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

Gökçe Aydos[1], Goerschwin Fey[1,b]

[a] *University of Bremen, Bremen, Germany*
[b] *German Aerospace Center, Bremen, Germany*

Abstract

Local triple modular redundancy (LTMR) is often the first choice to harden the FFs of a flash-based FPGA application against radiation-induced bitflips in space, but LTMR leads to an area overhead of roughly 300%. To cope with this significant overhead, we propose an error detection based approach. 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 29% up to 36% of the area overhead caused by LTMR.

Keywords: fault-tolerance, FPGA, LTMR, parity, error-detection, retry

1. Introduction

Field-programmable gate arrays (FPGAs) are often utilized in space avionics due to their processing efficiency, reprogrammability, and extensible interface capabilities; providing flexibility for a range of mission requirements. The avionics must be protected from ionizing radiation in space. In the absence of a shield (e.g., magnetic field of the earth), high energy particles can traverse through a digital circuit and cause errors. These errors can be caused by permanently damaging the semiconductor structure or induce significant amount of charge leading to a transient voltage pulse on a net, which can eventually lead to hard or soft errors, respectively.

The most common functional transient radiation effects that happen on the gate level, which can cause a soft error, are the *single event-transient* (SET) and *-upset* (SEU). An SET can be seen as a transient voltage pulse on a circuit net. If such a change happens on a data net and then latched by a FF, this transient can lead to an upset of the FF-bit and thus to an SEU. SEUs are not permanent and can be corrected e.g., with a reset.

Fault-tolerance against SEUs can be implemented at various levels of a circuit, e.g., process- or design-level. Hardening a circuit at the design-level is referred as *radiation hardening by design* (RHBD) [1] and involves the wise use of available design elements by the designer. RHBD is preferred if the designer has merely access to a commercially available *integrated circuits* (ICs) and IC manufacture processes, respectively.

The right RHBD techniques depend on the underlying FPGA architecture. Currently, three memory architectures exist on the market dependent on how the the programmable logic is configured. These are SRAM-, flash- and antifuse-based architectures. On SRAM-based FPGAs, the circuit programming information, i.e., configuration, is stored on SRAM. SRAM has a high SEU sensitivity, therefore, compared to antifuse- and flash-based configurations, the configuration of SRAM-based parts must additionally be protected.

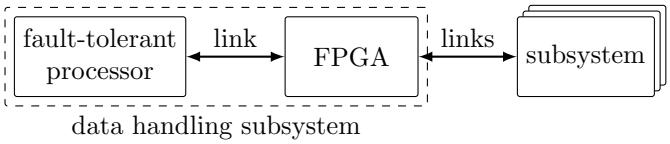

Figure 1: Overview of the reference data handling architecture. Processor communicates with the subsystems through the FPGA.

In this work, we assume a flash-based FPGA architecture. In flash-based FPGAs, SEUs mainly happen in the flip-flops (FFs) of an FPGA application. The FPGA configuration bits do not have to be protected, because flash memory has a negligible soft error rate due to SEUs.

The state-of-the-art solution against single bitflips for flash-based FPGAs is the *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 cir-

Email addresses: goekce@cs.uni-bremen.de (Gökçe Aydos), goerschwin.fey@dlr.de (Goerschwin Fey)

cuit is functionally isolated, then recovered and the software finally 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) [2],[3]. Also *error detection with retry* for achieving error correction was proposed, e.g., in [4]. 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 [5]. 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 [6],[7]. Software- and hardware-based techniques have their tradeoffs, therefore these can also be used together [6].

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., [8, 9]) 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
- fault tolerance analysis of transaction-based processing, which is an important part of EDSR
- 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 an example implementation for our approach. In Section 3, we explain LTMR and EDSR and the implementations which are compared. In Section 4, we generalize the processing approach shown in Section 3 and discuss its fault tolerance. Section 5 presents synthesis results based on a known flash-based FPGA. We end the paper with a brief conclusion.

2. Reference Architecture

We use a reference model of an on-board data handling unit (OBDH) for satellites [9] for our analysis. Using this example architecture we will explain how EDSR is implemented in particular, because LTMR is mostly architecture-independent. First, we describe an overview of the system, then the FPGA design, and finally the communication protocol between the processor and the FPGA.

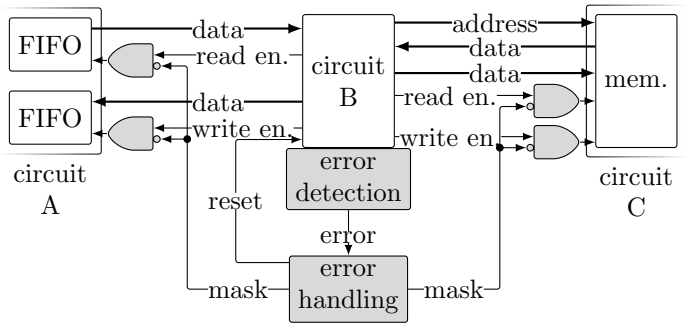

Figure 2: Excerpt from the FPGA design. Circuit B is hardened by PBED using the gray components. Other circuits are immune to soft errors.

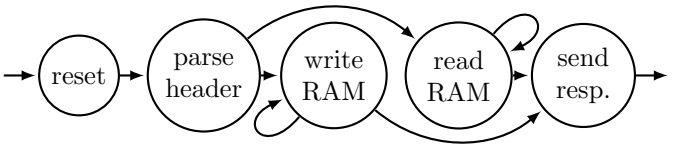

Figure 3: Simplified state diagram of circuit B, which parses the remote memory packets sent by the mission software (i.e., the processor).

2.1. 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 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.

2.2. 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 simplified FPGA model consists of three functional blocks: sequential circuits 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. The memories transfer one word per cycle. Circuit A and C including the FIFOs and RAM are assumed to be sufficiently protected against soft errors (e.g., by LTMR and error correcting and detecting code). Circuit B must be hardened by design.

The FIFOs and the memory need a single clock cycle for reading or writing a single word, which enables the

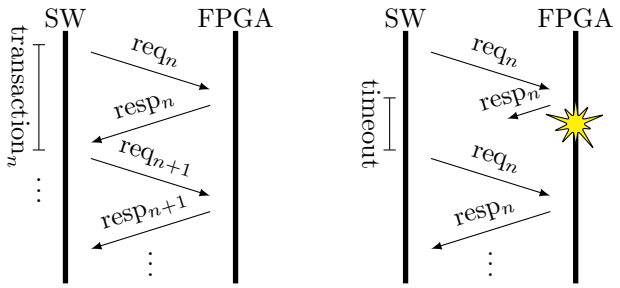

Figure 4: Sequence diagram of the communication protocol, which is based on transactions. A transaction consists of a request (*req*) and a response (*resp*). The left diagram shows a normal sequence: every request is followed by a response. On the right, the error behavior is visualized: if still no response after a timeout is received, the last transaction is repeated.

masking a single word access operation in the same clock cycle.

2.3. Communication Protocol

The communication protocol between the processor and the FPGA is visualized in Fig. 4. The protocol consists of two kinds of messages: *request* and *response*, which both make up a single transaction. The processor sends memory access requests for a specific address or address interval to the FPGA and the FPGA (more precisely, circuit B answers 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 a timeout, e.g., due to a soft error, the last request is repeated.

The communication protocol can send one word per cycle and the messages can be composed of multiple words. The validity of a single message is dependent on the last word sent. If the last word flags an error or is not present, then all the words until the last valid packet are discarded. Consequently, in case of an error, already transmitted words of a packet are discarded and the transaction fails.

3. Compared Hardening Techniques

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

3.1. Local Triple Modular Redundancy (LTMR)

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 (cf. Fig. 5). 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.

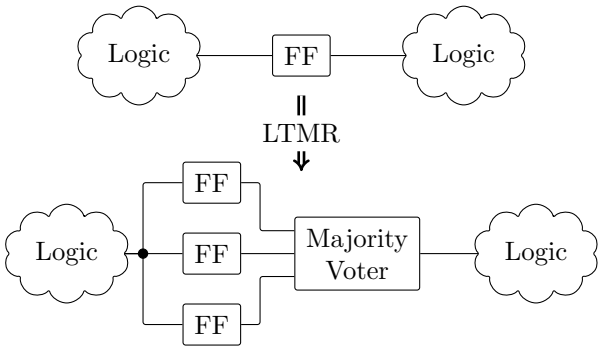

Figure 5: In LTMR, every application FF is triplicated and their output is voted. One single bitflip in one clock cycle will be corrected.

3.2. Error Detection with Software-retry (EDSR)

EDSR is based on hardware error detection and software-retry. We use PBED as the error detection technique in EDSR. First, we show the implementation details and discuss the system impacts of PBED and then software-retry.

3.2.1. Parity-based Error Detection (PBED)

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 [2]. 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, Circuit B must be recovered to an operational state. Often, this is done by resetting the circuit 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. Circuit B is also an example for a data exchanging circuit.

Reissuing a request introduces extra processing delays, which should be negligible if the soft error rates are low. This will be analyzed in Section 5.

Fig. 2 shows PBED applied on circuit B. The *error detection* block continuously generates and checks the parity. 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. 6 shows the generic implementation of the error detection in a single cluster. The number of clusters is given by c_{cl} (c: count,

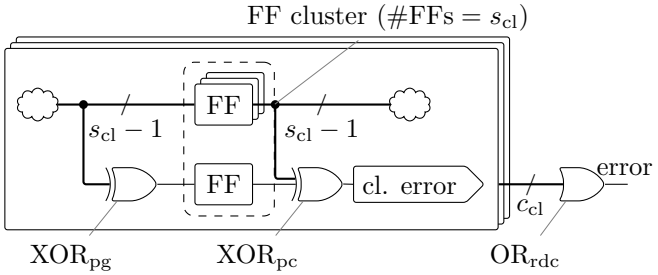

Figure 6: Generic Implementation of PBED. $\mathrm{XOR_{pg}}$ generates an even parity bit from the inputs of the FF cluster with the size of s_{cl}. $\mathrm{XOR_{pc}}$ continuously checks the integrity of the data stored in the cluster. If a bitflip in the cluster occurs, $\mathrm{XOR_{pc}}$ outputs a logical one, which indicates a cluster error. Typically, there are many clusters in a circuit. $\mathrm{OR_{rdc}}$ reduces the c_{cl} cluster error signals to a single error signal, which is fed to the error handling block. The error handling block masks the control signals of the circuit and resets the circuit.

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 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.

The mask signal deactivates the control signals (i.e., FIFO and memory control signals) of Circuit B. The reset signal recovers the circuit from a possibly erroneous state to its initial state. After recovery, the error flag is deasserted and the unreliable module begins data processing again.

The masking of the outputs allows for a rapid functional isolation of the slave, and enables a recovery window that span multiple clock cycles, which eases the timing closure of the hardened circuit. For instance, if the recovery had to take only one clock cycle, then a recovery based on a reset would be practically a synchronous reset, which can have a significant negative effect on the timing.

In this implementation, the error handler is basic and has merely a reset and mask functionality. A more advanced error handler can also notify the processor in case of an error by implementing an FSM, which can carry out a more complex recovery procedure.

3.2.2. Software-retry

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 the software application issues writes to a memory location which triggers an operation (e.g., transmitting a command to a subsystem), then retrying retriggers the last operation, which can be undesirable and dangerous. For example, given that a single address is written during one single clock cycle, assume a write operation to ten addresses. If an error is detected after the fifth address is written, the module will fall back to the reset state. As the software is not aware about the addresses which are successfully written, the software needs to synchronize itself and determine the state of the remote memory.

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 indeterminable system state.

There is no need for single cycle, 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.

4. Transaction-based Processing

In our previous example, we have shown a communication protocol based on transactions. On the example system, we achieve tolerance against SEUs by collaboration of hardware and software. The hardware detects an error, stops the transaction and the software retries the transaction. Compared to error correction on hardware, which mostly occurs in every clock cycle ensuring that an error does not cause data corruption, a bit error can lead to data corruption and hence to an unexpected loss of processing context in a circuit. To ensure deterministic data processing in this context, the processing for the mission must be carried out in smaller *chunk*s, each acknowledged by Circuit B that no corruption due to bitflips has taken place. We call this kind of handshaked data processing *transaction-based processing*.

In this section, we generalize our approach by providing a system specification, which will be then used to show that the system will not fail under the fault model that we presume.

4.1. System Specification

A *data processing circuit* (cf. Circuit B shown in Fig. 2) is a clocked circuit with internal memory which can transfer a data *word* in every clock cycle during processing. Processed data is transferred to or from a *buffer memory*. A buffer memory is for instance a random-access (RAM) or first-in first-out (FIFO) memory, like two FIFOs and the RAM shown in Fig. 2.

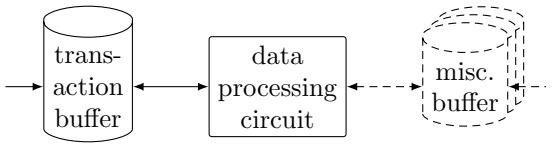

Figure 7: Data processing circuit receives a request from the transaction buffer and writes the response after processing. For communicating with other circuits, miscellaneous buffers are used.

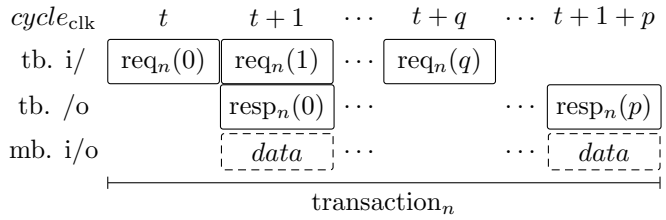

Figure 8: An example transaction visualized on cycle level. The processing circuit (cf. Fig. 7) processes request words and writes the response words back to the transaction buffer (tb.). During one clock cycle ($cycle_{clk}$), one request word ($\text{req}_n(i)$) of a request (req_n) or one response word ($\text{resp}_n(i)$) of a response (resp_n) can be transferred. During the transaction, also data transfer to/from miscellaneous buffers (mb.) is possible. i/: input, /o: output, i/o: input or output. Note that a response does not have to start at $t+1$, but may start later.

A *transaction buffer* (cf. the FIFOs in Fig. 2) is always present and used for getting processing data input and writing back the output. Other buffers can be present for communicating with other circuits (cf. RAM for memory-mapped communication interface in Fig. 2 and they are called *miscellaneous buffer*s. This generalized view on the data processing circuit is visualized in Fig. 7. All buffers are sufficiently protected against soft errors, for instance by using an error detection and correction code.

Processing data is sent by a *master* (cf. processor in Fig. 1) and the sent data is called a *request*. The data processing circuit processes the request as a *slave* and writes the output on the transaction buffer, which is called a *response*. Request and response consist of at least one or many consecutive *word*s. A request and the response to this request make up a processing *transaction*. A transaction on cycle level is visualized in Fig. 8.

A transaction fails, if the last word of the respective response is not present in the transaction buffer after a timeout. In this case the respective request is repeated. Many consecutive transactions make up a data processing *mission*.

4.2. Fault model

There are numerous effects caused by the radiation in space. SEUs and SETs are the most common functional transient radiation effects that happen on the gate level. An SET can happen on every net of a circuit and can be seen as a transient voltage pulse on a net. If such a change happens on a data net and then latched by a FF, this transient can lead to a bitflip in the FF. But an SET can also happen directly on a net inside the FF itself and possibly flip the state of the FF. An upset of the FF bit due to a single energetic particle is called an SEU.

An SET, and thus also an SEU, are asynchronous events by nature. If an SET occurs during setup or hold times of an FF, this can lead to metastability and thus to a indeterminable state of the FF. An SET can be detrimental on global nets like clock or reset but also on shared data nets. A recommended fault-tolerance strategy against an SET is to triplicate global signals or to use temporal redundancy by introducing delay elements, which introduce signal delays that are longer than the maximum duration of a voltage pulse caused by an SET and compare a net with its delayed value. On the other hand, space redundancy like LTMR is used against an SEU on FFs [10]. Consequently, a sufficient fault-tolerance strategy against functional errors should accommodate both temporal and space redundancy.

In this work, we concentrate on SEUs only which occur directly inside the FF and not on shared nets, which can cause multiple bitflips. Our fault model is based on the following assumptions:

- only SEUs happen
- SEUs happen on a discrete time domain
- SEUs happen synchronously to the circuit

Consequently:

- the faults appear as single bitflip errors
- it is not relevant where an SEU happens inside a clock cycle
- if an SEU happens during a clock cycle, then the error is only observable in the next clock cycle and subsequent cycles

4.3. Fault-tolerance Analysis

The goal of our approach is to ensure that the mission is completed without any erroneous data in the mission output. In this subsection, we show that our proposed approach meets the fault-tolerance goal. Note that data will be corrupted due to SEUs, but as long as the erroneous data do not propagate from the slave to the master or other neighboring circuits, it is not an error from the mission perspective.

If an SEU happens in a clock cycle, in the next clock cycle a bitflip in a cluster will be observable. PBED can detect this error and mask the circuit outputs in the same cycle. At the same time, the recovery is activated and the circuit is brought to a known state by a reset. As long as the circuit is in recovery, the circuit outputs stay masked. In summary, in PBED

- a bit error is detected in the next clock cycle

- a bit error cannot propagate outside the circuit and eventually cause silent data corruption

Consequently, if an error is detected during a transaction, the master will not get a response and subsequently retry the transaction without any data corruption.

A transaction succeeds or fails as a whole, but the slave processes the data on every clock cycle. Consequently, the master cannot know the state of a miscellaneous buffer after an SEU. We already discussed this problem in Subsection 3.2 and given a solution for the example system architecture. Nevertheless, the actual solution is application dependent and the master should pay attention to this behavior.

5. Experimental Results

We compared needed processing time for an example mission and synthesis results 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 ProASIC3 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.

5.1. Processing Time Penalty

To verify our PBED implementation tool and compare the runtime performance of LTMR and EDSR under injection of 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. 9. 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. 10 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.

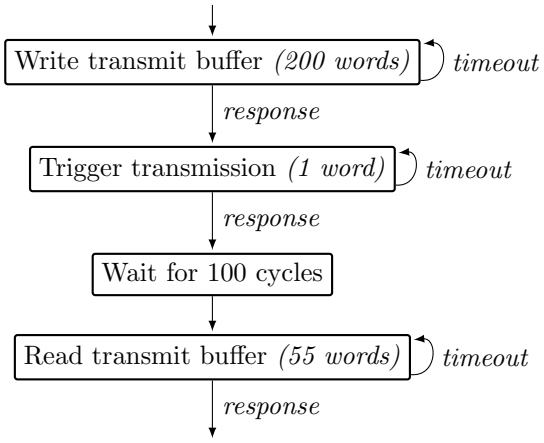

Figure 9: Simplified flow diagram of one single memory access block. It consists of three transactions. The transactions are retried by the software if there is no response after the timeout has passed.

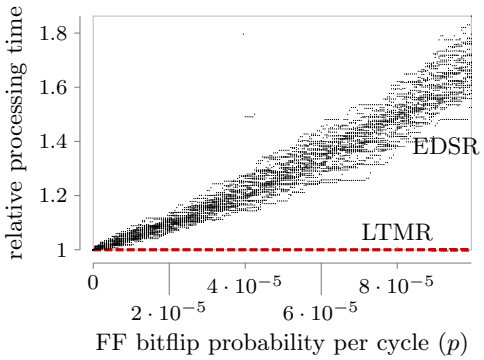

Figure 10: Scatterplot of relative processing time for a given mission. The factor is relative to the processing time of LTMR.

For comparison, note that, assuming one year mission in L2 orbit under $1/\mathrm{cm}^2$ shielding, a programmed circuit with 5000 FFs on a ProASIC RTPE3000L FPGA has four SEUs [11]. Assuming that this design runs at 20 MHz, then p for this mission is calculated by dividing the errors per year by the number of cycles in one year:

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

Assuming the error rate from Eq. 1 and transactions

Table 1: Synthesis results for different sizes of circuits. PBED saves the hardening overhead from 29% up to 35.9%.

c_{FF}			A			$f_{\max}$ (MHz)			$t_{\mathrm{crit}+}$ (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	LT	PB	LT	PB	LT	PB	PB
25	75	39	144	218	196	122.0	97.3	105.6	2.1	1.3	74	52	3.0	2.1	29.7%
49	147	75	298	443	391	116.5	96.2	99.4	1.8	1.5	145	93	3.0	1.9	**35.9%**
73	219	111	406	671	592	114.6	94.4	93.2	1.9	2.0	265	186	3.6	2.5	29.8%
97	291	147	547	898	793	114.4	95.9	92.4	1.7	2.1	351	246	3.6	2.5	29.9%
121	363	183	684	1118	987	117.4	92.2	83.7	2.3	3.4	434	303	3.6	2.5	30.2%
145	435	219	839	1352	1203	113.2	93.2	73.2	1.9	4.8	513	364	3.5	2.5	**29.0%**
169	507	255	968	1578	1388	117.4	95.0	85.6	2.0	3.2	610	420	3.6	2.5	31.1%
193	579	291	1099	1805	1587	116.8	89.2	73.2	2.6	5.1	706	488	3.7	2.5	30.9%
217	651	327	1209	1985	1748	116.3	91.8	71.0	2.3	5.5	776	539	3.6	2.5	30.5%
241	723	363	1336	2227	1934	120.7	93.3	75.8	2.4	4.9	891	598	3.7	2.5	32.9%
265	795	399	1472	2433	2140	115.5	91.8	73.0	2.2	5.0	961	668	3.6	2.5	30.5%
290	867	436	1707	2719	2383	114.9	92.5	68.5	2.1	5.9	1012	676	3.5	2.3	33.2%
314	939	472	1816	2949	2600	116.6	91.9	74.0	2.3	4.9	1133	784	3.6	2.5	30.8%
338	1011	508	1932	3172	2779	112.9	92.8	71.5	1.9	5.1	1240	847	3.7	2.5	31.7%
362	1083	544	2076	3397	2990	116.9	91.4	72.5	2.4	5.2	1321	914	3.6	2.5	30.8%
386	1155	580	2230	3625	3199	116.9	89.4	63.8	2.6	7.1	1395	969	3.6	2.5	30.5%
410	1227	616	2381	3866	3414	118.7	82.8	67.8	3.7	6.3	1485	1033	3.6	2.5	30.4%
434	1299	652	2555	4082	3620	118.8	91.1	73.3	2.6	5.2	1527	1065	3.5	2.5	30.3%
458	1371	688	2705	4344	3826	117.1	88.4	68.3	2.8	6.1	1639	1121	3.6	2.4	31.6%
482	1443	724	2857	4549	4030	114.8	88.4	67.7	2.6	6.1	1692	1173	3.5	2.4	30.7%

with a maximum length of 10^3 cycles, make the time penalty per year insignificant.

5.2. Synthesis Results

To compare the synthesis impacts, we created circuits of different sizes by multiple instantiations of circuit B and demultiplexing the output to not exhaust the input output buffers of the FPGA. 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. Compared to the results in our previous work [12], we used an asynchronous reset for the circuit and introduced a FF for the reset signal, which could alleviate the timing requirements for an immediate synchronous reset and resulted in slightly better timings. The results are shown in Table 1. 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.

For bigger circuit areas, the impact of PBED on the critical path (and thus on the maximum frequency) is mostly higher than the LTMR's. PBED reduces the hardening overhead of LTMR by 29% up to 35.9%.

6. Conclusion

Like the electronics for mission critical applications, the FPGAs used in space applications must be protected against radiation induced errors, which is often done by redundancy. LTMR is often used on FPGA designs, which can correct the induced errors locally. If the logic resources are scarce and the SEU rate on the FPGA during a mission is low, then the error correction functionality can be shifted to the software, leaving an FPGA circuit only with error detection, which we call error detection with software-based retry (EDSR).

We applied LTMR and parity-based EDSR on a reference architecture, discussed their system impacts, 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 29% of the area overhead caused by the LTMR can be saved by implementing PBED and correcting the errors with time redundancy. The impact on the critical path of the circuit is significant for bigger circuit sizes.

Acknowledgment

This work has been supported by the European Union's Horizon 2020 research and innovation program under grant agreement No 637616 (MaMMoTH-Up) and by the University of Bremen's Graduate School SyDe, funded by the German Excellence Initiative.

References

[1] J. D. Black, Best practices in radiation hardening by design: CMOS, in: J. D. Cressler, H. A. Mantooth (Eds.), Extreme Environment Electronics, CRC Press, 2013, Ch. 43.

[2] M. Nicolaidis, Y. Zorian, On-line testing for VLSI - a compendium of approaches, Journal of Electronic Testing Theory and Applications (JETTA) 12 (1998) 7–20. doi:10.1023/A:1008244815697.

[3] M. Gössel, V. Ocheretny, E. Sogomonyan, D. Marienfeld, New Methods of Concurrent Checking, Vol. 42 of Frontiers In Electronic Testing, Springer Netherlands, 2008. doi:10.1007/978-1-4020-8420-1.

[4] M. Nicolaidis, Time redundancy based soft-error tolerance to rescue nanometer technologies, in: 17th IEEE VLSI Test Symposium, 1999, pp. 86–94. doi:10.1109/VTEST.1999.766651.

[5] K. Mohanram, N. Touba, Cost-effective approach for reducing soft error failure rate in logic circuits, in: International Test Conference (ITC), Vol. 1, 2003, pp. 893–901. doi:10.1109/TEST.2003.1271075.

[6] M. Rebaudengo, M. Reorda, M. Violante, B. Nicolescu, R. Velazco, Coping with SEUs/SETs in microprocessors by means of low-cost solutions: a comparison study, IEEE Transactions on Nuclear Science 49 (3) (2002) 1491–1495. doi:10.1109/TNS.2002.1039689.

[7] O. Goloubeva, M. Rebaudengo, M. S. Reorda, M. Violante, Software-implemented hardware fault tolerance, Springer, 2006. doi:10.1007/0-387-32937-4.

[8] K. Varnavas, W. H. Sims, J. Casas, The use of field programmable gate arrays (FPGA) in small satellite communication systems, in: T. Pham, J. C. Casas, C.-P. Rückemann (Eds.), Seventh International Conference on Advances in Satellite and Space Communications (SPACOMM), 2015.

[9] C. J. Treudler, J.-C. Schröder, F. Greif, K. Stohlmann, G. Aydos, 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.

[10] M. Berg, Design for radiation effects, presentation from Military and Aerospace Programmable Logic Devices (MAPLD) Workshop (2008).

[11] N. Battezzati, L. Sterpone, M. Violante, Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications, Springer, 2011, Ch. 7. doi:10.1007/978-1-4419-7595-9_7.

[12] G. Aydos, G. Fey, Empirical results on parity-based soft error detection with software-based retry, in: Nordic Circuits and Systems Conference (NORCAS), IEEE, 2015. doi:10.1109/NORCHIP.2015.7364378.