	623	47151	8	815	68444
b14	26	1223	197298	8	1562	270331
b15	67	2617	357090	19	1927	497782
b17	131	3879	1033822	60	4170	1797585
b18	293	8993	2234146	293	8993	2234146
b19	392	9180	4493610	392	9180	4493610
b20	57	2431	444563	22	2183	641227
b21	60	2620	437615	20	2176	613314
b22	89	3242	665340	33	2604	991980
PicoAES	59	657	48709	14	1086	71671

We can notice here that for benchmarks that have small context size (b01-b11,b13), our placement strategy will reduce the number of context frame with a huge augmentation of critical frames and critical bits. This overhead of critical bits is mainly due to inputs/outputs locations, where the bounding box was not always located nearby I/O, inducing an augmentation of used wires and so the size of critical bits. However for other benchmarks, it is clear that context-aware resources placement strategy minimizes the number of context and critical frames with a small overhead of the total number of critical bits (mainly b15, b20, b21, b22). This reduction of critical frames is beneficial for the fault detection latency.

We noticed also that our new placement algorithm (CARP) cannot find better placement which can be routed and imple-

mented for b18 and b19 due to their huge tile usage.
Moreover, the maximum frequency for each benchmark is exactly identical for both PAR and CARP strategy.

TABLE III
TIME OVERHEAD REDUCTION IN BER USING CARP.

Circuit	FD	HC/HR
b15	26.37%	71.64%
b20	10.20%	61.40%
b21	16.95%	66.67%
b22	19.68%	62.92%

Table III gives percentage of time overhead reduction due to the use of our new context-aware resources placement with Backward Error Recovery fault tolerant approach. For example the fault detection latency of the circuit b22 ($Latency_{detection} = 5.83ms$) is reduced by more than 19% ($4.68ms$) when using CARP. Similarly, $T_{checkpoint_i} = 160.2\mu s$ and $T_{recovry_i} = 162\mu s$ are reduced by more than 62% ($T_{checkpoint_i} = 25.2\mu s$ and $T_{recovry_i} = 27\mu s$).

VI. CONCLUSION AND FUTURE WORK

In this work, we presented a new resources placement strategy which reduces significantly the time overhead of our fault mitigation approach. This later uses Partial Dynamic Reconfiguration to protect SRAM-based applications against transient faults. First, the time overhead of such an approach is evaluated and then optimized using a new Context-Aware Resources Placement algorithm (CARP). The CARP approach is implemented and evaluated against Xilinx Place and Route (PAR) tool. Results show a reduction of both context frames and critical frames, which subsequently reduces the execution time for fault detection, hardware checkpoint and hardware recovery. In our future work, we target to integrate the routing strategy to the Context-Aware Resources Placement to reduce both the size of context frames and the size of critical frames.

REFERENCES

[1] F. Sahraoui, F. Ghaffari, M. El Amine Benkhelifa, and B. Granado. An efficient ber-based reliability method for sram-based fpga. In *Design and Test Symposium (IDT), 2013 8th International*, pages 1–6.

[2] C. Bolchini and C. Sandionigi. Fault classification for sram-based fpgas in the space environment for fault mitigation. *Embedded Systems Letters, IEEE*, 2(4):107–110, 2010.

[3] Jason A. Cheatham, John M. Emmert, and Stan Baumgart. A survey of fault tolerant methodologies for fpgas. *ACM Trans. Des. Autom. Electron. Syst.*, 11(2):501–533, 2006.

[4] F.L. Kastensmidt, L. Carro, and R.A. da Luz Reis. *Fault-tolerance Techniques for SRAM-based FPGAs*, volume 32. Springer Verlag, 2006.

[5] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin. Improving fpga design robustness with partial tmr. In *Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International*, pages 226–232.

[6] S. Yousuf, A. Jacobs, and A. Gordon-Ross. Partially reconfigurable system-on-chips for adaptive fault tolerance. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–8.

[7] A. Ilias, K. Papadimitriou, and A. Dollas. Combining duplication, partial reconfiguration and software for on-line error diagnosis and recovery in sram-based fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 73–76.

[8] H. Quinn, K. Morgan, P. Graham, J. Krone, M. Caffrey, and K. Lundgreen. Domain crossing errors: Limitations on single device triple-modular redundancy circuits in xilinx fpgas. *Nuclear Science, IEEE Transactions on*, 54(6):2037–2043, 2007.

[9] M. Lanuzza, P. Zicari, F. Frustaci, S. Perri, and P. Corsonello. Exploiting self-reconfiguration capability to improve sram-based fpga robustness in space and avionics applications. *ACM Trans. Reconfigurable Technol. Syst.*, 4(1):1–22, 2010.

[10] J. Kok, L. F. Gonzalez, and N. Kelson. Fpga implementation of an evolutionary algorithm for autonomous unmanned aerial vehicle on-board path planning. *Evolutionary Computation, IEEE Transactions on*, 17(2):272–281, 2013.

[11] K. Asami, H. Hagiwara, and M. Komori. Visual navigation system based on evolutionary computation on fpga for patrol service robot. In *Consumer Electronics (GCCE), 2012 IEEE 1st Global Conference on*, pages 295–298.

[12] A. Grunewald, S. Hardt, M. Mielke, and R. Bruck. A decentralized charge management for electric vehicles using a genetic algorithm: Case study and proof-of-concept in java and fpga. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–7.

[13] François C. Allaire, Mohamed Tarbouchi, Gilles Labonté, and Giovanni Fusina. Fpga implementation of genetic algorithm for uav real-time path planning. *J. Intell. Robotics Syst.*, 54(1-3):495–510, 2009.

[14] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin. Improving fpga design robustness with partial tmr. In *Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International*, pages 226–232.

[15] K. Morgan, M. Caffrey, P. Graham, E. Johnson, B. Pratt, and M. Wirthlin. Seu-induced persistent error propagation in fpgas. *Nuclear Science, IEEE Transactions on*, 52(6):2438–2445, 2005.

[16] Huang Keheng, Hu Yu, Li Xiaowei, Hua Gengxin, Liu Hongjin, and Liu Bo. Exploiting free lut entries to mitigate soft errors in sram-based fpgas. In *Test Symposium (ATS), 2011 20th Asian*, pages 438–443.

[17] S. Yousuf, A. Jacobs, and A. Gordon-Ross. Partially reconfigurable system-on-chips for adaptive fault tolerance. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–8.

[18] M. Berg, C. Poivey, D. Petrick, D. Espinosa, A. Lesea, K. A. LaBel, M. Friendlich, H. Kim, and A. Phan. Effectiveness of internal versus external seu scrubbing mitigation strategies in a xilinx fpga: Design, test, and analysis. *Nuclear Science, IEEE Transactions on*, 55(4):2259–2266, 2008.

[19] B. Dutton and C. Stroud. Built-in self-test of embedded seu detection cores in virtex-4 and virtex-5 fpgas. In *International Conference on Embedded Systems and Applications*, pages 149–155.

[20] U. Legat, A. Biasizzo, and F. Novak. Seu recovery mechanism for sram-based fpgas. *Nuclear Science, IEEE Transactions on*, 59(5):2562–2571, 2012.

[21] Lee Ju-Yueh, Chang Cheng-Ru, Jing Naifeng, Su Juexiao, Wen Shijie, R. Wong, and He Lei. Heterogeneous configuration memory scrubbing for soft error mitigation in fpgas. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 23–28.

[22] Ken Chapman. Seu strategies for virtex-5 devices, 2010.

[23] Ken Chapman. Virtex-5 seu critical bit information. *XAPP864*, 2010.

[24] A. G. Schmidt, Huang Bin, R. Sass, and M. French. Checkpoint/restart and beyond: Resilient high performance computing with fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 162–169.

[25] Xilinx. Virtex-5 fpga configuration user guide ug191 (v3.11).

[26] Xilinx. Command line tools user guide (ug628 v14.2), 2012.

[27] S. G. Hansen, D. Koch, and J. Torresen. High speed partial run-time reconfiguration using enhanced icap hard macro. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 174–180.

[28] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. Rapidsmith: Do-it-yourself cad tools for xilinx fpgas. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 349–355.

[29] C. Beckhoff, D. Koch, and J. Torresen. The xilinx design language (xdl): Tutorial and use cases. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pages 1–8.