Empirical Results on Parity-based Soft Error Detection with Software-based Retry

Gökçe Aydos†
†University of Bremen - Reliable Embedded Systems
Bremen Germany goekce@cs.uni-bremen.de

Goerschwin Fey†‡
‡German Aerospace Center - Institute of Space Systems
Bremen Germany goerschwin.fey@dlr.de

***Abstract*—Local triple modular redundancy (LTMR) is often the first choice to harden a flash-based FPGA application against soft errors in space. In this work, we compare parity-based error detection with software-based retry, and LTMR on a reference architecture regarding maximum frequency, area overhead and processing time. Our results show that our solution based on parity-based error-detection saves from 30 % up to 45 % of the area overhead caused by LTMR.**

I. Introduction

Field-programmable gate arrays (FPGAs) are often utilized in space avionics. The avionics must be protected from ionizing radiation in space. In the absence of a shield (e.g., magnetic field of the earth), a high energy particle can traverse through a digital circuit and induce significant amount of charge, which can cause soft errors. These errors are not permanent and can be corrected e.g., with a reset. In flash-based FPGAs, soft errors mainly happen in the flip-flops (FFs) of an FPGA application in form of bitflips. The FPGA configuration bits do not have to be protected, because flash memory has a negligible soft error rate.

The state-of-the-art solution for flash-based FPGAs is *local triple modular redundancy* (LTMR), i.e., triplicating the application FFs and voting their outputs. Unfortunately, triplication has a significant area overhead. Alternatively, a part of the space redundancy in the FPGA may be eliminated by implementing additional time redundancy, e.g., in software, if the FPGA acts as a co-unit beside an already radiation-hardened processor. An example architecture is depicted in Fig. 1, where the FPGA implements the communication protocol interfaces needed for communicating with the satellite subsystems and the processor runs the mission software. The FPGA circuit which has to be hardened, only implements error detection. In case of an error, this circuit is functionally isolated and the software instructs the circuit to reprocess the last request. With this collaborative approach, error correction is achieved and the overhead of local error correction is eliminated in the FPGA. This technique will be referred as *error detection with software-based retry* (EDSR). In this paper, *parity-based error detection* (PBED) is used in EDSR.

Parity-based codes and *triplication* are well-known *concurrent error detection* techniques (CED) [1],[2]. Also *error detection with retry* for achieving error correction was proposed, e.g., in [3]. In recent years, one the one hand, partial hardening techniques were proposed due to the relatively high overhead of CED techniques, which selectively harden susceptible parts of the circuit [4]. On the other hand, software-based fault-tolerance techniques are also popular due to the flexibility and relatively loose constraints of software, e.g., regarding memory requirements, compared to hardware [5],[6]. Software- and hardware-based techniques have their tradeoffs, therefore these can also be used together [5].

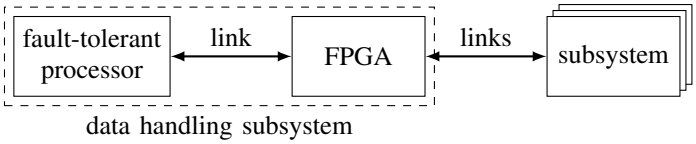

Fig. 1. Overview of the reference data handling system

This work applies parity-based EDSR on an example data handling architecture based on a commercially-available flash-based FPGA and provides an experimental comparison to LTMR. Up to now, there is no detailed comparison based on a state-of-the-art (e.g., [7]) flash-based FPGA. Due to the limited resources of space-proven flash-based FPGAs, area savings can be the key for fitting the application onto the FPGA. Our contributions are

- EDSR in the context of the full system stack including the discussion of requirements for the application and
- empirical comparison of LTMR versus EDSR for circuit area overhead, maximum circuit frequency, and overall system latency due to error correction on a representative system in space-proven technology.

In the following sections, we firstly present the reference data processing system, which is used as the testbench. Then, we explain LTMR and EDSR and the implementations which are compared. Afterwards, experimental results based on a known flash-based FPGA are presented.

II. Reference Architecture

We use a reference model of an on-board data handling unit (OBDH) for satellites [7] for our analysis. In this section, we describe an overview of the system, the FPGA design, and the communication protocol between the processor and the FPGA.

A. Overview

Fig. 1 shows an overview of the architecture. OBDH comprises of two main processing modules: a processor and an FPGA. The processor runs the mission software, which involves communicating with different subsystems on-board

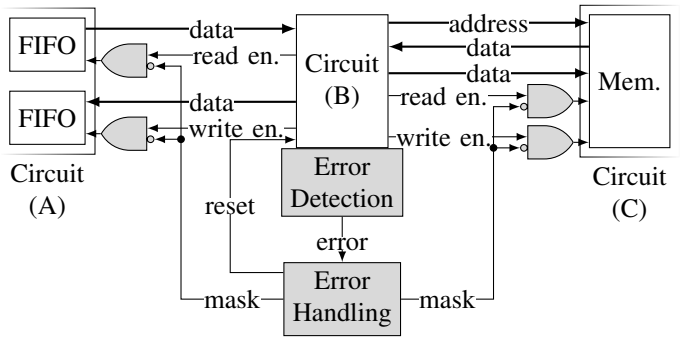

Fig. 2. Excerpt from the FPGA design. Circuit (B) is hardened by PBED using the gray components. Other circuits are immune to soft errors.

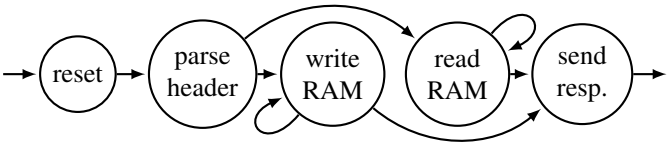

Fig. 3. Simplified state diagram of circuit B, which parses the remote memory packets sent by the mission software (i.e., the processor).

of the space system. The communication is done through the FPGA, which acts as an interface component and implements the various communication interfaces needed by the subsystems (e.g., RS232, CAN). We assume that the processor, the communication line between the processor and the FPGA, and the subsystems are sufficiently protected against soft errors.

B. FPGA Design

From the processor point of view, the FPGA is a remote memory bus, where the implemented link interfaces are memory-mapped. The processor utilizes these interface modules by reading and writing the respective memory areas.

The FPGA model consists of three functional blocks: circuit A, B, and C as shown in Fig. 2. Circuit A serves the memory access requests from the processor to circuit B, which issues memory accesses on circuit C and finally returns the data to the processor using the FIFO interface of circuit A. In Fig. 3, circuit B is shown more in detail. Circuit C with a memory block inside resembles the memory-mapped interfaces. Circuit A and C are assumed to be sufficiently protected against soft errors (e.g., by LTMR). Circuit B must be hardened.

The FIFOs and the memory need a single clock cycle for reading or writing a single word, which renders the masking a single word access operation in the same clock cycle possible.

C. Communication Protocol

The communication protocol between the processor and the FPGA is visualized in Fig. 4. It consists of two kinds of messages: *request* and *response*. The processor sends memory access requests for a specific address or address interval to the FPGA and the FPGA (more precisely, circuit B) responds with the according response: A read request is responded with read data and a write request is acknowledged after the write operation. Every request is acknowledged with a response and a second request cannot be sent before the response to the first request has been received. If the FPGA does not respond after

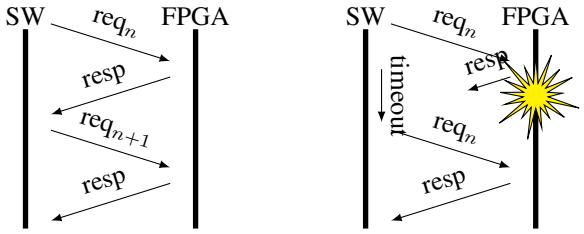

Fig. 4. Sequence diagram of the communication protocol.

a timeout, e.g., due to a soft error, the last request is repeated.

III. Compared Hardening Techniques

In this section, LTMR and EDSR, and their characteristics are discussed. EDSR's implementation on the reference system is discussed more in detail due to its system impacts.

In LTMR, one FF from the application is triplicated and the outputs of the resulting three FFs are input to a voter, which outputs the majority value. LTMR detects and corrects a bitflip on an FF locally, hence it can be automatically applied on top of a circuit. This makes LTMR functionally transparent to the rest of the system, consequently the circuit mostly does not require a redesign before mapping to an FPGA.

PBED is a well-known error detection technique, which adds a parity bit to every data word being stored, e.g., by XORing the data bits [1]. Upon reading the data word, the parity is calculated again, compared to the stored parity value and in case of a mismatch, an error signal is asserted. Subsequently, an error handler can react and initiate a recovery scheme to correct the error.

After an error, a module must be recovered to an operational state. Often, this is done by resetting the module to its initial state. This in turn leads to a loss of the processing context that must be brought back, which involves periodically backing up the processing context, i.e., checkpointing. If the processing context does not contain any information which is needed for a long time, i.e., when a module regularly falls back to a defined state after a short time period, then the overhead of checkpointing in the circuit may be eliminated by reissuing a processing request after an error. Examples for such a module are a protocol converter or a module which exchanges data between two modules after reformatting data. Reissuing a request introduces extra delays, which should be negligible if the soft error rates are low.

Fig. 2 shows PBED applied on circuit B. The *error detection* block continuously generates the data redundancy and checks the integrity of data. If an error is detected, the *error* signal is asserted and the *error handling* block immediately masks the control signals on either side of the unreliable circuit.

FFs in the unreliable circuit are segmented to groups and for each group one parity FF is introduced. One single group with a parity FF is called a *cluster*. Fig. 5 shows the generic implementation of the error detection in a single cluster. The number of clusters is given by c_{cl} (c: count, cl: cluster). Each cluster contains $s_{\mathrm{cl}} - 1$ user FFs plus one parity FF (s: size). Even parity is generated by XORing the inputs to the user FFs

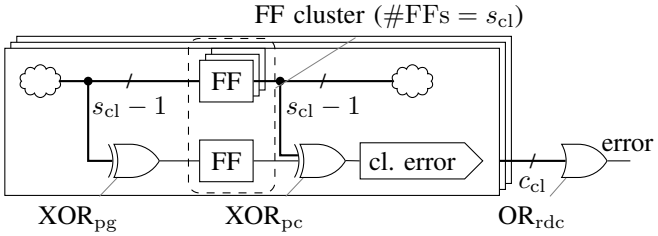

Fig. 5. Generic implementation of PBED.

by the $\mathrm{XOR_{pg}}$. The integrity of the stored bits is checked by the $\mathrm{XOR_{pc}}$ with s_{cl} inputs and the *cluster error* is generated by each cluster. Finally, c_{cl} cluster error signals are reduced to a single *error* signal by an OR gate. Error handling is done by generating the *reset* and *mask* signals using the error signal.

If an incomplete or no response is received by the processor in the timeout window, then a recovery of the software processing context depends on the state: If an error happens during processing of a read request, then this request is repeated. If an error occurs in the middle of a write transaction, the software cannot know which part of the transaction was completed and the software can synchronize itself by reading these addresses again or simply retry the last transaction. If a write to a memory location triggers an operation (e.g., transmitting a command to a subsystem), then retrying retriggers the last operation, which can be undesirable and dangerous. In case of such *action-triggering* memory locations, the software can issue single memory write operations only. This has the advantage that every atomic memory write operation is acknowledged separately and the software knows exactly which single memory operation did not succeed, avoiding an undeterminable system state. This requirement can be loosened, if a memory area is written which does not trigger an action, i.e., the output of the target system does not change after the transaction. An example is the transmit buffer of a communication interface module, where the transmit operation must be first triggered by setting a bit in a control register allowing to start a data transfer to a subsystem. In this case, the processor would first try to write the transmit payload-data to the buffer with one write request and in the subsequent request the transmission operation would be triggered using another write request.

IV. Experimental Results

We compared needed processing time for an example mission and synthesis impacts on different sizes of circuits. As circuit B, we implemented a module, which is functionally a concrete instantiation of the FSM in Fig. 3. For PBED, we chose the cluster size $s_{\mathrm{cl}} = 3$, which fits to the ProASIC architecture with three-input LUTs and should give area-efficient results. In the tested implementation, the error handling comprises of (a) masking the circuit outputs and (b) resetting the circuit. In the following, the results are shown.

A. Processing Time Penalty

To verify our PBED implementation tool and compare the runtime performance of LTMR and EDSR under injection of

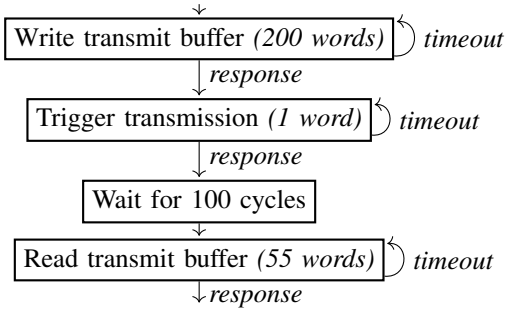

Fig. 6. Simplified flow diagram of one single memory access block. The packets are retransmitted by the software if there is no response after a timeout.

bitflips, we implemented a bitflip injection tool and a testbench which performs a mission. The mission consists of 100 memory access blocks. Each memory access block consists of three subsequent memory accesses. One single memory access block is visualized in Fig. 6. The block starts with a write transaction consisting of 200 words, which resembles data that should be sent to a subsystem by the FPGA. After the data are written, the subsystem data transmission is activated by a single word access. The subsystem responds in a predefined time window of 100 cycles. After a delay of 100 cycles, the subsystem response consisting of 55 words is read. At the end of the mission, the time needed for the whole mission is measured.

At every clock cycle, the bitflip injection tool iterates over all FFs in the target circuit and flips the FF bits according to the given probability p randomly. Probability p is defined as the bitflip probability per clock cycle for a single FF. The random numbers generated for the bitflip injection are dependent on a seed. We run the mission for $0 \leq p \leq 0.0001$, and for one single p, the simulation was run with 32 different seeds.

In LTMR, the error is corrected in the same clock cycle, but EDSR requires that the error is corrected by the software by repeating the failed memory access request, which in turn causes additional processing delays. Fig. 7 shows relative processing time needed by EDSR for the given mission. The processing time of EDSR is plotted relative to the LTMR processing time, which is constant. For PBED, the processing time increases with increasing bitflip probability p, as a failed memory access request must be repeated. The time loss due to retransmission is at least the time required to transmit the failed request. At higher p, if the bitflip rate equals to the memory access request rate, the processing time would be infinite. Therefore, the processing time grows exponentially in respect to p. Note that, at the simulated p interval, there were no undetected errors (e.g., multiple bitflips in a PBED cluster) for both techniques.

For comparison, note that, assuming one year mission in L2 orbit under 1/cm^2 shielding, a programmed circuit with 5000 FFs on a ProASIC RTPE3000L FPGA has four SEUs [8]. Assuming that this design runs at 20 MHz, then p for this mission can be calculated by:

$$p = 4/5000/365/24/60/60/(20 \times 10^6) \approx 1.3 \times 10^{-18} \quad (1)$$

Assuming the error rate from Eq. 1 makes the time penalty per year insignificant.

TABLE I
SYNTHESIS RESULTS FOR DIFFERENT SIZES OF CIRCUITS. PBED SAVES THE HARDENING OVERHEAD FROM 30.3 % UP TO 44.8 %.

c_{FF}			A			$f_{\max}$ (MHz)			t_{crit} (ns)			$t_{\mathrm{crit}+}$ (ns)		A_+		$\frac{A_+}{c_{\mathrm{FF,ba}}}$		$1-\frac{A_{+,\mathrm{PB}}}{A_{+,\mathrm{LT}}}$
ba	LT	PB	ba	LT	PB	ba	LT	PB	ba	LT	PB	LT	PB	LT	PB	LT	PB	PB
25	75	38	163	240	211	79.9	73.7	66.3	12.5	13.5	15.0	1.0	2.5	77	48	3.0	1.9	37.6 %
73	219	110	500	725	636	77.3	72.6	53.0	12.9	13.7	18.8	.8	5.9	225	136	3.0	1.8	39.5 %
121	363	182	827	1212	1050	81.2	71.1	49.0	12.3	14.0	20.3	1.7	8.0	385	223	3.1	1.8	42.0 %
169	507	254	1204	1772	1517	76.5	70.6	48.0	13.0	14.1	20.8	1.1	7.8	568	313	3.3	1.8	**44.8** %
193	579	290	1309	1901	1666	78.2	68.6	47.5	12.7	14.5	21.0	1.8	8.3	592	357	3.0	1.8	39.6 %
241	723	362	1636	2402	2094	74.9	65.6	43.9	13.3	15.2	22.7	1.9	9.4	766	458	3.1	1.9	40.2 %
289	867	434	2100	2942	2641	81.9	69.6	44.9	12.2	14.3	22.2	2.1	10.0	842	541	2.9	1.8	35.7 %
337	1011	506	2435	3452	3072	76.2	65.6	45.4	13.1	15.2	22.0	2.1	8.9	1017	637	3.0	1.8	37.3 %
385	1155	578	2837	3976	3630	79.7	66.2	43.3	12.5	15.0	23.0	2.5	10.5	1139	793	2.9	2.0	**30.3** %
433	1299	650	3085	4396	3959	76.2	61.9	41.0	13.1	16.1	24.3	3.0	11.2	1311	874	3.0	2.0	33.3 %
481	1443	722	3390	4893	4291	71.4	61.5	41.8	13.9	16.2	23.8	2.3	9.9	1503	901	3.1	1.8	40.0 %

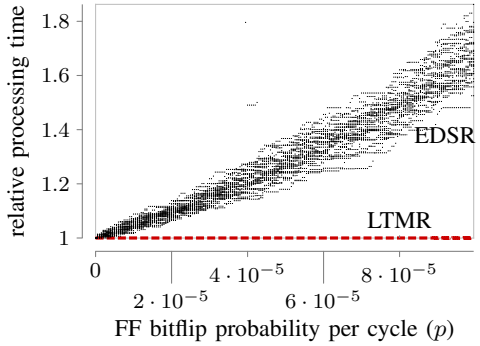

Fig. 7. Scatterplot of relative processing time for a given mission. The factor is relative to the processing time of LTMR.

B. Synthesis Results

To compare the synthesis impacts, we created circuits of different sizes by multiple instantiations of circuit B. The circuits were synthesized using the tool *Synplify* for ProASIC A3P250. LTMR and PBED were applied using Synplify and a newly-implemented tool which generates the PBED circuitry on top of an RTL design, respectively. The output netlists were then placed and routed using *Designer* from Microsemi. The results are shown in Table I. The parameters shown are: FF count (c_{FF}), circuit area (A), maximum frequency ($f_{\max}$), critical path length (t_{crit}), critical path overhead ($t_{\mathrm{crit}+}$), circuit area overhead (A_+), circuit area overhead per FF ($\frac{A_+}{c_{\mathrm{FF,ba}}}$) and redundancy saving by PBED with respect to LTMR ($1-\frac{A_{+,\mathrm{PB}}}{A_{+,\mathrm{LT}}}$). The parameters are shown for the bare (ba), LTMR applied (LT) and PBED applied (PB) circuit.

Note that in ProASIC3 architecture, every *configurable logic block* (CLB) can be either configured as an FF or LUT. Consequently, in this work, circuit area A is defined as the total count of FFs and LUTs in the circuit.

The impact of PBED on the critical path (and thus on the maximum frequency) is significant due to the synchronous reset in the error handling. PBED saves the hardening overhead from from 30.3 % up to 44.8 % compared to LTMR.

V. CONCLUSION

We applied LTMR and PBED with software-based retry on a reference architecture and experimentally compared circuit area overhead, maximum frequency and needed processing time using an example mission under fault injection. The results show that at least 30 % of the area overhead caused by the LTMR can be saved by implementing PBED and correcting the errors with time redundancy. In our implementation the impact on the critical path of the circuit is significant, but a solution based on asynchronous reset and pipelined error detection will be investigated as future work.

ACKNOWLEDGMENT

This work has been supported by the University of Bremen's Graduate School SyDe, funded by the German Excellence Initiative.

REFERENCES

[1] M. Nicolaidis and Y. Zorian, "On-line testing for VLSI - a compendium of approaches," *Journal of Electronic Testing Theory and Applications (JETTA)*, vol. 12, pp. 7–20, February 1998.

[2] M. Gössel, V. Ocheretny, E. Sogomonyan, and D. Marienfeld, *New Methods of Concurrent Checking*, ser. Frontiers In Electronic Testing. Springer Netherlands, 2008, vol. 42.

[3] M. Nicolaidis, "Time redundancy based soft-error tolerance to rescue nanometer technologies," in *17th IEEE VLSI Test Symposium*, 1999, pp. 86–94.

[4] K. Mohanram and N. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits," in *International Test Conference (ITC)*, vol. 1, Sept 2003, pp. 893–901.

[5] M. Rebaudengo, M. Reorda, M. Violante, B. Nicolescu, and R. Velazco, "Coping with SEUs/SETs in microprocessors by means of low-cost solutions: a comparison study," *IEEE Transactions on Nuclear Science*, vol. 49, no. 3, pp. 1491–1495, Jun 2002.

[6] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-implemented hardware fault tolerance*. Springer, 2006.

[7] C. J. Treudler, J.-C. Schröder, F. Greif, K. Stohlmann, G. Aydos, and G. Fey, "Scalability of a base level design for an on-board-computer for scientific missions," in *Proceedings of the Data Systems in Aerospace (DASIA) Conference*, 2014.

[8] N. Battezzati, L. Sterpone, and M. Violante, *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications*. Springer, 2011, ch. 7.