Exploiting Error Detection Latency for Parity-based Soft Error Detection

Gökçe Aydos*
*University of Bremen
Bremen, Germany
goekce@cs.uni-bremen.de

Goerschwin Fey*‡
‡German Aerospace Center
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. Unfortunately, LTMR leads to at least 300% area overhead. We propose a parity-based error detection approach, to use the limited resources of space-proven flash-based FPGAs more area-efficiently; this method can be the key for fitting the application onto the FPGA.**

A drawback of parity-based hardening is the significant impact on the critical path. To alleviate this error detection latency, pipeline structures in the design can be utilized. According to our results, this eliminates from 22% to 65% of the critical path overhead of the unpipelined error detection. Compared with LTMR, the new approach increases the critical path overhead of LTMR by a factor varying from 2 to 7.

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., by 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.

One of the popular flash-based FPGAs is the ProASIC3 [1]. This FPGA has 130 nm feature size and a lower logic density compared to SRAM-based FPGAs in the market, but it is the state-of-the-art FPGA for space applications ([2], [3]). This FPGA does not have inherent protection against soft errors in FFs, and the standard hardening solution is *local triple modular redundancy* (LTMR). In LTMR, the application FFs are triplicated and their outputs are voted, which protects against single bitflips. LTMR is shown in Figure 1. Unfortunately, triplication has a significant area overhead of at least 300%.

Hardening a circuit against soft errors on design level is mostly done by redundancy (e.g., in LTMR by space redundancy). LTMR detects and corrects errors locally, which comes at the cost of additional space. Alternatively, a part of the area redundancy in the FPGA may be eliminated by combining space and time redundancy, such as: *only* error detection on hardware and on-demand transaction-retry (i.e., recomputation) on software, if the error rate is less than a dozen bitflips per year. In our previous work [4], we evaluated a parity-based error detection (PBED) approach to use the limited resources of space-proven flash-based FPGAs more area-efficiently, which can be the key for fitting the application onto the FPGA.

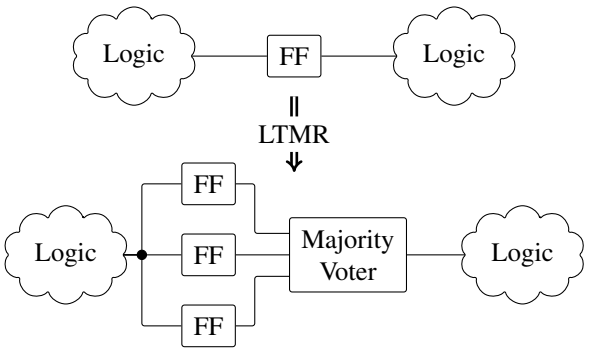

Fig. 1. Application of local triple modular redundancy on user logic

Triplication [5] and *parity-based code*, which is a *concurrent error detection* technique (CED) [6], are well-known. *Recomputation* for achieving error correction has also been proposed [7], [8].

In our previous work [4], we applied LTMR and PBED-based hardening on a reference architecture and experimentally compared circuit area overhead, maximum frequency and the needed processing time using an example mission under fault injection. The results for a fixed cluster size showed that at least 30% of the area overhead caused by LTMR can be saved by implementing PBED and correcting the errors with time redundancy. In this previous implementation the impact on the critical path of the circuit was significant. Our contributions in this work consist of:

- a pipelined error detection and handling approach to alleviate the impact of PBED on the critical path of the circuit
- the concept of a parity-based hardening tool, which implements this approach

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.

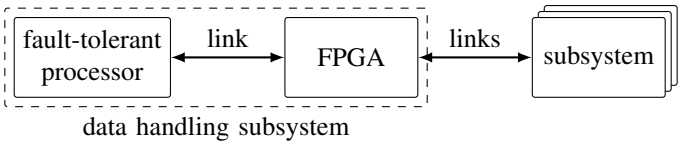

Fig. 2. Example on-board data handling architecture. FPGA must be hardened by design. Figure reused from [4].

- empirical comparison of the pipelined- with approach from our previous work [4]
- empirical comparison of PBED and LTMR using the new approach

In the following sections, we will firstly present how PBED is applied on an example processing architecture and the system requirements for applying PBED. In Section III, we introduce the pipelined error detection approach and describe how to apply it on a circuit automatically. In Section IV, we compare pipelined error detection to our previous approach and LTMR using experimental results.

II. Error detection based hardening on an example design

LTMR detects and corrects the bit error locally in the same clock cycle, but PBED detects an error on the module level and this error can live for further cycles until correction. In the latter approach, there is an *error correction latency* and this latency places additional demands on the system. These are discussed in the following example.

Figure 2 shows an on-board data handling architecture. The processor runs the mission software and uses the FPGA as an extensible interface module for communicating with the subsystems. In this architecture the processor and subsystems are assumed to be sufficiently hardened against soft errors, but the FPGA must be hardened by design.

The subsystems are manipulated via memory access, i.e., the mission software sends a data packet, which is written to a particular memory area in the FPGA. In turn, this memory access can trigger an action in the subsystem. The circuit in the FPGA for decoding the data packets are shown in Figure 3. Circuit (A) queues packets sent by the processor, circuit (B) decodes the packets and writes to respective memory addresses in circuit (C). Circuits (A) and (C) are assumed to be sufficiently hardened (e.g., by using LTMR) and circuit (B) is protected by error detection based hardening.

The gray components in Figure 3 are used for hardening the circuit (B). The error detection module implements a concurrent error detection technique (in this work, parity-based). The error handling module implements the actions that must be taken after an error is detected. Error handling has two roles:

- recovering circuit (B) from the unknown state
- isolating circuit (B) from the rest of the system while it is in an unknown state

After processing one packet, circuit (B) sends a response packet to the software and falls back to reset state. Therefore the recovery can be done by a reset.

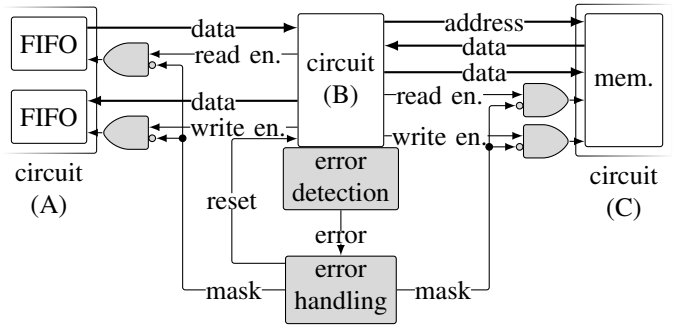

Fig. 3. Example circuit for decoding remote memory access requests. Circuit (B) is hardened using the gray components. Figure reused from [4].

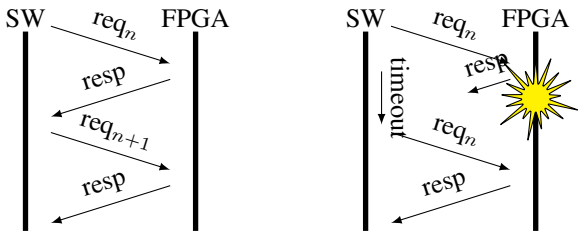

Fig. 4. Transaction-based processing and transaction-retry after a soft error in hardware. Figure reused from [4].

Sending a response is not only important for flow control but also for detecting an error. Due to the fact that we allow errors in circuit (B) that cannot be corrected immediately, the software must use transaction-based processing. This gives the software the opportunity to repeat the last processing request (i.e., resend the last packet) after a timeout, if circuit (B) cannot send any response due to a recovery event. Transaction-based processing is visualized in Figure 4.

The goal of isolation is that an error in circuit (B) does not propagate to the rest of the system. This can be achieved by masking the output signals. If the circuit interface includes control signals, further resources can be saved by only masking the control signals like write- and read-enable in Figure 3.

III. Pipelined Error Detection and Handling

In PBED, the error signal can be generated directly or by using pipelining, where the latter gives better timing results with insignificant area overhead. In this section, we explain the latter technique more in detail and show a comparison with direct error signal generation based on synthesis results. Finally, we present a toolchain to implement these techniques on an FPGA.

A. Pipelining

PBED is based on two modules: *error detection* and *error handling*. The error detection module itself consists of many smaller error detection clusters and additional circuitry which reduces all the error signals output by the clusters to a single error signal. Figure 5 shows the structure of an error detection cluster $\text{cluster}_{\text{ED}}$. One cluster consists of k application FFs FF_{a}, one parity FF FF_{p} and two XORs: one for parity

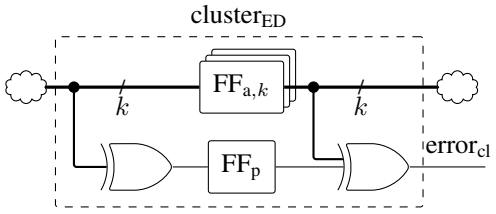

Fig. 5. Error detection ($_{\mathrm{ED}}$) cluster calculates the parity for a variable count of FFs and checks the parity in the following cycle. In case of mismatch, the cluster error signal is activated. The FF subscripts a and p denote *application* and *parity* respectively. In one cluster there are k application FFs and one parity FF.

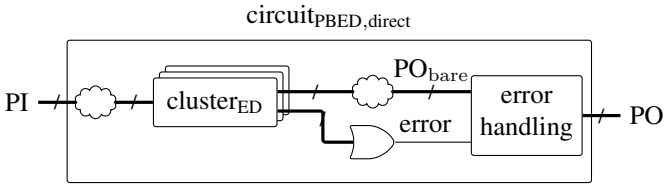

Fig. 6. PBED with direct error detection

generation and one for parity checking. The XORs are logical; they resemble an XOR tree dependent on the number of respective input signals and used FPGA architecture. If one bit in the cluster flips, the cluster error signal error$_{\mathrm{cl}}$ is active. FF$_{\mathrm{a}}$ count k in a cluster is an input parameter and will be expressed using cluster size $s_{\mathrm{cl}} = k + 1$.

Generally, a PBED-hardened circuit contains many of these clusters, whose error outputs can be reduced to a single error signal by ORing them (Figure 6). We call this approach PBED with *direct* error detection. This operation creates a deep OR-tree, dependent on the size of the hardened circuit. This in turn has a negative impact on the critical path and maximum frequency. For example, on a circuit with 121 FFs, PBED causes a critical path overhead $t_{\mathrm{c+}}$ of 8 ns (added to the critical path of the original circuit), which is five times the $t_{\mathrm{c+}}$ of LTMR, reducing the maximum frequency of the bare (i.e., unhardened) circuit from 81 MHz to 49 MHz, compared to the LTMR version achieving 71 MHz. [4]

Alternatively, a long error detection path can be broken into shorter paths by using inherent pipeline structures in a circuit. A data processing circuit, e.g., an instruction processor, utilizes many stages to process an instruction before it is evaluated. This latency introduced by a circuit can be exploited for error detection on the module level. For example, if a memory write instruction takes five cycles before corresponding memory signals are activated, and the data word is written, then it is sufficient to handle a bitflip in this particular instruction five cycles later. In other words, in the same cycle when this word is written to memory. In this work, the approach is called PBED with *pipelined* error detection.

In pipelined error detection, FFs are grouped according to their *sequential distance* d_{seq} to any primary output (PO) of the circuit. d_{seq} is defined as the minimum number of cycles that a bit needs to be visible at PO. For example, a FF whose

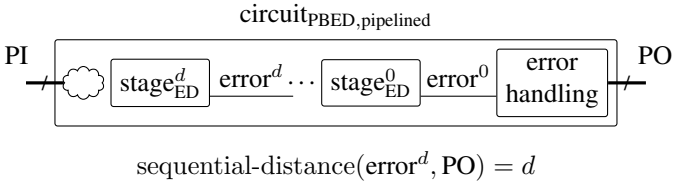

Fig. 7. Interconnect between pipelined error signal propagation and error handling.

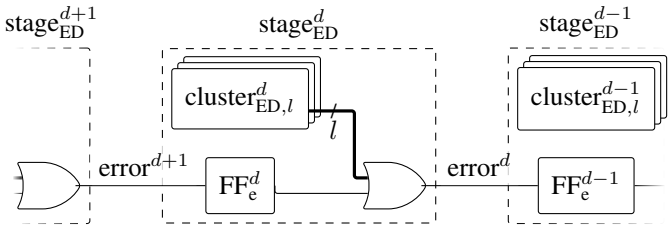

Fig. 8. PBED with pipelined error detection

output is a PO of the circuit has $d_{\mathrm{seq}} = 0$. FFs with $d_{\mathrm{seq}} = d$ belong to a particular *error detection stage*, which is named stage$^{d}_{\mathrm{ED}}$. These stages are visualized in Figure 7.

The inner structure of a stage is shown in Figure 8. Analogous to direct error detection, the FFs are grouped in clusters within a stage. Stages contain an *error FF* FF$^{d}_{\mathrm{e}}$, which stores the error signal that is coming from the previous stage, with the exception of the leftmost stage with the greatest d_{seq}. The error signal of stage$^{d}_{\mathrm{ED}}$, error^{d}, is generated by ORing the buffered error signal from the last stage and the error signals from the clusters within the stage.

The error signal generated by direct and pipelined error detection (error and error^{0}, respectively) are fed to the *error handling* module. The error module is responsible for (1) recovering the circuit from the unknown state and (2) isolating the circuit from the rest of the system while it is in the unknown state. An example implementation is shown in Figure 9. Isolation is realized by masking the appropriate nets from the POs of the bare circuit (PO$_{\mathrm{bare}}$), which can alter the state of the system, e.g., the control signals of a memory interface. The masking starts in the same cycle when the error signal is visible at PO and remains active until the circuit is reset. As long as the circuit is in isolated state, the circuit can be asynchronously reset with the help of a shift register. The number of FFs in the shift register must be chosen such that all FFs in the circuit are guaranteed to be reset after the respective number of reset cycles.

B. Algorithm

PBED can be implemented on-top of a technology-level netlist using an automatic tool[1]. The pseudo code of the PBED application program is shown in Algorithm 1. Before processing, the netlist needs to be parsed, for which we used Verilog-Perl [9]. Then, all the FFs with enable input must be replaced with a basic FF and a multiplexer, which is normally also

[1]Tool available at https://gitlab.informatik.uni-bremen.de/goekce/pbed

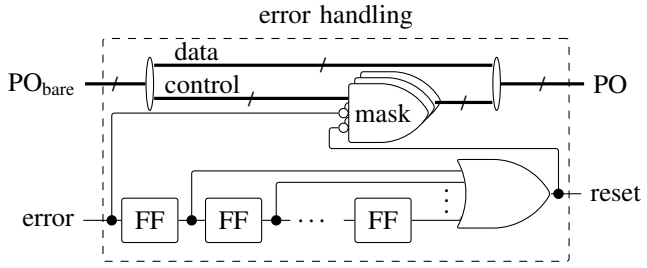

Fig. 9. Example implementation of error handling module. When the error signal is active then the control signals are masked to isolate the circuit in the same cycle. In subsequent cycles, the reset signal is hold active and the circuit falls back to reset state.

done in LTMR. The multiplexer emulates the enable behavior by switching between the output of the FF and the input data which must be fed to the FF when enable signal is active. This is crucial, because an enable FF is not updated in every cycle, but only when the enable input is activated. If a soft error happens on enable FFs, these errors can eventually accumulate and are undetectable for even number of bitflips in a cluster.

If pipelined error detection is used, d_{seq} for every FF must be determined. For this purpose, a FF-only dataflow graph is generated by setting the POs as sink vertexes and exploring the design using breadth-first search and only adding the FFs to the graphs. While traversing, the FFs are annotated with d_{seq} to each particular PO. Subsequently, the minimum of these d_{seq}s is determined, which corresponds to d_{seq} to the output.

In the next step, the FFs are put to clusters according to cluster size, and clock and reset signals of FFs. The FFs in a cluster must be sensitive to the same edge. Furthermore, all the FFs in a cluster must have the same reset type: all active-low or -high. These countermeasures enable the connection of the parity FF to the same clock and reset signal of the application FFs in the cluster possible. After the clusters are generated, the generation of stages is done according to Section III-A. In case of direct error detection, d_{seq} does not play a role, therefore only clusters are created and the cluster errors are reduced.

Finally, the netlist must be recompiled along with the error handling module. Before that, the synthesizer must be instructed to not optimize the redundancies away, because the tools normally presume no external effects (e.g. bitflips) during optimizations. The error handling module is designed manually using HDL. After compilation, the module is ready to be verified under error injection.

C. Overall toolflow

In previous sections, we explained the concept of pipelined- and direct error detection in detail. The implementation of these techniques requires further attention regarding the design flow to follow. In this subsection, we show the overall toolflow that we used with some remarks.

Figure 10 shows an overview of our toolflow, which is partly analogous to classic FPGA design flow. Often, verification of the technology netlist is skipped in the FPGA design, and

```
Data: technology netlist
Result: PBED applied technology netlist
foreach FF do
    if has enable input then
        eliminate enable input;
    end
end
if pipelined error detection then
    foreach primary output (PO) do
        build a FF dataflow graph with this PO as sink
         vertex;
    end
end
foreach FF do
    if pipelined error detection then
        determine min. sequential distance to output by
         using the FF dataflow graphs;
        categorize according to clock-, reset-signal and
         min. sequential distance to output;
    else
        categorize according to clock- and reset-signal;
    end
    push to a cluster according to cluster-size and
     category;
end
foreach cluster do
    add parity-generation and -check circuitry;
end
if pipelined error detection then
    for sequential distance (d_seq)=max to 0 do
        put clusters with d_seq to a new stage;
        reduce cluster error signals to a single error
         signal;
        merge the error signal from the previous stage;
        add an error FF to the stage;
    end
else
    reduce cluster error signals to a single error signal;
end
```

Algorithm 1: Application of PBED to a technology netlist

the design is tested directly on an experimental board or real hardware. This also applies to LTMR, because it is a straightforward approach to harden a circuit against soft errors. But verification of complex redundancy techniques needs a verification on the technology level under error injection. This approach is also used for PBED, in which error handling module can be implemented differently for different target circuits.

To verify the hardened circuit against soft errors time-efficiently, the testbench should be designed in such a way that:

- the RTL-design under test can be easily replaced with a technology netlist, which is mostly achieved by operating on the primary inputs and outputs of the design under test,

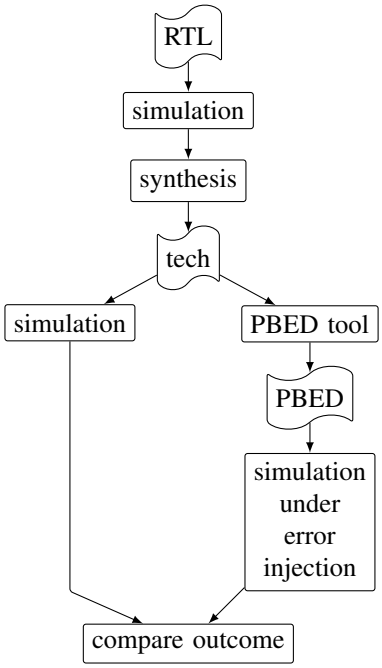

Fig. 10. Simulation flow for verifying the PBED-applied circuit.

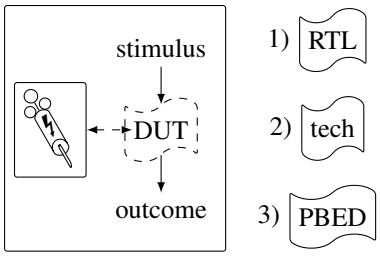

Fig. 11. PBED testbench overview. Design is tested using RTL-, technology- and finally PBED-applied-netlist. In the last case, error injection is also activated.

but not directly on the internal signals, which will be not accessible in a technology netlist due to optimization
- the test scenarios should deal with the unresponsive states of the circuit, e.g., when an error is detected and the recovery of the circuit is in action
- the simulation software allows access to the elements of the netlist, especially to FFs for error injection

The PBED testbench with these countermeasures implemented is visualized in Figure 11. Design is tested using RTL-, technology- and finally PBED-applied-netlist. In the last case, error injection is also activated.

For error injection in our simulation flow, we used the Foreign Language Interface [10] for the simulation software Questa (Mentor Graphics), which allows access to the netlist elements and allows to simulate bitflips. In reality, a soft error can normally be overwritten in the next cycle by the new data. To reproduce this behavior in the simulation, we flipped the target bit by simply setting the signal to the inverted value (instead of forcing and releasing a signal), which can be overwritten in the next cycle.

		$A_{\mathrm{comb}+}$		$A_{\mathrm{FF}+}$		A_+		$t_{\mathrm{c}+}$ (ns)		
c_{st}	w_{d}	PD	PP	PD	PP	PD	PP	PD	PP	$1-\frac{t_{\mathrm{c}+,\mathrm{PP}}}{t_{\mathrm{c}+,\mathrm{PD}}}$
4	8	51	47	18	21	69	68	7.84	5.24	33.21%
4	16	94	91	34	37	128	128	8.79	6.83	22.32%
4	32	177	178	66	69	243	247	9.89	7.74	**21.75%**
4	64	346	331	130	133	476	464	11.71	8.08	31.03%
8	8	94	93	34	41	128	134	7.84	4.47	43.06%
8	16	176	169	66	73	242	242	9.28	5.47	41.01%
8	32	347	349	130	137	477	486	10.41	6.16	40.85%
8	64	693	663	258	265	951	928	11.68	7.51	35.74%
16	8	177	177	66	81	243	258	8.80	4.86	44.81%
16	16	347	334	130	145	477	479	10.01	5.30	47.00%
16	32	692	685	258	273	950	958	11.23	5.26	53.15%
16	64	1369	1330	514	529	1883	1859	11.66	6.94	40.48%
32	8	347	341	130	161	477	502	10.50	3.66	**65.10%**
32	16	692	663	258	289	950	952	10.82	4.00	63.03%
32	32	1370	1361	514	545	1884	1906	11.50	5.05	56.06%
32	64	2735	2654	1026	1057	3761	3711	13.35	6.75	49.40%

TABLE I
PBED SYNTHESIS RESULTS FOR DIRECT (PD) AND PIPELINED (PP) ERROR SIGNAL PROPAGATION WITH FIXED CLUSTER SIZE $s_{\mathrm{cl}} = 3$.

IV. SYNTHESIS RESULTS AND COMPARISON WITH LTMR

In this section, we compare pipelined error detection to our previous approach and LTMR using experimental results.

A. Comparison of pipelined- with direct-error detection

To show the positive effect of pipelined error detection on the critical path of a PBED-applied design, we synthesized a generic shift register without any combinatorics using following input parameters:

- data width $w_{\mathrm{d}} = \{8, 16, 32, 64\}$
- stage count $c_{\mathrm{st}} = \{4, 8, 16, 32\}$

Then, we applied PBED with direct (PD) and pipelined (PP) error detection on the netlists with cluster size $s_{\mathrm{cl}} = 3$ and gathered the following output parameters for both approaches:

- combinational area overhead $A_{\mathrm{comb}+}$ (number of tiles added to the original circuit)
- FF area overhead $A_{\mathrm{FF}+}$
- total area overhead $A_+ = A_{\mathrm{comb}+} + A_{\mathrm{FF}+}$
- critical path overhead $t_{\mathrm{c}+}$

Using the $t_{\mathrm{c}+}$, we additionally determined how much of the critical path overhead of PD can be saved by using PP by calculating $1 - \frac{t_{\mathrm{c}+,\mathrm{PP}}}{t_{\mathrm{c}+,\mathrm{PD}}}$, which we will refer as $t_{\mathrm{c}+}$ saving.

The synthesis was done for the ProASIC3 FPGA A3PE3000L using Precision (Mentor Graphics) for synthesis and Designer (Microsemi) for placing and routing. The results are shown in Table I.

PP outperforms PD regarding $t_{\mathrm{c}+}$ with negligible area overhead difference. PP can save at least 22% of $t_{\mathrm{c}+}$ of PD and for designs with more stages the saving goes up to 65%. No correlation can be noticed between w_{d} and $t_{\mathrm{c}+}$ saving.

			$A_{\text{comb}+}$		$A_{\text{FF}+}$		A_+		$t_{\text{c}+}$ (ns)		$\frac{A_+}{A_{\text{FF,BA}}}$			
c_{st}	w_{d}	t_{c} (ns)	LT	PP	LT	PP	LT	PP	LT	PP	LT	PP	$1-\frac{A_{+,\text{PP}}}{A_{+,\text{LT}}}$	$\frac{t_{\text{c}+,\text{PP}}}{t_{\text{c}+,\text{LT}}}$
4	8	1.89	32	47	64	21	96	68	2.42	5.24	3.00	2.13	**29.17%**	2.16
4	16	1.88	64	91	128	37	192	128	2.19	6.83	3.00	2.00	33.33%	3.12
4	32	1.90	128	178	256	69	384	247	2.24	7.74	3.00	1.93	35.68%	3.46
4	64	1.90	256	331	512	133	768	464	1.67	8.08	3.00	1.81	39.58%	4.85
8	8	2.77	64	93	128	41	192	134	2.06	4.47	3.00	2.09	30.21%	2.17
8	16	2.77	128	169	256	73	384	242	1.10	5.47	3.00	1.89	36.98%	4.99
8	32	3.10	256	349	512	137	768	486	0.94	6.16	3.00	1.90	36.72%	**6.54**
8	64	2.71	512	663	1024	265	1536	928	1.38	7.51	3.00	1.81	39.58%	5.46
16	8	2.94	128	177	256	81	384	258	1.82	4.86	3.00	2.02	32.81%	2.66
16	16	3.02	256	334	512	145	768	479	0.89	5.30	3.00	1.87	37.63%	5.96
16	32	3.65	512	685	1024	273	1536	958	1.34	5.26	3.00	1.87	37.63%	3.92
16	64	3.30	1024	1330	2048	529	3072	1859	1.08	6.94	3.00	1.82	39.49%	6.41
32	8	2.77	256	341	512	161	768	502	2.11	3.66	3.00	1.96	34.64%	**1.74**
32	16	3.47	512	663	1024	289	1536	952	1.22	4.00	3.00	1.86	38.02%	3.29
32	32	3.38	1024	1361	2048	545	3072	1906	0.94	5.05	3.00	1.86	37.96%	5.37
32	64	3.44	2048	2654	4096	1057	6144	3711	1.12	6.75	3.00	1.81	**39.60%**	6.04

TABLE II
SYNTHESIS RESULTS FOR LTMR AND PIPELINED PBED WITH CLUSTER SIZE $s_{\text{cl}} = 3$

s_{cl}	$A_{\text{comb}+}$	$A_{\text{FF}+}$	A_+	$t_{\text{c}+}$ (ns)	$1-\frac{A_{+,\text{PP}}}{A_{+,\text{LT}}}$	$\frac{t_{\text{c}+,\text{PP}}}{t_{\text{c}+,\text{LT}}}$
2	369	265	634	6.38	17.45%	6.79
3	349	137	486	6.16	36.72%	6.54
4	312	97	409	6.55	46.74%	6.96
5	350	73	423	6.34	44.92%	6.74
6	348	65	413	5.73	46.22%	**6.09**
7	311	57	368	6.63	52.08%	**7.04**
8	345	49	394	6.05	48.70%	6.43
9	343	41	384	6.55	50.00%	6.96
10	326	41	367	6.51	52.21%	6.92
11	331	41	372	6.37	51.56%	6.77
12	321	33	354	6.36	53.91%	6.76

TABLE III
PIPELINED PBED SYNTHESIS RESULTS FOR STAGE COUNT $c_{\text{st}} = 8$, DATA WIDTH $w_{\text{d}} = 32$, AND VARIOUS CLUSTER SIZES s_{cl}.

B. Comparison with LTMR

We have shown that PP delivers better results than PD regarding $t_{\text{c}+}$. In this section we compare PP with the state-of-the-art hardening approach LTMR (LT) using the same shift register configurations from Subsection IV-A. The results are shown in Table II. In addition to the known parameters from Table I, following parameters are shown:

- critical path of the unhardened circuit t_{c}
- area overhead per user FF $\frac{A_+}{A_{\text{FF,BA}}}$ (BA: bare circuit)
- area overhead saving if PP is used instead of LTMR $1-\frac{A_{+,\text{PP}}}{A_{+,\text{LT}}}$
- critical path overhead factor of PP $\frac{t_{\text{c}+,\text{PP}}}{t_{\text{c}+,\text{LT}}}$

According to the results, at least 29% of the area overhead caused by LTMR can be saved by using PP. If w_{d} increases, PP can save more area, but it has a negative impact on $t_{\text{c}+}$. PP has up to 6.5 times the critical path overhead of LTMR. Note that we compared not the *absolute* critical path, which determines the maximum frequency, but only the *overhead*, which adds up to the critical path of the bare circuit.

We gathered additional results by setting $c_{\text{st}} = 8$ and $w_{\text{d}} = 32$ and varying s_{cl} between 2 and 12. PP saves more area with increasing s_{cl}, but s_{cl} does not have a significant effect on $t_{\text{c}+}$.

V. CONCLUSION

We have presented a new approach for PBED by exploiting pipeline structures in a circuit, and shown that it can save from 22% to 65% of the critical path overhead caused by the direct error detection approach. LTMR has still better timing performance and the pipelined PBED increases the critical path overhead by a factor of 2 to 7. As future work, PBED will be applied on a soft-processor core to get results on a more complex design. Formalization of transaction-based processing is required to safely utilize PBED in a system; this is also planned as future work.

REFERENCES

[1] *Radiation-Tolerant ProASIC3 Low Power Spaceflight Flash FPGAs Datasheet*, Microsemi, November 2013.

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

[3] 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.

[4] G. Aydos and G. Fey, "Empirical results on parity-based soft error detection with software-based retry," in *Nordic Circuits and Systems Conference (NORCAS)*, Oct 2015.

[5] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, April 1962.

[6] 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.

[7] J. H. Patel and L. Y. Fung, "Concurrent error detection in ALU's by recomputing with shifted operands," *IEEE Transactions on Computers*, vol. C-31, no. 7, pp. 589–595, July 1982.

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

[9] W. Snyder, "Verilog-Perl distribution," http://www.veripool.org/wiki/verilog-perl, 2015.

[10] *Foreign Language Interface Manual*, Mentor Graphics.