2017

PARITY-BASED ERROR DETECTION WITH RECOMPUTATION FOR FAULT-TOLERANT SPACEBORNE COMPUTING

GÖKÇE AYDOS

GÖRSCHWIN FEY
Advisor

UNIVERSITY OF BREMEN
Faculty of Mathematics and Computer Science

Abstract

In radiation environment (e.g., space, nuclear reactor), electronics can fail due to bitflips in the flipflops of integrated circuits. A common solution is to triplicate the flipflops and connect their outputs to a voter. If one of the three bits is flipped, then the voter outputs the majority value and tolerates the error. This method is called triple modular redundancy (TMR).

TMR can cause about 300% area redundancy. An alternative way is error detection with on-demand recomputation, where the recomputation is done by repeating the failed processing request to the processing circuit. The computation is done in consecutive transactions, which we call transaction-based processing.

We implemented and evaluated the aforementioned alternative approach using parity checking on the Microsemi ProASIC3 FPGA, which is often used in space applications. The results show that parity-based error detection with our system recovery approach can save up to 54% of the area overhead that would be caused by the TMR, and achieve in most circuits slightly better timing results than TMR on ProASIC3. This area saving can be the key for fitting the application to a space-constrained chip.

Zusammenfassung

In einer Strahlungsumgebung wie im All oder in der Nähe eines Atomreaktors können elektronische Geräte durch Bitkipper in den Flipflops integrierter Schaltungen ausfallen. Eine gängige Methode gegen die Bitkipper ist triple modular redundancy (TMR), bei der jedes Flipflop der Schaltung dreifach instanziiert wird und die Ausgänge der Flipflops zu einem Voter angeschlossen werden. Falls eins von den drei Bits gekippt wird, dann gibt der Voter den Majoritätswert aus und toleriert somit diesen Fehler.

TMR kann etwa 300% Flächenaufwand verursachen. Eine alternative Methode ist Fehlerdetektion mit anschließender Neuverarbeitung der letzten Daten. Die Neuverarbeitung der Daten wird durch die Wiederholung der letzten Datenverarbeitungsanfrage zur Schaltung realisiert. Die Verarbeitung der Daten erfolgt durch nacheinanderfolgende Transaktionen und diese Art von Datenverarbeitung nennen wir transaktions-basierte Datenverarbeitung in dieser Arbeit.

Wir haben die obenerwähnte Methode implementiert und bewertet, wobei wir als Fehlerdetektionsmethode Paritätsprüfung eingesetzt haben. Die Bewertung erfolgte auf dem FPGA Microsemi ProASIC3, das bei Avionikanwendungen sehr verbreitet ist. Die Ergebnisse zeigen, dass unsere Methode bis zu 54% des Flächenaufwands einsparen kann, der sonst vom TMR verursacht wäre. Andererseits kann unsere Methode in den meisten Schaltungen etwas besseres Timing als TMR erzielen. Die Flächeneinsparung könnte maßgeblich für die Implementierung einer Anwendung auf einer begrenzen Chipfläche sein.

Contents

1 Introduction **1**
1.1 Application on a processing architecture 3
1.1.1 Overview 3
1.1.2 FPGA Design 3
1.1.3 Communication Protocol 4
1.1.4 Hardening 5
1.2 Next chapters and background work 6

2 Preliminaries **9**
2.1 Concepts of dependable computing 9
2.1.1 Systems and threats to dependability 10
2.1.2 Means for dependability 15
2.1.3 Fault tolerance 16
2.2 Transient effects on sequential circuits 18
2.3 Fault model used in this work 20
2.4 Fault tolerance techniques against SEUs 22
2.4.1 Fabrication process level techniques 23
2.4.2 Chip layout level techniques 24
2.4.3 Logic level techniques 25
2.4.4 Triple modular redundancy on logic level 26
2.4.5 Architecture level 28
2.4.6 Software level 28
2.4.7 Algorithm level 29
2.5 FPGAs used in mission-critical applications 29
2.6 Microsemi ProASIC3 FPGA 31
2.7 Error detection–based fault tolerance 31

3 Related work **35**
3.1 Error detection by duplicated instructions 35
3.2 Limitations of software-based techniques 39
3.3 Cross-layer exploration for architecting resilience 41
3.3.1 General discussion 41
3.3.2 Parity checking 43

4 Parity-based error detection 45
4.1 Concept . . . 45
4.2 Analytical evaluation . . . 48
4.2.1 Prerequisites . . . 48
4.2.2 Critical path delay . . . 49
4.2.3 Circuit area overhead . . . 53
4.2.4 Multiple bit error susceptibility . . . 55
4.3 Experimental evaluation . . . 58
4.3.1 Finite state machine (FSM) circuit . . . 59
4.3.2 I99T circuits . . . 68
4.4 Automatic application . . . 75

5 Pipelined cluster error signal reduction 83
5.1 Concept . . . 83
5.2 Experimental evaluation . . . 84
5.2.1 Finite state machine (FSM) circuit . . . 85
5.2.2 I99T circuits . . . 87
5.3 Automatic application . . . 90

6 Transaction-based processing & recovery 93
6.1 Recovery in the target circuit . . . 93
6.1.1 Circuit isolation . . . 94
6.1.2 Circuit reset . . . 96
6.2 Transaction-based processing . . . 97
6.2.1 Concept . . . 97
6.2.2 Fault tolerance analysis . . . 99
6.3 Experimental evaluation . . . 100
6.3.1 FSM . . . 101
6.3.2 I99T circuits . . . 104
6.3.3 Processing time penalty . . . 106
6.4 Automatic application . . . 107
6.4.1 Logical masking of control signals . . . 107
6.4.2 Reset circuit . . . 109

7 Conclusion 111

Acknowledgments and statutory declaration 113

References 114

Chapter 1

Introduction

Electronics for airborne and space systems, called avionics, must often be protected from ionizing radiation in and coming from space. In the absence of a shield like the magnetic field of the earth, high energy particles can traverse through the digital circuit fabric and cause bitflips in the flipflops of a circuit.

Also terrestrial computing systems at sea level are exposed to some amount of radiation, but the probability that a bit in a flipflop is flipped is relatively low compared to higher attitudes in atmosphere or space. Still, as more bits can be stored on the same chip area, the computing systems at sea level also show signs of bitflips.

Where some of the induced bitflips can vanish unnoticed by the computing system, some bitflips can lead to a restart or freeze of a system. This is not an issue if a personal computer restarts, but this should not happen with a critical system like a server which tracks financial transactions or a computer, which guides a space vehicle. Such systems must be dependable and be able to tolerate possible threats in their working environment, e.g., component failure due to aging or high energy particles present in space striking through system components.

Dependable computers often use modular redundancy against component failures. Modular redundancy means that a module is present many times that failure of a module can be tolerated by switching to the redundant modules. If the effect of a threat is permanent, i.e., a component cannot be used after a failure, then a system with n equal components can tolerate up to $n-1$ component failures.

In contrast, if a threat is only temporary, e.g., by recovering a failed component by a restart, there is no need to include many redundant modules. A well-known fault tolerance approach against temporary module failures is the *triple modular redundancy (TMR)*, which provides a straightforward error detection and recovery approach by triplicating a module and connecting the modules to a voter entity, and the voter entity selects the trusted output. For instance, a majority voter outputs *yes* if two of the modules output *yes* and one *no* by trusting the majority. Consequently, TMR can tolerate a failure of a single module and enables the continuation of the service. Still, while one module is in failure, module recovery

must be initiated to avoid a failure of the system (consisting of these three modules), because a second module failure will cause a system failure.

The idea of TMR can also be applied on components of avionics. Digital sequential circuits which are part of avionics also often implement TMR. If TMR is implemented solely on the flipflops, it is called *local TMR* (*LTMR*). In LTMR, for every application flipflop in the circuit two redundant flipflops are created, which store the same bit as the application flipflop. The outputs of these three flipflops are then connected to a majority voter.

The LTMR approach has the advantage of built-in error detection and recovery: If a flipflop bit is flipped during a clock cycle, then this bit error is masked by the voter. In the next cycle, the flipped flipflop will be overwritten with the correct bit coming from the combinatorics, leading to the recovery from the erroneous state of the flipflop.

LTMR can be easily applied on a circuit and is often applied using commercial available software tools. Unfortunately, LTMR comes at a significant cost of additional space for redundant flipflops. Additionally, redundant flipflops can also account for excess power consumption.

On space- and power-constrained applications, an alternative is to apply an error detection approach instead of LTMR, because error detection generally incurs less resources than error correction.

A circuit which implements error detection can only flag an error, but cannot correct erroneous data, or recover itself from an erroneous state. In this situation, this circuit can be recovered externally from the failure state by another system component and the last processing request to this circuit can be retried. In this work, we propose this approach, and call it *error detection-based fault tolerance* and will be abbreviated as *EDFT*.

In chronological order, EDFT involves the following actions:

- the detection of the error
- system recovery on the circuit using isolation and error handling
- error detection and system recovery on the user application retrying the last processing request to the circuit

The last action can be carried out by a request and response-based processing protocol between two systems. In this work, this processing technique is called *transaction-based processing*.

The three actions of EDFT can be implemented by different approaches. To evaluate EDFT in detail and compare to the state of the art, not only generic specifications but also concrete implementation of EDFT's components are needed. In the next section, we will present a data processing architecture, on which EDFT can be applied. With the help of this example architecture, we will describe the EDFT's components more in detail and then evaluate EDFT by using the concrete implementations.

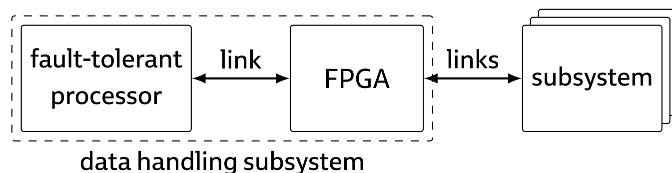

Figure 1.1: Overview of the reference processing architecture. Processor communicates with other subsystems through the FPGA.

1.1 Application on a processing architecture

In this section, we describe a reference model of an on-board data handling unit (OBDH) for satellites [Tre+14]. Using this example architecture we will briefly explain in the following section how EDFT is implemented. The detailed implementation will be discussed in the following chapters. Moreover, this particular implementation will also be used for comparing EDFT with state of the art in the following chapters.

First, we describe an overview of the system, then the target circuit, and finally the communication protocol between the processor and the circuit.

1.1.1 Overview

Figure 1.1 shows an overview of the architecture. The OBDH subsystem 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., UART, CAN). We assume that the processor, the communication line between the processor and the FPGA, and the subsystems are sufficiently protected against soft errors.

1.1.2 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 figure 1.2. 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 figure 1.3, circuit B is described as a finite state machine (FSM). Circuit B reads the memory access request packets sent by the processor from the FIFO and transforms them in memory accesses for circuit C. Circuit C with a memory block inside resembles the memory-mapped interfaces. The memories transfer one word per

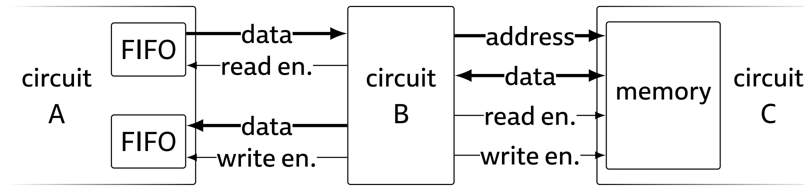

Figure 1.2: Excerpt from the FPGA design. Circuit B must be hardened by design. Other circuits are immune to soft errors.

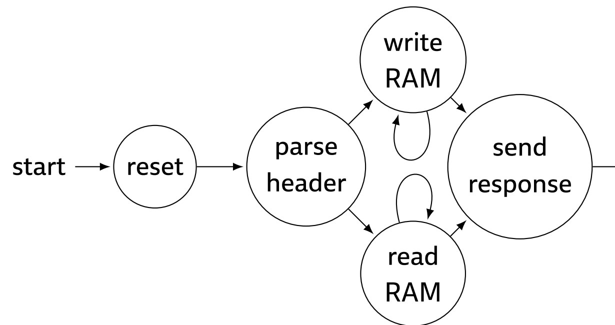

Figure 1.3: Simplified state diagram of circuit B, which parses the remote memory packets sent by the mission software (i.e., the processor)

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, thus the data flow to the memories can be controlled with a single word granularity.

1.1.3 Communication Protocol

The communication protocol between the processor and the FPGA is visualized in figure 1.4. 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 after a time-

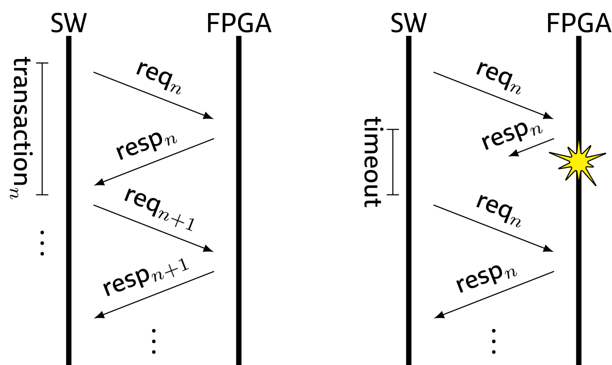

Figure 1.4: Sequence diagram of the communication protocol, which is based on transactions. A *transaction* consists of a request (*req*) and a response (*resp*). The left diagram shows a normal sequence: every request is followed by a response. On the right, the behavior in case of an error in the FPGA is visualized: if still no response after a timeout is received, the last transaction is repeated.

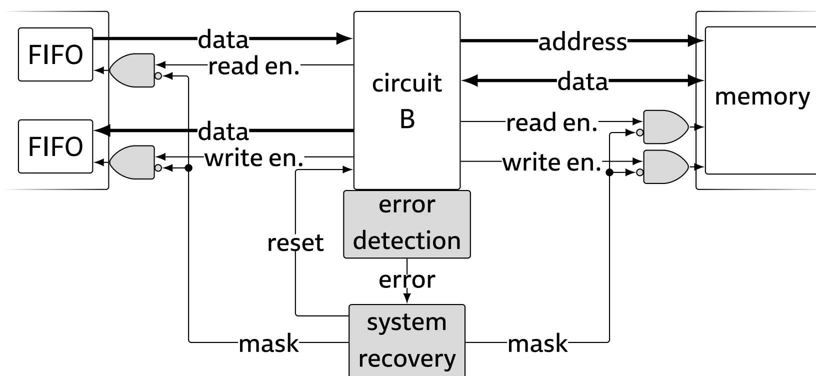

Figure 1.5: Design from figure 1.2 with error detection-based fault tolerance (EDFT) applied on circuit B

out, 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.

1.1.4 Hardening

Now, we describe how EDFT can be applied on the reference architecture. Figure 1.5 shows EDFT hardware components attached to the target circuit. If a bit in circuit B flips, then the error signal is activated by the error detection module. The error signal activates the error handling module, which immediately masks the target circuit's outputs to isolate the circuit. While the circuit stays in isolated state, the error handler recovers circuit from the erroneous state by activating the

reset signal of circuit B.

As circuit B was reset, no valid response could be sent. Consequently, the last request to circuit B is repeated according to the communication protocol and the processing can continue.

Compared to the state-of-the-art approach TMR, EDFT has a lower area (thus also power) overhead, which can be the key factor to fit an application on a space-constrained chip. The critical path overhead is similar to TMR, so the hardened application can run at similar clock frequencies. Compared to TMR, EDFT requires a software component which retries a failed transaction, which should not have a significant overhead.

From the theoretical perspective, the concepts used by EDFT are not novel approaches. Still, this work contributes to the existing work by:

- applying EDFT on a real processing architecture
- evaluating EDFT using a state-of-the-art FPGA for space applications
- evaluating EDFT analytically and experimentally using placed-and-routed circuits
- describing the automatic application of EDFT using a parity-based approach.

1.2 Next chapters and background work

After we have shown an overview of EDFT using an example, in next chapters we will discuss it more in detail. First, some preliminaries important for understanding EDFT will be handled in chapter 2. The chapter 3 will address the related work. Then, chapter 4 analyzes an example implementation of the error detection module: parity-based error detection. A possible drawback of parity-based error detection is the timing impact. In chapter 5, we propose a pipelining approach which can alleviate this impact. The remaining components of EDFT - error handling and transaction-based processing will be discussed in chapter 6. Finally, we will conclude the work by giving some recommendations regarding testing of an EDFT-applied system, summarizing important points of the work and giving some suggestions for the future work.

The following publications make up the background work for the following chapters:

- [AF15b] introduces the idea of EDFT in general, which was already done in section 1.1.
- in [AF15c] we give a first insight to the performance of EDFT using parity-based error detection by comparing our approach with LTMR analytically. Synthesis results using a real circuit is gathered in [AF15a]. These contributions make up part of chapter 4.

- the work for pipelined parity approach in chapter 5 originates from [AF16b].
- [AF16a] is an extended version of [AF15a] and provides a more detailed specification, and fault tolerance analysis of transaction-based processing. The contributions in this work were used in chapter 6.

Chapter 2

Preliminaries

We begin this chapter by introducing general goals and concepts in dependable computing. In section 2.2 discuss about transient effects in digital circuits, in section 2.3 about our fault model. In section 2.4, we give an overview of techniques for achieving fault tolerance against bitflips in flipflops of a digital circuit. In section 2.5, we give additional information about FPGAs for radiation environment, as our evaluations in next chapters and the implementation of our work is based on an FPGA. Section 2.6 is dedicated to the FPGA that we used in our evaluations, the ProASIC3. After the introduction of these concepts and background information, in section 2.7, we present our proposed approach more in detail.

2.1 Concepts of dependable computing

Some terms or common concepts used in this work regarding fault tolerance, e.g., fault, error, fault handling, error handling, can have different meanings in different fields of science or even different perceptions by different persons in the same field. Because of this reason, we give the definitions of some terms and common concepts of fault tolerance that are used in this work. The definitions are based on the well-known work [Avi+04], which compiles the common terms and concepts belonging to dependable and secure computing. The terms and concepts introduced in [Avi+04] are very broad and we will confine us to the terms and concepts relevant to this work and give examples by applying these terms and concepts on systems used in embedded computers and digital circuits.

In what follows, we first introduce the terms important for the concept of dependability, fault tolerance and soft error. Then, we present the means for achieving dependability and fault tolerance of a system.

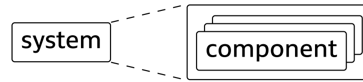

Figure 2.1: A system consists of components. A component itself is a system too.

2.1.1 Systems and threats to dependability

A *system* is an entity that interacts with other entities. A system can deliver *services* to other systems in its environment via its *use interfaces* and can receive services from other systems. For instance, an onboard data handling system (OBDH) on a space satellite stores and periodically transmits gathered data during a mission to earth, and also handles the communication between other systems on a satellite. The OBDH delivers a data communication service to the systems onboard the satellite and a housekeeping data transmission service to the satellite operator on earth. Note that a satellite operator and thus a human can also be abstracted as a system. A use interface of the OBDH can be a software module for decoding the data packets received from other systems, if we observe the OBDH from the software point of view. On the other hand, from the hardware point of view OBDH, a plug mounted on the case of the data handling system would be the use interface. It is obvious that a system can be perceived differently in different abstraction levels.

A system consists of one or more *components*, which contribute to the service delivered by the system. A system is a recursive term, a component of a system is also a system itself. For instance, the circuit board component carrying the main processor chip of an OBDH is itself a system which interacts with other circuit boards inside the OBDH enclosure. The main processor chip is a system which can run software and process data as a service, consisting of the circuit die and the chip pins. The circuit on the die consists of digital and analog circuitry, where the digital circuitry consists of combinational and sequential logic elements. A flipflop as a sequential element can also be abstracted as a system which can store a Boolean value as a service.

A dependable system tries to deliver a *correct* service to its users, but there are threats against the service delivery. A service *failure* is an event that causes a transition to a system state, where the system cannot deliver its service to the users on its use interface in an expected way and the failure leads to an *incorrect* service. A failure is caused by one or more *errors* inside the system. An error is a deviation from the correct system state, which can lead to a system failure, but not every deviation from the correct system state must end up in a failure. A system service has an *external* state, which determines how the service is delivered at the use interface. The rest of the system state is defined by the *internal* state. Only a deviation from the correct external service state can be perceived by the user, and thus is a failure. Consequently, an error in a system must propagate (through components) and change the external system state to cause a failure. A *fault* is

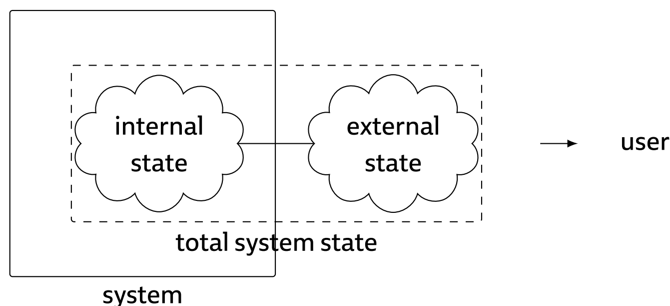

Figure 2.2: Illustration of a system delivering services to a user. The total system state controls how the services are delivered to the user. The total system state consists of the internal and external state. The part of the total system state which can be observed by the user is the external state, the remaining part is the internal state. The arrow shows the flow of information.

the cause of an error. A fault can be *internal* or *external*. A fault must be *active* that it can cause an error. During the time period when a fault does not lead to an error, the fault is *dormant*. Note that a failure in a system component does not have to cause a system failure, but this failure can cause an error in a neighboring system component which in turn can activate an error in the external service state and cause a system failure. The concepts discussed until now are illustrated in figures 2.1, 2.2, 2.3, and 2.4.

Imagine a data processing circuit that receives a memory access request at its use interface, accesses a memory circuit according to the memory access request and transmits a response to the user. A correct service is delivered, if the circuit accesses the memory correctly and responds according to the request. A write access request to a (normally) not used address causes an assertion of both the read and write enable signals of the memory, and no write operation can be executed in return. Let this behavior due to a bug in the synthesizable hardware description of the data processing circuit due to a wrong reasoning of a developer. In this case, the wrong reasoning is an error of the developer, which have caused a dormant fault in the circuit. Note that according to [Avi+04], a human can be a component of a system, and thus a human can also be modeled as a system. Only if the mentioned particular write access request happens, this dormant development fault sets a flipflop in the circuit, which in turn activates the read signal of the memory circuit and leads to the failure.

The same error can be caused by an external fault. Assume that this circuit is operated in space and the circuit is not sufficiently protected against the energetic particles present at the operated orbit, e.g., by not using a chip with radiation-hardened flipflops. Then, the energetic particle, which traverses through the circuit and induces enough charge to flip the flipflop bit controlling the read enable signal, is an external fault. The bitflip event in the flipflop is an error. In most cases, this bit gets overwritten with a correct value by a predecessor flipflop, and

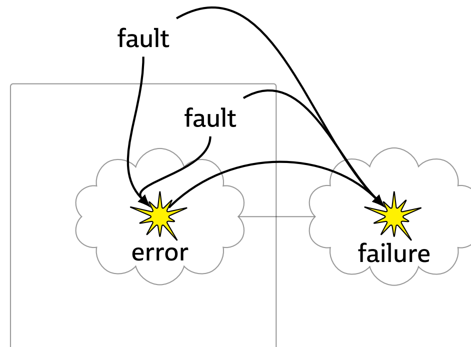

Figure 2.3: Fault, error and failure concepts illustrated on the system from figure 2.2. An error is a deviation from the correct system state. If the error happens in the external state of a service delivered by the system, thus can be observed by the user, then it is a failure. An error can be caused internally or externally.

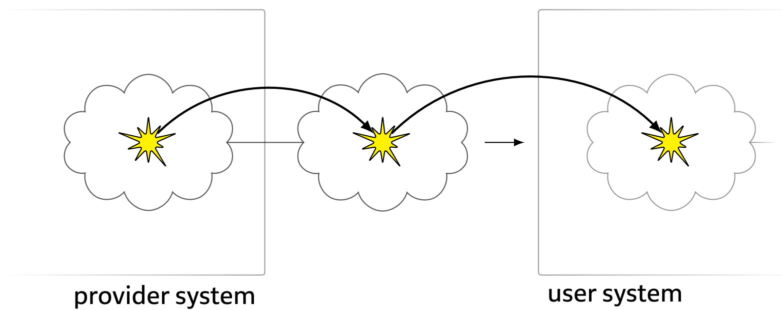

Figure 2.4: Error propagation between systems. A provider failure can be fault from the perspective of a user and cause an error in the user system.

the bit error is corrected until a memory write request is received. In other cases, it causes a failure.

Faults which are continuously present in time are called *permanent* faults, otherwise we talk of *transient* faults. For instance, imagine a power transistor was soldered incorrectly to a circuit board and a transistor pin is detached due to strong vibrations during the launch of a space satellite. This is a permanent fault, because an inspection and repair is generally not part of a satellite mission. Another important fault source is the radiation in space. Radiation can cause permanent faults in circuits, if a circuit is exposed to radiation long enough that physical structures on a die are damaged irreparably. The flipflop bitflips due to radiation which can be overwritten in the next clock cycles are in contrast transient faults.

Some faults can be activated systematically in a determinable way, these are *hard* faults. Hard in this context means that a fault do not seem to change its reaction and stays "hard", when a determined input stimulus is applied on the system. If the fault seems to be activated sporadically, then we talk of a *soft* fault. Generally, soft faults can only be reproduced under very complex and rare internal and external conditions. A hard fault in the context of our example data processing circuit, would be the development bug in the circuit, which easily gets activated when a particular stimulus is applied to the system. On the other hand, if the fault was not activated during the verification of the circuit due to an insufficient verification coverage and only happens a single time in a month during the operation, we talk of a soft fault. Note that hard and soft are terms about the fault activation reproducibility and is dependent on the perception of a fault.

[Avi+04] identifies fault classes which are divided into three major groupings: development, physical, and interaction faults. Development faults are caused during engineering phase of a system. Physical faults are faults which are caused on the hardware. Interaction faults arise due to faults at the use interface of systems mainly by humans or generally by interference between systems. These groupings are overlapping, i.e., one fault can belong to two groupings, for instance an insufficient verification coverage can lead to a physical development fault.

The authors identify that there are no transient development faults. Due to the similarity between the perceptions of soft development faults and transient physical faults, which cannot be easily reproduced, these two categories are bundled as *intermittent* faults. This classification is illustrated in figure 2.5. Errors caused by intermittent faults are called *soft errors*. If an error present in the system is not noticed, then the error is *latent*, otherwise *detected*.

Note that the research community involved in the fault tolerance for mission-critical digital systems mostly uses the terms *soft error* for temporary upsets, and *hard error* for permanent errors in electronics due to electromagnetic radiation [Nic11], [BSV11, ch. 3], [KCR06, ch. 1], [Pet11, ch. 2]. The difference between the meanings of *hard* and *soft* in fault tolerance community can be seen in figures 2.5, 2.6, and 2.7. The errors are caused by various single event effects (SEE), which we introduce separately in section 2.2.

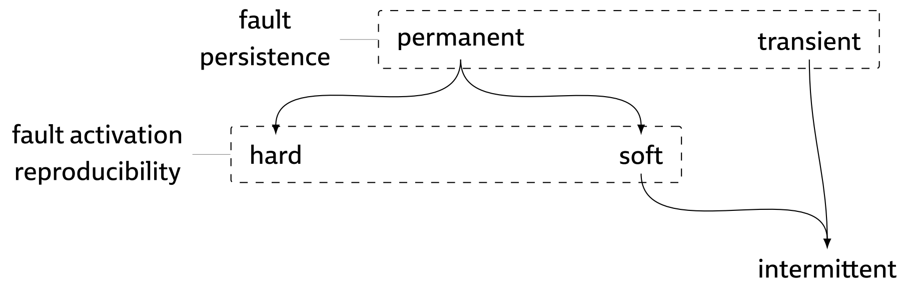

Figure 2.5: Due to similarity between soft development faults and transient physical faults, these are bundled as intermittent faults. Figure adapted from [Avi+04].

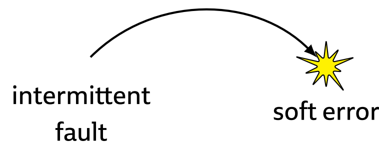

Figure 2.6: Soft errors are caused by intermittent faults. This definition of soft error in [Avi+04] differs from the definition commonly used in the fault tolerance community (figure 2.7).

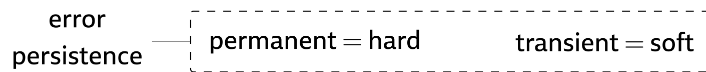

Figure 2.7: The meanings of soft and hard in fault tolerance community. Compared to the taxonomy in [Avi+04] illustrated in figures 2.5 and 2.6, fault tolerance community usually uses the terms *hard* and soft for error persistence.

Even we will not use the terms *hard* and *soft* as described in [Avi+04], other fault tolerance concepts presented from [Avi+04] are generally used in fault tolerance community.

2.1.2 Means for dependability

In last subsection, we introduced the concepts of fault, error and failure, which are three main threats to dependable computers. It is obvious that faults are the source for errors as well as failures. Consequently, the concepts for the means for ensuring dependability are based on the word fault and are called:

- fault avoidance (goal: fault-free system)
  - fault prevention
  - fault removal
- fault acceptance (goal: living with faults)
  - fault forecasting
  - fault tolerance

Fault prevention happens mostly during development phase of a system and aims to avoid generation of faults by enabling more robust development processes. For instance, there are coding styles or standards for hardware descriptions used in avionics, which limit the use of some coding language constructs or coding techniques which may lead to faults in code. [CPB10] compiles some guidelines for the hardware description language (HDL) VHDL.

Fault removal happens during development and operation phase of a system. For instance, verification during HDL development is carried out to remove the faults in the code. The faults in the HDL code are also called bugs. During operation, fault removal is mainly done during system maintenance. Maintenance is an external countermeasure and a maintenance follows a failure or is done periodically as preventive means. Due lack of physical access, a physical fault removal is not practicable for space satellites, but a fault in the system software can be removed for instance by removing the fault and reuploading to the satellite.

Fault forecasting tries to foresee faults by evaluating the system behavior. The evaluation can be done during development as well as in operation. For instance as part of the quality assurance for satellite systems a fault detection, isolation and recovery (FDIR) is prepared by analyzing the fault sources in the system and checking the presence of the means against the faults.

Fault tolerance tries to avoid system failures during operation with the help of fault tolerance techniques. If the use environment of a dependable system includes external faults, then fault avoidance is not practicable and this system must implement fault tolerance and/or fault forecasting.

2.1.3 Fault tolerance

From the four means for dependability, the fault tolerance is the key concept for this work and will be described more in detail.

[Avi+04] uses the following classification for fault tolerance techniques:

- error detection
  - concurrent detection
  - preemptive detection
- recovery
  - error handling
    - compensation
    - rollback
    - rollforward
  - fault handling
    - diagnosis
    - isolation
    - reconfiguration
    - reinitialization

Error detection is the localization of errors in the system state. We speak of *concurrent* detection or concurrent error detection (CED) if the error detection can be carried out continuously. Some examples are comparators for duplex systems or error detecting codes for registers in circuits, which are active continuously in time. An overview of CED techniques against bitflips will be handled in subsection 2.4.3 more in detail.

Preemptive detection takes place outside the actual operation window of a system. This means that the component delivering the the service is paused and the tester component is active in the system. An example is the checking the integrity of data in the random access memories on a circuit after power-on, to prevent a failure during data processing.

System recovery or simply recovery is the reaction to a detected error and tries to create a system state which is free of errors (*error handling*), and undertakes actions that the faults do not cause any errors (*fault handling*).

The first error handling technique is the *compensation*. Compensation masks the erroneous part of the system state, if sufficient redundancy for the system state is present. For instance, if the state machine of a circuit is encoded using Hamming-code, then single bit errors can be compensated by this system.

Rollback tries to go back to a error-free system state. This technique has the advantage of restoring a prior state of the system with minimal data and time loss and restarting processing from this state, especially if the system needs high

amount of time to reach this state again. But this advantage comes at the cost of extra space for saved system states, which are also called checkpoints. If a rollback is not possible, *rollforward* can be tried. Compared to an old, saved state, rollforward restores a new error-free system state. Imagine a data processing circuit which periodically stores checkpoints in a memory. Subsequently, a radiation-induced bitflip happens in the state machine, which gets detected due to one-hot encoding. First, a rollback is tried, but the checkpoint is too old to restore. As fallback, rollforward is tried by resetting the state machine and the start state is restored.

Diagnosis is the evaluation of an error to find out the fault that led to the error. Diagnosis normally happens in complex systems, when the cause of an error is not obvious, where an error can be caused by a long chain of threats.

Isolation keeps the fault in a defined area by means of logical or electrical masking with the aim that the fault does not affect neighboring systems by propagating through the system boundaries. For instance, if a bit error is detected in a sequential circuit, then circuit's outputs can be logically masked to avoid propagation of erroneous data to neighboring circuits.

Reconfiguration involves reassigning of tasks to spare components in the system. For instance, a data processing system with numerous identical processing components can reassign a task from a failed processing component to another.

Finally, *reinitialization* means a restart of the system, bringing the system back to its initial state. In case of a complex hardware, this is mostly achieved by turning the system off and on again. In case of a sequential circuit this equals to a reset of the circuit.

A rollback or rollforward is usually followed by fault handling, especially if a hard fault is expected in the system. For instance, let some data read from a flash memory block has a bit error. The error is corrected with the help of Hamming code, but the error handling determines that it is a hard fault. So, fault handling proceeds and marks the damaged area in the flash memory that this area is not used in future, otherwise the hard fault (the damaged flash cell) can be reactivated again and cause another error. In this example error recovery is done by compensation and isolation.

Fault handling can also precede error handling, if fault handling can react faster than error handling. For instance, assume a sequential circuit in radiation environment. During operation the circuit detects an error using parity. As the system was designed for radiation environment, the system assumes this is a soft error. Error handling is done using rollforward by a reset in the component, where the error is detected, but the reset takes many clock cycles. So, the system isolates the component that the error does not propagate to other components, by immediately logically masking the primary outputs that can propagate the error, for instance the control signals like *write enable* in memory interfaces. This isolation is called fault isolation, because if an error propagates to the neighboring system, it is an external fault from the neighboring system's perspective. Note that this

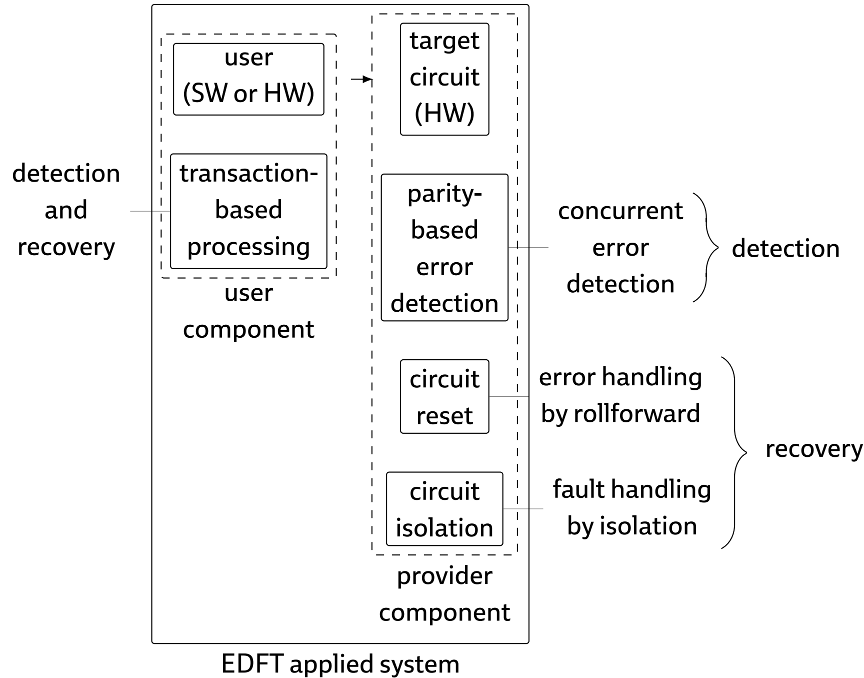

Figure 2.8: EDFT applied reference architecture labeled according to the taxonomy in [Avi+04]

example reflects the idea behind our recovery technique in chapter 6. In this case fault tolerance is achieved by concurrent error detection with on-demand system recovery, where fault handling (isolation) is done before error handling (rollforward).

Error detection and system recovery is also abbreviated as *detection and recovery*.

Figure 2.8 shows error detection–based fault tolerance (EDFT) applied on the reference processing architecture from section 1.1, which is labeled according to the taxonomy we introduced in this section.

2.2 Transient effects on sequential circuits

This work concentrates on the transient bit errors in the flipflops of sequential circuits, which are mainly caused by radiation. Although we have introduced many terms and concepts for dependability in last sections, we will further introduce terms used in fault tolerance against radiation-induced errors. These are important for understanding our fault model in this work. Different works on fault tolerance can have different working terms and we will present the definitions introduced in [Pet11, ch. 2], but also add some remarks on the use of these terms in fault tolerance community.

A local effect caused on a system by interaction of a single energetic particle is called a *single event effect* (SEE). Other (not caused by only a single particle) radiation effects are due to cumulative dose of these particles on a long term, e.g., total ionizing dose, which can affect the circuit performance in long term. An SEE can result in local corruption of information stored in a node, which is called a *single event upset* (SEU). In other words, an SEU is a corrupted electrical state. An upset in turn can result in *transient*, *permanent*, or *static errors*.

Transient errors are visible as deviations from the normal signal state in a limited time interval generally less than the duration of a clock cycle, for instance a transient peak on the output of a circuit gate that lasts only a fraction of the clock cycle duration.

Permanent errors are mostly caused by damage on circuit components, e.g., a destroyed power transistor due to radiation. These errors are also called hard errors. Permanent errors are not the motivation of this work.

Static errors, which are caused by transient errors getting latched by the circuit, can be corrected by, e.g., a reset, and these errors are also called *soft errors*. Soft errors often happen in the memory elements in form of bitflips and if uncorrected, these may propagate through the circuit and may lead to a system failure. Nevertheless, there are many inherent structures on a circuit, which prevent the radiation-induced faults from causing errors. These structures are latching window of sequential components, as well as electrical- and logical-barriers of combinational components [Lid+94].

Note that according to the taxonomy in [Avi+04], a soft error can be caused by soft permanent faults or transient faults, which is a more broad definition. In fault tolerance community, soft errors are usually transient bit errors, which are caused by SEEs, and which can be recovered from by a reset. As this work is motivated by radiation-induced transient faults in flipflops, we will refer to the bit errors by using the term soft error, same as how the fault tolerance community calls it.

Moreover, the term transient error introduced formerly is usually called single event transient (SET) in fault tolerance community, but [Pet11] does not use this term at all. Also, the term SEU is used in [Pet11] in a more general context making the transient pulses on a net an SEU, so these transient pulses on electrical nets are seen as a corrupted electrical state. Even the definitions make sense in their context, in fault tolerance community, these two terms are mostly used as follows:

- SEU as bitflips in memory components
- SET as transient pulses on combinational nets

As working terms, we will use SET for transient voltage pulses on a circuit net, and SEU or bitflip for flipped bit in a flipflop, as these terms are more common in the fault tolerance community.

Figures 2.9 and 2.10 summarize the discussion about different terms used in [Pet11] and fault tolerance community.

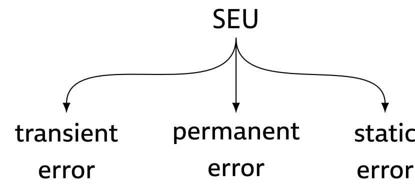

Figure 2.9: Classification of SEUs according to [Pet11]. [Pet11] does not use the term SET.

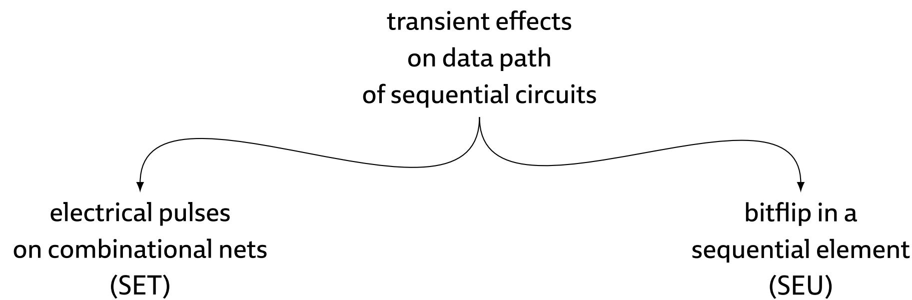

Figure 2.10: Compared to [Pet11], the terms SET and SEU are used to differentiate the effects on combinational on sequential elements. These terms will also be used in this work as working terms.

2.3 Fault model used in this work

The motivation of this work is to recover a sequential circuit from the erroneous state caused by the bitflips in flipflops, and bitflips can be caused by various ways. In this section, we describe the faults that we are hardening our system against.

SEUs and SETs are the most common functional transient radiation faults 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 flipflop, this transient can lead to a bitflip in the flipflop. But an SET can also happen directly on a net inside the flipflop itself and possibly flip the state of the flipflop. An upset of the flipflop bit due to a single energetic particle is called an SEU.

Bitflips due to SETs are rarer than SEUs, because:

- a striking particle must induce enough energy on a circuit net to cause an SET, which depends on the electrical capacitance of the regarding net
- the combinatorics must pass the SET to the input of a flipflop to have the chance to be registered
- the SET must be effective during the time window when a sequential element is transparent that the SET gets registered

With shrinking feature sizes, the electrical capacitance of circuits nets decrease, which increases the error rate due to both SETs and SEUs. With increasing frequencies, especially the probability that an SET gets registered increases. Therefore the errors due to SETs are frequency dependent.

An SET, and thus also an SEU, are asynchronous events by nature. If an SET occurs during setup or hold times of an flipflop, this can lead to metastability and thus to a indeterminable state of the flipflop. 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 flipflops [Ber08]. Consequently, a sufficient fault tolerance strategy against functional errors should accommodate both temporal and space redundancy.

In this work, we focus only on SEUs which occur directly inside the flipflops, 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 clock

Consequently:

- the faults appear as single bitflip errors
- an SEU happens inside a single clock cycle and 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

We focus only on SEUs, because most of the evaluations in this work are based on the the well-known FPGA for space applications, the ProASIC3. According to [PGG11], bitflips caused in ProASIC3's flipflops are mainly due to SEUs. ProASIC3 is discussed in section 2.6 more in detail.

With feature sizes further decreasing, one would expect that the errors due to SETs increase compared to the errors caused by SEUs. Recent technology nodes show an opposite behavior. For instance, [GSZ09] states that the error rate due to combinational elements is below 30% of error rate caused by sequential elements at 32 nm feature size, even it was predicted that the contribution of combinational and sequential elements should be equal at this technology node. [Sei+12] states that the error rate of 22 nm technology shows very small increase in error rate due

to combinational SETs compared to sequential SEUs and notes that the error rate due combinational SETs remain below the projections in earlier publications.

We assume that SEUs happen on a discrete time domain and synchronously to the circuit clock, because an analysis on a continuous time domain depends on the path delays of a routed circuit, and on the setup- and hold-times of the flipflops.

2.4 Fault tolerance techniques against SEUs

The fault tolerance against SEUs can be implemented at various abstraction levels of a computing system, e.g.:

- fabrication process level
- chip layout level
- logic level
- architecture level
- software level
- algorithm level

Some fault tolerance techniques are based on combination of techniques present on many abstraction levels. For example, a software component reacts to an exception which was caused by the arithmetic unit of a processor.

At the lowest level of abstraction, a digital circuit can be made fault-tolerant by selecting special materials or a special chip manufacturing process. Hardening at this level is usually called *radiation hardening by process* (RHBP). When the chip manufacturing process is fixed, we arrive at the design level. Design means using the available building blocks to create a system, where the building blocks usually start at transistors can go to individual software modules and further. Hardening a system at design level is also called *radiation hardening by design* (RHBD). RHBD depends strictly on the wise use of components that the designer has access to. For instance an FPGA circuit designer can only use the building blocks of the chosen FPGA, namely the configurable logic blocks (CLB) and the routing infrastructure (also called interconnect). It is noteworthy that some systems which are not explicitly hardened can still show an inherent fault tolerance against radiation. [Bla12] calls this kind of hardening by luck *radiation hardening by serendipity* (RHBS).

Now, we will traverse through various abstraction levels and give example fault tolerance techniques at each level.

Implementing a system at a high abstraction level can be less time consuming and can come at lower costs due to reuse of existing solutions. This saving also applies to fault tolerance. For instance, implementing fault tolerance at the software

level provides more flexibility and it is usually cheaper compared to the incorporation of a special chip manufacturing process, as manufacturing a custom chip creates high costs compared to software. Therefore, many systems for deploying in radiation environment are implemented using broadly available, i.e., commercial chips along with RHBD. This rule can be also repeated at other abstraction levels: It is usually cheaper to use commercially, broadly available systems than a system for a niche market, or to develop it from scratch.

Nevertheless it is important to state that the advantage of implementing fault tolerance in higher levels is not always true. RHBD has also its limits, and dependent on the mission requirements, the designer *must* implement fault tolerance additionally at lower levels. A remarkable example is hardening against the total ionizing dose (TID) in space, which can slowly degrade the performance of a digital circuit through the mission time. At the application level of an FPGA usually there is no way to harden against TID, and hardening against TID effects is usually achieved at the process level [Bla12]. Another example is system level TMR. For instance, first it may seem trivial to triplicate an on-board computer for a satellite for fault tolerance, but the voting and the system recovery in case of an error or failure still needs a significant portion of engineering work and may not be cheap as using two on-board computers with fault tolerance implemented at lower abstraction levels.

In the next subsections, we present an overview of fault tolerance techniques against SEUs at various abstraction levels.

TMR is a well-known technique, which can be implemented on various abstraction levels of a system. Our evaluations in the coming chapters are based on TMR implemented on logic level, therefore this technique will be discussed in its own subsection.

Last but not least, it is noteworthy that we discuss in this section only about fault tolerance against SEUs, because most our fault model is based on SEUs (see section 2.3). Generally, a digital circuit for a mission-critical application in space should also pay attention not only to other radiation effects, but also incorporate additional means to ensure dependability in various abstraction levels, e.g., fault removal, fault forecasting, fault prevention, which we discussed in subsection 2.1.2.

2.4.1 Fabrication process level techniques

Fault tolerance at the *fabrication process level* or shortly *process level* resembles the means used in the respective chip manufacturing process. Usually, chips are manufactured in fabs using commercial very-large-scale integration (VLSI) processes which are aimed at high yields and low costs. For applications with special needs, other manufacturing processes can be used. For instance radiation-hardened processes incorporate dense delay elements like resistors and capacitors that can be used in the feedback path of latches. These resistors and capac-

itors can then increase the delay of the feedback path and protect against SEUs. Usually, commercial processes do not provide this kind of dense delay elements, so obtaining a similar feedback path with a commercial process could be impossible or with additional area penalties.

Another example for a process level fault tolerance technique is using *silicon on insulator* (SOI) technology in the wafer manufacturing process. [Col04, part 8.3.2] discusses the use of SOI in circuits for radiation environments and states that memories based on SOI technology have a lower soft error rate than their counterparts, which are based on the conventional CMOS (complementary metal-oxide-semiconductor) technology. For example, a comparison of SEU cross section (the number of SEUs in a specific radiation test normalized by the total number of irradiated particles and total memory bits, i.e., a normalized error rate) between an SOI and conventional PowerPC processor with similar feature size shows that cross section of the SOI processor is about one magnitude lower than the cross section of the conventional counterpart [Iro+03].

2.4.2 Chip layout level techniques

Chip layout level resembles the drawing of polygons, vias, transistors, use of cells, gates, and floorplanning to create a chip. At the low level, a chip designer can reduce the SEU vulnerability for instance by layouting in such a way that a sufficient critical charge is ensured. A higher critical charge results in a lower SEU vulnerability, because a striking energetic particle has to bring then more energy to neutralize this charge and cause a voltage level that represents the opposite logic level, namely a bitflip. [Bla12] discusses some RHBD techniques at the chip layout level not only limited to hardening against SEUs.

A well-known RHBD technique at chip layout level is *dual interlocked state cell* (DICE) for memory elements [CNV96]. DICE is based on space redundancy and is implemented as follows: A standard SRAM memory cell is based on one bistable element. Basically, DICE introduces three additional bistable elements which are chained in a loop, see figure 2.11. Every bistable element has two neighboring bistable elements, which can isolate their neighbors from each other dependent on the memory state. So, if there is an SET on one of the bistable elements, this transient is not propagated to other elements. When the transient effect has ended, the feedback from the neighboring bistable element restores the state of the corrupt element. Usually, a standard SRAM cell is based on 6 transistors and DICE uses 12 transistors, so DICE has an area overhead of 1. For instance DICE cells are included in many space-grade FPGAs like Atmel ATF280F, Aeroflex UT6325 and Xilinx Virtex-5QV [Ber12].

Radiation susceptibility of the DICE can be further improved by using a different layout approach called LEAP [Kel+10]. According to [Kel+10], LEAP-DICE improves the error rate of DICE and conventional flipflop by a factor of 5 and 2000, respectively. Additionally, LEAP-DICE flipflops are less susceptible to multiple bit-

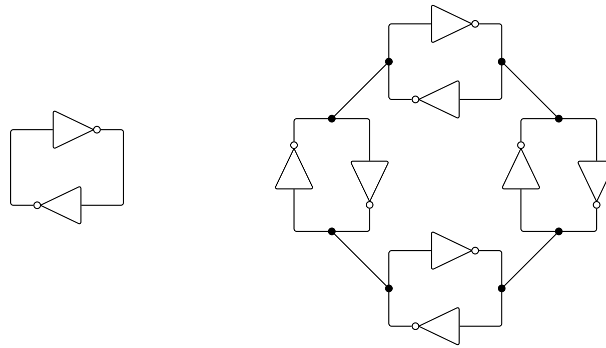

Figure 2.11: The left figure shows a bistable element, i.e., one bit memory abstracted by two inverters. On the right, a DICE cell is depicted, which consists of four bistable elements that are chained in a loop.

flips caused by a single particle.

2.4.3 Logic level techniques

Logic level techniques work mainly on the bit level can be applied on various abstraction levels that work with bits.

Error detecting codes and *error detecting and correcting codes* are based on the bit level redundancy. The aim of these codes is to achieve better results than duplication in case of error detection, and triplication in case of error detection and correction. In digital circuits, they are used in online testing [NZ98] and concurrent error detection (CED). Some examples against bit corruption are parity, dual-rail, m-out-of-n, and arithmetic code for error detection, and Hamming, BCH, and Reed-Solomon code for error correction.

Many *concurrent error detection* (CED) is based on logic level techniques. CED is based on error detection during the normal operation of a system. CED is usually implemented by an additional checker circuit, which checks an invariant property of the target system [NZ98]. Note that CED can also be implemented on various other system state properties like temperature and power, but in this work we confine us to the techniques relevant against bitflips.

In *parity checking* (called *parity-based error detection* (PBED) in this work), a parity bit is added to every data word being stored, e.g., by XORing the data bits and storing the result along with the data word. 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 system recovery scheme. Parity checking is used for instance in the level 1 cache of the processors of the IBM S/390 G5 system [SG99] and on the execution unit registers of a SPARC processor [And+03].

Error correcting codes add enough redundancy to data to enable correcting of bit errors. Hamming code is commonly used in circuits to encode memory data.

Even it can also be used for hardening the registers in a circuit, simple replication of registers is preferred (e.g., triplication and voting), because replication has less impacts on the critical path of a circuit. Some techniques are based on arithmetic properties of special functional units. For instance *residue codes* are based on the following equality:

$$(x \cdot y) \text{ mod } m = ((x \text{ mod } m) \cdot (y \text{ mod } m)) \text{ mod } m$$

Modulo for binary numbers can be efficiently implemented by calculating modulo of single bits and summing them. This principle is for instance used in a multiply unit of a SPARC processor [And+03].

If a circuit has only bitflips in one direction, i.e., only one to zero, or zero to one, then a *sum code* can be used. In sum code, the number of ones or zeros are coded in binary and attached to the information word. So, an information word with n bits has $\lceil \log_2(n+1) \rceil$ checkbits. This code is also called *Berger code* [Ber61].

2.4.4 Triple modular redundancy on logic level

A well-known RHBD technique which can be used at most abstraction levels is the *triple modular redundancy* (TMR). The principle of TMR originates from 1960s [Arm61; LV62]. In TMR one module is triplicated and the outputs of the three modules are input to a voter, which outputs the majority value. A *module* in this sense can be anything from a whole system to a small functional block or simply a gate.

In coming chapters, we will evaluate our proposed fault tolerance technique and this evaluation includes also a comparison to the state-of-the-art hardening technique TMR on a flash-based FPGA. Due to this reason, we present some common TMR techniques applied on the application level of an FPGA.

There are various TMR techniques based on the reliability requirements of a circuit. Following list depicts a list of TMR techniques for FPGAs according to [Ber08], which can be applied at the application level of an FPGA. These principles can also be applied to digital circuits:

- local TMR
- distributed TMR
- global TMR

In local TMR (LTMR), a combinational net being registered by a flipflop is connected to two additional flipflops and the outputs of the three flipflops are connected to a majority voter. The distributed TMR additionally triplicates the combinational data paths, so the combinatorics including the majority voter is also triplicated. Finally, the global TMR takes also transient effects on clock nets into account and triplicates the clock net, where every clock net supplies one particular flipflop of a triplicated data path. Local, distributed, global TMR and their differences are illustrated in figures 2.12, 2.13 and 2.14.

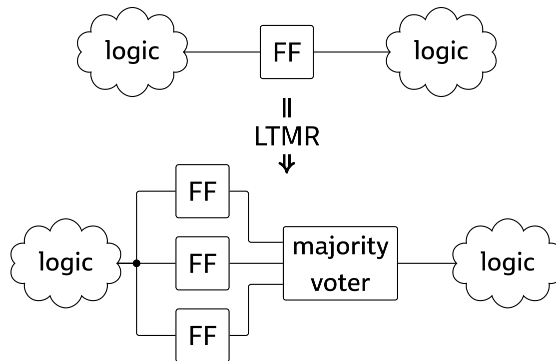

Figure 2.12: Application of local triple modular redundancy on user logic

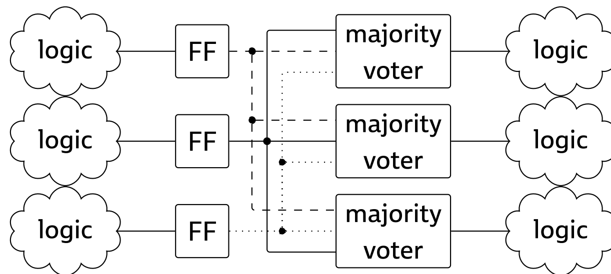

Figure 2.13: Distributed TMR triplicates every data path. So, every data path for combinatorics requires its own majority voter compared to LTMR, where the data path for combinatorics is shared.

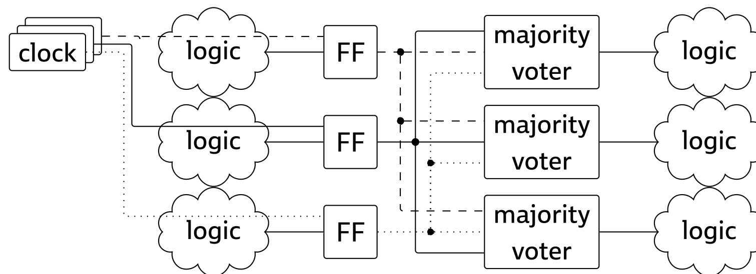

Figure 2.14: Global TMR additionally triplicates the clock lines for each data path compared to DTMR

LTMR protects against SEUs, but if an SET on combinatorics is latched by a flipflop, it leads to an SEU. With increasing circuit frequency, the latching probability also increases. DTMR makes the SEUs due to SETs frequency independent, but SETs can still happen on global clock nets and affect many flipflops at once. Global TMR solves this problem by additionally triplicating the clock lines.

In this work, only bitflips in flipflops are considered. Consequently, the LTMR is used as the compared TMR technique in our evaluations.

Presented TMR techniques detect and correct a single bit error on an flipflop locally using a majority voter, hence the TMR can be automatically applied on top of a circuit. This makes TMR functionally transparent to the rest of the system, consequently the circuit mostly does not require a redesign before mapping to an FPGA.

2.4.5 Architecture level

Architecture or micro-architecture resembles the specific implementation of a processor, including how the pipeline is structured, how many cores are integrated, how is the cache organized.

Fault tolerance techniques at architecture level try to exploit the flexibility provided at this level. For instance, instructions can be checked for integrity or threads can be run multiple times to detect errors.

Modular redundancy approach is also used to detect and correct errors in functional units of a processor core. For example, IBM S/390 G5 processor uses duplicated instruction-fetch and -execution units in the pipeline. If the outputs are the same then the recovery unit places the new state for the pipeline in a buffer. If an error is detected, then the instruction is retried, if the error repeats, then the processor halts [SG99].

Modular redundancy causes at least 100% overhead. Alternative approaches use abstracted information of an application to monitor the pipeline. For instance, the static control- and data-flow graph of an application binary can be loaded into a monitoring unit, which checks then for error during runtime [MBS08].

2.4.6 Software level

Software approaches generally work on the instruction level by augmenting the compiled binary with additional instructions for checking the control- and data-flow of an application. Well-known techniques are runtime software assertions, control- and data-path checking, and instruction duplication. Some techniques will be discussed in chapter 3 in detail.

The main advantage of software approaches is the flexibility due to higher level of abstraction. The application developer can use this flexibility to implement the fault tolerance needed by a specific application.

The downside of software-only approaches is the limited error rate improvement. For example, a software application on a processor with LEAP-DICE hard-

ened flipflops can achieve an error-rate improvement factor up to 5000, where a software-only approach based on duplicated instructions can achieve only up to 38 [Che+16b].

2.4.7 Algorithm level

The algorithm level techniques exploit the characteristics of a specific algorithm to check the integrity of intermediate or end results. For this purpose, the algorithm is augmented with additional checks and recovery.

For instance, [HA84] detects and corrects errors in matrix computations by augmenting the operands with additional checksums and distributes the computations to multiple processing units to avoid single point of failure.

The difference of algorithm- to software-level is that algorithm level techniques are application specific, which is also the main disadvantage.

2.5 FPGAs used in mission-critical applications

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.

FPGAs store the software for their circuit programming information, i.e., configuration, in the configuration memory. Currently, the commercially available FPGAs used in mission-critical applications use the following types of configuration memory:

- SRAM
- flash
- antifuse

The majority of the off-the-shelf FPGAs are SRAM-based. Known companies for SRAM-based FPGAs are Xilinx, Altera, Siliconblue (acquired by Lattice Semiconductor in 2011) and Atmel. Most of the commercially-available FPGAs manufactured by these companies are not designed for space. Although it is possible to use some of the ordinary (not mission-critical applications) FPGAs in noncrucial experimental payloads in space, for mission-critical applications like onboard data handling unit, space-grade FPGAs are preferred. Some SRAM-based FPGAs for space are Virtex-5QV (also referred as single event immune reconfigurable FPGA (SIRF)), Virtex-4QV designed by Xilinx, and ATF280F, ATFEE560 (two ATF280Fs with two EEPROMs in a package) designed by Atmel. All the mentioned FPGAs but the Virtex-4QV have built-in fault tolerance against radiation-induced faults and usually no further hardening on application-level (e.g., applying TMR at the netlist level) is needed [Xili14; Atme15a; Atme15b].

Table 2.1: Number of SEUs in a circuit with 5000 flipflops and 8 Kib memory during a one year mission in L2 orbit under 1/cm^2 shielding for different FPGAs based on a fault model. Data taken from [BSV11, ch. 7].

device	conf. mem.	RAM	FF
Virtex-4QV	344430	3747	2188
RTPE3000L (RT ProASIC3)	0	62	4
ATF280F	~ 0	~ 0	~ 0

The flash- and antifuse-based FPGAs were brought to market by Actel, which was acquired by the semiconductor company Microsemi in 2010. Another manufacturer of antifuse-based FPGAs is Aeroflex. The antifuse and flash memories have lower vulnerability to radiation-induced faults compared to SRAM, therefore FPGAs based on these memory technologies are popular in radiation environment. Another advantage is the instant availability of the FPGA application after powerup, because the configuration does not have to be loaded from an external memory compared to SRAM-based FPGAs. Moreover, SRAM-based FPGAs need usually external non-volatile configuration memory to be additionally deployed on the system board or in the chip package (e.g., ATFFE560 FPGA). Last but not least, antifuse and flash memories consume less power than SRAM, because SRAM is volatile, in other words, energy is needed to keep the data on SRAM. Some popular space-grade FPGAs from Microsemi are RTAX [Micr15b], RT ProASIC3 [Rez10] and RTG4 [Micr16]. Aeroflex provides the antifuse-based FPGA UT6325 [Aero13].

RTAX is antifuse-based and was the main FPGA choice for space applications before the space-grade flash- and SRAM-based FPGAs were available. The fact that this FPGA is available more than ten years ([Wan04]) gives RTAX also an additional advantage of heritage. This is contradictory to the short life cycle of commercial digital circuits, but heritage of components is one of the key factors in space that can be seen as on-field testing of a component and contributes to the trust attributed to the component.

The most important drawback of antifuse- and flash-based FPGAs is that they often do not provide much resources as their SRAM-based counterparts. In the sparsely populated area of space-grade FPGAs, Microsemi recently introduced the FPGA RTG4 with comparable resources, though. Another drawback is the limited reprogrammability compared to SRAM. Antifuse-based FPGAs are one-time programmable and flash memories have usually a limited program/erase cycle.

Table 2.1 compares vulnerabilities of three different FPGAS: one FPGA with built-in fault tolerance (ATF280F), and two FPGAs that have to be hardened on the application level, SRAM-based Virtex-4QV and flash-based RTPE3000L.

2.6 Microsemi ProASIC3 FPGA

In our evaluations we use a popular off-the-shelf flash-based field-programmable gate array (FPGA) for mission-critical applications, the ProASIC3. ProASIC3 FPGA family was introduced back in 2005 [Mor05] by the company Actel. In the following years, other family products based on the same architecture with additional features like more interfaces and low-power were introduced, especially the RT ProASIC3. RT stands for radiation-tolerant. RT ProASIC3 (also called RT3P) is based on a low-power product (A3PL) and is available (introduced in 2010 [Rez10], commercially available since 2012 [EEJo12]) in an airtight (hermetically-sealed) ceramic package and is tested against military standards. The flash-based configuration memory and the availability of a special chip package for extreme environments as well as additional testing against military standards (RT ProASIC) makes ProASIC3 very popular for mission-critical applications like aerospace.

As mentioned in section 2.5, heritage is a key factor in mission-critical applications. Even the ProASIC3 architecture dates back to 2005 and its space-grade package is available since 2010, it is still state-of-the-art for space missions [VSC15; Tre+14].

Usually, FPGAs realize a given application by using the building blocks available on the chip, namely configurable logic blocks (CLB). In ProASIC3, the CLBs can be either configured as a flipflop or three-input look-up table (LUT), which is contrary to popular CLB architectures where a CLB can simultaneously be configured a flipflop and LUT.

ProASIC3 is based on a semiconductor process with 130 nm feature size. According to irradiation tests on RTPE3000L [PGG11], the direct SEU effects inside the flipflops are more significant compared to the SETs on combinational components latched by the flipflops. Due to the same reason, [PGG11] observed that hardening against SETs using filters on the flipflop inputs (ANDing the delayed and undelayed flipflop input signal) does not have any significant effect on the error rate of the irradiated circuit.

2.7 Error detection-based fault tolerance

After the introduction of fault tolerance concepts, we present our approach more in detail.

In our approach, the target system that has be hardened can be abstracted as two systems. The first one is hardware, i.e., is implemented as a circuit, and provides a service. The second one is the user, which can be both hardware or software. Figure 2.15 visualizes these two systems.

On hardware, detection of an error requires space or time redundancy, but often less redundancy resources than both detection and correction. If the resources on a device are scarce and costly, then implementing a local error correction scheme can become a hurdle. In this case, system recovery can be done

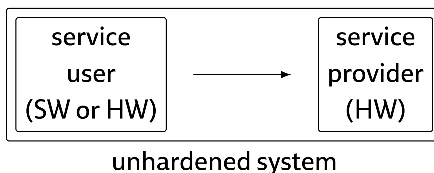

Figure 2.15: Abstract model of the reference architecture presented in section 1.1

by recomputation, e.g., by retrying the last processing request in software or additional circuit buffering the last request. Issuing a non-local error correction requires more recovery time than a local correction, beginning from the detection until the hardware is recovered from the erroneous state and recomputation is done. Nevertheless, if the error rate of the system is low, then an on-demand system recovery can be practicable.

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, then the overhead of checkpointing in the circuit may be eliminated by reissuing a processing request. Examples for such a module are a protocol converter or simply a module which exchanges data between two modules after reformatting data. These modules do not have to store an information for a long time and have a defined state after a chunk of data or a transaction is processed. The example circuit B that was presented in figure 1.1 falls also in this category, as it only exchanges data between two modules and moves to its initial state after a request is processed. If an error occurs during processing of a request, then the error handler can reset the processing module and flag an error to the processor that a processing request can be reissued, i.e., software-based retry. Alternatively, instead of flagging, the request can be reissued after a nonresponsive timeout. In this case, the time penalty caused by an error is negligible, if the FPGA SEU rates during a mission due to space radiation are low.

We refer to this technique as *error detection-based fault tolerance* (EDFT) in this work. We evaluated EDFT using parity-based error detection, circuit reset, and circuit isolation on the service provider side, and transaction-based processing on the service user side. Figure 2.16 shows EDFT applied on the system already shown in figure 2.15. EDFT's components will be presented and evaluated separately using example implementations in next chapters.

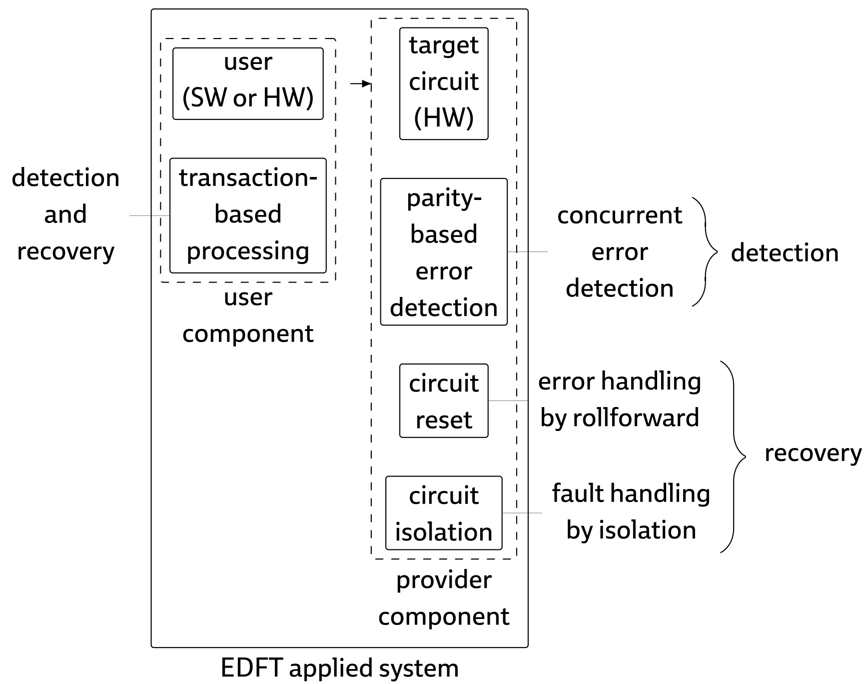

Figure 2.16: EDFT applied on the system in figure 2.15. The components are classified according to [Avi+04]

Chapter 3

Related work

In section 2.4, we gave an overview of fault tolerance techniques at different abstraction levels. In this chapter, we discuss close related work.

An important contribution of this work is that it evaluates a concrete implementation of parity-based error detection with recomputation on a known FPGA for space applications. To the best of our knowledge, there is no work in literature, which evaluates a similar fault tolerance technique on a similar device. So, we will present related work from a more general perspective.

Our work proposes a hybrid approach by combining fault tolerance at software and hardware level to use the advantages of high-level and low-level fault tolerance techniques. We will present first a well-known software-level fault tolerance approach, then a work showing the limits of software approaches. Finally, we discuss a more general and recent work, which combines well-known fault tolerance techniques from various levels to achieve a system fault tolerance in terms of area, timing, power, and error detection improvement.

3.1 Error detection by duplicated instructions

EDDI's fault model is based on single bitflips in a processor. EDDI tries to detect the errors which happen during program execution by executing the instructions twice on two different sets of general purpose registers and program memory addresses. The error detection happens before executing a *branch* or *store* instruction. EDDI is applied as follows.

Firstly, a instruction dependency graph for the program is generated, which shows the data dependencies between particular instructions and plays an important rule in instruction scheduling. Then, using the sequenced instructions, the *storeless* basic block graph (SBB) of the program is constructed. First let us explain the concept of basic block.

The concept *basic block* is well-known in compiler design. It defines a sequence of instructions, where it is always guaranteed that every instruction in the sequence

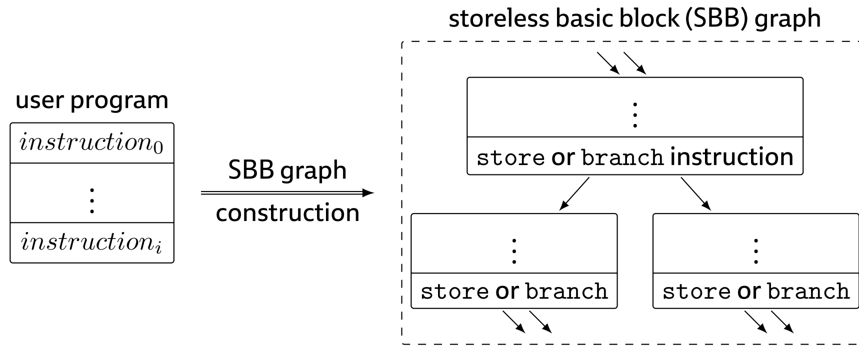

Figure 3.1: Storeless basic block (SBB) construction. The well-known concept of basic block is augmented by the property *storeless*, because the error detection is done before a branch or store instruction in EDDI.

will be executed before the coming instructions in the sequence. Put differently, a basic block can only be entered by a jump to the first instruction, and exited by using the last instruction in the sequence without any jumps to the instructions between the first and last instruction. Examples for the last instruction in a basic block are jump, branch, and return instructions.

EDDI augments the basic block concept by adding the *storeless* property, because the error detection happens prior to a branch or store instruction. SBB construction is illustrated in figure 3.1.

The fault model of EDDI is based on bitflips in memory such as registers in the processor, program and code memory. To detect the bitflips in registers and program memory, these are partitioned in master and shadow sections. The instructions of the user program are limited to the master components. For illustration, see figure 3.2.

Then, the instructions are duplicated, and transformed such that the duplicated instructions operate on the shadow registers and memory, see figure 3.3 for illustration. The duplicated SBB is then added to the dependency graph.

After instruction duplication, the instructions for error detection are introduced into the dependency graph. The instruction compares the registers which decide the outcome of a branch instruction, or which will be stored in memory, and jumps to system recovery code in case of a mismatch. See figure 3.4.

Finally, the scheduling is carried out. The instructions must be scheduled such that the error detection probability is high. For instance, if the master and shadow (duplicated) instructions are interleaved, i.e., if a shadow instruction always follows a master instruction, this leads to an error detection probability of about 0.5 for bitflips causing unintentional jumps in code. In case of interleaved instructions, if the unintentional jump is to a master instruction, then both the master and shadow instructions will operate on the same data values and the comparison at the end of an SBB will not yield an error. The solution presented in [OSM02] is

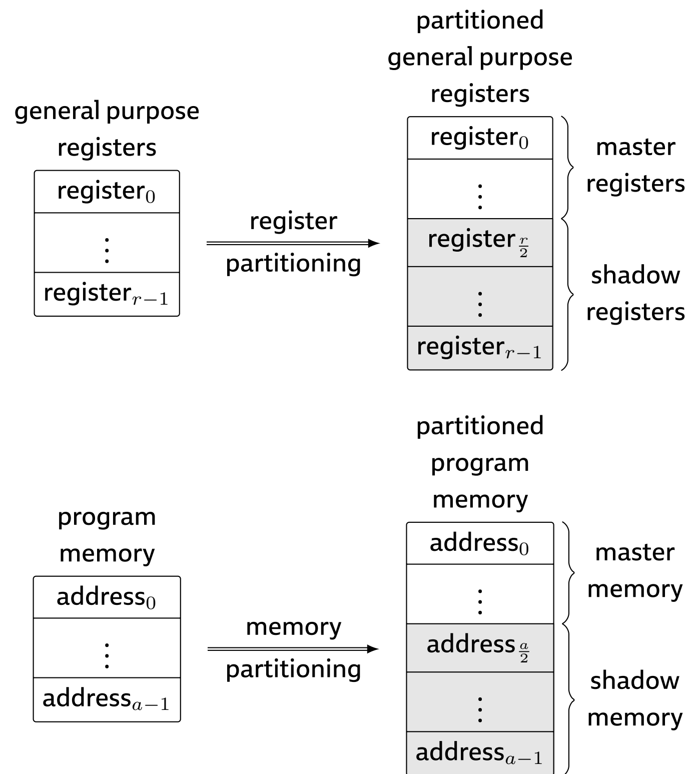

Figure 3.2: Partitioning of general purpose registers and program memory

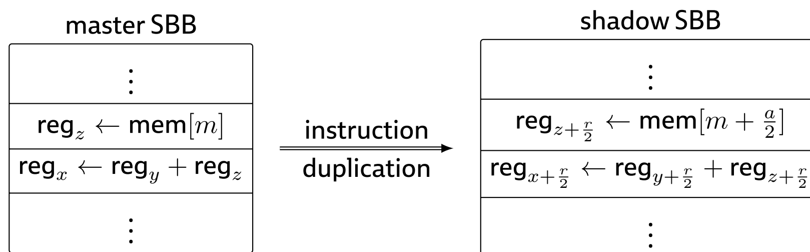

Figure 3.3: Instructions of the user program are duplicated and transformed to operate on the shadow registers and memory

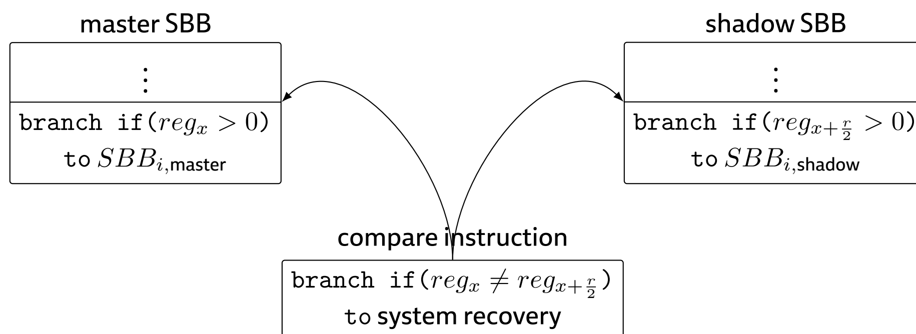

Figure 3.4: Error detection is done before a branch or store instruction. For this purpose, the register which decides a branch or will be stored in memory is compared to its shadow.

to have two master instructions at the beginning of an SBB.

There are also other errors which can go undetected, but they are not covered here. The error detection coverage is dependent on the application. [OSM02] states an error detection coverage of about 0.98 to 0.99 for EDDI according to fault injection experiments. The same measure for unhardened programs varies from 0.93 to 0.54.

EDDI causes an execution time overhead due to duplicated instructions and added comparisons, which should be greater than 1. But the authors' are motivated by a superscalar processing architecture, where EDDI can be used to maximize instruction level parallelism.

EDDI's execution time overhead is also application dependent and varies between 0.45 and 1.14 on a superscalar processor which can issue four instructions per cycle.

Duplication technique can also be implemented on source code level, e.g, duplicating the variables and operations on the variables in C code and comparing a variable with its duplicate whenever a variable is used like in figure 3.5 [Reb+99]. This approach lowers the error detection coverage and also yields a worse execution overhead.

A recovery procedure must be implemented in EDDI, but [OSM02] do not give any details about the recovery procedure nor if the recovery was included in the evaluation.

It is also important to note that the fault injection was done only on the code section of the program. Fault injection on the flipflop level can lead to less error detection rate, e.g., [Che+16b] states an error coverage of 0.86.

Moreover, [Che+16b] notes that reading and comparing the values after storing them to memory (store-readback) can lead to a higher error detection coverage.

user program

```
int a, b;
⋮
a = b+5;
⋮
```

hardened program

```
int a, b,
  a_dupl, b_dupl;
⋮
a = b+5;
a_dupl = b_dupl+5;

if (a != a_dupl)
  recovery();
⋮
```

Figure 3.5: Application of the EDDI technique on source code level. Every variable is duplicated. An operation on a variable is repeated on its duplicate. After a variable is used as a tight operand, the variable is compared to its duplicate. A recovery procedure handles the error.

3.2 Limitations of software-based techniques

In section 3.1, we discussed EDDI, a well-known software-based fault tolerance technique, which uses only bitflips on memory elements as the fault model and is evaluated on a superscalar processor. In this section, we discuss [Aza+11], which evaluates additional software-based techniques and also includes errors caused by transients in combinational components. This work also analyzes the fault coverage contribution of the evaluated software techniques one by one.

The compared techniques are:

- instruction duplication (EDDI)
- signatures for basic blocks
- inverted branches

While EDDI concentrates on the data, the latter two techniques try to observe the control flow by detecting unintentional jumps in the program flow. For instance, an error affecting the program counter can lead to such an error, which can sometimes cannot be detected by EDDI.

Signatures for basic blocks can be used to observe the program flow. In this technique, a distinct signature given to every basic block. The signature is loaded to a global memory resource whenever a basic block is entered, and the signature is checked, whenever a basic block is exited. This principle is illustrated in figure 3.6.

The technique *inverted branches* is based on duplication of branch instructions for checking whether the branch operation was executed correctly or not. A branch operation has generally two possible jump positions, either the next ad-

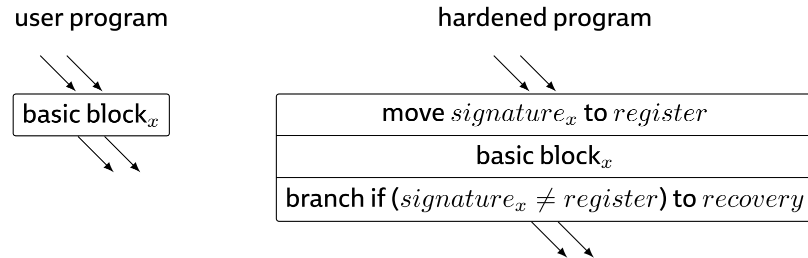

Figure 3.6: Error detection in the control flow of software by using signatures for basic blocks. The signature is loaded at the beginning and checked at the end of a basic block.

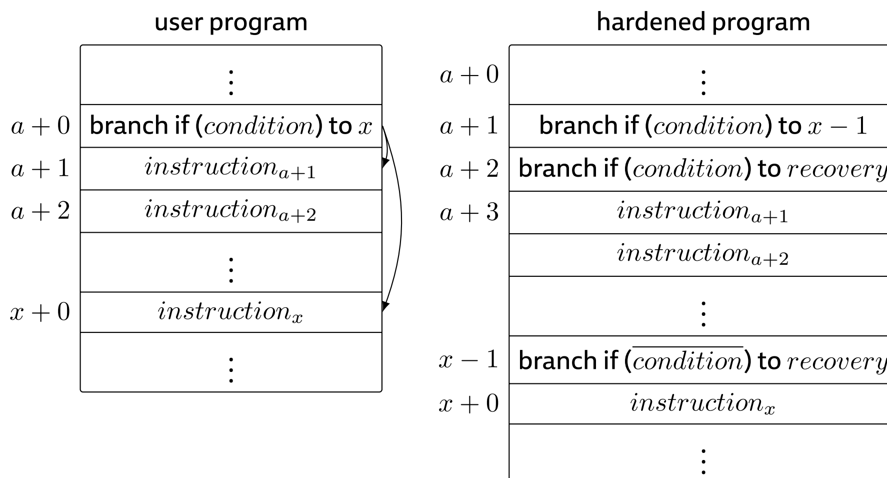

Figure 3.7: Error detection in the control flow of software by duplication of branch instructions

dress or the branch address, so the duplicate is placed on both potential destinations. Branch instruction duplication and jump to recovery can be fusioned in one instruction by inverting the branch condition in the branch instruction placed at the branch address, and leaving the branch condition the same in the next address, which should be taken if the branch condition is false. The technique is illustrated in figure 3.7.

Inverted branches can detect a wrong branch decision, because:

- if the branch condition should be *true*, but the program does *not* branch, the branch condition in the branch instruction at the *next* address will be true, and the program will proceed with recovery
- if the branch condition should be *false*, but the program does branch, the *inverted* branch condition in the branch instruction at the *branch* address will be true, and the program will proceed with recovery

assuming that the probability that both two sequential branch instructions are affected by an error is low enough that the duplicated branch instruction will detect the error.

Fault injections on sequential and combinational components of a processor show a fault coverage of:

- 0.77 to 0.84 for EDDI
- 0.04 to 0.09 for signatures for basic blocks
- 0.01 for inverted branches

EDDI can detect most of the injected faults, where the latter two techniques can not. For all techniques combined, the authors state a fault coverage of 0.79 to 0.88.

The authors state that most of the undetected errors are due to unintentional jumps from a basic block to the same basic block. So, the authors suggest adding additional fault tolerance, but state that full fault coverage is unlikely to be reached.

The presented software level techniques are very flexible but their fault coverage is probably insufficient for most mission-critical applications. In our approach, we propose a hardware error detection approach to catch the errors directly on the bit level and so reach almost full fault coverage. If multiple bit errors are not allowed, our approach leads to full fault coverage (discussed in section 4.2.4).

3.3 Cross-layer exploration for architecting resilience

The work [Che+16a] proposes a fault tolerance framework spanning various abstraction layers of a system. The framework combines known fault tolerance techniques to find cost-effective, area- and power-efficient combinations.

Compared to the conference publication [Che+16a], the extended eprint version [Che+16b] contains details of the parity checking approach, which is important as related work and will be discussed separately from the general aspects of this work.

First we discuss the general aspects, and then parity checking used in this work.

3.3.1 General discussion

The fault model of the work are the single and multiple bitflips on general purpose processors due to radiation in terrestrial environments. An in-order and a more complex out-of-order processor are used in the evaluations. The evaluated techniques include some of the techniques that we presented in section 2.4 and in this chapter, e.g., an improved version of DICE, parity checking and EDDI. Where error correction techniques can be evaluated alone, the error detection techniques are

analyzed both alone and by combining with recovery mechanisms for processors, e.g., instruction retry.

The authors emphasize the importance of an automatized approach for implementing fault tolerance, as fault tolerance is generally implemented on systems based on experience and common practice. So, they propose a cross-layer fault tolerance by combining low- and and high-level techniques and picking the combination that provides the error rate improvement needed by a mission.

The framework consists of four components:

- reliability analysis using bitflip injection and execution time evaluation using RTL models and benchmark application
- area, power, energy, clock frequency evaluation by physical synthesis and layout
- a resilience library consisting of ten error detection and correction techniques and four system recovery techniques for general purpose processors
- an evaluation component, which compares the error rate improvements of different combinations

Compared to the related work presented in 3.1 and 3.2, the authors use a more detailed classification of erroneous outcomes in the benchmarks:

- silent data corruption (SDC)
- detected but uncorrected error (DUE)

SDC happens if the system cannot detect an error, the system continues processing and the error corrupts the program output. In DUE, the system also cannot detect the error, but the system crashes and is not usable without human intervention. In context of this work, if the benchmark program terminates normally, but the program output data differs, this corresponds to an SDC. If the benchmark program terminates unexpectedly, does not terminate in two times the nominal execution time, or if the system recovery is not successful after an error is detected, this is a DUE.

According to the concepts we introduced in section 2.1.1, both SDC and DUE are failure events from the user perspective, because corrupted program output and a system crash are likely unwanted events for the user. The severity of these two failures depend on the service expectation of the user. If the user expects that the service should run without interruption, then a DUE is a more severe failure. On the other hand, if an incorrect program output should be avoided, then the user can favor a DUE instead of an SDC. All in all, a mission-critical system must not have any failures.

As a fault tolerance measure to compare the evaluated techniques, mainly SDC and DUE *improvement* are used, where improvement is defined by comparing the

number of erroneous outcomes of the unhardened and hardened design after a benchmark run:

$$\text{improvement} = \frac{\text{number of erroneous outcomes of the unhardened design}}{\text{number of erroneous outcomes of the hardened design}}$$

The concepts SDC- and DUE-improvement are better measures for comparing the fault tolerance techniques compared to error detection coverage, because not all flipflop-bitflips on a processor leads to an SDC or DUE. This means an unhardened design can have already an intrinsic error detection coverage, if only the program output is observed for evaluations. For instance, the authors found out that about 39% of flipflops in the out-of-order processor design do not lead to an erroneous outcome during the benchmarks at all.

The flipflops not leading to any erroneous belong to components like branch prediction or trap register, which do not play a crucial role for the correctness of the system but performance. In this case, these components can be left unhardened.

The authors advocate the fault model of bitflips in flipflops, as this model is sufficient enough to reflect the actual behavior of current systems, and test only for errors at the flipflop level. They additionally mention that injecting faults at higher levels, e.g., at register or application level, can cause highly inaccurate results.

For instance, EDDI achieves an SDC improvement factor of about 3 when bitflips are injected on the flipflop level, 2 on the register level, and 13 on the application level when the bitflips are injected into program variables.

The authors mention dual and triple modular redundancy (DMR, TMR) at the architecture level, but do not evaluate them due to their high overhead of about 200% and 300% in area and power.

High-level techniques at the software and algorithm level do not provide an SDC improvement of more than 38 and therefore they propose augmenting low-level techniques at the circuit and logic level.

The framework found out that a combination of algorithm-based fault tolerance, parity, LEAP-DICE and architectural recovery approach can achieve an SDC improvement factor of 50 with about 1% area, 2% power, and 3% energy overhead for the in-order processor. So, they conclude that new approaches aim for better error rate improvements than the particular techniques used in this combination.

3.3.2 Parity checking

The authors use parity checking for flipflops as a circuit-level error detection technique. Pipelining and flipflop grouping for parity checking is also discussed.

In parity checking, parity bit is calculated for a group of flipflops. In their methodology, flipflop group size (in our work, we call this cluster size) can be 16 or 32 bits, as the authors experimentally determined that these two group sizes lead to the lowest resource costs. Parity checking is implemented for a flipflop group size of

32 flipflops. If this technique does not meet the timing of the original design, then the group size of 16 along with parity pipelining is used.

The authors do not present any algorithm details about how the pipelining is implemented on the design. Based on the presented figure in the work, we assume that this is traditional pipelining which can lead to error detection latency. In our pipelining approach in chapter 5, we use a different approach based on sequential distance.

In parity checking, flipflops can be grouped in various ways. The authors evaluated the following heuristics for creating the flipflops groups:

- fixed group size
- sorted by vulnerability
- sorted by flipflop locations
- sorted by timing slack

Their evaluations show that grouping the flipflops by their locations and using the former methodology (use group size of 32 flipflops, if timing not met, use group size of 16 with pipelining) yielded the most area-efficient and power efficient result with an overhead of 11% and 23%, respectively.

Sorting by locations was done by using the hierarchy in the design. Flipflops belonging to a processor component, e.g., instruction fetch, were grouped together. Note that we used the place and route results for grouping the flipflops in the design.

In technology nodes featuring very small feature sizes, a single particle strike can also account for multiple bitflips. To cope with this threat, in their parity checking approach the authors enforce that two flipflops in the same parity group are not adjacent, and also try to maximize the average distance between the flipflops in the same parity group.

Chapter 4

Parity-based error detection

In previous chapters we introduced error detection–based fault tolerance (EDFT). The error detection block in EDFT simultaneously checks for any bit errors in the target circuit. For this purpose a concurrent error detection approach can be used. To allow a more precise evaluation of the EDFT, a concrete error detection approach must be chosen. We chose parity checking approach for this purpose.

Our evaluation is in this chapter is based on a commercially available FPGA for mission-critical applications using benchmarks circuits. We also provide an automatized implementation of parity-based error detection. Even parity checking is a well-known technique, our work enable a more precise evaluation, and complex comparison of two different fault tolerance approaches - EDFT and LTMR.

In this chapter, we first present the idea behind parity-based error detection and provide the specification of our implementation in section 4.1. Using this specification and the reference processing architecture introduced in section 1.1, we give an analytical evaluation of PBED in section 4.2 and compare with the state of the art approach LTMR. In section 4.3, we do the comparison using synthesis results based on various circuits. Finally, section 4.4 presents the automatic application of PBED.

4.1 Concept

Parity checking is the most basic error detection technique and it is well-known [NZ98]. Parity can detect an odd number of bit errors in a data word by adding a parity bit to the data word so that the number of 1-bits in the word is even (*even parity*) or odd (*odd parity*). Upon reading the data word along with the parity bit, the parity is calculated again, compared to the used parity property (even or odd) 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.

Now, we will describe the implementation details of our parity checking approach for sequential circuits and we will refer to our implementation as *parity-*

based error detection (PBED). A circuit which must be hardened will be called *target circuit*.

In PBED, application flipflops in the target circuit are partitioned into clusters and for each cluster one parity flipflop is introduced. This problem can be formulated as follows:

- $\mathcal{F}^{\mathsf{a}} = \{f_1^{\mathsf{a}}, \ldots, f_n^{\mathsf{a}}\}$ is the set of n application flipflops in the target circuit
- $\mathcal{C}^{\mathsf{a}} = \{C_1^{\mathsf{a}}, \ldots, C_m^{\mathsf{a}}\}$ is the set of m flipflop clusters, which is a partitioning of $\mathcal{F}^{\mathsf{a}}$, where:
  - $\bigcup_{C_i^{\mathsf{a}} \in \mathcal{C}^{\mathsf{a}}} C_i^{\mathsf{a}} = \mathcal{F}^{\mathsf{a}}$
  - $C_i^{\mathsf{a}} \cap C_j^{\mathsf{a}} = \emptyset$ for $i \neq j$
  - $|C_i^{\mathsf{a}}| \leq k$
  - k is the number of flipflops in a cluster

The partitioning $\mathcal{C}^{\mathsf{a}}$ is then altered to include the parity flipflops:

- $\mathcal{F}^{\mathsf{p}} = \{f_1^{\mathsf{p}}, \ldots, f_m^{\mathsf{p}}\}$ is the set of m parity flipflops, where:
- $\mathcal{C}^{\mathsf{p}} = \{C_1^{\mathsf{p}}, C_2^{\mathsf{p}}, \ldots, C_m^{\mathsf{p}}\}$ is the set of m flipflop clusters which are hardened by parity, where
  - $C_i^{\mathsf{p}} = C_i^{\mathsf{a}} \cup \{f_i^{\mathsf{p}}\}$
  - $f_i^{\mathsf{p}} = f_1^{\mathsf{a}} \oplus f_2^{\mathsf{a}} \oplus \cdots \oplus f_q^{\mathsf{a}}$ for even parity
  - $f_i^{\mathsf{p}} = f_1^{\mathsf{a}} \oplus f_2^{\mathsf{a}} \oplus \cdots \oplus f_q^{\mathsf{a}} \oplus 1$ for odd parity
  - $\{f_1^{\mathsf{a}}, \ldots, f_q^{\mathsf{a}}\} = C_i^{\mathsf{a}}$
  - $\oplus$ is the XOR operator

In the following, we also include the XOR gates for parity generation and checking in the flipflop cluster for convenience. Figure 4.1 shows the generic implementation of the error detection in a single cluster $\text{cluster}_{\text{ED}}$ in detail. One cluster consists of k application flipflops FF_{a}, one parity flipflop FF_{p} and two XORs: one for parity generation and one for parity checking. Note that even the whole block is named as $\text{cluster}_{\text{ED}}$, only the XORs and the FF_{p} belong to the error detection module as visualized in figure 1.5.

Normally, a PBED-hardened circuit contains many clusters. To generate the error signal for the whole circuit, the cluster signals must be reduced to a single error signal. The straightforward approach for reducing the cluster error signals is to OR these signals as shown in figure 4.2. This approach will be abbreviated as *direct PBED*.

Note that there is an extended version of this approach, which reduces the cluster error signals using a pipelining approach, which will be covered and analyzed independently in the following chapter 5. This chapter is only about the direct PBED approach, therefore the short abbreviation PBED will only refer to direct PBED here.

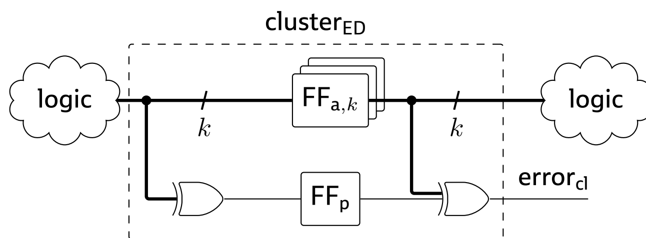

Figure 4.1: Parity-based error detection in a single cluster. A single error detection cluster cluster$_{ED}$ houses k application flipflops (FF$_a$) and one parity flipflop (FF$_p$). The (even) parity is calculated by XORing k inputs to the FF$_a$s and the data integrity is checked by XORing $k + 1$ flipflop output signals in the cluster$_{ED}$. The error signal error$_{cl}$ is active in case of an odd number of bit errors.

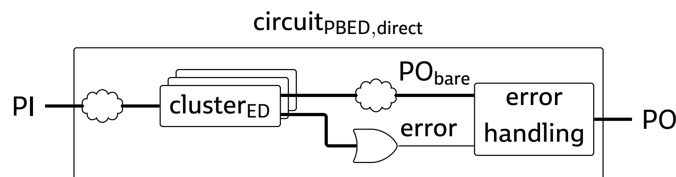

Figure 4.2: PBED-hardened circuit with direct cluster error signal reduction. The error$_{cl}$ signals (figure 4.1) are reduced by using a single logical OR gate to the signal 'error', which is then input to the error handler. The primary output of the bare (i.e., unhardened) circuit is also input to the error handling module for isolation of the circuit (refer to section 6.1 for more details).

4.2 Analytical evaluation

Like most error detection techniques applied on design-level, also PBED introduces redundancy to the circuit and thus affects the circuit area and critical path. In this section, we will evaluate these circuit impacts analytically and compare them to LTMR. Moreover, we will analyze the multiple bit error susceptibility of PBED and LTMR. The goal of this analysis is to get first theoretical limits before we proceed with the experimental evaluation.

The following subsections are structured as follows: We will first describe the prerequisites for the analysis. Then, the critical path delay and circuit area overhead impacts of both approaches will be analyzed and compared. Finally, we will discuss about the multiple bit susceptibility.

4.2.1 Prerequisites

The circuit analysis will be done by a hypothetical synthesis of the PBED-hardened circuit for the Microsemi flash-based FPGA ProASIC3, i.e., the primitives of the ProASIC3 will be used as building blocks. In ProASIC3 architecture, every configurable logic block (CLB) can be configured either as a three-input LUT or a flipflop.

Microsemi ProASIC3 [Micr15a] is chosen because it is state of the art for space missions (e.g., [VSC15; Tre+14]) and it is commercially available in a special integrated circuit package for space environment. ProASIC3's broad availability and space provenance makes it more reasonable to do the synthesis on this FPGA than using a custom ASIC design kit.

Note that in this chapter we confine the evaluation only to the error detection block to provide an independent analysis of EDFT's components. Nevertheless, we assume that the error output of the error detection module is connected to a flipflop in the error handler to enable a more precise analysis of the critical path impact.

Many of the following comparison parameters are dependent on:

- the size of one cluster s_{cl}, where $s_{\text{cl}} \geq 2$
- the total cluster count in the target circuit c_{cl}.

Consequently, the measurement parameters will be determined only by using the flipflop count in the target circuit - the combinatorics will be arbitrary in this analysis. According to the figure 4.3, $s_{\text{cl}} = k + 1$ and $c_{\text{cl}} = m$.

The parameters will be determined for $s_{\text{cl}} \stackrel{!}{=} 3^x$ and $c_{\text{cl}} \stackrel{!}{=} 3^y$, where $x, y \in \mathbb{N}$, which fits into the ProASIC3 architecture with three-input LUTs. This selection of input parameters makes the most timing-efficient use of the FPGA area for a specific logic depth. With the increasing value of s_{cl} and c_{cl} more LUTs are needed for parity generation and the reduction of cluster error signals, respectively. With increasing number of LUTs on a critical path, longer delay is introduced on this path. However, the additional delay is only proportional to the logarithm of s_{cl} and c_{cl}.

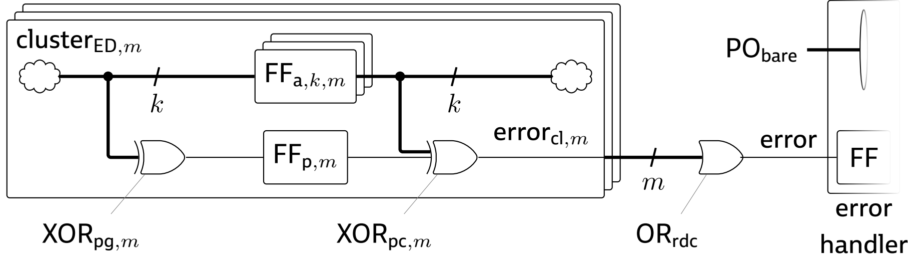

Figure 4.3: Direct PBED with labeled components for analysis. $m = c_{cl}$ error detection clusters are connected to the error signal reduction gate OR_{rdc}. $_{pg}$ and $_{pc}$ stand for parity-generation and -check, respectively.

Consequently, the critical path of a benchmark circuit only changes for different values $x, y \in \mathbb{N}$, leading to such selection of s_{cl} and c_{cl} values. This behavior is visualized in figure 4.6.

To differentiate the comparison parameters of circuit with different hardening techniques, the parameters of the bare circuit (i.e., target circuit, hardening not implemented) are labeled with the subscript $_{bare}$ and the parameters of the circuits with LTMR and PBED with $_{LTMR}$ and $_{PBED}$, respectively. An overhead in a measurement parameter by the applied technique is labeled with the subscript $_{+}$.

For the analysis of the critical path delay, interconnect delays are not considered. The interconnect delays depend significantly on the CLB placing and clocking resource utilization in an FPGA, which makes a general analysis not feasible. Nevertheless, these analytical values will be compared with experimental values in section 4.3.

4.2.2 Critical path delay

The critical path delay t_{crit} limits the maximum frequency of a circuit and increases with additional serial logic on the critical path. In what follows, we first determine LTMR's then PBED's critical path delay, and then compare them.

LTMR

In LTMR, every bit must be decoded by a majority voter (MAJ3) before it is propagated to the combinational logic, which causes an extra delay. Consequently the actual critical path delay $t_{crit,bare}$ is extended by the propagation delay of the majority voter. The subscript $_{pd}$ stands for propagation delay.

$$t_{crit+,LTMR} = t_{pd,MAJ3} \tag{4.1}$$

Figure 4.4 visualizes the critical path overhead caused by the LTMR.

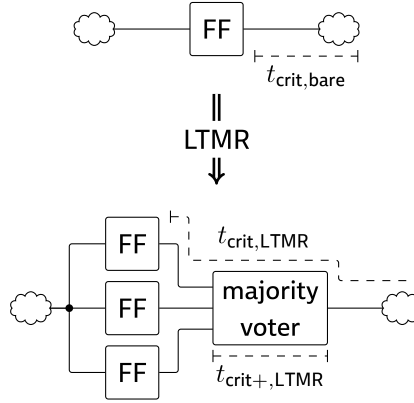

Figure 4.4: The critical path overhead of the LTMR visualized. Critical paths of the bare and LTMR applied circuit are denoted as $t_{\text{crit,bare}}$ and $t_{\text{crit,LTMR}}$. Note that even t_{crit} literally denotes the maximum time duration that a flipflop output signal requires to reach another flipflop, in this figure, t_{crit} denotes the path that this critical flipflop output signal travels.

PBED

In PBED, there are two critical path candidates:

1. the actual critical path plus the overhead added serially by PBED, i.e., the critical path of the bare circuit plus the parity generation path, $t_{\text{crit,PBED,1}}$
2. a newly created parallel path by PBED, the parity check and the cluster error signal reduction path, $t_{\text{crit,PBED,2}}$

These two paths are visualized in figure 4.5.

The first path delay can be calculated as follows: The parity has to be generated before the combinational signals are registered. The propagation delay of the gate XOR_{pg} is called $t_{\text{pd,XOR}_{\text{pg}}}$.

$$t_{\text{crit+,PBED,1}} = t_{\text{pd,XOR}_{\text{pg}}} \tag{4.2}$$

The second path $t_{\text{crit,PBED,2}}$ consists of the XOR_{pc} and OR_{rdc}.

$$t_{\text{crit,PBED,2}} = t_{\text{pd,XOR}_{\text{pc}}} + t_{\text{pd,OR}_{\text{rdc}}} \tag{4.3}$$

The gates XOR_{pc}, XOR_{pg} and OR_{rdc} can have more than three inputs, so they will be synthesized as a tree of LUTs on the ProASIC3. The synthesis of a gate with s_{input} inputs to a tree with a depth of d is shown in figure 4.6.

The propagation delay generated by a gate with an input size s_{input} is called $t_{\text{pd}}(gate, s_{\text{input}})$ and can be calculated by determining the depth d of the tree and multiplying it with the propagation delay of the respective three-input macro (e.g., OR3 for an OR gate), as the interconnect delays are not considered:

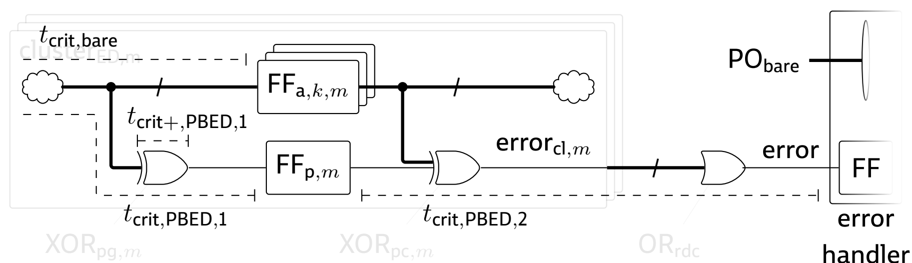

Figure 4.5: The two critical path candidates in direct PBED, $t_{\text{crit,PBED,1}}$ and $t_{\text{crit,PBED,2}}$. Note that $t_{\text{crit,PBED,1}}$ is generated by adding the critical path overhead of PBED $t_{\text{crit+,PBED,1}}$ to the existing critical path of the target circuit $t_{\text{crit,bare}}$ while $t_{\text{crit,PBED,2}}$ is newly generated by PBED.

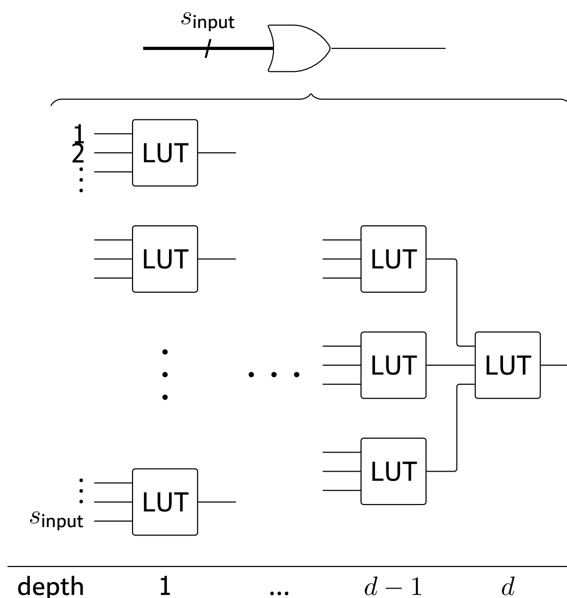

Figure 4.6: The figure shows how a gate with input size of s_{input} is mapped to an FPGA architecture with three-input LUTs. After mapping, a LUT tree with a depth of $d = \lceil \log_3 s_{\text{input}} \rceil$ is created. Note that if s_{input} is not a power of three, then not all the leafs of the tree exist.

$$
\begin{aligned}
t_{\mathsf{pd}}(gate, s_{\mathsf{input}}) &= d_{gate} \cdot t_{\mathsf{pd},macro} \\
&= \lceil \log_3 s_{\mathsf{input}} \rceil \cdot t_{\mathsf{pd},macro}
\end{aligned} \tag{4.4}
$$

With eq. 4.4, the propagation delays of the three defined gates can be calculated:

$$
t_{\mathsf{pd,XOR_{pg}}} = \lceil \log_3 (s_{\mathsf{cl}} - 1) \rceil \cdot t_{\mathsf{pd,XOR3}} \tag{4.5}
$$

$$
t_{\mathsf{pd,XOR_{pc}}} = \lceil \log_3 s_{\mathsf{cl}} \rceil \cdot t_{\mathsf{pd,XOR3}} \tag{4.6}
$$

$$
\begin{aligned}
t_{\mathsf{pd,OR_{rdc}}} &= \lceil \log_3 c_{\mathsf{cl}} \rceil \cdot t_{\mathsf{pd,OR3}} \\
&= \left\lceil \log_3 \left\lceil \frac{c_{\mathsf{FF,bare}}}{s_{\mathsf{cl}}} \right\rceil \right\rceil \cdot t_{\mathsf{pd,OR3}}
\end{aligned} \tag{4.7}
$$

With equations 4.5 to 4.7, we can finally calculate the critical paths for PBED by only using our input parameter variables s_{cl} and $c_{\mathsf{FF,bare}}$.

The critical path delays $t_{\mathsf{crit+,PBED,1}}$ and $t_{\mathsf{crit,PBED,2}}$ have the input parameter variables s_{cl} and $c_{\mathsf{FF,bare}}$. The remaining input parameters, i.e., gate propagation delays, are assumed to be constant values. Propagation delay of the macros are dependent on environment factors like the temperature, but we assume a constant environment in our analysis. Consequently, only s_{cl} and $c_{\mathsf{FF,bare}}$ limit the maximum frequency of the circuit.

In ProASIC3, at a junction temperature of 70°C and worst-case supply voltage 1.14 V, $t_{\mathsf{pd,MAJ3}}$, and $t_{\mathsf{pd,XOR3}}$ are 1.09 ns, and 1.37 ns respectively [Micr15a]. The propagation delay $t_{\mathsf{pd,OR3}}$ was neither available in the datasheet or macro library documentation. We assume the value of $t_{\mathsf{pd,OR3}} = 0.777$ ns, which is taken from the timing report of a layouted netlist which uses the primitive OR3. With these data the critical path caused by the flipflops and combinational elements can be calculated for various s_{cl} and $c_{\mathsf{FF,bare}}$ parameters.

Comparison

Table 4.1 shows the critical path delays $t_{\mathsf{crit+,1}}$ and $t_{\mathsf{crit,2}}$ for various values of the input parameter (x, y). The parameters s_{cl} and $c_{\mathsf{FF,bare}}$ are determined using (x, y), where $s_{\mathsf{cl}} = 3^x$, cluster count $c_{\mathsf{cl}} = 3^y$ and flipflop count in the bare circuit $c_{\mathsf{FF,bare}} = (s_{\mathsf{cl}} - 1) \cdot c_{\mathsf{cl}}$. With increasing depth of $\mathsf{XOR_{pg}}$, $t_{\mathsf{crit+,1}}$ grows for PBED, i.e., every time when s_{cl} reaches a higher power of 3. The additional path delay $t_{\mathsf{crit+,1}}$ of LTMR is independent of the input parameters. For $s_{\mathsf{cl}} = 3$ LTMR and PBED have a similar critical path overhead. PBED has additionally the $t_{\mathsf{crit+,2}}$, which grows with increasing depth of $\mathsf{XOR_{pc}}$ and $\mathsf{OR_{rdc}}$ gates.

Compared to the $t_{\mathsf{crit+,PBED,1}}$, which is always relative to the existing critical path $t_{\mathsf{crit,bare}}$, $t_{\mathsf{crit,PBED,2}}$ is generated in parallel to the bare circuit. Due to this reason, $t_{\mathsf{crit,PBED,2}}$ stays uncritical up to a certain depth of parity check and reduction gates.

				$t_{\text{crit}+,1}$ (ns)		$t_{\text{crit},2}$ (ns)
(x,y)	s_{cl}	c_{cl}	$c_{\text{FF,bare}}$	LTMR	PBED	PBED
(1,2)	3	9	18	1.09	1.37	2.92
(1,3)	3	27	54	1.09	1.37	3.7
(1,4)	3	81	162	1.09	1.37	4.48
(2,2)	9	9	72	1.09	2.74	4.29
(2,3)	9	27	216	1.09	2.74	5.07
(2,4)	9	81	648	1.09	2.74	5.85
(3,2)	27	9	234	1.09	4.11	5.66
(3,3)	27	27	702	1.09	4.11	6.44
(3,4)	27	81	2106	1.09	4.11	7.22

Table 4.1: Critical path impacts of LTMR and PBED for different numbers of application flipflops in the circuit and cluster sizes

4.2.3 Circuit area overhead

Assuming that the circuit area is proportional to the CLB count, we define the parameter $Area$ as the CLB count. For comparison, we are interested in the area overhead $Area_+$, i.e., the CLB overhead $c_{\text{CLB}+}$:

$$Area_+ = c_{\text{CLB}+} \tag{4.8}$$

In ProASIC3 architecture, every CLB can be either configured as an flipflop or LUT. Then, the circuit area overhead can be calculated by adding the count of additionally introduced LUTs and flipflops:

$$c_{\text{CLB}+} = c_{\text{LUT}+} + c_{\text{FF}+} \tag{4.9}$$

LTMR

In the LTMR applied circuit, the flipflops are triplicated, i.e., two redundant flipflops are added for each application flipflop:

$$c_{\text{FF}+,\text{LTMR}} = 2 \cdot c_{\text{FF,bare}} \tag{4.10}$$

LTMR requires one LUT for each application flipflop as voter:

$$c_{\text{LUT}+,\text{LTMR}} = c_{\text{FF,bare}} \tag{4.11}$$

In total, the area overhead for LTMR is:

$$Area_{+,\text{LTMR}} = c_{\text{CLB}+,\text{LTMR}} = 3 \cdot c_{\text{FF,bare}} \tag{4.12}$$

PBED

In PBED, for each cluster one parity flipflop is needed:

$$c_{\text{FF+,PBED}} = c_{\text{cl}} \tag{4.13}$$

XOR_{pg}-, XOR_{pc}- and OR_{rdc}-gates consist of LUTs:

$$c_{\text{LUT+,PBED}} = c_{\text{cl}}(c_{\text{LUT,XOR}_{\text{pg}}} + c_{\text{LUT,XOR}_{\text{pc}}}) + c_{\text{LUT,OR}_{\text{rdc}}} \tag{4.14}$$

As shown in figure 4.6, a gate with s_{input} inputs creates a tree, so the needed maximum LUT count for a tree of depth d can be determined by the following formula, assuming that every new level of the tree introduces 3^d LUTs at maximum. A gate with s_{input} is symbolized as $gate_{s_{\text{input}}}$ in the following.

$$\begin{aligned} c_{\text{LUT},gate_{s_{\text{input}}},\text{max}} &= \sum_{i=0}^{d_{gate_{s_{\text{input}}}}-1} 3^i \\ &= \frac{1}{2} \cdot (3^{d_{gate_{s_{\text{input}}}}} - 1) \end{aligned} \tag{4.15}$$

Using the formula for depth $d = \lceil \log_3 s_{\text{input}} \rceil$ (figure 4.6):

$$c_{\text{LUT},gate_{s_{\text{input}}},\text{max}} = \frac{1}{2} \cdot (3^{\lceil \log_3 s_{\text{input}} \rceil} - 1) \tag{4.16}$$

If s_{input} is a power of 3 (in case of XOR_{pc} and OR_{rdc}), then the equation can be further simplified:

$$\begin{aligned} s_{\text{input}} \overset{!}{=} 3^x, x \in \mathbb{N} &\implies 3^{\lceil \log_3 s_{\text{input}} \rceil} = s_{\text{input}} \\ &\implies c_{\text{LUT},gate_{s_{\text{input}}}} = \frac{1}{2} \cdot (s_{\text{input}} - 1) \end{aligned} \tag{4.17}$$

If $s_{\text{input}} + 1$ is a power of 3 (in case of XOR_{pg}), the same amount of LUTs are required. This is due to the fact that in one cluster, a gate with s_{input} inputs in this case will contain a single two-input LUT with the rest being three-input LUTs. Assuming no logic optimization like logic packing, a two- and a three-input LUT both occupy one CLB, thus the same area.

$$\begin{aligned} s_{\text{input}} + 1 \overset{!}{=} 3^x, x \in \mathbb{N} &\implies 3^{\lceil \log_3 s_{\text{input}} \rceil} = s_{\text{input}} + 1 \\ &\implies c_{\text{LUT},gate_{s_{\text{input}}}} = \frac{1}{2} \cdot s_{\text{input}} \end{aligned} \tag{4.18}$$

The logical gates XOR_{pc} and OR_{rdc} have an input size of $k + 1 = s_{\text{cl}} = 3^x$ and $m = c_{\text{cl}} = 3^y$, respectively (cf. subsection 4.2.1 and figure 4.3) and are assumed to have an input size which is a power of 3. So, with eq. 4.17, $c_{\text{LUT,XOR}_{\text{pc}}}$ and $c_{\text{LUT,OR}_{\text{rdc}}}$ can be determined.

$$s_{\text{input,XOR}_{\text{pc}}} = s_{\text{cl}} \implies c_{\text{LUT,XOR}_{\text{pc}}} = \frac{1}{2} \cdot (s_{\text{cl}} - 1) \tag{4.19}$$

$$s_{\mathsf{input,OR_{rdc}}} = c_{\mathsf{cl}} \implies c_{\mathsf{LUT,OR_{rdc}}} = \frac{1}{2} \cdot (c_{\mathsf{cl}} - 1) \tag{4.20}$$

The logical gate $\mathsf{XOR_{pg}}$ has as input size of $k = s_{\mathsf{cl}} - 1 = 3^x - 1$. Consequently, $s_{\mathsf{input}} + 1$ is a power of 3. So, with eq. 4.18, $c_{\mathsf{LUT,XOR_{pg}}}$ can be determined.

$$s_{\mathsf{input,XOR_{pg}}} = s_{\mathsf{cl}} - 1 \implies c_{\mathsf{LUT,XOR_{pg}}} = \frac{1}{2} \cdot (s_{\mathsf{cl}} - 1) \tag{4.21}$$

By using the LUT counts of $\mathsf{XOR_{pc}}$, $\mathsf{OR_{rdc}}$ and $\mathsf{XOR_{pg}}$ from the last three equations, the LUT overhead of PBED in eq. 4.14 can be rewritten:

$$c_{\mathsf{LUT+,PBED}} = c_{\mathsf{cl}}(c_{\mathsf{LUT,XOR_{pg}}} + c_{\mathsf{LUT,XOR_{pc}}}) + c_{\mathsf{LUT,OR_{rdc}}} \tag{4.14 revisited}$$

$$\begin{aligned} c_{\mathsf{LUT+,PBED}} &= c_{\mathsf{cl}}\Big(\frac{1}{2} \cdot (s_{\mathsf{cl}} - 1) + \frac{1}{2} \cdot (s_{\mathsf{cl}} - 1)\Big) + \frac{1}{2}(c_{\mathsf{cl}} - 1) \\ &= c_{\mathsf{cl}}(s_{\mathsf{cl}} - 1) + \frac{1}{2}(c_{\mathsf{cl}} - 1) \end{aligned} \tag{4.22}$$

Finally, with eq. 4.9, 4.13 and 4.22, total area overhead for PBED equals to:

$$\begin{aligned} Area_{+,\mathsf{PBED}} &= c_{\mathsf{cl}} + c_{\mathsf{cl}}(s_{\mathsf{cl}} - 1) + \frac{1}{2}(c_{\mathsf{cl}} - 1) \\ &= c_{\mathsf{cl}} + c_{\mathsf{cl}}s_{\mathsf{cl}} - c_{\mathsf{cl}} + \frac{c_{\mathsf{cl}}}{2} - \frac{1}{2} \\ &= c_{\mathsf{cl}}(s_{\mathsf{cl}} + \frac{1}{2}) - 0.5 \end{aligned} \tag{4.23}$$

$c_{\mathsf{FF,bare}}$ is a main input parameter, therefore it is better to rewrite c_{cl} using $c_{\mathsf{FF,bare}}$:

$$Area_{+,\mathsf{PBED}} = \frac{c_{\mathsf{FF,bare}}}{s_{\mathsf{cl}} - 1}(s_{\mathsf{cl}} + \frac{1}{2}) - 0.5 \tag{4.24}$$

Comparison

Table 4.2 shows the area overhead $Area_+$ and area overhead caused by a single application flipflop $Area_+ : c_{\mathsf{FF,bare}}$ for various values of s_{cl} and $c_{\mathsf{FF,bare}}$ parameters. Area overhead $Area_+$ is related to $c_{\mathsf{FF,bare}}$ instead of the whole circuit including combinatorics, because the area overhead is only dependent on $c_{\mathsf{FF,bare}}$ and the combinatorics LUT count is arbitrary.

PBED leads to an area overhead of circa 1.7 LUTs per application flipflop for $s_{\mathsf{cl}} = 3$. The area overhead of PBED decreases with increasing s_{cl} and c_{cl} to approximately 1.1 LUTs per application flipflop. The LTMR area overhead is independent of the input parameters. Overall, $s_{\mathsf{cl}} = 3$ is a reasonable choice for saving significant amount of FPGA resources and at the same time for having as little impact on the critical path as possible. If the maximum frequency is not important, then higher s_{cl} values can be the choice.

4.2.4 Multiple bit error susceptibility

LTMR and PBED techniques both are immune against one bitflip in a clock cycle, but not against multiple- and even-number of bit errors in one cluster, respectively. In this section, we will compare the LTMR and PBED regarding multiple bit

Table 4.2: Area impacts of LTMR and PBED for different numbers of application flipflops in the circuit and cluster sizes

				$Area_+$		$Area_+ : c_{\text{FF,bare}}$	
(x, y)	s_{cl}	c_{cl}	$c_{\text{FF,bare}}$	LTMR	PBED	LTMR	PBED
(1,2)	3	9	18	54	31	3	1.72
(1,3)	3	27	54	162	94	3	1.74
(1,4)	3	81	162	486	283	3	1.75
(2,2)	9	9	72	216	85	3	1.18
(2,3)	9	27	216	648	256	3	1.19
(2,4)	9	81	648	1944	769	3	1.19
(3,2)	27	9	234	702	247	3	1.06
(3,3)	27	27	702	2106	742	3	1.06
(3,4)	27	81	2106	6318	2227	3	1.06

error susceptibility by calculating the probability that an error cannot be detected in the circuit.

We assume that every flipflop in LTMR and PBED is updated in every clock cycle with a correct value, otherwise the bitflips can accumulate and lead to uncorrectable errors.

If a single particle travels through the circuit, then it can cause single or multiple bit errors dependent on the amount of energy transferred to the circuit and the size of the IC structures. In this analysis, we assume that the CLBs are far enough from each other to consider bitflips as independent events and all the flipflops have the same bitflip probability. In the following, we use p as the bitflip probability of one flipflop in one clock cycle.

In what follows, we calculate the probability for an undetectable multiple bit error in a hardened circuit under the former assumptions. The probability for a multiple bit error is abbreviated as p_{MBE}.

LTMR

We apply the definition of a cluster also on LTMR and define an LTMR cluster as the group of three flipflops after triplication. So, if two or three bits flip in a cluster during a clock cycle, then this cluster outputs a wrong value. If i is the number of bits flipped in a cluster:

$$
\begin{aligned}
p_{\text{MBE,cl,LTMR}} &= \sum_{i=2}^{3} \binom{3}{i} p^i (1-p)^{3-i} \\
&= 3p^2(1-p) + p^3 \\
&= 3p^2 - 2p^3
\end{aligned}
\tag{4.25}
$$

There is one cluster for each application flipflop. So there are $c_{\text{FF,bare}}$ LTMR clusters in total. For an undetectable error, at least one LTMR cluster must have an undetectable number of bitflips.

$$p_{\text{MBE,LTMR}} = \sum_{i=1}^{c_{\text{FF,bare}}} \binom{c_{\text{FF,bare}}}{i} p^i_{\text{MBE,cl,LTMR}} (1 - p_{\text{MBE,cl,LTMR}})^{c_{\text{FF,bare}} - i} \tag{4.26}$$

It is easier to calculate the complement of this event, which simplifies the sum. So, we calculate the probability that LTMR works without any undetected error. This means all of the LTMR clusters have detectable number of bitflips. Then, we subtract the complementary event from 1.

$$\begin{aligned} p_{\text{MBE,LTMR}} &= 1 - \sum_{i=0}^{0} \binom{c_{\text{FF,bare}}}{i} p^i_{\text{MBE,cl,LTMR}} (1 - p_{\text{MBE,cl,LTMR}})^{c_{\text{FF,bare}} - i} \\ &= 1 - (1 - p_{\text{MBE,cl,LTMR}})^{c_{\text{FF,bare}}} \\ &= 1 - (1 - 3p^2 + 2p^3)^{c_{\text{FF,bare}}} \end{aligned} \tag{4.27}$$

PBED

In a PBED cluster, (positive) even number of bitflips cannot be detected.

$$\begin{aligned} p_{\text{MBE,cl,PBED}} &= \sum_{\substack{i=2,\\ i=2n, n\in\mathbb{N}}}^{s_{\text{cl}}} \binom{s_{\text{cl}}}{i} p^i (1-p)^{s_{\text{cl}} - i} \\ &= \begin{cases} \sum_{n=1}^{\frac{s_{\text{cl}}}{2}} \binom{s_{\text{cl}}}{2n} p^{2n} (1-p)^{s_{\text{cl}} - 2n} & s_{\text{cl}} = 2j, j \in \mathbb{N} \\ \sum_{n=1}^{\frac{s_{\text{cl}}-1}{2}} \binom{s_{\text{cl}}}{2n} p^{2n} (1-p)^{s_{\text{cl}} - 2n} & \\ \quad + \sum_{i=s_{\text{cl}}}^{s_{\text{cl}}} \binom{s_{\text{cl}}}{i} p^i (1-p)^{s_{\text{cl}} - i} & s_{\text{cl}} = 2j+1, j \in \mathbb{N} \end{cases} \end{aligned} \tag{4.28}$$

Analogous to LTMR, for an undetectable error, at least one of c_{cl} PBED clusters must have an undetectable number of bitflips in one clock cycle. But like in Like in eq. 4.27, it is easier to calculate the complementary event.

$$\begin{aligned} p_{\text{MBE,PBED}} &= \sum_{i=1}^{c_{\text{cl}}} \binom{c_{\text{cl}}}{i} p^i_{\text{MBE,cl,PBED}} (1 - p_{\text{MBE,cl,PBED}})^{c_{\text{cl}} - i} \\ &= 1 - \sum_{i=0}^{0} \quad \text{———— '' ————} \\ &= 1 - (1 - p_{\text{MBE,cl,PBED}})^{c_{\text{cl}}} \end{aligned} \tag{4.29}$$

Comparison

Assuming one year mission in the second Lagrangian point (L2 orbit, 1.5 million km away from the earth), under 1/cm² shielding, a programmed circuit with 5000 flipflops on an RTPE3000L FPGA has four SEUs [BSV11, ch. 7]. If this circuit runs

Table 4.3: Comparison of LTMR and PBED regarding multiple bit error probability of one cluster $p_{\text{MBE,cl}}$ and whole circuit p_{MBE}

				$p_{\text{MBE,cl}}$		p_{MBE}	
(x,y)	s_{cl}	c_{cl}	$c_{\text{FF,bare}}$	LTMR	PBED	LTMR	PBED
(1,2)	3	9	18	4.84e-36	4.84e-36	8.71e-35	4.35e-35
(1,3)	3	27	54	4.84e-36	4.84e-36	2.61e-34	1.31e-34
(1,4)	3	81	162	4.84e-36	4.84e-36	7.84e-34	3.92e-34
(2,2)	9	9	72	4.84e-36	5.81e-35	3.48e-34	5.23e-34
(2,3)	9	27	216	4.84e-36	5.81e-35	1.05e-33	1.57e-33
(2,4)	9	81	648	4.84e-36	5.81e-35	3.14e-33	4.7e-33
(3,2)	27	9	234	4.84e-36	5.66e-34	1.13e-33	5.1e-33
(3,3)	27	27	702	4.84e-36	5.66e-34	3.4e-33	1.53e-32
(3,4)	27	81	2106	4.84e-36	5.66e-34	1.02e-32	4.59e-32

at 20 MHz, then p can be calculated by:

$$\begin{aligned} p &= 4/5000/365/24/60/60/(20 \times 10^6) \\ &\approx 1.27 \times 10^{-18} \end{aligned} \tag{4.30}$$

Table 4.3 shows a comparison of multiple bit error probabilities for various s_{cl} $c_{\text{FF,bare}}$ parameters. The multiple bit error probability of one cluster $p_{\text{MBE,cl}}$ should be lower for the PBED, as PBED can detect an odd number of bit errors. The calculation does not show any differences for $p_{\text{MBE,cl}}$, because the assumed bit error rate p is very low and multiple bit errors greater than two practically do not happen.

For $s_{\text{cl}} = 3$, p_{MBE} of PBED is approximately half of the LTMR's. When the cluster size s_{cl} for PBED increases, then p_{MBE} of PBED also increases - at $s_{\text{cl}} = 27$, p_{MBE} of PBED is approximately five times of LTMR's.

4.3 Experimental evaluation

After the analytical evaluation, we provide experimental results, which allows a more precise evaluation of PBED. For experiments, we used:

- an FSM design, which was replicated various times to analyze the impact of PBED on circuit-timing and -area for various input circuit areas in detail but with a fixed circuit type.
- I99T benchmark circuits, which allows to assess the PBED impacts on various circuit types.

The temperature and supply voltage settings for the timing analysis of the layouted circuits are the same as in the analytical evaluation (junction temperature of 70°C and worst-case supply voltage 1.14 V).

The evaluations are also based on the comparison with LTMR like in the analytical comparison.

LTMR and PBED were applied using the synthesis tool Synplify and a newly-implemented tool which generates the PBED circuitry on top of an RTL design (introduced in section 4.4), respectively. This tool and thus the experiments have parameters like cluster size range, placer try count and partitioning try count. These parameters along with the tool will be introduced in section 4.4. In what follows, we only give brief description of these parameters.

The circuits were synthesized using Synplify with automatic constraining, which maps the design with different clock constraints to achieve the highest clock frequency possible. The output netlists were then layouted (in other words placed and routed) using Designer from Microsemi. Every synthesized circuit were layouted ten times with different seeds (i.e., $placing\ try\ count = 10$) and the layout with the best timing was picked for the results.

For PBED, we varied the cluster size from 2 to 9 and partitioned the flipflops according to their location in the layouted bare circuit. The partitioning technique does not always find the optimal solution. Therefore the partitioning is repeated $partitioning\ try\ count$ times and the best solution was chosen. In all experiments we used $partitioning\ try\ count = 100$. The partitioning uses layout data (coordinates of the cell placements) of the layouted bare circuit netlist, for this purpose the layout data of the bare circuit with the best timing out of the four placed designs were used.

4.3.1 Finite state machine (FSM) circuit

In this subsection, we present synthesis results using an implementation of the FSM circuit shown in the reference processing architecture (figure 1.3). To get various circuit sizes, we instantiated this FSM multiple times. To not exhaust the input-output ports of the FPGA due to excessive number of instantiations, which would make the circuit unplaceable on the FPGA, we connected the circuit outputs to a demultiplexer.

The circuits were synthesized for the ProASIC3 with the smallest available area, the A3P250. We chose an area-constrained FPGA to compare the performance LTMR and PBED additionally at a high utilization of the FPGA.

The following synthesis results show the circuit input parameters:

- circuit name (circ.), which corresponds to the FSM circuit instantiation count
- PBED cluster size (s_{cl}) for PBED-hardened circuits

and the circuit output parameters for the bare, LTMR applied and PBED applied circuits after synthesis:

- flipflop count c_{FF}
- total area A
- critical path delay t_{crit}

The parameters are for the bare (ba), LTMR applied (LT) and PBED applied (PB) circuit. Using the output parameters, we derived the following comparison parameters:

- the overheads caused by the hardening techniques on the respective output parameters of the bare circuit, which are marked by the plus symbol in the subscript ($_+$), e.g., $A_{+,\mathsf{PBED}} = A_{\mathsf{PB}} - A_{\mathsf{ba}}$
- area overhead of the respective hardening technique per application flipflop $\frac{A_+}{c_{\mathsf{FF,ba}}}$
- area overhead ratio PBED to LTMR $\frac{A_{+,\mathsf{PB}}}{A_{+,\mathsf{LT}}}$

We did not normalize the area overhead using the whole area, because both LTMR and PBED harden directly the flipflops of the circuit. So, area overhead is mainly dependent on the number of flipflops.

Note that in ProASIC3 architecture, every CLB can be either configured as a flipflop or lookup table (LUT). Consequently, in this work, circuit area A is defined as the total count of flipflops and LUTs in the synthesized circuit.

We first do our evaluation by fixing the cluster size at 3. Table 4.4 shows an excerpt of the obtained parameters from the synthesis results and the table 4.5 shows the derived parameters.

Firstly, we analyze the table 4.4. The FSM has 25 flipflops, which was instantiated up to 43 times, until the bare circuit could not be fit into the FPGA. In circuits where the FSM was replicated, one of the 25 FFs is always synthesized away in the replica FSMs, because the synthesizer bound a particular primary output net of all the replicas having always the same value to the same flipflop. The FSMs have the same input and are connected to a demultiplexer. Consequently, the circuit with a single instantiated FSM has 25 flipflops, and with with every instantiated FSM $25 - 1 = 24$ additional flipflops are added to the circuit. This rule does not always apply, because as the design gets bigger, some cells may have to drive a higher amount of other cells, which are called *high fanout* cells. If the synthesizer encounters a sequential or combinational cell with a high fanout, then this cell gets replicated to divide the fanout on two cells. The replication is needed because a cell output has a maximum current it can drive, hence a limited fanout. The circuits where a flipflop replication happens are for instance 12 and 25, in these cases 25 additional flipflops are added compared to the last circuit.

Compared to the fixed additional number of flipflops after an additional instantiation of the FSM, a repeating pattern in the additional number of combinational cells cannot be recognized. This is probably due to the heuristics used in optimizations on combinational cells (LUTs).

Table 4.4: Synthesis results for multiple instantiations of the FSM circuit. The critical path for LTMR (LT), PBED (PB) and bare (ba) do not exist for circuits > 26, > 30, and > 42, respectively, because the place-and-router could not route or fit these circuits into the FPGA due to excessive circuit area. For these circuits, the critical path delays (t_{crit}) are marked with minus (-). Flipflop count and area for PBED-hardened circuit 43 do not exist, as the PBED tool requires a placed and routed bare circuit, but bare version of circuit 43 cannot be placed and routed.

	c_{FF}			A			t_{crit} (ns)		
circ.	ba	LT	PB	ba	LT	PB	ba	LT	PB
1	25	75	40	144	218	190	7.96	9.61	9.51
2	49	147	76	299	444	388	8.54	10.26	9.72
3	73	219	111	406	670	585	8.40	10.32	10.05
4	97	291	147	546	891	783	8.40	10.35	10.44
5	121	363	183	674	1107	971	8.46	10.45	10.52
6	145	435	219	831	1361	1186	8.43	10.52	10.84
7	169	507	255	975	1586	1387	8.47	10.43	10.67
8	193	579	291	1070	1772	1545	8.39	10.81	11.06
9	217	651	327	1198	1993	1730	8.54	10.86	11.37
10	241	723	363	1328	2202	1920	8.13	10.59	11.59
11	265	795	399	1494	2442	2145	8.66	10.73	11.64
12	290	867	436	1659	2725	2367	8.70	10.78	11.87
13	314	939	472	1833	2978	2600	8.69	10.99	11.98
14	338	1011	508	1967	3172	2796	8.65	10.66	11.96
15	362	1083	544	2101	3411	2988	8.50	10.88	12.37
16	386	1155	580	2315	3701	3262	8.71	11.04	12.53
17	410	1227	616	2383	3857	3387	8.64	11.09	12.47
18	434	1299	652	2547	4092	3609	8.12	10.76	13.04
19	458	1371	688	2638	4316	3758	8.41	10.83	12.54
20	482	1443	724	2786	4566	3969	8.38	10.88	12.95
21	506	1515	760	2923	4787	4163	8.75	11.21	13.02
22	530	1587	796	3089	5035	4392	8.38	11.09	12.94
23	554	1659	832	3248	5266	4613	8.66	10.99	13.46
24	578	1731	870	3420	5495	4841	8.73	11.28	13.27
25	603	1803	906	3542	5753	5020	8.73	11.09	13.51
26	627	1875	942	3712	5987	5248	8.74	11.35	14.12
27	651	1947	978	3872	6223	5470	8.89	-	13.65
⋮									
30	723	2163	1086	4282	6881	6060	8.93	-	14.00
31	747	2235	1122	4399	7137	6236	9.02	-	-
⋮									
42	1012	3027	1519	6092	9722	8569	8.88	-	-
43	1036	3099	-	6287	9952	-	-	-	-

Table 4.5: Derived parameters for the FSM circuit using the synthesis results in the table 4.4. The last row shows the average (avg.) value of the last five derived parameters.

	$c_{\mathsf{FF}+}$		A_+		$t_{\mathsf{crit}+}$ (ns)		$\frac{A_+}{c_{\mathsf{FF,ba}}}$		
circ.	LT	PB	LT	PB	LT	PB	LT	PB	$\frac{A_{+,\mathsf{PB}}}{A_{+,\mathsf{LT}}}$
1	50	15	74	46	1.65	1.54	2.96	1.84	0.62
2	98	27	145	89	1.71	1.18	2.96	1.82	0.61
3	146	38	264	179	1.92	1.65	3.62	2.45	0.68
4	194	50	345	237	1.96	2.04	3.56	2.44	0.69
5	242	62	433	297	1.98	2.05	3.58	2.45	0.69
6	290	74	530	355	2.09	2.42	3.66	2.45	0.67
7	338	86	611	412	1.96	2.20	3.62	2.44	0.67
8	386	98	702	475	2.41	2.67	3.64	2.46	0.68
9	434	110	795	532	2.32	2.83	3.66	2.45	0.67
10	482	122	874	592	2.46	3.46	3.63	2.46	0.68
11	530	134	948	651	2.07	2.98	3.58	2.46	0.69
12	577	146	1066	708	2.09	3.17	3.68	2.44	0.66
13	625	158	1145	767	2.31	3.29	3.65	2.44	0.67
14	673	170	1205	829	2.01	3.31	3.57	2.45	0.69
15	721	182	1310	887	2.39	3.87	3.62	2.45	0.68
16	769	194	1386	947	2.32	3.82	3.59	2.45	0.68
17	817	206	1474	1004	2.45	3.84	3.60	2.45	0.68
18	865	218	1545	1062	2.64	4.92	3.56	2.45	0.69
19	913	230	1678	1120	2.42	4.13	3.66	2.45	0.67
20	961	242	1780	1183	2.50	4.57	3.69	2.45	0.66
21	1009	254	1864	1240	2.46	4.27	3.68	2.45	0.67
22	1057	266	1946	1303	2.71	4.56	3.67	2.46	0.67
23	1105	278	2018	1365	2.33	4.80	3.64	2.46	0.68
24	1153	292	2075	1421	2.54	4.53	3.59	2.46	0.68
25	1200	303	2211	1478	2.35	4.78	3.67	2.45	0.67
26	1248	315	2275	1536	2.61	5.38	3.63	2.45	0.68
27	1296	327	2351	1598	-	4.76	3.61	2.45	0.68
⋮									
30	1440	363	2599	1778	-	5.07	3.59	2.46	0.68
31	1488	375	2738	1837	-	-	3.67	2.46	0.67
⋮									
42	2015	507	3630	2477	-	-	3.59	2.45	0.68
43	2063	-	3665	-	-	-	3.54	-	-
						avg.	3.59	2.42	0.67

The place-and-router could fit 26 LTMR-hardened, 30 PBED-hardened and 42 not-hardened (bare) FSMs into the FPGA.

Now, we look at the derived values. The derived values include the overheads in area ($c_{\mathsf{FF}+}$, A_+) and critical path ($t_{\mathsf{crit}+}$), as well as the normalized area overhead ($\frac{A_+}{c_{\mathsf{FF,ba}}}$) and the area overhead ratio between PBED and LTMR ($\frac{A_{+,\mathsf{PB}}}{A_{+,\mathsf{LT}}}$) for comparison.

In average, LTMR causes an area overhead per application flipflop of 3.59, and PBED 2.42. The area overhead ratio is 0.67, so PBED saves 33% of the area overhead caused by LTMR.

Comparison of the average area overhead values with the analytical results in table 4.2 shows that the experimental values differ. The difference is about $3.59 - 3 = 0.59$ in case of LTMR and $2.42 - 1.75 = 0.67$. This difference is caused by the remapping of enable flipflops to pairs of a multiplexer and a flipflop. This remapping ensures that the flipflop is updated in every clock cycle, which in turn avoids the accumulation of bitflips in PBED or LTMR cluster. In section 4.4, enable flipflop conversion is discussed more in detail.

The area overheads of the circuits 1 and 2 stand out. These two circuits have significantly lower area overhead per flipflop than other circuits, for instance 2.96 for LTMR. The reason is again the enable flipflop conversion, but in this case the conversion happened during the synthesis of the bare circuit.

In ProASIC3 architecture, the CLBs can be either configured as a three-input LUT or flipflop. An enable flipflop with a clear/preset input requires a four inputs (clock, clear/preset, data input, and data enable), and in this case the clear/preset input must be connected to a global routing path. Most sequential circuits have a reset input net, which is connected to most flipflops in the circuit, so normally apart from the clock signals also the reset signals with high fanout are routed using global routing paths in ProASIC3, and this is not a significant restriction in designs with common clear/preset inputs for the flipflops. If the clear/preset input of an enable flipflop is not connected to a net with a high fanout (dependent on the settings) by the synthesizer, then this enable flipflop is converted to a multiplexer, and three-input flipflop (with clock, clear/preset, and data input) by the layout tool. This conversion is analogous to the enable flipflop conversion mentioned earlier. In the opposite case, if a net has a high fanout, then the layout tool promotes this net to a global resource and the net is routed using the limited global routing resources.

In circuits 1 and 2, all the enable flipflops (15 and 30, respectively) are converted to three-input flipflop and multiplexer pairs already during the synthesis of the bare circuit, because the net connected to the clear/preset inputs of the enable flipflops does not have enough fanout to be promoted to a global resource. This in turn increased the area of the bare circuit and decreased the area overhead per application flipflop to values similar to the achieved in the analytical results in table 4.2 (3 for LTMR, 1.7 for PBED), 2.96 and 1.84 in case of circuit 1 and 2.96 and 1.82 in circuit 2. The empirical values for PBED are higher than the analytical results,

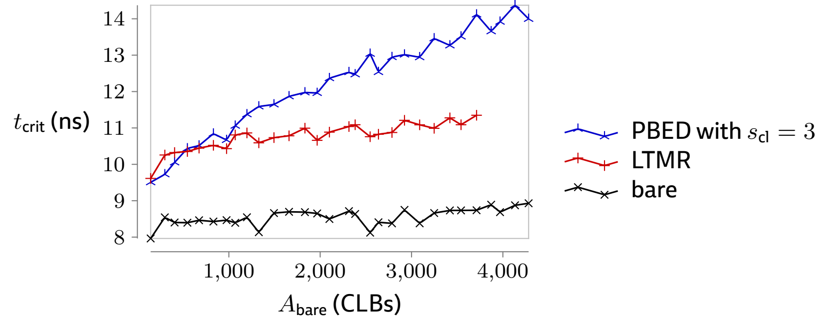

Figure 4.7: Critical path delay data for the FSM circuits plotted from table 4.4. The x-axis shows the area of the bare circuit (A_{bare}) in number of FPGA CLBs. The y-axis shows the absolute critical path delay (t_{crit}) of the bare and hardened circuits in the plot in ns. Every point represents a successfully routed circuit and points belonging to the PBED-, LTMR-hardened or bare circuits are connected by a line, respectively. Beginning from $A_{\text{bare}} \geq 3872$ CLBs, the points do not exist for LTMR-hardened circuits, as these could not be fit into the FPGA.

because we assumed an optimal utilization of the CLBs in the analysis. Beginning from circuit 3, the required fanout by the synthesizer is achieved by the clear/preset net and the clear/preset net uses a global route on the FPGA. So, the enable flipflops do not have to be converted, require only one tile, and the enable flipflop conversion only occurs during the hardening by LTMR and PBED.

Although it is possible to avoid the conversion of enable flipflops also in bare versions of the circuits 1 and 2 by promoting the reset/preset nets manually to global resources, we chose an approach where we tried to work with as much default settings in the synthesis and layout tools as possible.

In conclusion, the area overhead of PBED and LTMR is highly dependent on the use of enable flipflops in the circuit. The obtained analytical results regarding the area overhead can only be achieved in a circuit without enable flipflops.

The critical path overhead for PBED in the synthesis results is for circuit 1 nearly the same as in the analytical evaluation (see table 4.1), $t_{\text{crit+,1,analytical}} = 1.37$ ns ≈ 1.54 ns $= t_{\text{crit+,experimental}}$. With increasing circuit size the critical path overhead increases, which can be better seen in figure 4.7 with absolute values and in figure 4.8 with overhead values.

By looking into the critical path details in the timing reports of the layouted circuits, we observed that in PBED-hardened versions of circuits 1 to 4 and 6, the critical path is caused by the parity generation ($t_{\text{crit+,1}}$, figure 4.5) and in the rest of the circuits, the path caused by the OR-tree ($t_{\text{crit,2}}$, figure 4.5) becomes the critical path.

In PBED-hardened circuits, the growth of the critical path overhead decreases with the circuit size, and the graph has a logarithmic shape. The reason is that the circuit size is linearly proportional to the flipflop count, but the critical path of the

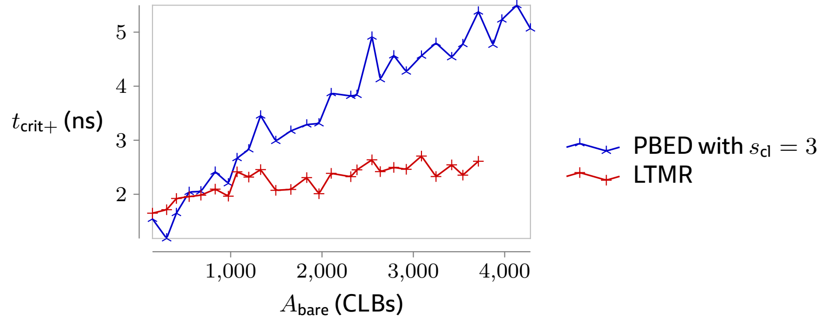

Figure 4.8: Critical path delay overhead data for the FSM circuits plotted from table 4.5. The x-axis shows the area of the bare circuit (A_{bare}) in number of FPGA CLBs. The y-axis shows the critical path overheads ($t_{\text{crit}+}$) of the LTMR- and PBED-hardened circuits relative to the critical path of the bare circuit in ns. The rest is similar to figure 4.7.

OR-tree increases logarithmically (see equation 4.7).

The critical path overhead of most of the LTMR-hardened circuits stays between 1.5 ns and 3 ns. LTMR needs only local routing, therefore the timing overhead is fairly constant.

The former analytical timing results in table 4.1 differ significantly from the experimental results of most circuits, because we did not incorporate the routing delays in our analytical evaluation. For instance, circuit 7 with 169 application flipflops has $t_{\text{crit},2} = 4.48$ ns $+ 0.777$ ns $= 5.257$ (because $162 < 169 < 486$ according to the analytical results in table 4.1, and $t_{\text{crit},2}(x, y+1) = t_{\text{crit},2}(x, y) + t_{\text{pd,OR3}}$). The critical path in the experimental results is 10.67 ns, so the routing makes about half of the whole critical path. In case of LTMR-hardened circuits, the ratio is similar: 1.09 ns without routing versus 1.5 to 3 ns with routing.

The increasing area has minimal impact on the critical path of the bare circuit. This is due to the low complexity of the FSM circuit and their isolation from each other, which does not need long routing paths.

A longer critical path leads to a negative maximum frequency impact on the circuit, therefore PBED-hardened circuit can achieve a lower frequency than LTMR if the circuit that has to be hardened includes more than 975 CLBs and 169 flipflops (circuit 7 in table 4.4).

Until now, the evaluation was for a fixed cluster size of 3. Now, we present results with cluster sizes from 2 to 9 to analyze the impact of the cluster size on the critical path and area overhead. The figures 4.9 and 4.10 illustrate the area and critical path overheads for all circuits, respectively.

For small circuits (e.g., circuits 1 to 6, up to 145 application flipflops), increasing the cluster size also increases the critical path overhead, because in small circuits the parity generation is on the critical path, and the parity generation path grows logarithmically for greater cluster sizes (see equation 4.5).

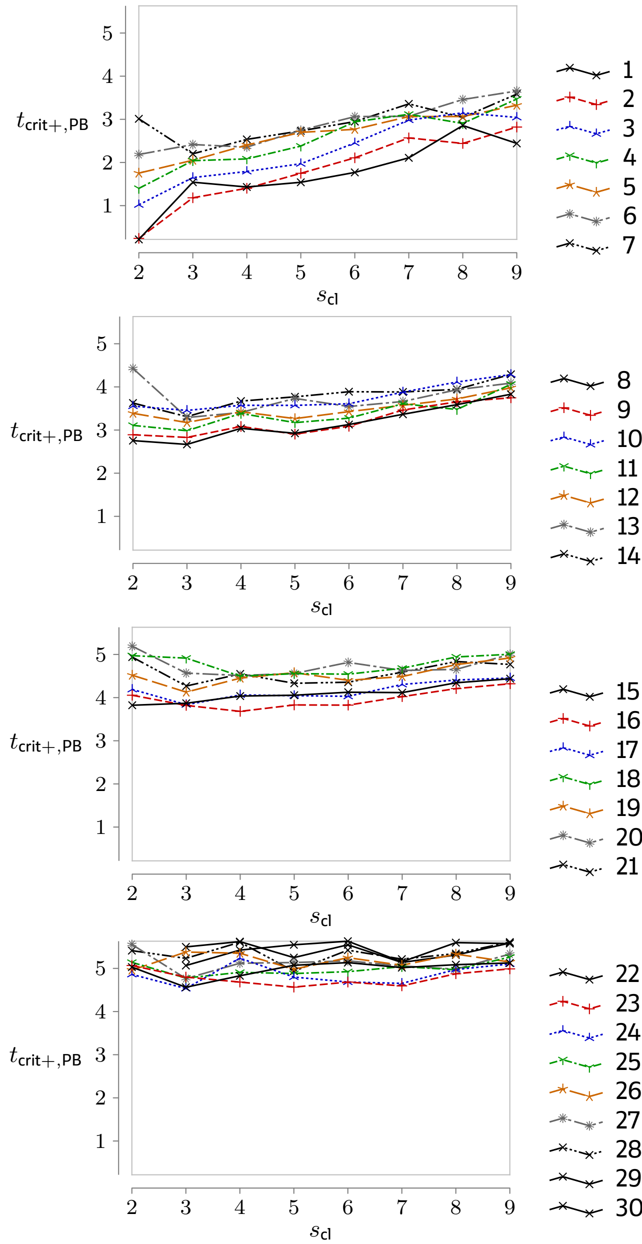

Figure 4.9: Critical path overhead delay ($t_{\text{crit+}}$) for PBED hardened FSM circuits with cluster sizes (s_{cl}) varying from 2 to 9. A single PBED cluster corresponds to a group of one parity bit and $s_{\text{cl}} - 1$ application flipflops. The critical path overhead is relative to the bare circuit. Every point corresponds to a placed and routed circuit. The lines connect $t_{\text{crit+,PB}}$ values for a particular circuit. The point for the circuit 22($s_{\text{cl}} = 2$) does not exist, because it could not be fit into the FPGA.

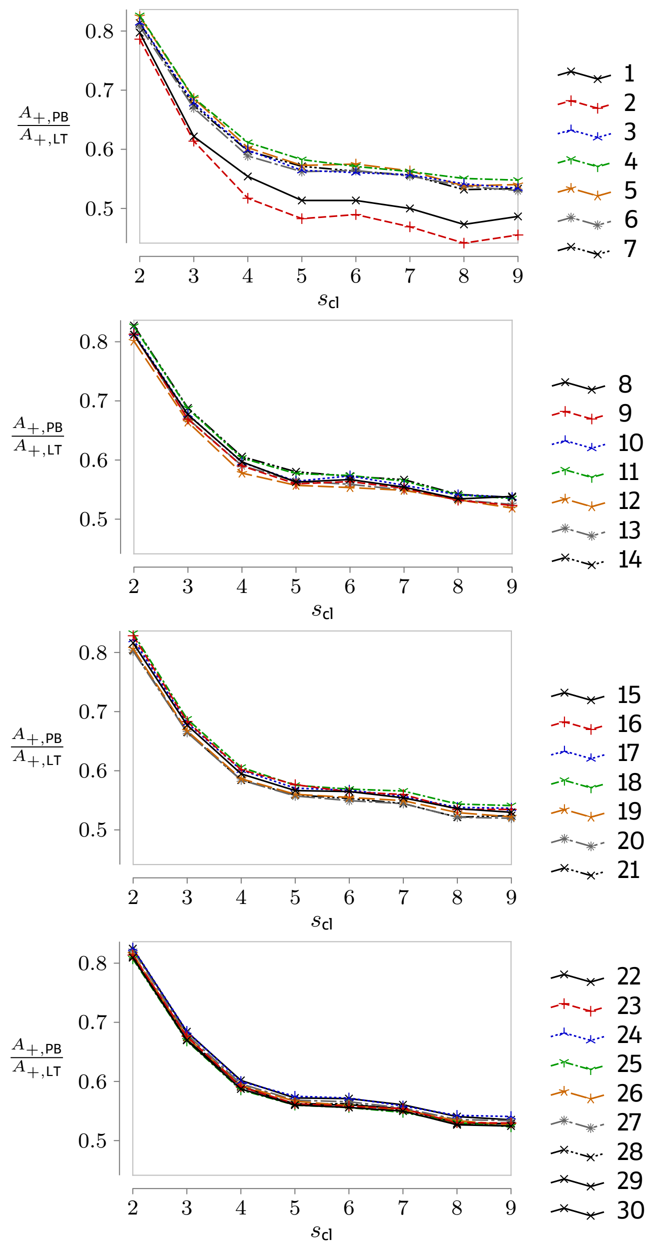

Figure 4.10: Area overhead (A_+) per application flipflop ($\frac{A_+}{c_{\text{FF,ba}}}$) for PBED hardened FSM circuits with cluster sizes varying from 2 to 9. Every point corresponds to a placed and routed circuit. The lines connect y-axis values for a particular circuit.

Table 4.6: Minimum and maximum values for derived parameters for PBED hardened FSM circuits for various cluster sizes (s_{cl})

s_{cl}	$t_{\mathrm{crit}+}$ (ns)	$\frac{A_{+}}{c_{\mathrm{FF,ba}}}$	$\frac{A_{+,\mathrm{PB}}}{A_{+,\mathrm{LT}}}$
2	0.22 - 5.55	2.33 - 2.98	0.79 - 0.84
3	1.18 - 5.50	1.82 - 2.46	0.61 - 0.69
4	1.40 - 5.62	1.53 - 2.18	0.52 - 0.61
5	1.54 - 5.55	1.43 - 2.07	0.48 - 0.58
6	1.77 - 5.63	1.45 - 2.08	0.49 - 0.58
7	2.11 - 5.42	1.39 - 2.03	0.47 - 0.57
8	2.44 - 5.60	1.31 - 1.97	0.44 - 0.55
9	2.44 - 5.66	1.35 - 1.96	0.46 - 0.55

For other circuits, the critical path overhead stays similar for different cluster sizes, but a decrease is noticeable in the critical path overhead of most circuits, if the cluster size of 3 is chosen instead of 2. All in all, the critical path overhead variation over cluster size stays under 1 ns for circuits 8 to 30 (approximately more than 200 application flipflops, and 1000 CLBs according to table 4.4).

If we look at the area overhead ratio of PBED and LTMR, we see that the graph shape is similar for all of the circuits. The circuits 1 and 2 have less overhead, because the enable flipflops were already converted in the bare version of the circuit, as discussed at the beginning of this section.

The decay in the graphs continues until the cluster size of 6, at which there is a local maximum. So, the cluster size of 6 should be avoided. For relatively big circuits (from circuit 8 upwards), cluster size of 9 can be selected without any significant compromise on the critical path overhead and thus on the maximum frequency.

Finally, table 4.6 summarizes the minimum and maximum values for the derived values. The upper bound for the critical path overhead stays between 5 to 6 ns and the lower bound for the same parameter increases slightly up to 2.44 ns. The upper and lower bound are caused by the biggest and smallest circuits, respectively. We observe that greater cluster size does only affect relatively small circuits.

At the highest experimented cluster size of 9, PBED can achieve and area overhead of 1.33, and can save up to 55% of the area overhead caused by the LTMR.

4.3.2 I99T circuits

We applied LTMR and PBED also on the I99T benchmark circuits, which are part of the ITC'99 benchmark circuits distributed by the CAD group at Politecnico di Torino. The circuits are introduced in [CRS00] and their VHDL descriptions can be obtained from [CADP16]. These RT-level circuits have only one clock signal, do

Table 4.7: Synthesis results for I99T benchmark circuits. The results are ordered by the bare circuit area. See subsection 4.3.1 for the description of the parameters and abbreviations used in the table.

	c_{FF}			A			t_{crit} (ns)		
cir.	ba	LT	PB	ba	LT	PB	ba	LT	PB
b02	4	12	8	14	26	23	4.59	6.24	5.14
b01	10	30	17	27	58	47	4.46	6.18	5.83
b06	8	24	13	32	56	47	5.50	7.20	6.67
b08	21	63	33	91	153	130	10.49	12.26	11.86
b03	31	90	48	97	185	155	8.20	10.22	9.13
b09	28	84	43	100	182	152	9.66	11.43	10.70
b10	24	72	38	103	175	149	6.75	8.56	7.86
b13	56	168	85	152	333	274	8.14	9.89	9.07
b07	44	132	67	178	308	259	13.57	15.65	14.84
b11	35	105	54	256	359	319	17.12	18.62	18.16
b04	66	198	100	338	589	515	23.81	25.70	25.71
b05	41	108	64	395	490	468	24.59	26.48	25.64
b12	122	357	184	551	1005	878	16.30	17.89	16.84
b14	216	645	325	3484	4310	4060	47.47	51.49	50.14
b15	437	1278	659	4501	5999	5545	33.70	36.09	35.22
b20	435	1290	655	7649	9281	8806	45.70	49.56	49.31
b21	432	1290	650	7771	9419	8925	44.56	48.79	48.63
b22	622	1839	937	11177	13496	12826	45.20	49.35	49.11
b17	1390	4026	2093	13493	18286	16868	34.23	37.02	36.01
b18	3219	9207	4867	34576	45473	42493	46.69	49.11	49.45
b19	6384	18417	9644	61165	83037	76869	49.59	-	-

not have any internal memories other than flipflops and are synchronous.

In the benchmark package (itc99-poli2-vhd.tar.xz, version 7 Sept. 2014), additional circuits are available, which are b14_1, b15_1, b17_1, b18_1, b19_1, b20_1, b21_1, b22_1 and b30. These circuits are not included in our evaluation, because all from this list but the b30 are minor modified versions of the original circuits, and b30 was not a compilable circuit description. Moreover, the parametric circuit b16 was not available in the package, even it is documented in [CRS00].

Synthesis of the circuits were carried out the same way as in subsection 4.3.1, but this time we used the ProASIC3 FPGA with the highest area resources, the A3PE3000L. Table 4.7 shows the synthesis results for PBED cluster size of 3 and the table 4.8 shows the derived parameters.

Of all benchmark circuits only the hardened b19 cannot be layouted on the used FPGA for both LTMR, and PBED for cluster sizes lower than 4, so b19 does not have any timing data on tables 4.7 and 4.8.

Table 4.8: Derived parameters using the I99T synthesis results in the table 4.4. The last row shows the average (avg.) value of the last three derived parameters.

	$c_{\mathsf{FF}+}$		A_+		$t_{\mathsf{crit}+}$ (ns)		$\frac{A_+}{c_{\mathsf{FF,ba}}}$		
cir.	LT	PB	LT	PB	LT	PB	LT	PB	$\frac{A_{+,\mathsf{PB}}}{A_{+,\mathsf{LT}}}$
b02	8	4	12	9	1.65	0.55	3.00	2.25	0.75
b01	20	7	31	20	1.72	1.38	3.10	2.00	0.65
b06	16	5	24	15	1.69	1.17	3.00	1.88	0.63
b08	42	12	62	39	1.77	1.36	2.95	1.86	0.63
b03	59	17	88	58	2.02	0.92	2.84	1.87	0.66
b09	56	15	82	52	1.77	1.04	2.93	1.86	0.63
b10	48	14	72	46	1.81	1.11	3.00	1.92	0.64
b13	112	29	181	122	1.76	0.93	3.23	2.18	0.67
b07	88	23	130	81	2.08	1.27	2.95	1.84	0.62
b11	70	19	103	63	1.50	1.04	2.94	1.80	0.61
b04	132	34	251	177	1.89	1.91	3.80	2.68	0.71
b05	67	23	95	73	1.90	1.05	2.32	1.78	0.77
b12	235	62	454	327	1.60	0.54	3.72	2.68	0.72
b14	429	109	826	576	4.02	2.67	3.82	2.67	0.70
b15	841	222	1498	1044	2.39	1.52	3.43	2.39	0.70
b20	855	220	1632	1157	3.86	3.62	3.75	2.66	0.71
b21	858	218	1648	1154	4.23	4.07	3.81	2.67	0.70
b22	1217	315	2319	1649	4.16	3.91	3.73	2.65	0.71
b17	2636	703	4793	3375	2.80	1.78	3.45	2.43	0.70
b18	5988	1648	10897	7917	2.42	2.75	3.39	2.46	0.73
b19	12033	3260	21872	15704	-	-	3.43	2.46	0.72
						avg.	3.27	2.24	0.68

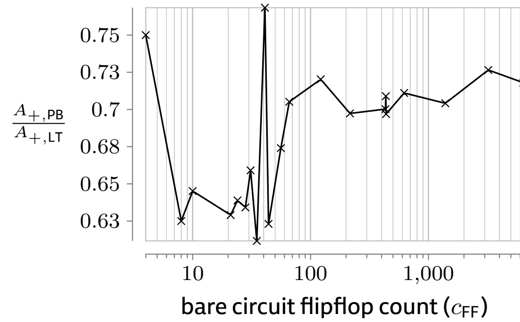

Figure 4.11: Area overhead ratio PBED ($s_{\text{cl}} = 3$) to LTMR for all I99T circuits plotted over the flipflop count in the bare circuit. The x-axis is drawn in logarithmic scale.

Circuits with a low number of flipflops have an overhead per application flipflop of about 3 in case of LTMR and less than 2 in case of PBED due to the enable flipflop conversion already done in the bare circuit, which we discussed in last subsection. In the rest of the circuits, the area overhead per application flipflop shown in table 4.8 goes up to 3.82 in case of the circuit b14 for LTMR, and 2.68 in case the circuit b04 for PBED.

The area overhead value of 3.82 shows that LTMR theoretically can have an area overhead per application flipflop of 4, if the bare circuit flipflops are only enable flipflops. The reason is as follows. An additional multiplexer for the enable flipflop conversion is needed for every application flipflop, so the area overhead is $3 + 1 = 4$. The same rule also applies to PBED, and the analytical results for the area overhead per application flipflop for PBED in table 4.2 can increase by 1.

As this additional overhead is needed by both of the hardening techniques, it is better to look at the area overhead ratio PBED to LTMR over the bare circuit flipflop count, which is shown in figure 4.11.

Figure 4.11 shows that for bare circuits with more than 60 flipflops, the area overhead ratio is between 0.7 and 0.73. The rest of the circuits have an area overhead ratio between 0.6 and 0.65 with the exception of two circuits, the b05 and b02. b05 and b02 make up the two peaks in figure 4.11.

The circuit b05 has an area overhead for LTMR of 2.32, which is extraordinary low for LTMR. As this circuit has a bare circuit flipflop count of 41, the LTMR area overhead should be near 3. The reason for the extraordinary low area overhead is that five flipflops have to be replicated during synthesis of the bare circuit, because they have a high fanout. LTMR triplicates the flipflops and connects their outputs to a majority voter, so during synthesis of the LTMR-hardened circuit, instead of flipflops, majority voters get replicated. Therefore, the LTMR-hardened b05 has $(41 - 5) \times 3 = 108$ flipflops. The extraordinary low area overhead for LTMR results in a high area overhead ratio of 0.77, which is one of the peaks in figure 4.11.

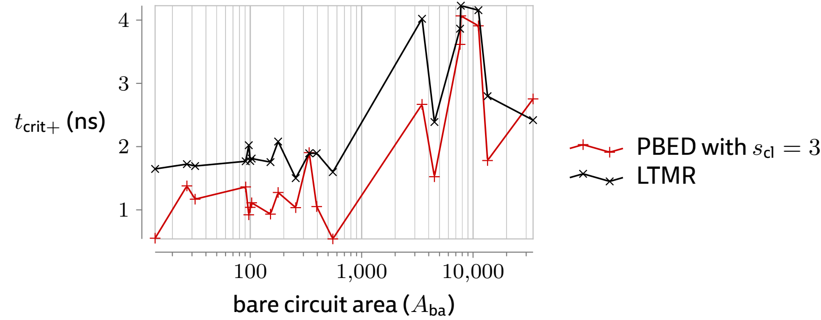

Figure 4.12: Critical path overhead delay ($t_{\text{crit}+}$) for every I99T circuit and hardening type, sorted according to bare circuit area. Plot template same as in figure 4.8, but the x-axis is drawn in logarithmic scale.

The bare version of the circuit b02 has a low flipflop count of four, three of them have clear inputs and one has preset input. As PBED only adds flipflops with the same clear/preset signal to a cluster (see section 4.4), three clusters are created in this case, which in turn creates three parity flipflops with parity-generation and -check circuitry. With the additional flipflop for the error output, PBED causes the high overhead of 2.25 and area overhead ratio of 0.75, which makes up the second exceptional peak in figure 4.11. For circuits with a low flipflop number but many not fully-utilized clusters, PBED results in a high area overhead.

Now, we look at the critical path. Figure 4.12 shows the critical path overhead delay for PBED is in all circuits but in b18 lower than LTMR. In critical path details we observed that in all off the I99T circuits the parity generation path causes the critical path and not the error signal reduction path. This is the reason why $t_{\text{crit}+}$ for PBED in figure 4.12 does not increase with increasing circuit area like in the synthesis results of the replicated FSMs in figure 4.8.

Until now, the evaluation was for a fixed cluster size of 3. Now, we present results with cluster sizes from 2 to 9 to analyze the impact of the cluster size on the critical path and area overhead. The figures 4.13 and 4.14 illustrate the critical path- and area-overheads for all circuits, respectively.

As the critical path of all the circuits is caused by the parity generation path, the critical path increases in most circuits with increasing cluster size. The reason is that more inputs to the XOR gate for parity generation creates more routing delay and more CLBs.

Compared to the critical path overheads in the FSM circuits, the cluster size has a noticeable effect on the critical path overhead. This is probably due to the regularity of the FSM circuits, as they were created by simple replication of a base circuit. Most of the I99T circuits are standalone circuits, and this makes the routing of the clusters more difficult. Consequently, there is a tradeoff between the critical path overhead of the circuit and the area overhead.

The form of the area overhead ratio plot is similar for most of the circuits, with

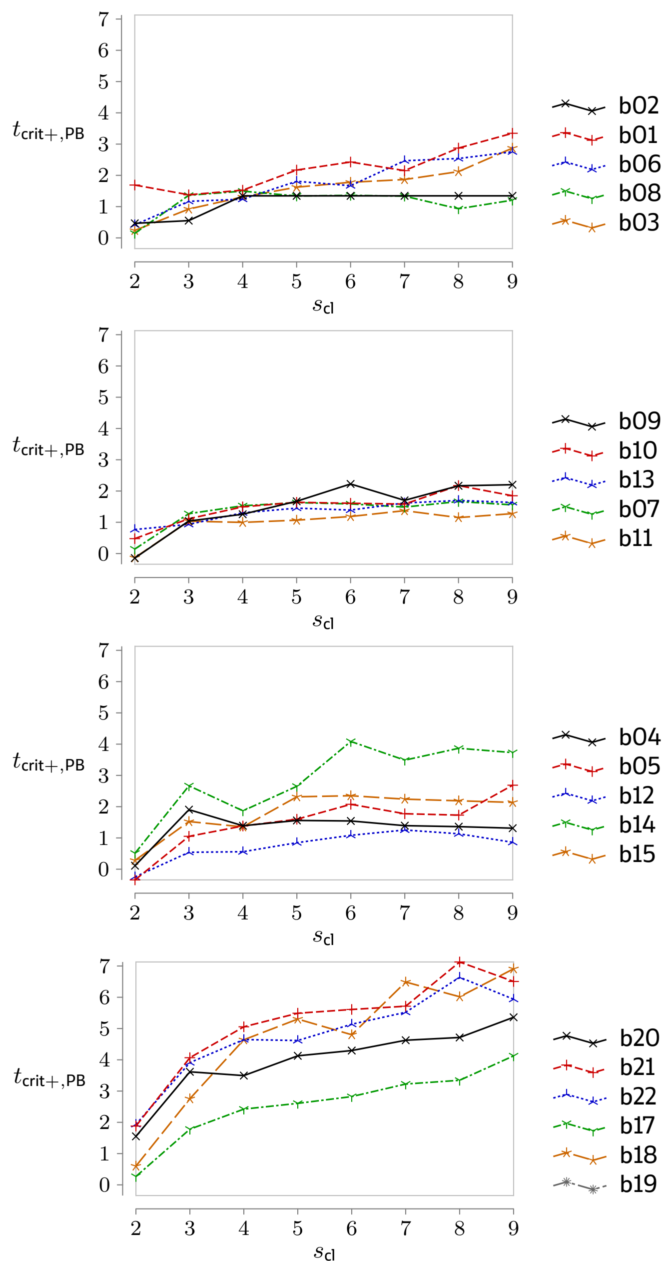

Figure 4.13: Critical path overhead delay ($t_{\text{crit}+}$) for PBED hardened I99T circuits with cluster sizes (s_{cl}). Plot template same as figure 4.9. The points for the circuits b19 do not exist, because these could not be placed and routed.

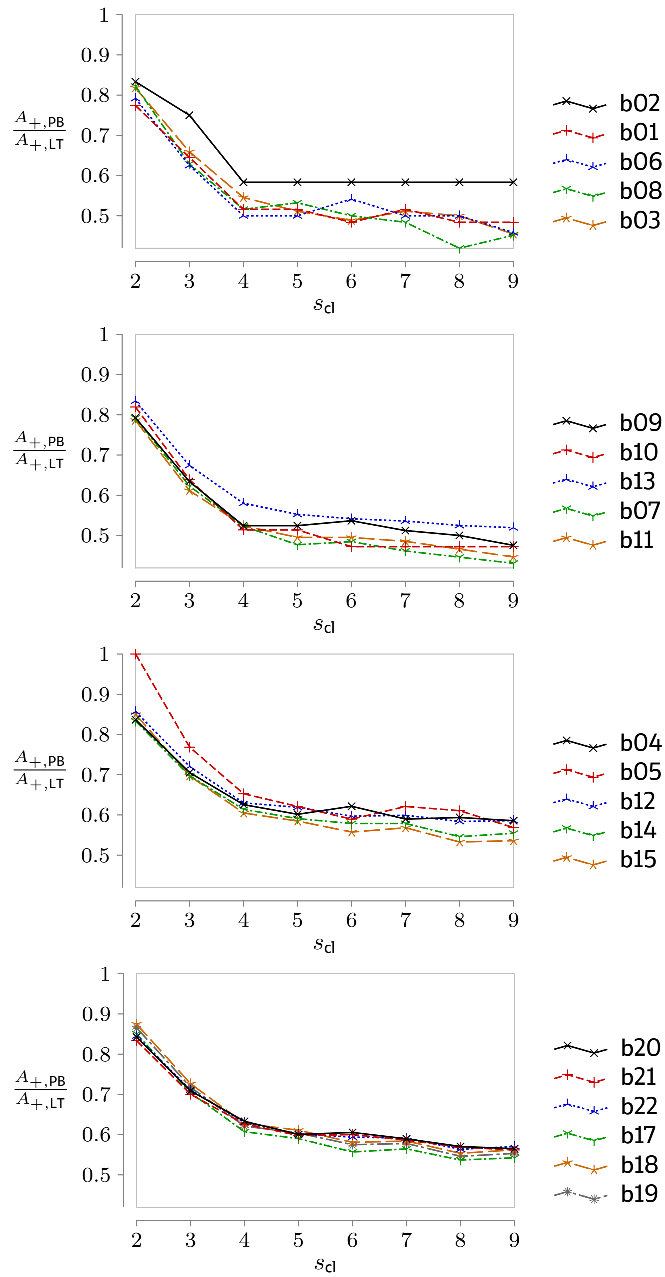

Figure 4.14: Area overhead (A_+) per application flipflop ($\frac{A_+}{c_{\mathsf{FF,ba}}}$) for PBED hardened I99T circuits with cluster sizes varying from 2 to 9. Plot template same as figure 4.10.

Table 4.9: Minimum and maximum values for derived parameters for PBED hardened I99T circuits for various cluster sizes (s_{cl})

s_{cl}	$t_{\mathsf{crit}+}$ (ns)	$\frac{A_+}{c_{\mathsf{FF,ba}}}$	$\frac{A_{+,\mathsf{PB}}}{A_{+,\mathsf{LT}}}$
2	-0.34 - 1.93	2.31 - 3.19	0.77 - 1.00
3	0.54 - 4.07	1.78 - 2.68	0.61 - 0.77
4	0.56 - 5.06	1.50 - 2.39	0.50 - 0.65
5	0.85 - 5.49	1.41 - 2.30	0.48 - 0.62
6	1.08 - 5.61	1.37 - 2.36	0.47 - 0.62
7	1.25 - 6.49	1.36 - 2.24	0.46 - 0.62
8	0.93 - 7.13	1.24 - 2.26	0.42 - 0.61
9	0.86 - 6.91	1.27 - 2.23	0.43 - 0.59

the exception of b02 and b05. The decay is visible until cluster size 5. At 6 a local peak is present, followed by a slow decay. The plots of the bigger circuits, the two bottom plots in figure 4.14, are positively shifted in y-axis compared to the smaller circuits. This shows that the PBED has more area overhead in bigger circuits compared to LTMR. For example, the circuits in the second plot, b09 to b11, have an area overhead ratio between 0.5 and 0.6, where the circuits in the third and fourth plots, b04 to b18, have an area overhead ratio of 0.6.

Finally, table 4.9 summarizes the minimum and maximum values for the derived values. The upper bound for the critical path overhead is caused by the b19, which could not be layouted for the cluster sizes 2 and 3, so the upper bound for the cluster sizes 2 and 3 differ from the rest.

According to the table 4.9, PBED can save up to 58% of the LTMR area overhead. The upper bound for the area overheads is caused by the circuit b05, therefore they differ significantly to the upper bounds in the FSM circuits.

4.4 Automatic application

PBED can be applied on-top of a technology-level netlist using an automatic tool. The tool is for Microsemi ProASIC3 FPGA primitives and is available at [Ayd16]. The pseudo code of the direct PBED application program is shown in algorithm 4.1.

We used various third-party tools for our tool. For the technology-level netlist processing, the circuit description needs to be parsed and a new circuit description must be generated. We used Verilog-Perl [Sny16] for this purpose. The place-and-router tool Designer from Microsemi can output two-dimensional coordinates of the primitives after a place-and-route run. We used this information for partitioning. We explain the algorithm more in detail in what follows.

A proper partitioning of the flipflops can have a significant impact on the timing of the routed circuit. For instance, if two application flipflops which are very far

Data: technology-level netlist, placing try count, cluster size, partitioning try count
Result: direct PBED applied technology-level netlist

```
1  for t = 1 to placing try count do
2  |   placer seed = t;
3  |   place the netlist using the placer tool in Designer;
4  |   route the placed netlist using the router tool in Designer;
5  end
6  pick the routed netlist with the shortest critical path;
7  extract flipflop coordinates from the picked routed netlist;
8  foreach flipflop do
9  |   if has enable input then
10 |   |   eliminate enable input;
11 |   end
12 |   if has negated output then
13 |   |   eliminate negated output;
14 |   end
15 |   categorize according to clock- and reset-signal;
16 end
```

Algorithm 4.1: Application of direct PBED to a technology-level netlist p.1

from each other are put into the same cluster, then the input nets to the XOR for parity generation (XOR_{pg}) will have longer routes than in the case of two neighboring flipflops. These long routes in turn can pose a higher critical impact on the critical path. For this reason, before altering the technology-level netlist, we gather the physical information about the application flipflops in lines from 1 to 7.

As the placing and routing process is usually based on heuristics, we run the place-and-router multiple times ($placing\ try\ count$ in line 1) with different seeds. In this work, we used $placing\ try\ count = 8$ for the experiments. The placer and router tool optimizes for the best timing, i.e., the shortest critical path.

At the end of the runs, we select the run with the best timing and use this run for flipflop location extraction. A primitive on ProASIC3 can be located by a two-dimensional coordinate.

At line 8, we begin processing the technology-level netlist. All the flipflops which have an enable input or a negated output must be replaced with a basic flipflop with clock input, data input, data output, and reset input if applicable. See figure 4.15 for a visualization. Note that this conversion must also be done in LTMR. This conversion or remapping is called *two tile implementation* by the layout tool, because the multiplexer and flipflops require one tile (i.e., CLB) each. The elimination of the negated output strictly requires an additional tile, but usually this negation is propagated to the gates connected to the flipflop output and does not create additional area overhead.

In the two tile implementation, the multiplexer emulates the enable behavior

```
   Data: technology-level netlist, placing try count, cluster size, partitioning
         try count
   Result: direct PBED applied technology-level netlist
18 if location-aware partitioning then
19   foreach flipflop category do
20     unclustered flipflops = flipflops in this flipflop category;
21     clusters with min. distance = ∅;
22     min. total distance = ∞ ;
23     clusters for this try = ∅;
24     total distance for this try = 0;
25     for t = 1 to partitioning try count do
26       while there are unclustered flipflops do
27         cluster = new cluster;
28         master = pick a random flipflop;
29         push master to cluster;
30         while there are unclustered flipflops and cluster is not full do
31           neighbor = pick the nearest unclustered flipflop to
              master;
32           push neighbor to cluster;
33           total distance for this try + = distance between master
              and neighbor;
34         end
35         push cluster to clusters for this try;
36       end
37       if total distance for this try < min. total distance then
38         min. total distance = total distance for this try;
39         clusters with min. distance = clusters for this try;
40       end
41     end
42     use clusters with min. distance as partitioning;
43   end
44 else // random partitioning
45   foreach flipflop category do
46     unclustered flipflops = flipflops in this flipflop category;
47     while there are unclustered flipflops do
48       cluster = new cluster;
49       while there are unclustered flipflops and cluster is not full do
50         random flipflop=pop from unclustered flipflops;
51         push random flipflop to cluster;
52       end
53     end
54   end
55 end
56 foreach cluster do
57   add parity-generation and -check circuitry;
58 end
59 reduce cluster error signals to a single error signal;
```

Algorithm 4.2: Application of direct PBED to a technology-level netlist p.2

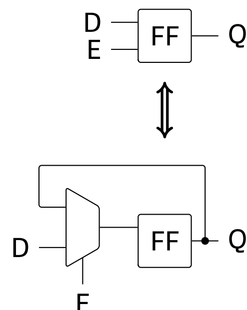

Figure 4.15: Conversion of a flipflop with enable input to a basic flipflop with multiplexer. This is done to be able to update the flipflop content in case of a bitflip.

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

In the next step, the flipflops are categorized according to cluster size, and clock and reset signals of flipflops. The flipflops in a cluster must be sensitive to the same clock signal and edge. Furthermore, all the flipflops in a cluster must have the same reset type: all active-low or -high. These constraints enable the connection of the parity flipflop to the same clock and reset signal of the application flipflops in the cluster.

The loop beginning at line 19 carries out the partitioning of the flipflops. The partitioning creates flipflop clusters with a constant size for each flipflop category by using the physical information about the application flipflops gathered in lines 1 to 7.

The clusters are created around a *master* flipflop, where *neighboring* flipflops are picked from the list of unclustered flipflops list by the distance to the master flipflop (loop at line 26). As distance, the two-dimensional Euclidean distance without square root is used:

$$distance = (x_{\text{master}} - x_{\text{neighbor}})^2 + (y_{\text{master}} - y_{\text{neighbor}})^2 \tag{4.31}$$

New clusters are created until there are no unclustered flipflops left.

This approach does not find the solution with the minimal distance sum over all clusters, therefore we repeat the partitioning many times (loop at line 25), which is determined by the input parameter *partitioning try count*. For the experiments in this work, we set this parameter to 1000, i.e., $partitioning\ try\ count = 1000$.

As the figure of merit for each partitioning, we use the sum of all distances between neighboring and master flipflops in all clusters (line 33), which is called *total distance for this try* in the algorithm. For each flipflop category, we pick the partitioning with the minimum total distance (if block at line 37).

The partitioning of the flipflops was done on the technology-netlist level, because the layout tool does not provide any programming interface to implement the partitioning inside the layout tool.

Instead of location-aware partitioning, random partitioning can be used. In this case, the clusters are filled with random flipflops without respecting their location (else block at line 44.)

After the partitioning is completed, the parity-generation and -check circuitry is added to every cluster. The error signals of all clusters are then reduced to a single error signal. This signal is added as an additional primary output to the netlist.

Our partitioning solution is an approximate solution. The partitioning problem can be solved with k-means clustering with a fixed cluster size and the k-means clustering problem is NP-hard [MNV12]. For our tool we need partitioning with fixed cluster sizes, so we decided to implement a simple heuristic as a proof of concept.

In the following, we will analyze the computational complexity of the algorithm. For this purpose, we introduce the following variables:

- c_{FF} number of all flipflops in the netlist
- c_{cat} number of flipflop categories
- cat_i i'th flipflop category
- $c_{\mathsf{FF},i}$ number of flipflops in category cat_i
- s_{cl} cluster size
- $c_{\mathsf{FF},i,\mathsf{mst}}$ number of master flipflops in category cat_i
- $c_{\mathsf{FF},i,\mathsf{uncl},j}$ number of unclustered flipflops in category cat_i before j'th iteration
- c_{parttry} partitioning try count (constant)
- c_{placetry} placing try count (constant)

In the following, we analyze the time complexity of the algorithm dependent on the number flipflops c_{FF} in the circuit.

The loop in lines from 1 to 5 is run c_{placetry} times but depends on the third-party place-and-route tool, therefore the time complexity cannot be estimated. Line 6 picks from c_{placetry} routed netlists the one with maximum frequency and can be processed in c_{parttry} steps, which is constant. Line 7 is done by a third-party parser and is linear to the components present in the circuit netlist. Lines 8 to 16 iterates over all flipflops in c_{FF} steps, because categorizing a flipflop means adding it to an array which includes only flipflops belonging to this category. The loop in lines from 19 to 43 will be analyzed beginning from the next paragraph. The loop in lines from 56 to 58 iterates over all clusters and applies parity-generation and -check circuitry, which in turn iterates over all flipflops in a cluster. So, the loop

in lines from 56 to 58 can be processed in c_{FF} steps. Line 59 iterates over all the cluster error signals and ORs them, which also corresponds to c_{FF} steps.

The loop from 19 to 43 iterates over all flipflop categories, and partitions the flipflops in a category c_{parttry} times in lines from 25 to 41. For each partitioning, clusters are generated in lines from 26 to 36. For each master flipflop in a cluster, $s_{\mathsf{cl}} - 1$ nearest flipflops are picked, which means iterating over all the unclustered flipflops in the current category.

The number of master flipflops in category cat_i corresponds to:

$$c_{\mathsf{FF},i,\mathsf{mst}} = \left\lceil \frac{c_{\mathsf{FF},i}}{s_{\mathsf{cl}}} \right\rceil \tag{4.32}$$

For each master flipflop in a category, $s_{\mathsf{cl}} - 1$ nearest flipflops are picked in lines from 30 to 34. Before the first iteration, after the master flipflop for the first cluster has been picked, there are $c_{\mathsf{FF},i} - 1$ unclustered flipflops, and before the second iteration, there are $c_{\mathsf{FF},i} - s_{\mathsf{cl}} - 1$ unclustered flipflops. So, the number of unclustered flipflops before j'th iteration corresponds to:

$$c_{\mathsf{FF},i,\mathsf{uncl},j} = c_{\mathsf{FF},i} - (j-1)s_{\mathsf{cl}} - 1 \tag{4.33}$$

The master flipflops are picked randomly, thus the partitioning for each flipflop category is done c_{parttry} times. So, the number of comparisons corresponds to:

$$\sum_{i=1}^{c_{\mathsf{cat}}} \left(c_{\mathsf{parttry}} \cdot \sum_{j=1}^{c_{\mathsf{FF},i}} (c_{\mathsf{FF},i,\mathsf{mst}} \cdot c_{\mathsf{FF},i,\mathsf{uncl},j})\right) \tag{4.34}$$

After we have developed the number of steps in general form, we can carry on with best- and worst-case analysis.

In best case, every single flipflop belongs to another category, and no comparison needs to be done:

$$c_{\mathsf{cat}} = c_{\mathsf{FF}} \implies c_{\mathsf{FF},i,\mathsf{uncl},j} = 0 \tag{4.35}$$

In this case, only the lines from 32 to 40 are executed, which are processed in $O(1)$ time. So, in best case the partitioning is accomplished in $c_{\mathsf{parttry}} \cdot c_{\mathsf{FF}}$ steps, which corresponds to $\Omega(c_{\mathsf{FF}})$.

In worst case, there is only one flipflop category, and the partitioning is done in $c_{\mathsf{parttry}} \cdot c_{\mathsf{FF}}^2$ steps, which corresponds to $O(c_{\mathsf{FF}}^2)$.

The lines other than the loop in lines 19 to 43 are processed in c_{FF} steps, so the time complexity of the algorithm is determined by the loop in lines 19 to 43.

For the evaluation of our location-aware partitioning approach, we synthesized the PBED-hardened FSM and I99T circuits using cluster size of 3 both with location-aware and random partitioning. The results are plotted in figures 4.16 and 4.17.

Contrary to our expectations, location-aware partitioning does not always result in a better timing. In case of the FSM circuits (figure 4.16), the critical path difference is less than 1 ns, and in case of the I99T circuits, less than 1.5 ns. The

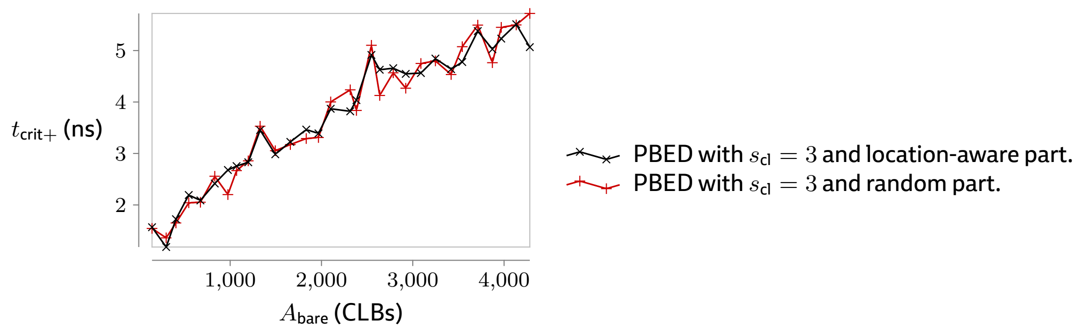

Figure 4.16: Impact of location-aware partitioning on the critical path of FSM circuits. In this plot, critical path overhead delay ($t_{\text{crit}+}$) for every FSM circuit hardened with PBED for both with location-aware and random partitioning (part.) are shown, sorted according to bare circuit area. The x-axis is drawn in logarithmic scale.

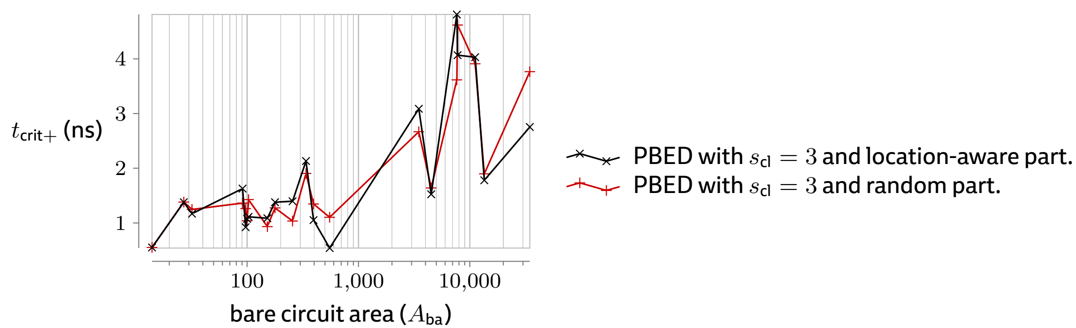

Figure 4.17: Impact of location-aware partitioning on the critical path of I99T circuits. Critical path overhead delay ($t_{\text{crit}+}$) for every I99T circuit hardened with PBED for both with location-aware and random partitioning (part.) are shown, sorted according to bare circuit area. The x-axis is drawn in logarithmic scale.

number of sequential elements for a PBED-hardened circuit does not differ for both partitioning techniques, and the difference between the number of LUTs is less than 5. We cannot observe any significant impact of the location-aware partitioning. For the best timing, both partitioning techniques should be tried. For our experimental evaluations in this work, we synthesized the PBED-hardened circuits with both partitioning techniques and have picked the design with the best timing.

Chapter 5

Pipelined cluster error signal reduction

In the previous chapter, we introduced PBED with direct cluster error reduction, which can have negative impact on the circuit critical path and thus on the timing. In this chapter, we show an alternative approach based on pipelining of the cluster error signal reduction. The following sections are structured similar to the PBED chapter with the exception of the analytical evaluation section. The analytical comparison was done to assess the theoretical limits of direct PBED and pipelined PBED is only evaluated using synthesis results.

In the following sections, we first introduce the pipelined PBED. Then, we present the experimental evaluation and finally discuss the automatic application of this PBED approach.

5.1 Concept

PBED reduces the cluster error signals to a single circuit error signal. In circuits with numerous flipflops, this can create a long error detection path. 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, this approach is called PBED with *pipelined* cluster error signal reduction and will be abbreviated as *pipelined PBED* in the following.

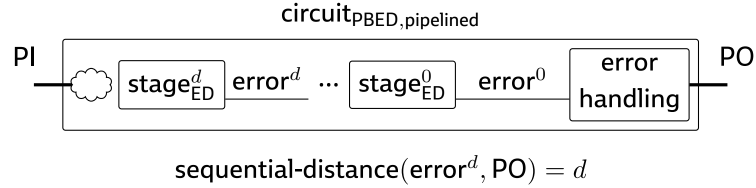

Figure 5.1: PBED with pipelined cluster error signal reduction: top view on a hardened circuit with $d + 1$ stages

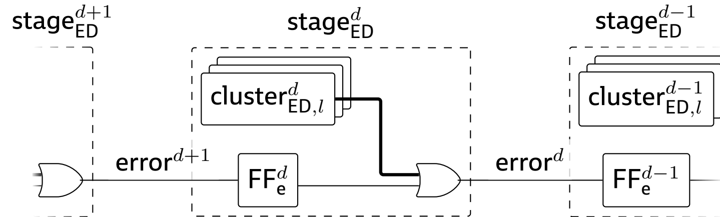

Figure 5.2: PBED with pipelined cluster error signal reduction: single stage with neighbor stages

In contrast to direct PBED, in pipelined PBED, flipflops are grouped according to their *sequential distance* d_{seq} to any primary output 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 output is a PO of the circuit has $d_{\text{seq}} = 0$. flipflops with $d_{\text{seq}} = d$ belong to a particular *error detection stage*, which is named stage$^d_{\text{ED}}$. These stages are visualized in figure 5.1.

The inner structure of a stage is shown in figure 5.2. Analogous to direct PBED, the flipflops are grouped in clusters within a stage. Stages contain an *error flipflop* FF$^d_{\text{e}}$, 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$^d_{\text{ED}}$, error^d, is generated by ORing the buffered error signal from the last stage and the error signals from the clusters within the stage.

On the one hand, pipelined PBED shrinks the OR-tree for error signal reduction, which can result in a shorter critical path if the OR-tree is on the critical path. On the other hand, pipelined PBED results in more overhead, stages introduce another level of flipflop category (like reset-,clock-signal) and flipflops from different categories cannot be clustered together. This can result in more incomplete clusters and thus more area overhead.

5.2 Experimental evaluation

In section 5.1, we saw that pipelined PBED is a timing enhancement on the direct PBED. Hence, the experimental evaluation will show the differences between

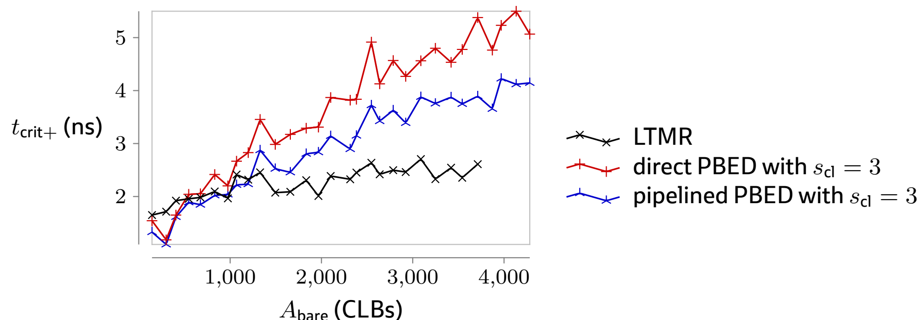

Figure 5.3: Critical path delay overhead ($t_{\text{crit}+}$) over bare circuit area (A_{bare}) for LTMR- and PBED-hardened FSM circuits with PBED cluster size of 3

these two approaches in overall and additionally use the former comparisons with LTMR from the last chapter.

Same circuits and same experiment conditions from section 4.3 are used in the following, so the introductions to the evaluations will be omitted and the subsection will immediately start with the description and evaluation of the experimental results.

5.2.1 Finite state machine (FSM) circuit

Table 5.1 shows the synthesis results for the FSM circuits which are hardened by pipelined PBED with cluster size of 3. The table contains absolute output parameters as well as the difference (diff.) to the direct PBED approach.

The area parameters, flipflop count and area do not show a significant difference (i.e., $c_{\text{FF,diff}}$, A_{diff}). At least one flipflop is added to every circuit, as the FSM circuit has two stages and pipelined PBED requires one flipflop for every stage but the last stage. In some circuits, flipflop count difference raises to three, which is probably due to optimizations like adding of additional primitives to balance the load on high fanout nets. The possible overhead due to optimizations also apply to combinational elements.

The area difference fluctuates more than the flipflop count. Pipelined PBED introduces stages, which should have a negative impact on area as the cluster error signals for particular stages are reduced independently. This in turn causes a non-exhaustive utilization of the three input XOR LUTs and thus more additional area. But this effect is not clearly observable because the FSM has only two stages. All in all, the average and maximum difference for sequential elements and total area stay below 4 and 10 tiles, respectively.

For all of the circuits, the critical path is shorter. Figure 5.3 plots the critical path delay overheads of pipelined-, direct-PBED and LTMR over the bare circuit area for a detailed evaluation.

Table 5.1: Synthesis results for the FSM circuits hardened by pipelined PBED with cluster size of 3. Absolute (abs.) values as well as differences (diff.) to the direct PBED are shown. A positive difference means a higher value for the pipelined PBED. The last row shows the average values for the differences.

	c_{FF}		A		t_{crit} (ns)	
circ.	abs.	diff.	abs.	diff.	abs.	diff.
1	41	1	191	1	9.29	-0.21
2	77	1	389	1	9.64	-0.09
3	112	1	585	0	10.02	-0.04
4	148	1	783	0	10.29	-0.15
5	184	1	970	-1	10.31	-0.21
6	220	1	1186	0	10.45	-0.39
7	256	1	1389	2	10.50	-0.16
8	292	1	1544	-1	10.61	-0.45
9	328	1	1731	1	10.78	-0.60
10	364	1	1921	1	11.01	-0.58
11	400	1	2142	-3	11.19	-0.46
12	437	1	2365	-2	11.15	-0.72
13	475	3	2603	3	11.49	-0.48
14	511	3	2800	4	11.48	-0.48
15	547	3	2989	1	11.64	-0.73
16	583	3	3264	2	11.61	-0.92
17	619	3	3386	-1	11.79	-0.69
18	655	3	3613	4	11.85	-1.19
19	689	1	3761	3	11.83	-0.70
20	725	1	3970	1	12.01	-0.94
21	763	3	4168	5	12.14	-0.88
22	799	3	4395	3	12.26	-0.68
23	835	3	4609	-4	12.42	-1.04
24	873	3	4842	1	12.61	-0.66
25	908	2	5021	1	12.48	-1.03
26	944	2	5244	-4	12.63	-1.48
27	980	2	5472	2	12.54	-1.11
28	1015	1	5619	-8	12.91	-1.00
29	1052	2	5850	9	12.99	-1.38
30	1087	1	6051	-9	13.07	-0.92
31	1123	1	6236	0	-	-
	avg.	1.83		0.02		-0.68

Table 5.2: Minimum and maximum values for derived parameters for PBED hardened FSM circuits for various cluster sizes (s_{cl})

s_{cl}	$t_{\mathsf{crit}+}$ (ns)	$\frac{A_+}{c_{\mathsf{FF,ba}}}$	$\frac{A_{+,\mathsf{PB}}}{A_{+,\mathsf{LT}}}$
2	0.06 - 4.42	2.31 - 2.97	0.78 - 0.84
3	1.10 - 4.23	1.84 - 2.46	0.62 - 0.69
4	1.28 - 4.25	1.55 - 2.18	0.52 - 0.61
5	1.67 - 4.54	1.49 - 2.11	0.50 - 0.58
6	1.88 - 4.55	1.51 - 2.10	0.51 - 0.59
7	2.25 - 4.71	1.39 - 2.04	0.47 - 0.57
8	2.35 - 5.10	1.41 - 1.98	0.48 - 0.55
9	2.25 - 5.13	1.39 - 1.96	0.47 - 0.54

For the bare circuit areas greater than 500 CLBs, the critical path overhead is improved by 0.5 to 2 ns, but LTMR has still about 1 to 2 ns less critical path overhead. For circuit area less than 500, all hardening techniques lead to similar results.

We omit detailed results for each circuit and cluster size like we did in section 4.3, because pipelining does not significantly change the impact of cluster size. Instead of detailed results, table 5.2 summarizes the minimum and maximum values for the derived values for each cluster size.

5.2.2 I99T circuits

Table 5.3 shows synthesis results for I99T circuits hardened by pipelined PBED with cluster size of 3.

In all circuits the flipflop count difference to the direct PBED is greater than the total stage count without the last stage ($c_{\mathsf{FF,diff.}} >= c_{\mathsf{stage}} - 1$), which supports the plausibility of the resulted flipflop count.

Pipelined PBED can result in shorter critical path, but not always. The critical path delay difference to direct PBED is less than 1.25 ns. Figure 5.4 shows the critical path delay of the hardened circuits sorted according to the bare circuit area. In most cases PBED results in a shorter critical path than LTMR. The exceptions are the circuits b18 and b21, in which pipelined-PBED results in a longer critical path than the LTMR.

Like in subsection 5.2.1, we omit detailed analysis by each cluster size and circuit and minimum and maximum values for the derived parameters for each cluster size in table 5.4, as the pipelining does not change the impact of cluster size variation.

Finally, we analyzed the relative flipflop count over the stages in I99T circuits. Figure 5.5 shows the relative flipflop count in percent for a particular circuit.

We observe that in most circuits about half of the flipflops have a distance of

Table 5.3: Synthesis results for I99T circuits hardened by pipelined PBED with cluster size of 3. Additionally, in the last column the total stage count (c_{stage}) gathered by the PBED tool is printed. For the remaining parameters confer to table 5.1.

	c_{FF}		A		t_{crit} (ns)		
circ.	abs.	diff.	abs.	diff.	abs.	diff.	c_{stage}
b02	9	1	23	0	5.56	0.42	1
b01	19	2	49	2	5.85	0.01	2
b06	15	2	48	1	6.37	-0.30	1
b08	35	2	133	3	11.55	-0.30	2
b03	55	7	160	5	9.12	-0.00	5
b09	55	12	161	9	9.79	-0.91	8
b10	40	2	151	2	7.82	-0.04	2
b13	90	5	277	3	9.12	0.05	3
b07	69	2	258	-1	14.87	0.03	2
b11	58	4	322	3	17.24	-0.92	2
b04	104	4	516	1	24.98	-0.73	4
b05	65	1	470	2	25.61	-0.02	1
b12	189	5	881	3	16.68	-0.16	4
b14	328	3	4062	2	49.05	-1.09	2
b15	664	5	5549	4	34.40	-0.82	3
b20	658	3	8808	2	48.73	-0.58	3
b21	653	3	8930	5	49.16	0.53	3
b22	940	3	12827	1	49.27	0.16	3
b17	2099	6	16867	-1	36.32	0.31	5
b18	4872	5	42495	2	50.69	1.25	5
b19	9649	5	76864	-5	-	-	5
	avg.	3.90		2.05		-0.16	

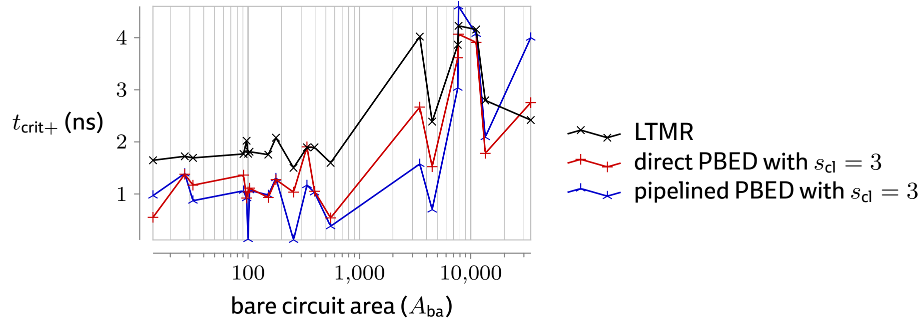

Figure 5.4: Critical path delay overhead ($t_{\text{crit}+}$) over bare circuit area (A_{bare}) for LTMR- and PBED-hardened circuits with PBED cluster size of 3

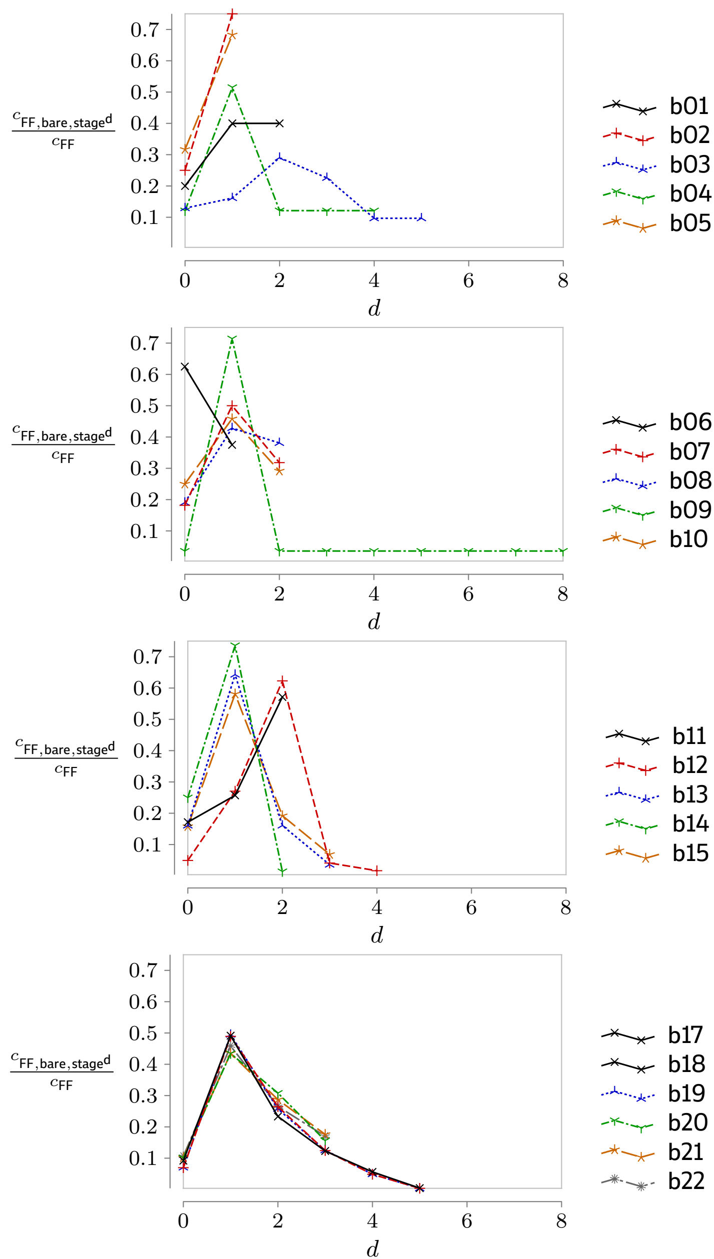

Figure 5.5: The distribution of flipflops over the stages for every I99T circuit. stage^d is a stage with a sequential distance of d to the primary output of the circuit.

Table 5.4: Minimum and maximum values for derived parameters for PBED hardened I99T circuits for various cluster sizes (s_{cl})

s_{cl}	$t_{\mathsf{crit}+}$ (ns)	$\frac{A_+}{c_{\mathsf{FF,ba}}}$	$\frac{A_{+,\mathsf{PB}}}{A_{+,\mathsf{LT}}}$
2	-0.44 - 1.51	2.31 - 3.20	0.79 - 1.01
3	0.12 - 4.60	1.82 - 2.70	0.62 - 0.79
4	-0.01 - 9.34	1.56 - 2.50	0.55 - 0.75
5	0.30 - 11.32	1.46 - 2.34	0.50 - 0.75
6	0.52 - 6.60	1.50 - 2.42	0.51 - 0.75
7	0.60 - 5.95	1.54 - 2.41	0.52 - 0.75
8	0.49 - 6.38	1.41 - 2.32	0.48 - 0.75
9	0.33 - 7.01	1.36 - 2.30	0.46 - 0.75

one cycle to the output. Although this allows for a pipelining in cluster error signal reduction, a more homogeneous distribution of the flipflops should result in better timing for pipelined PBED.

5.3 Automatic application

Also pipelined PBED can be automatically applied to a technology-level netlist. Our tool that we introduced in the last chapter also supports pipelined PBED. The pseudo code of the pipelined PBED application program is shown in algorithm 5.1.

Compared to the direct PBED approach, in pipelined PBED, the sequential distance to PO d_{seq} of every flipflop must be determined. For this purpose, a flipflop-only dataflow graph is generated by setting the POs as sink vertexes and exploring how the flipflops are connected to each other and to the primary output by using breadth-first search in the loop at line 7+1. While traversing, the flipflops are annotated with $d_{\mathsf{seq,PO}_i}$ to each single PO. Subsequently, the minimum of these $d_{\mathsf{seq,PO}_i}$s is determined at line 14+1, which corresponds to d_{seq} to the output:

$$d_{\mathsf{seq}} = \min_{\forall i} d_{\mathsf{seq,PO}_i} \tag{5.1}$$

In the next step (line 15), the flipflops are put to clusters like in direct PBED, but cluster generation in pipelined approach additionally respects the sequential distance to the primary output, that not only the flipflops with the same clock edge and reset are allowed to be in the same cluster, but also with the same sequential distance.

Then, the clusters are generated. In the loop at line 45+1, the clusters are put into error detection stages with respective sequential distance to the primary output. Finally, the stages are interconnected.

In the following, we analyze the time complexity introduced by the pipelined PBED approach when compared to the direct approach. For this purpose, we in-

Result: pipelined PBED applied technology-level netlist

⋮

7+1 **foreach** *primary output (PO)* **do**

7+2 build a flipflop dataflow graph with this PO as sink vertex and annotate the flipflops with sequential distance to this PO;

7+3 **end**

8 **foreach** *flipflop* **do**

⋮

14+1 determine $min.\ sequential\ distance\ to\ output$;

15 categorize according to ... and $min.\ sequential\ distance\ to\ output$;

16 **end**

⋮

59 ~~reduce cluster error signals to a single error signal~~;

59+1 **for** *sequential distance (d_{seq})=max to 0* **do**

59+2 put clusters with d_{seq} to a new stage;

59+3 reduce cluster error signals to a single error signal;

59+4 merge the error signal from the previous stage;

59+5 add an error flipflop to the stage;

59+6 **end**

Algorithm 5.1: Application of pipelined PBED to a technology-level netlist. Only the differences to algorithm 4.1 are shown. Added lines are labeled as $l + a$, which means a'th line added after the l'th line from algorithm 4.1.

troduce following variables additional to the variables that we introduced in section 4.4:

- c_{LUT} number of all LUTs in the netlist
- c_{PO} number of primary outputs in the netlist
- $d_{\text{seq,max}}$ maximum sequential distance in the netlist
- c_{cl} number of clusters in the circuit

The loop in line 7+1 does a breadth-first search on a connected graph (the netlist) for each primary output. Firstly, we do the worst case analysis. A dense connected graph can be processed in $O(n^2)$ time, where n is the number of vertexes. So, building the flipflop data-graph for all primary outputs corresponds to $O(c_{\text{PO}} \cdot (c_{\text{LUT}} + c_{\text{FF}})^2)$. Generally, most of the components of the netlist are traversed after some of the primary outputs have been processed, and the connectivity information (i.e., the neighbors of a flipflop) can be cached. Therefore, the time complexity can be reduced to $O((c_{\text{LUT}} + c_{\text{FF}})^2)$. In best case, we have a sparse graph, which corresponds to $\Omega(c_{\text{LUT}} + c_{\text{FF}})$. If we assume that $c_{\text{LUT}} = const \cdot c_{\text{FF}}$, the time complexity can be reduced to $O(c_{\text{FF}}^2)$ and $\Omega(c_{\text{FF}})$.

The line 14+1 corresponds to comparison of c_{PO} values, but c_{PO} cannot be determined exactly. If we assume that the number of primary outputs do not exceed the number of flipflops in the circuit, the line 15 can be processed in $O(c_{\text{FF}})$ and $\Omega(1)$ time. These lines are processed for each flipflop, so they correspond to $c_{\text{FF}} \cdot (c_{\text{FF}} + 1)$ steps and to $O(c_{\text{FF}}^2)$ and $\Omega(c_{\text{FF}})$.

The loop in line 45+1 includes operations which iterate over the clusters, that have a specific sequential distance. So, in total, this loop iterates over all flipflop clusters, and is processed in c_{cl} steps. In worst case, every flipflop belongs to a different category, and in best case to the same category. This corresponds to $O(c_{\text{FF}})$ and $\Omega(1)$.

Time complexity of the direct PBED approach corresponds to $O(c_{\text{FF}}^2)$ and $\Omega(c_{\text{FF}})$, which was discussed in section 4.4. We see that the time complexity is not changed by the additional steps introduced by the pipelined PBED approach.

Chapter 6

Transaction-based processing & recovery

In previous chapters, we presented and evaluated the error detection part of EDFT. A fault-tolerant system should provide end-to-end reliability, i.e., after error detection, the system should recover itself from the erroneous state. As shown in figure 6.1, the rest of EDFT is based on:

- system recovery in the target circuit
  - circuit isolation
  - circuit reset
- detection and recovery in the user component
  - transaction-based processing

System recovery reacts to a detected error and ensures a system state without errors. Due to error detection-only (parity) approach instead of error-correcting codes in our approach, it is not possible to compensate an error immediately and recover the system. Instead, recovery is done by circuit isolation (fault masking), and circuit reset (rollforward). Transaction-based processing is for error detection and system recovery in the user component.

In the next sections, we first present and evaluate the system recovery components for the target circuit, and then for the user component. Finally, we discuss their automatic application.

6.1 Recovery in the target circuit

System recovery in the target circuit consists of the circuit isolation and circuit reset components. Like PBED, system recovery can be implemented in a transparent fashion to the target circuit as shown in figures 6.2 and 6.3.

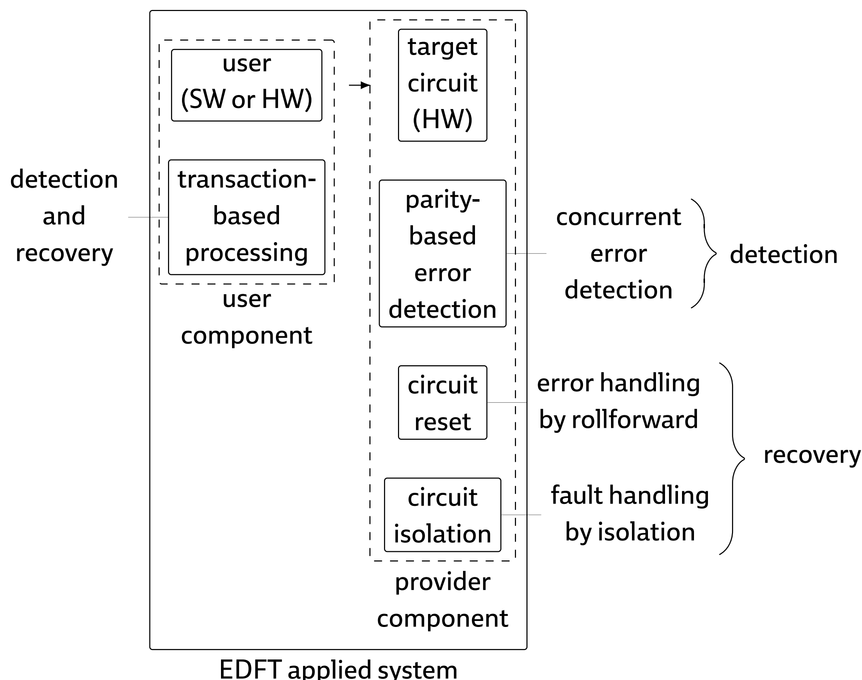

Figure 6.1: EDFT applied on the reference architecture (Figure 2.8 reused)

We present the two components in detail in subsections 6.1.1 and 6.1.2.

6.1.1 Circuit isolation

The goal of isolation is that an error in the target circuit does not propagate to the rest of the system. As the circuit recovery can take multiple clock cycles (e.g., an asynchronous reset over multiple clock cycles), an erroneous data word in target circuit can propagate to neighboring circuits and cause additional errors. Consequently, the reaction latency to an error must be bounded.

Isolation can be achieved, e.g., by stopping the clock for the target circuit (clock gating) or masking its output signals. In our work, we concentrated on masking.

Generally, circuit interfaces contain control signals, which control the data flow. An example is the write-enable signal on a memory interface. As long as the write-enable signal is not activated, no data will be transferred to the neighbor circuit. So, if the circuit interface includes control signals, further resources can be saved by only masking the control signals like write- and read-enable as shown in figure 6.3.

An example implementation of the circuit isolation with logical masking of the control signals is shown in figure 6.4. As long as the reset signal is active and the circuit is being recovered, the output control signals stay masked.

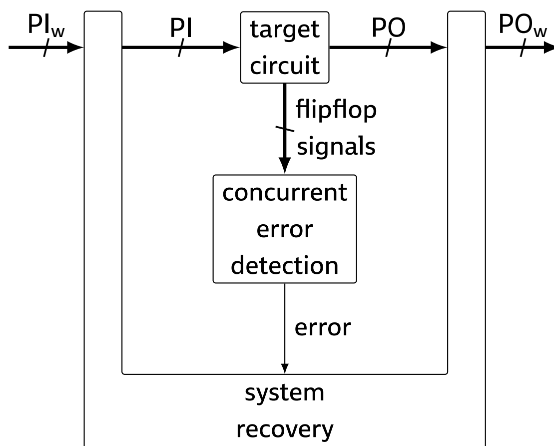

Figure 6.2: Overview of target circuit's system recovery components. PI and PO stand for primary-input and -output of the target circuit, respectively. The subscript $_w$ stands for *wrapped*. Error handling module wraps the target circuit for recovering and isolating the circuit.

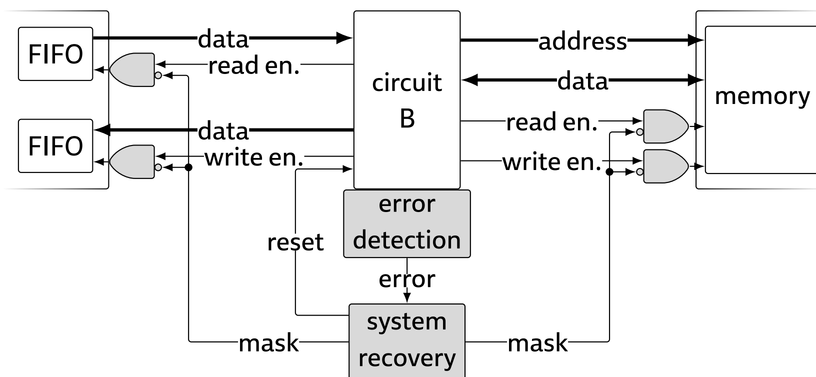

Figure 6.3: Detection and recovery applied on the reference processing architecture from section 1.1. To save resources, only the control signals are masked. (Figure 1.5 reused.)

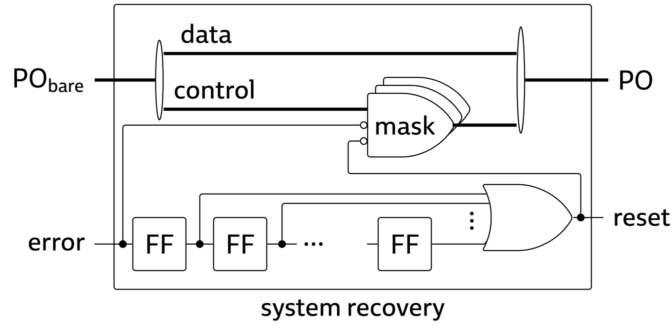

Figure 6.4: Example implementation of the system recovery for the target circuit. When the error signal is active then the control signals are masked to isolate the circuit logically in the same cycle. In subsequent cycles, the asynchronous reset signal is hold active and the circuit is reset.

6.1.2 Circuit reset

After an error, the circuit can be in an indeterminable state. The aim of error handling is to put the target circuit to a determined state that data processing can continue, e.g., to the state at the end of the last successful transaction.

The easiest recovery approach is to put the circuit to the start state by activating the reset signal, which is feasible if the target circuit always lands on the start state after one transaction is processed. We chose this approach for our evaluation. In opposite cases, where the circuit is at different states at the beginning of a transaction, the recovery must concurrently observe the state of the circuit and recover the circuit to the respective start state.

In figure 6.4, we show an example of an asynchronous reset-based recovery approach. A shift register enables a reset that is active many clock cycles. The number of flipflops in the shift register must be chosen such that all flipflops in the circuit are guaranteed to be reset after the respective number of clock cycles. Consequently, the size of the shift register s_{SR} is dependent on the longest reset path to a flipflop (critical reset path $t_{\text{crit,rst}}$), the circuit clock period t_{clk}, and can be calculated by the following equation:

$$s_{\text{SR}} = \left\lceil \frac{t_{\text{crit,rst}}}{t_{\text{clk}}} \right\rceil \tag{6.1}$$

It is obvious that the circuit reset is a sequential circuit as well and susceptible to soft errors. Consequently, it must be sufficiently protected due to following threats:

- During the initial state of the shift register, one or more bitflips in the latter flipflops of the shift register activates the reset of the target circuit, which results in a shorter reset duration.
- During the active state of the shift register, in other words, when the circuit

is being reset, one or more bitflips can interrupt the reset of the circuit. In this case, a bitflip would cause a short reset, which has a low probability due to the relatively small area of the shift register compared to the target circuit.

In both cases not all flipflops in the circuit are initialized and this can cause a circuit failure. Consequently, the reset circuit should be hardened by LTMR.

6.2 Transaction-based processing

Many sequential circuits process data by receiving a request and transmitting a response. Sending a response is not only important for the flow control but also for detecting an error. Due to the fact that we allow errors in the target circuit that cannot be corrected immediately, the system environment which utilizes the target circuit can use transaction-based processing. This gives the environment the opportunity to repeat the last processing request (i.e., resend the last packet) after a timeout, if the target circuit cannot send any response due to a recovery event. Therefore, a system implementing EDFT should incorporate a transaction-based processing scheme.

In this section, we first present the concept and provide a specification in subsection 6.2.1. Using this specification, we show that the system will not fail under the fault model that we presume in subsection 6.2.2.

6.2.1 Concept

In our previous example in section 1.1, we have proposed a communication protocol based on transactions, which is re-shown in figure 6.5. In this 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 like LTMR, which mostly occurs in every clock cycle ensuring that an error does not cause data corruption, a bit error in EDFT can lead to data corruption and hence to an unexpected loss of processing context in a system, in which this circuit is incorporated. 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.

A *data processing circuit* (cf. circuit B shown in figure 1.1) 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 figure 1.1.

A buffer stores one or many data words. These words can be used in two ways:

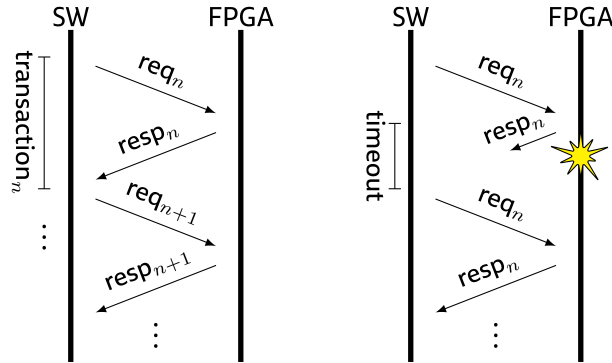

Figure 6.5: Sequence diagram of the communication protocol of our reference system in section 1.1, which is based on transactions. A *transaction* consists of a request (*req*) and a response (*resp*). The left diagram shows a normal sequence: every request is followed by a response. On the right, the error behavior is visualized: if still no response after a timeout is received, the last transaction is repeated.

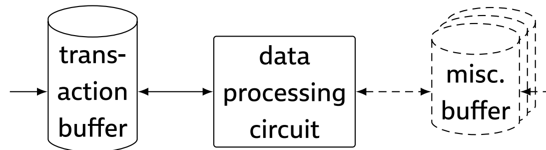

Figure 6.6: Data processing circuit receives a request from the transaction buffer and writes the response after processing. For communicating with other circuits, miscellaneous buffers are used.

- as a memory-mapped communication interface, e.g., writing a specific word to a specific address starts sending UART bits to a peer. A memory location holding such a word is called an *action triggering* address. When the communication is completed, a status word indicates if a communication was successful or has failed.

- as an input or output for data processing, e.g., a checksum circuit reads the input words, processes the checksum and writes checksum words back to the buffer. We call a memory location holding such a word a *passive* address.

A *transaction buffer* (cf. the FIFOs in figure 1.1) 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 figure 1.1 and they are called *miscellaneous buffer*s. This generalized view on the data processing circuit is visualized in figure 6.6. 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 figure 1.1) and the sent

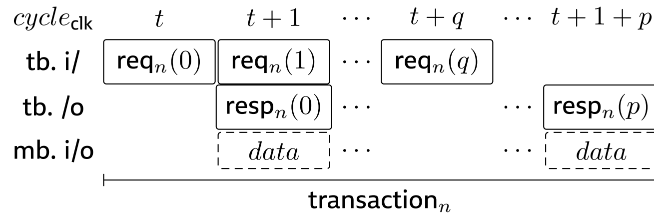

Figure 6.7: An example transaction visualized on cycle level. The processing circuit (cf. figure 6.6) processes request words and writes the response words back to the transaction buffer (tb.). During one clock cycle ($cycle_{clk}$), one request word (req$_n(i)$) of a request (req$_n$) or one response word (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.

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 figure 6.7.

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*.

6.2.2 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 durina a clock cycle, then a bitflip in a cluster will be observable in the next clock cycle. EDFT 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 EDFT:

- 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. To avoid this, the master must pay attention to how the the requests to the slave are built. Although the actual solution is application dependent, in the following we provide an example approach how the requests can be built.

If an incomplete or no response is received by the master in the timeout window, then a recovery procedure by the master is initiated.

If an error happens during processing of a transaction involving only passive addresses, then the simplest approach is to retry the last request. The reason is:

- after a read request, the state of the miscellaneous buffers do not change
- after a write request, part of the miscellaneous buffers may change, but this does not trigger an action. Consequently, the last request can be retried.

In both cases also a readback with partial write request can be issued, if this is less time-consuming than retrying.

In case of action triggering addresses, if an error happens during processing of a read transaction, then the simplest approach is the retry of the last request similar to the above reason. If this is a write action (e.g., triggering a data transmission to a subsystem), then retrying retriggers the last operation, which can be undesirable and dangerous. In this case, first a read request to the status register of the respective memory-mapped interface should be issued to see if the triggered action succeeded or not. Then, a write request can be issued accordingly. It is obvious that not only the design but also the correct use of the communication protocol is important for the fault tolerance of the system.

6.3 Experimental evaluation

In this section, we evaluate the system impacts of system recovery and transaction-based processing. System recovery is evaluated in subsections 6.3.1 and 6.3.2 by its area and timing impacts similar in the last two chapters 4 and 5. Transaction-based processing is based on recomputation and thus evaluated by its processing time impact in subsection 6.3.3.

The evaluation of the impacts of transaction-based processing on the hardware and software components is not straightforward, because the transaction handling depends on the communication protocol between the user and provider component, which can be arbitrary.

If we assume a bidirectional protocol where the provider component has to reply every request from the user component. In this case, the protocol must be able to retransmit the last request to the provider component, if there is no response or the response is negative. In terms of programming resources, this means that the user program should buffer every request until there is a positive response to the last request.

Transaction-based processing is not possible on a protocol where the user component sends a request to the provider component and assumes that the request will be processed correctly. This applies to protocols, in which not every request is responded, and also to unidirectional protocols. In this case, the protocol must be changed to support responses from the provider component, which can result in redesigning both user and provider component.

For the evaluation of the system recovery circuit, we took the pipelined PBED as error detection component and input the error signal to the system recovery circuit. The system recovery circuit was hardened using LTMR and for the reset circuit, we chose the reset duration of three cycles for all circuits. As we discussed in section 6.1.1, only control outputs should be masked to save area. In the FSM circuits, we indeed masked the control signal outputs. In the I99T circuits, we masked all the output signals, because there is no detailed documentation available about the semantics of the primary output signals.

6.3.1 FSM

Table 6.1 shows the area differences (diff.) of the PBED with system recovery component compared to the pipelined PBED approach without system recovery for the cluster size of three. The reset duration of three cycles was realized by a two bit counter by the synthesizer. This counter triplicated results in six flipflops and in the evaluation of PBED, we used one flipflop as a placeholder for the system recovery circuit to include the timing effects of the parity check and OR-tree. Consequently, all circuits have a flipflop difference of five.

The combinational area difference is not directly presented on table 6.1, but can be obtained by subtracting flipflop count from the area ($A - c_{\mathsf{FF}}$). Combinational area difference is caused by majority voters for the triplicated flipflops, gates for masking of the control signals, and the counter. The area difference fluctuates probably due to optimizations.

System recovery component causes 13 to 25 CLBs for all circuits, therefore the impact is significant for circuits with relatively small area. We additionally calculated the area overhead (bare circuit area as reference) per application flipflop and the area overhead ratio to LTMR approach. In average, the overhead per application flipflop is about 2.5 and EDFT approach saves in average 30% of the overhead that would be caused by LTMR for cluster size of 3.

The critical path difference to pipelined PBED is less than 2 ns and is about 1 ns in average. The critical path difference is visualized in figure 6.8. In worst case, a critical path overhead of about 5 ns is caused, which is about 2.5 ns more than LTMR.

System recovery does not change the impact of cluster size variation, therefore we only summarize the minimum and maximum values for the derived parameters in table 6.2. We see that EDFT can save up to 54% of the area overhead caused by LTMR with a maximum critical path overhead of 6 ns.

Table 6.1: Synthesis results for the FSM circuits hardened by pipelined PBED with cluster size of 3 and system recovery (abbreviated as PPR meaning pipelined PBED with recovery). Absolute (abs.) values as well as differences (diff.) to the version without system recovery from chapter 5 (was abbreviated as PP) are shown. A positive difference means a higher value for the version with system recovery. The last row shows the average values for the differences.

	c_{FF}		A		t_{crit} (ns)			
circ.	abs.	diff.	abs.	diff.	abs.	diff.	$\frac{A_{+,\mathsf{PPR}}}{c_{\mathsf{FF,ba}}}$	$\frac{A_{+,\mathsf{PPR}}}{A_{+,\mathsf{LT}}}$
1	46	5	205	14	9.46	0.17	2.44	0.82
2	82	5	403	14	9.94	0.30	2.12	0.72
3	117	5	598	13	10.03	0.01	2.63	0.73
4	153	5	800	17	10.31	0.03	2.62	0.74
5	189	5	987	17	10.29	-0.02	2.59	0.72
6	225	5	1204	18	10.59	0.14	2.57	0.70
7	261	5	1403	14	11.09	0.59	2.53	0.70
8	297	5	1559	15	11.30	0.69	2.53	0.70
9	333	5	1745	14	11.82	1.04	2.52	0.69
10	369	5	1935	14	11.79	0.78	2.52	0.69
11	405	5	2162	20	11.84	0.66	2.52	0.70
12	442	5	2385	20	11.82	0.67	2.50	0.68
13	480	5	2620	17	12.32	0.83	2.51	0.69
14	516	5	2815	15	12.62	1.14	2.51	0.70
15	552	5	3006	17	12.80	1.16	2.50	0.69
16	588	5	3280	16	12.62	1.01	2.50	0.70
17	624	5	3407	21	12.75	0.97	2.50	0.69
18	660	5	3629	16	13.12	1.27	2.49	0.70
19	694	5	3777	16	13.14	1.31	2.49	0.68
20	730	5	3985	15	13.16	1.16	2.49	0.67
21	768	5	4184	16	13.49	1.35	2.49	0.68
22	804	5	4409	14	13.49	1.23	2.49	0.68
23	840	5	4624	15	13.42	1.00	2.48	0.68
24	878	5	4856	14	13.69	1.08	2.48	0.69
25	913	5	5038	17	13.45	0.97	2.48	0.68
26	949	5	5268	24	13.58	0.95	2.48	0.68
27	985	5	5490	18	13.73	1.19	2.49	0.69
28	1020	5	5640	21	13.88	0.96	2.48	0.68
29	1057	5	5867	17	14.26	1.26	2.48	0.68
30	1092	5	6076	25	14.07	1.00	2.48	0.69
31	1128	5	6253	17	-	-	2.48	0.68
	avg.	5.00		18.57		0.83	2.49	0.70

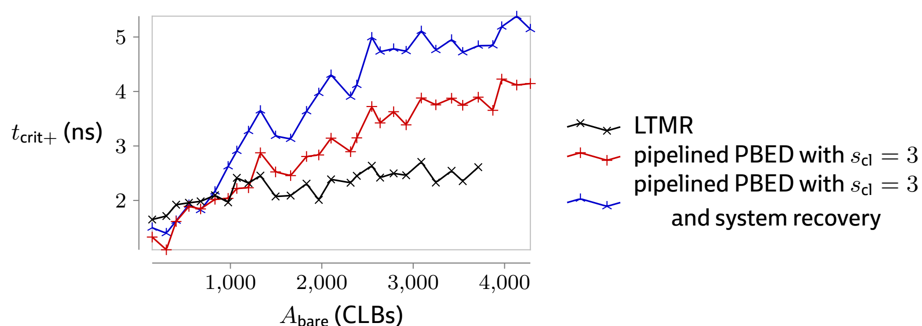

Figure 6.8: Critical path delay overheads ($t_{\text{crit}+}$) over bare circuit area (A_{bare}) for LTMR and both PBED techniques with cluster size of 3. Points for the LTMR-hardened circuits for $A_{\text{bare}} > 3750$ do not exist, because they did not fit into the FPGA.

Table 6.2: Minimum and maximum values for derived parameters for PBED hardened FSM circuits with recovery for various cluster sizes (s_{cl})

s_{cl}	$t_{\text{crit}+}$ (ns)	$\frac{A_+}{c_{\text{FF,ba}}}$	$\frac{A_{+,\text{PB}}}{A_{+,\text{LT}}}$
2	0.06 - 5.30	2.31 - 3.14	0.78 - 0.96
3	1.10 - 5.38	1.84 - 2.63	0.62 - 0.82
4	1.23 - 5.45	1.55 - 2.37	0.52 - 0.76
5	1.41 - 5.24	1.49 - 2.27	0.50 - 0.72
6	1.70 - 5.35	1.51 - 2.30	0.51 - 0.69
7	1.90 - 5.55	1.39 - 2.21	0.47 - 0.68
8	2.33 - 5.60	1.41 - 2.11	0.48 - 0.68
9	2.25 - 5.72	1.39 - 2.18	0.47 - 0.68

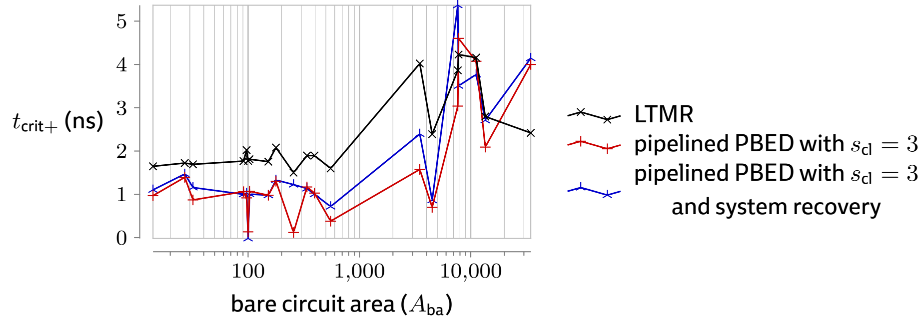

Figure 6.9: Critical path delay overheads (t_{crit+}) over bare circuit area (A_{bare}) for LTMR and both PBED techniques with cluster size of 3

6.3.2 I99T circuits

The structure of data presented in this subsection (table 6.3, 6.4, figure 6.9) is similar to the last subsection 6.3.1.

Table 6.3 shows that the flipflop difference is nearly constant at 5 as in the results of the FSM circuit. A significant fluctuation can be observed in the area difference compared to the FSM circuit. The reason is that every circuit has a different number of primary output nets, and we did not only mask the control signals in the primary output nets, but all primary output nets. For instance, b17 has 98 primary output nets, which creates a relatively high area overhead due to needed masking gates. Still, EDFT can save 27% of the area overhead caused by the LTMR in b17. On the other hand, EDFT can also cause more overhead than LTMR like in relatively small circuits like b06, which has an area of 64 CLBs. The circuits b02 and b05 also cause more area overhead than LTMR, and the reason for these circuits was explained in subsection 4.3.2.

The critical path impact can be better analyzed using figure 6.9, which also plots LTMR and pipelined PBED without system recovery critical path overheads. In most cases, PBED with and without system recovery have similar critical path overhead and the EDFT causes less critical path overhead than LTMR. Only b18 and b20 result in a longer critical path than LTMR and the critical path difference stays below 2 ns in these cases.

Similar to subsection 6.3.1, we only summarize the minimum and maximum values for the derived parameters and for different cluster sizes in table 6.4. Compared to the minimum and maximum values without recovery in table 5.4, the upper bounds for the critical path overheads are not changed for most cluster sizes, and if changed, the increase is below 1 ns. We see that EDFT can save up to 54% of the area overhead caused by LTMR.

Table 6.3: Synthesis results for I99T circuits hardened by pipelined PBED with cluster size of 3 and system recovery. For the remaining parameters confer to table 6.1.

	c_{FF}		A		t_{crit} (ns)			
circ.	abs.	diff.	abs.	diff.	abs.	diff.	$\frac{A_{+,PPR}}{c_{FF,ba}}$	$\frac{A_{+,PPR}}{A_{+,LT}}$
b02	14	5	35	12	5.69	0.13	5.25	1.75
b01	24	5	62	13	5.92	0.08	3.50	1.13
b06	20	5	64	16	6.66	0.29	4.00	1.33
b08	40	5	147	14	11.49	-0.06	2.67	0.90
b03	60	5	175	15	9.23	0.11	2.52	0.89
b09	60	5	173	12	9.64	-0.15	2.61	0.89
b10	45	5	168	17	7.76	-0.06	2.71	0.90
b13	95	5	296	19	9.12	0.00	2.57	0.80
b07	74	5	277	19	14.90	0.03	2.25	0.76
b11	63	5	341	19	18.35	1.11	2.43	0.83
b04	109	5	536	20	24.94	-0.04	3.00	0.79
b05	70	5	507	37	25.59	-0.03	2.73	1.18
b12	194	5	899	18	17.01	0.34	2.85	0.77
b14	333	5	4131	69	49.87	0.82	3.00	0.78
b15	669	5	5633	84	34.56	0.16	2.59	0.76
b20	663	5	8841	33	51.06	2.33	2.74	0.73
b21	658	5	8962	32	48.07	-1.10	2.76	0.72
b22	945	5	12858	31	48.96	-0.30	2.70	0.72
b17	2104	5	16982	115	36.93	0.62	2.51	0.73
b18	4877	5	42536	41	50.84	0.14	2.47	0.73
b19	9654	5	76923	59	-	-	2.47	0.72
	avg.	5.00		33.10		0.22	2.87	0.90

Table 6.4: Minimum and maximum values for derived parameters for PBED hardened I99T circuits with recovery for various cluster sizes (s_{cl})

s_{cl}	t_{crit+} (ns)	$\frac{A_{+}}{c_{FF,ba}}$	$\frac{A_{+,PB}}{A_{+,LT}}$
2	-0.44 - 2.29	2.31 - 5.75	0.79 - 1.92
3	-0.02 - 5.37	1.82 - 5.25	0.62 - 1.75
4	-0.01 - 14.12	1.56 - 5.25	0.55 - 1.75
5	-0.08 - 11.87	1.46 - 5.25	0.50 - 1.75
6	0.52 - 7.14	1.50 - 5.25	0.51 - 1.75
7	0.51 - 6.08	1.54 - 5.25	0.52 - 1.75
8	0.46 - 7.04	1.41 - 5.25	0.48 - 1.75
9	0.33 - 7.99	1.36 - 5.25	0.46 - 1.75

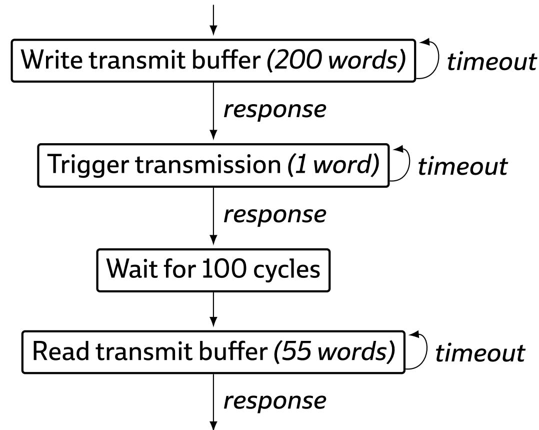

Figure 6.10: Simplified flow diagram of one single memory access block. It consists of three transactions. The transactions are retried by the software if there is no response after the timeout has passed.

6.3.3 Processing time penalty

Compared to local error handling of LTMR, EDFT handles an error by recomputation. This imposes a processing time penalty on the system. To compare the runtime performance of LTMR and EDFT 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 figure 6.10. 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 flipflops in the target circuit and flips the flipflop bits according to the given probability p randomly. Probability p is defined as the bitflip probability per clock cycle for a single flipflop. 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 EDFT requires that the error is corrected by the software by repeating the failed memory access request, which in turn causes additional processing delays. Figure 6.11 shows relative processing time needed by EDFT for the given mission. The processing time of EDFT is plotted relative to the LTMR processing time, which is constant. For EDFT, the processing time increases with increasing bitflip probability p, as a failed mem-

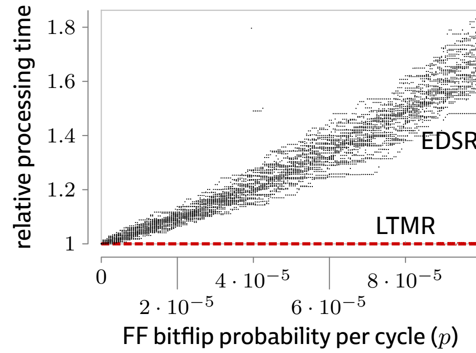

Figure 6.11: Scatterplot of relative processing time for a given mission. The factor is relative to the processing time of LTMR.

ory 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 the L2 orbit (second Lagrangian point, about 1.5 million km away from earth) under 1/cm^2 shielding, a programmed circuit with 5000 flipflops on a ProASIC RTPE3000L FPGA has four SEUs [BSV11, ch. 7]. 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:

$$\begin{aligned} p &= 4/5000/365/24/60/60/(20 \times 10^6) \\ &\approx 1.3 \times 10^{-18} \end{aligned} \tag{6.2}$$

Assuming the error rate from eq. 6.2 and transactions with a maximum length of 10^3 cycles, make the time penalty per year insignificant.

6.4 Automatic application

6.4.1 Logical masking of control signals

Logical masking of control signals can be easily implemented, if the control signals of the target circuit primary output are known. The synthesizable VHDL code in listing 6.1 describes a combinational circuit that masks the input signals, if the error signal is active.

Listing 6.1: Example circuit for masking of the control signals

```
entity signal_masker is
generic (
  -- This generic determines how the signal is deactivated
  -- in case 'signal_must_be_masked' signal is active.
  -- If false, the input control nets are active-low.
  SIGNAL_IS_ACTIVE_HIGH : boolean
);
port (
  -- Input --
  error                   : bit;
  reset_circuit_is_active : boolean;
  signal_input            : bit_vector;

  -- Output --
  signal_masked : out bit_vector
);
end entity;

architecture arch of signal_masker is
  signal signal_must_be_masked : boolean;
begin
  signal_must_be_masked <=
    true when reset_circuit_is_active or error = '1'
    else false;

  signal_masked <=
    (signal_input'range => '0')
      when signal_must_be_masked
           and     SIGNAL_IS_ACTIVE_HIGH else
    (signal_input'range => '1')
      when signal_must_be_masked
           and not SIGNAL_IS_ACTIVE_HIGH else
    signal_input;
end architecture;
```

6.4.2 Reset circuit

The reset circuit can also be automatically implemented by obtaining the critical reset path after placing and routing the target circuit and using the equation 6.1. The obtained shift register size equals to the reset duration of the target circuit. The synthesizable VHDL code in listing 6.2 describes a circuit which holds the reset signal active as long as the error signal is active.

Note that the reset signal `rst` is not immediately activated if the error signal is active. By doing so, a combinational loop would be created that can spuriously reset the circuit during the time window when the combinational signals settle before they get registered by the flipflops.

Listing 6.2: Example circuit for asynchronously resetting the target circuit

```
entity reset_circuit is
generic (
  RESET_DURATION: positive := 3;
  RESET_SIGNAL_IS_ACTIVE_HIGH : boolean
);
port (
  -- Input --
  clk, rst                   : bit;
  error                      : bit;
  target_circuit_rst_input : bit;

  -- Output --
  target_circuit_rst_wrapped : out bit;
  reset_circuit_is_active    : out boolean
);
end entity;

architecture arch of reset_circuit is
  signal counter: natural range 0 to RESET_DURATION;

  -- This circuit should be hardened by LTMR
begin
  counter_behavior: process (clk, rst)
  begin
    if rst then
      counter <= 0;
    elsif rising_edge(clk) then
      if counter = RESET_DURATION then
        counter <= 0;
      elsif error = '1' or counter > 0 then
        counter <= counter +1;
      end if;
    end if;
  end process;

  reset_circuit_is_active <=
    true when counter > 0
    else false;

  target_circuit_rst_wrapped <=
    '1' when reset_circuit_is_active
             and RESET_SIGNAL_IS_ACTIVE_HIGH else
    '0' when reset_circuit_is_active
             and not RESET_SIGNAL_IS_ACTIVE_HIGH else
    target_circuit_rst_input;
end architecture;
```

Chapter 7

Conclusion

High energy particles can cause bitflips on terrestrial and aerospace electronics. LTMR is often used as the straightforward approach to harden the flipflops of a sequential digital circuit for mission-critical applications, but LTMR incurs significant area and power overhead.

Nowadays, many applications are implemented on complex systems, which consist of many components. In such a system, it is advisable to implement a fault tolerance approach which exploits already available redundancy on more flexible components and reduces the fault tolerance overhead in scarce and costly components. With this motivation, we proposed an error detection-based approach with recomputation. To make a comparison with LTMR possible, we chose parity-based error detection (PBED) as the error detection approach.

We started evaluating our approach by showing the limits of the PBED approach on the ProASIC3 architecture analytically. The analytical comparison revealed that 60% of the area overhead that would be caused by LTMR can be saved by PBED for cluster size of 3. Additionally, we discussed the two critical path candidates for PBED, which are the parity generation path and error signal generation path.

In experiments, we found out that the particular application can significantly affect the overhead of PBED and LTMR. The overhead of both approaches was significantly dependent on the enable flipflops present in the original user circuit, because these flipflops have to be converted to a D-flipflop with a multiplexer. In larger circuits, we observed an increasing critical path and attenuated this effect by pipelining the error signal reduction. This is not a traditional pipelining and is based on sequential distance of flipflops to the output of the circuit to avoid adding additional pipelining register on the primary output of the circuit. In I99T circuits, pipelined-PBED can achieve up to 1 ns critical path saving compared to the direct-PBED approach with small area overhead. We observed that most of the flipflops in the circuits we analyzed have a sequential distance of 1, so pipelined-PBED cannot save significant critical path.

In a fault-tolerant system, error detection must be used with system recovery.

As system recovery on the target circuit, we used a circuit isolation approach by immediate masking of the primary outputs, and if available, only the control signals. During the isolated time, the circuit can be reinitialized using an approach, which can take multiple clock cycles.

We presume that a communication protocol will exist in a processing architecture with a user and provider system. System recovery on the user side is done by transaction-based processing. We specified this approach and discussed the points that will make the protocol between the user and provider fault-tolerant, and finally carried out a fault tolerance analysis based on our fault model.

All in all, we see that our end-to-end approach can achieve timing results better than LTMR in experiments. Our approach can save up to 54% of the area overhead that would be caused by LTMR and can achieve better timing than LTMR in most circuits.

Generally, in smaller circuits the area overhead factor can rise above 3, and LTMR is recommended for such circuits. But also a mid-sized circuit, where flipflops with high fanouts exist, can also cause more area overhead than LTMR, as these flipflops must be replicated in PBED. LTMR already triplicates every flipflop and no additional replication is needed for high-fanout flipflops. This underlines the application dependence of our approach's cost. Still, the less area overhead compared to LTMR may be the key to adopt sufficient functionality in a single chip.

We proposed error detection–based fault tolerance as an alternative to LTMR. As LTMR is an intrinsic error detecting and correcting technique, a comparison to an error detection–based technique is not straightforward. To achieve an accurate comparison, we have shown an error detection–based fault tolerance concept including recovery and transaction-based processing and implemented it on a known FPGA for space applications, which allowed us to achieve an accurate comparison of the timing and area resources. Moreover, we introduced pipelining for the error signal generation, which enables better timing.

This work provides a basis for future fault-tolerant data processing architectures that consist of massively parallel processing cores like a modern graphics processing unit. On such an architecture, it can be sufficient to implement an error detection–based technique on the processing cores. For processing, a job is divided into sub-jobs which can be processed in parallel. If a core fails to process a sub-job, then the processing request is repeated. Additionally, if a core is found out to have a permanent error, then it can be marked unusable.

We laid the foundations to enable area-efficient data processing for dependable spaceborne computing. Through our work, future on-board computers may provide higher computing performance.

Acknowledgments and Statutory Declaration

Acknowledgments

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

Many of the ideas leading to this work were developed in collaboration with my advisor Görschwin Fey, whom I express my utmost gratitude. I also thank to Alberto Garcia-Ortiz for fruitful discussions during my research. The typographically beautiful cover was designed by Alexandra Cor.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Bibliography

[And+03] H. Ando et al. “A 1.3-GHz fifth-generation SPARC64 microprocessor”. In: *IEEE Journal of Solid-State Circuits* 38.11 (2003), pp. 1896–1905. ISSN: 0018-9200. DOI: 10.1109/jssc.2003.818146.

[Arm61] Douglas B. Armstrong. “A general method of applying error correction to synchronous digital systems”. In: *The Bell System Technical Journal* 40.2 (Mar. 1961), pp. 577–593. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1961.tb01630.x.

[Atme15a] *ATF280F FPGA Datasheet*. Atmel. Nov. 2015. URL: http://www.atmel.com/Images/Atmel-7750-Rad-Hard-Reprogrammable-FPGA-ATF280F_Datesheet.pdf.

[Atme15b] *ATFEE560 datasheet*. Atmel. Sept. 2015. URL: http://www.atmel.com/Images/41041-ATFEE560_Datasheet.pdf.

[Avi+04] Algirdas Avižienis et al. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[Ayd16] Gökçe Aydos. *Parity-based Error Det. Tool for the ProASIC3 FPGA Architecture*. 2016. URL: https://gitlab.informatik.uni-bremen.de/goekce/pbed.

[AF15a] Gökçe Aydos and Görschwin Fey. “Empirical Results on Parity-based Soft Error Detection with Software-based Retry”. In: *Nordic Circuits and Systems Conference (NORCAS)*. IEEE, Oct. 2015. DOI: 10.1109/NORCHIP.2015.7364378. URL: http://www.cs.uni-bremen.de/agra/doc/konf/aydos2015ltmr_vs_pbed_exper.pdf.

[AF16a] — “Empirical Results on Parity-based Soft Error Detection with Software-based Retry”. In: *Microprocessors and Microsystems (MICPRO)* (Sept. 2016). DOI: 10.1016/j.micpro.2016.09.009.

[AF16b] — “Exploiting Error Det. Latency for Parity-based Soft Error Detection”. In: *Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, Apr. 2016. DOI: 10/bsf9. URL: http://www.cs.uni-bremen.de/agra/doc/konf/aydos2016exploiting_err_det_latency.pdf.

[AF15b] Gökçe Aydos and Görschwin Fey. "In-circuit Error Detection with SW-based Error Correction – An Alternative to TMR". In: *Formal Modeling and Verification of Cyber-Physical Systems*. Springer Fachmedien Wiesbaden, 2015, pp. 272–274. DOI: 10.1007/978-3-658-09994-7_10.

[AF15c] — "Parity-based Soft Error Detection with Software-based Retry vs. Triplication-based Soft Error Correction - An Analytical Comparison on a Flash-based FPGA Architecture". In: *INFORMATIK 2015*. Ed. by Douglas Cunningham et al. GI e.V. Sept. 2015, pp. 1415–1429. URL: http://subs.emis.de/LNI/Proceedings/Proceedings246/1415.pdf.

[Aza+11] José Rodrigo Azambuja et al. "Exploring the Limitations of Software-based Techniques in SEE Fault Coverage". In: *J Electron Test* 27.4 (Apr. 2011), pp. 541–550. DOI: 10.1007/s10836-011-5218-7.

[BSV11] Niccolò Battezzati, Luca Sterpone, and Massimo Violante. *Reconfigurable Field Programmable Gate Arrays for Mission-Critical Applications*. Springer, 2011. DOI: 10.1007/978-1-4419-7595-9.

[Ber08] Melanie Berg. "Design for Radiation Effects". Presentation from Military and Aerospace Programmable Logic Devices (MAPLD) Workshop. 2008.

[Ber12] — "Field Programmable Gate Arrays". In: *Industrial Electronics : Extreme Environment Electronics (1)*. Ed. by John D. Cressler and H. Alan Mantooth. CRC Press, 2012. Chap. 56.

[Ber61] J.M. Berger. "A note on error detection codes for asymmetric channels". In: *Information and Control* 4.1 (Mar. 1961), pp. 68–73. DOI: 10.1016/s0019-9958(61)80037-5.

[Bla12] Jeffrey D. Black. "Best Practices in Radiation Hardening by Design: CMOS". In: *Industrial Electronics : Extreme Environment Electronics (1)*. Ed. by John D. Cressler and H. Alan Mantooth. CRC Press, 2012. Chap. 43.

[CNV96] Teodor Calin, Michael Nicolaidis, and Raoul Velazco. "Upset hardened memory design for submicron CMOS technology". In: *IEEE Transactions on Nuclear Science* 43.6 (Dec. 1996), pp. 2874–2878. ISSN: 0018-9499. DOI: 10.1109/23.556880.

[Che+16a] Eric Cheng et al. "CLEAR: Cross-Layer Exploration for Architecting Resilience". In: *Proceedings of the 53rd Annual Design Automation Conference on - DAC'16*. Association for Computing Machinery (ACM), 2016. DOI: 10.1145/2897937.2897996.

[Che+16b] — *CLEAR: Cross-Layer Exploration for Arch. Resilience - Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores*. Version 2. June 23, 2016. arXiv: 1604.03062v2 [cs.AR].

[Col04] Jean-Pierre Colinge. *Silicon-on-Insulator Technology: Materials to VLSI*. 3rd. Springer US, 2004. DOI: 10.1007/978-1-4419-9106-5.

[CPB10] Philippa M. Conmy, Clive Pygott, and Iain Bate. "VHDL guidance for safe and certifiable FPGA design". In: *System Safety 2010, 5th IET International Conference on*. Oct. 2010, pp. 1–6. DOI: 10.1049/cp.2010.0832.

[CRS00] Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. "RT-level ITC'99 benchmarks and first ATPG results". In: *IEEE Design Test of Computers* 17.3 (July 2000), pp. 44–53. ISSN: 0740-7475. DOI: 10.1109/54.867894.

[GSZ09] Balkaran Gill, Norbert Seifert, and V. Zia. "Comparison of alpha particle and neutron-induced combinational and sequential logic error rates at the 32nm technology node". In: *2009 IEEE International Reliability Physics Symposium*. Institute of Electrical and Electronics Engineers (IEEE), Apr. 2009, pp. 199–205. DOI: 10.1109/irps.2009.5173251.

[HA84] Kuang-Hua Huang and J. A. Abraham. "Algorithm-Based Fault Tolerance for Matrix Operations". In: *IEEE Transactions on Computers* C-33.6 (June 1984), pp. 518–528. ISSN: 0018-9340. DOI: 10.1109/tc.1984.1676475.

[CADP16] *I99T benchmarks*. 2016. URL: http://www.cad.polito.it/downloads/tools/itc99.html (visited on 10/10/2016).

[Iro+03] Farokh Irom et al. "Single-event upset in evolving commercial silicon-on-insulator microprocessor technologies". In: *IEEE Transactions on Nuclear Science* 50.6 (Dec. 2003), pp. 2107–2112. ISSN: 0018-9499. DOI: 10.1109/TNS.2003.821820.

[KCR06] Fernanda Lima Kastensmidt, Luigi Carro, and Ricardo Reis. *Fault-Tolerance Techniques for SRAM-based FPGAs*. Springer, 2006.

[Kel+10] Lee Hsiao-Heng Kelin et al. "LEAP: Layout Design thr. Error-Aware Transistor Positioning for soft-error resilient sequential cell design". In: *2010 IEEE International Reliability Physics Symposium*. Institute of Electrical and Electronics Engineers (IEEE), May 2010, pp. 203–212. DOI: 10.1109/irps.2010.5488829.

[Lid+94] Peter Lidén et al. "On latching probability of particle induced transients in combinational networks". In: *24th International Symposium on Fault-Tolerant Computing (FTCS)*. June 1994, pp. 340–349. DOI: 10.1109/FTCS.1994.315626.

[LV62] Robert E. Lyons and Wouter Vanderkulk. "The Use of Triple-Modular Redundancy to Improve Computer Reliability". In: *IBM Journal of Research and Development* 6.2 (Apr. 1962), pp. 200–209. ISSN: 0018-8646. DOI: 10.1147/rd.62.0200.

[MNV12] Meena Mahajan, Prajakta Nimbhorkar, and Kasturi Varadarajan. "The Planar k-means Problem is NP-hard". In: *Theoretical Computer Science* 442 (July 2012), pp. 13–21. DOI: 10.1016/j.tcs.2010.05.034.

[MBS08] Albert Meixner, Michael E. Bauer, and Daniel J. Sorin. "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores". In: *IEEE Micro* 28.1 (Jan. 2008), pp. 52–59. ISSN: 0272-1732. DOI: 10.1109/mm.2008.3.

[EEJo12] *Microsemi Announces New Package for Radiation Tolerant Space Flight FPGAs. RT ProASIC3 Offered in Robust Ceramic Package*. Feb. 10, 2012. URL: http://www.eejournal.com/archives/news/20120210-06 (visited on 10/10/2016).

[Mor05] Kevin Morris. *Flash News Flash. Actel Unveils ProASIC3*. Jan. 25, 2005. URL: http://www.eejournal.com/archives/articles/20050125_flash (visited on 10/10/2016).

[Nic11] Michael Nicolaidis, ed. *Soft errors in modern electronic systems*. Vol. 41. Frontiers in Electronic Testing. Springer Science + Business Media, 2011. DOI: 10.1007/978-1-4419-6993-4.

[NZ98] Michael Nicolaidis and Yervant Zorian. "On-Line Testing for VLSI - A Compendium of Approaches". In: *Journal of Electronic Testing Theory and Applications (JETTA)* 12 (Feb. 1998), pp. 7–20. DOI: 10.1023/A:1008244815697.

[OSM02] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. "Error detection by duplicated instructions in super-scalar processors". In: *IEEE Transactions on Reliability* 51.1 (Mar. 2002), pp. 63–75. DOI: 10.1109/24.994913.

[Pet11] Edward Petersen. *Single Event Effects in Aerospace*. John Wiley & Sons, 2011. DOI: 10.1002/9781118084328.

[PGG11] Christian Poivey, M. Grandjean, and F. X. Guerre. "Radiation Characterization of Microsemi ProASIC3 Flash FPGA Family". In: *2011 IEEE Radiation Effects Data Workshop (REDW)*. July 2011, pp. 1–5. DOI: 10.1109/REDW.2010.6062510.

[Micr15a] *ProASIC3L FPGA Datasheet (DS0100)*. Version 14. Microsemi. June 2015. URL: http://www.microsemi.com/document-portal/doc_download/130702-ds0100-proasic3l-low-power-flash-fpgas-datasheet.

[Reb+99] Maurizio Rebaudengo et al. "Soft-error detection through software fault-tolerance techniques". In: *Proc. Int. Symp. Defect and Fault Tolerance in VLSI Systems DFT '99*. Institute of Electrical & Electronics Engineers (IEEE), Nov. 1999, pp. 210–218. DOI: 10.1109/dftvs.1999.802887.

[Rez10] Sana Rezgui. "New Reprogrammable and Non-Volatile Rad.-Tolerant FPGA: RT ProASIC3". In: *Aerospace Technologies Advancements*. Ed. by Thawar T. Arif. InTech, 2010. Chap. 6.

[Micr15b] *RTAX-S/SL and RTAX-DSP Radiation-Tolerant FPGAs datasheet*. Version 17. Microsemi. Feb. 2015. URL: http://www.microsemi.com/document-portal/doc_download/130713-rtax-s-sl-and-rtax-dsp-radiation-tolerant-fpgas-datasheet.

[Micr16] *RTG4 FPGA Datasheet (DS0131)*. Version 2. Microsemi. May 2016. URL: http://www.microsemi.com/document-portal/doc_download/135193-ds0131-rtg4-fpga-datasheet.

[Sei+12] Norbert Seifert et al. "Soft Error Susceptibilities of 22 nm Tri-Gate Devices". In: *IEEE Transactions on Nuclear Science* 59.6 (Dec. 2012), pp. 2666–2673. ISSN: 0018-9499. DOI: 10.1109/tns.2012.2218128.

[Sny16] Wilson Snyder. *Verilog-Perl distribution*. 2016. URL: http://www.veripool.org/projects/verilog-perl (visited on 10/10/2016).

[SG99] Lisa Spainhower and Thomas A. Gregg. "IBM S/390 Parallel Enterprise Server G5 fault tolerance: A historical perspective". In: *IBM Journal of Research and Development* 43.5.6 (Sept. 1999), pp. 863–873. DOI: 10.1147/rd.435.0863.

[Tre+14] Carl Johann Treudler et al. "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.

[Aero13] *UT6325 FPGA Datasheet*. Aeroflex. Nov. 2013. URL: http://ams.aeroflex.com/pagesproduct/datasheets/RadTolEclipseFPGA.pdf (visited on 10/10/2016).

[VSC15] Kosta Varnavas, William Herbert Sims, and Joseph 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)*. Ed. by Timothy Pham, Joseph C. Casas, and Claus-Peter Rückemann. 2015.

[Xili14] *Virtex-5QV Family Overview (DS192)*. Version 1.4. Xilinx. Nov. 12, 2014. URL: http://www.xilinx.com/support/documentation/data_sheets/ds192_V5QV_Device_Overview.pdf.

[Wan04] Jih-Jong Wang. *RTAX-S EDAC-RAM Single Event Upset Test Report*. June 2004. URL: http://www.microsemi.com/document-portal/doc_view/131377-rtax-s-see-data-for-the-edac-ram.