

“Stochastic Approaches for Speeding-Up the Analysis of the Propagation of Hardware-Induced Errors and Characterization of System-Level Mitigation Schemes in Digital Communication Systems”

Der Fakultät für Elektrotechnik und Informationstechnik der
Rheinisch-Westfälischen Technischen Hochschule Aachen vorgelegte Dissertation
zur Erlangung des akademischen Grades einer Doktorin der
Ingenieurwissenschaften

vorgelegt von

Diplom-Ingenieurin

Georgia Psychou, M.Sc.,

aus Tripoli, Griechenland

Berichter: Universitätsprofessor Dr.-Ing. Tobias G. Noll

Universitätsprofessor Dr.-Ing. Holger Blume

Universitätsprofessor Dr.-Ing. Tobias Gemmeke

Tag der mündlichen Prüfung: 29. September 2017

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online
verfügbar.

Acknowledgements

The author would like to acknowledge the support of former IMEC Netherlands researchers Jos Huisken and Dr.Tobias Gemmeke, that they provided, during their employment in IMEC Netherlands. During that time, there have been regular discussions on the progress and direction of the work and they offered valuable suggestions, which have made a number of publications and this work possible.

The research results published in this thesis have been in part published earlier in international peer-reviewed conferences and journals. Text, figures and tables are taken, in some cases, one-to-one from such own references.

Abstract

Today's nano-scale technology nodes are bringing reliability concerns back to the center stage of digital system design because of issues, like process variability, noise effects, radiation particles, as well as increasing variability at run time. Alleviations of these effects can become potentially very costly and the benefits of technology scaling can be significantly reduced or even lost. In order to build more robust digital systems, initially, their behavior in the presence of hardware-induced bit errors must be analyzed. In many systems, certain types of errors can be tolerated. These cases can be revealed through such an analysis. Overhead can be avoided and remedy measures can be applied only when needed. Communication systems are an interesting domain for such explorations: First, they have high societal relevance due to their ubiquity. Second, they can potentially tolerate hardware-induced errors due to their built-in redundancy present to cope with channel noise. This work focuses on analyzing the impact of such errors on the behavior of communication systems. Typically, error propagation studies are performed through time-consuming fault injection campaigns. These approaches do not scale well with growing system sizes.

Stochastic experiments allow a more time-efficient approach. On top, breaking down the system into subsystems and propagating error statistics through each of these subsystems further improves the speed-up and flexibility in the reliability evaluation of complex systems. As an initial step in this thesis, statistical moments are propagated through the signal flows of Linear-Time-Invariant (LTI) blocks. Such a scheme, although fast, can only be applied in the case that the signal lacks autocorrelation. However, autocorrelation can be introduced in the signal due to various reasons, like by signal processing blocks. In that case, other approaches are available to reduce the computational cost of the necessary (repetitive) experiments, like the Principal Component Analysis (PCA). Benefits of such a technique depend on several parameters and, therefore, a more broadly usable technique is required. To address this need, a framework is proposed that exploits the repetitive nature of fault injection experiments for speed-up in LTI blocks. Two cases are distinguished: One, in which all operators of the LTI block act in a linear time-invariant way, and one, in which non-linear operations due to finite wordlengths are present. To complement the subject matter, the broad range of hardware-based mitigation techniques at the higher system level are explored and characterized. In this way, the main properties of each mitigation category are identified and, therefore, suitable choices can be made according to the application needs.

Contents

Acknowledgements	i
Abstract	iii
1 Introduction	1
1.1 Thesis context and contributions	5
1.2 Thesis outline	8
2 Background and Context	9
2.1 Outline	9
2.2 Target systems and random processes	9
2.2.1 Target systems characteristics	9
2.2.2 Relevant terms from statistics and probability theory	12
2.3 Error injection and propagation studies	13
2.3.1 Error injection	14
2.3.2 Error propagation	16
2.4 Framework overview	19
2.5 Correlation between the error and error-free signal and comparison with noise sources	21
2.6 Summary and link to next chapter	26
3 Propagation of Corrupted, Uncorrelated Signal	27
3.1 Outline	27
3.2 Motivation and preliminaries	27
3.2.1 Statistical characteristics of the considered signal	28
3.2.2 Effects of error injection on Gaussian PMFs	29
3.2.3 Linear transformation of normal random variables	31
3.3 Propagating the corrupted signal using statistical moments	32

3.3.1	Set-up and observation	32
3.3.2	Modeling approach	36
3.4	Experimental results and related work discussion	38
3.4.1	Experimental set-up	38
3.4.2	Results	39
3.4.3	Related work	40
3.5	Limitations with using the statistical moments	42
3.5.1	Propagating the corrupted signal through LTI blocks	42
3.5.2	Additional related work	52
3.6	Summary and link to next chapter	52
4	Propagation of Corrupted, Autocorrelated Signal	53
4.1	Outline	53
4.2	Motivation and preliminaries	53
4.2.1	PCA steps and dimensionality reduction	55
4.3	Propagating the corrupted signal using PCA	57
4.3.1	Data organization for injection experiments	57
4.3.2	Illustration of the approach	61
4.4	Results and limitations with using the PCA for speeding-up the propagation of the corrupted, autocorrelated signal	66
4.5	Summary and link to next chapter	69
5	Propagation of Corrupted, Generic Signal	71
5.1	Outline	71
5.2	Motivation and preliminaries	71
5.2.1	Effect of finite wordlengths on error propagation	72
5.3	Error propagation approach	74
5.3.1	Error separation: propagation without finite wordlength effects	74
5.3.2	Error-free terms reuse: propagation combined with finite wordlength effects	75
5.3.3	Reduction of computational complexity.	77
5.4	Results	80
5.4.1	Set-up	80
5.4.2	Without non-linear effects	81
5.4.3	Including non-linear effects	81
5.5	Summary and link to next chapter	84

6	A Classification of Hardware-Based Resilience Techniques at the Higher Abstraction of Digital Systems	85
6.1	Outline	85
6.2	Introduction	85
6.3	Context and useful terminology	86
6.3.1	Resilient digital system design	86
6.3.2	Computing terminology	86
6.3.3	Rationale of the classification and its presentation	87
6.4	Platform hardware mitigation techniques	88
6.4.1	Forward execution - Additional HW modules provision	89
6.4.2	Forward execution - HW modules amount fixed	93
6.4.3	Backward execution - Additional HW modules provision	97
6.4.4	Backward execution - HW modules amount fixed	98
6.4.5	Overall platform hardware classification	102
6.5	Summary	102
7	Conclusions	105
	List of Publications	107
	List of Terms and Symbols	113

List of Figures

1.1	Thesis focus within the reliability assessment and mitigation flow . . .	6
2.1	Detection operation in binary baseband communication system	10
2.2	Example of BPSK-based communication line	15
2.3	Both signal values change by the same magnitude but in different directions leading to different amplitudes.	16
2.4	Views of the same system at different abstractions	17
2.5	Views of the error injection and propagation in a system at different abstractions	17
2.6	Example of splitting a communication system into subsystems	18
2.7	Error signal generation and propagation through the system blocks using a mixture of techniques	20
2.8	Two periods of the error-free (a) and the corrupted-by-channel-noise (b) sinusoidal signal	22
2.9	Channel-induced error signal (a) modeled and (b) resulting after the interaction with the error-free sinusoidal signal	22
2.10	Two periods of the non-quantized (a) and the quantized (b) sinusoidal signal	23
2.11	Quantization noise signal (a) modeled and (b) resulting after the interaction with the error-free sinusoidal signal	24
2.12	Quantization noise signal (a) modeled and (b) resulting after the interaction with the error-free sinusoidal signal	24
2.13	Two periods of the error-free (a) and the corrupted-by-bit-errors (b) sinusoidal signal	25
2.14	Bit-flip signal (a) and error signal (b)	26
3.1	Bit-flip samples PMF and effect of single bit-flips on signal PMF. . .	29
3.2	Error injection in simple signal-flow (2- <i>tap</i> FIR filter)	33
3.3	PMF at the filter input and output when $\nu_f = 7$ is corrupted at the input (2- <i>tap</i> filter)	34

3.4	PMF at the filter input and output when $\nu_f = 4$ is corrupted at the input (2- <i>tap</i> filter)	36
3.5	PMF at the filter input and output when $\nu_f = 4$ is corrupted at the input (5- <i>tap</i> filter)	36
3.6	Illustration of reference set-up vs. the implemented set-up to efficiently propagate bit-errors on the noise distribution through the filter at the receiver	39
3.7	Data path 1	42
3.8	PMFs of the signal at the different nodes of the data path 1	43
3.9	Creation of Gaussian components at data path 1	45
3.10	Data path 2	46
3.11	PMFs of the signal at the different nodes of the data path 2	47
3.12	Internal correlation due to the filter in the first section of data path 2	48
3.13	ACF plots at node (C) for data path 1 and 2	49
3.14	DFT signal flow	50
3.15	Input (node (A)) and output (node (B)) PMF after injection experiments in DFT block	51
3.16	ACF plots at node (A) of the DFT block	51
4.1	Set-up for propagating the corrupted, autocorrelated signal through the G operator in the reference domain (a) and in the reduced PCA domain (b)	54
4.2	Modified injection rate in order to speed-up the error injection campaign	58
4.3	Filter output when a single error sample is placed at the input	59
4.4	A long sample sequence is re-organized into sections of length S	60
4.5	PMF (a) and ACF (b) plot of the original autocorrelated signal	63
4.6	PMF (a) and ACF (b) plot of the autocorrelated signal after truncation (10 dimensions kept)	64
4.7	PMF (a) and ACF (b) plot of the reference corrupted signal \tilde{Y}	65
4.8	PMF (a) and ACF (b) plot of the re-projected (from PCA) corrupted signal $Y \approx \tilde{Y}'$	65
4.9	Histograms to illustrate gains in dimension reduction for multiple filters with various characteristics	67
4.10	Impulse and frequency responses of 5 representative low-pass filters (two upper rows) and 5 representative high-pass filters (two lower rows).	68
5.1	An example of non-linear operations due to the finite wordlengths in a direct-form filter	72

5.2	Quantization (truncation) and saturation characteristics and effect on error values	74
5.3	Two functionally equivalent approaches to calculate the error signal at the output	75
5.4	SFGs for transposed and lattice (only 2 stages) filters	79
5.5	SFG for a DFT block of length N	79
5.6	Coefficients for the 32-tap filter and PMF of the input signal	80
6.1	Top down splitting to create the classification tree and mapping of the related work	87
6.2	Basic classification ¹ for techniques at the platform HW	89
6.3	Classification for forward techniques that require additional HW modules	90
6.4	Lockstep execution in a <i>pair-and-spare</i> structure	91
6.5	Read out (7,4) Hamming codeword and syndrome generation for zero and one error with correction	93
6.6	Classification for forward techniques that keep the amount of HW modules fixed	94
6.7	Classification for backward techniques that require the provision of additional HW modules	97
6.8	Classification for backward techniques that reuse existing HW modules	98
6.9	Illustration of concepts in the platform-HW <i>backward</i> category	100
6.10	A local error can trigger all the CMP cores to roll-back in global checkpointing schemes	101
6.11	Overall proposed classification for techniques at the platform HW	103

¹The boxes in the classification figures include hyperlinks to the text. By clicking on each of the boxes, the reader will be transferred to the corresponding section in the text.

List of Tables

3.1	Propagation of distribution $\mathcal{N}(16, 2)$ through 2-tap FIR filter when $\nu_f = 7$ is flipped	34
3.2	Propagation of distribution $\mathcal{N}(16, 2)$ through 2-tap FIR filter when $\nu_f = 4$ is flipped	35
3.3	Comparison for 16-, 32-, 64-tap filter (input signal: $\mu=0, \sigma=0.562$) . .	40
3.4	Skewness and kurtosis at the DFT output	49
4.1	Parameters used in the current illustration	61
4.2	Correlation among first 5 rows and eigenvalues for uncorrelated and autocorrelated data	62
4.3	Computational cost for filtering (through filter 2) the autocorrelated data (autocorrelated through filter 1)	66
5.1	A small illustration of the effect of quantization on error propagation	73
5.2	Results for 32-tap direct-form filter under different input wordlengths without non-linear effects	81
5.3	Results for 8-bit input wordlength under different direct-form filter lengths without non-linear effects	81
5.4	Erroneous (non-masked) sample counts for 32-tap direct-form filter .	82
5.5	Results for 32-tap direct-form filter under different input wordlengths with non-linear effects	83
5.6	Results for 8-bit input wordlength under different direct-form filter lengths with non-linear effects	83
5.7	Results for 16-bit input wordlength under different all-zero lattice filter lengths with non-linear effects	83
5.8	Results for 16-bit input wordlength under different DFT block lengths with non-linear effects	83
7.1	LTI Processing of Stationary Random Process in the Principle Component Analysis (PCA) Domain	109
7.2	Trade-offs in HW-based resilience techniques-Part 1	110

7.3	Trade-offs in HW-based resilience techniques-Part 2	111
7.4	Trade-offs in HW-based resilience techniques-Part 3	112

Chapter 1

Introduction

The early concerns of John von Neumann [91] regarding building reliable computing entities out of unreliable components were largely forgotten with the gradual replacement of vacuum tubes by transistors and the following high-scale transistor integration [58]. Now, after some decades, reliability has come back to the forefront in the context of modern CMOS technology. The current reliability concerns originate from mechanisms that occur both during the manufacturing process and during the system's operational lifetime. The anomalous physical conditions that are created from these effects are called **faults**. Such anomalous cases include (permanent) physical defects or (temporal) deviations of physical properties. Typically, in CMOS circuits, faults cause deviations of the electrical potential of the circuit nodes and may manifest as (logic bit) **errors** captured at a logic circuit storage element e.g. flip-flop or a memory cell. In case the fault does not manifest as error, it is considered to be *masked*. Depending on the circuit operation, errors can persist in the memory element for long and/or propagate through the whole circuit. They may cause **failures** at system level. In that case, the system is not able anymore to fulfill its specification.

Some of the prevalent physical mechanisms, which contribute to the manifestation of faults and errors, are briefly discussed in the following:

(a) Mechanisms that concern devices include:

- (i) **Radiation particles** [5], [62] e.g. due to ionizing alpha-particles (generated in the package) or due to high-energy cosmic rays may lead to bit-flips. The particles generate (many) electron-hole pairs along their trajectory through the semiconductor material. In the electrical field of the depletion region of a reverse-biased pn-junction, electrons and holes get separated by Coulomb forces leading to charge packets. Depending on the amount of charge and impedance state of the associated electrical circuit nodes, their electric potential is altered significantly.

In case the electric potential at the input of a logic storage element at strobe time, e.g. of a flip-flop, deviates from the expected voltage intervals

$[V'_{\max}, V'_{\min}]$ or $[V'_{\max}, V'_{\min}]$ for the two logic levels, a wrong bit is stored. In addition, these faults can manifest as errors in the memory cells. In the early days, only Dynamic Random Access Memories (DRAMs) were affected, since, in DRAMs, the information is represented by electric charge on a floating (small) capacitance. A particle hit can lead to a significant alteration of the charge packet and, thereby, corrupt the stored bit. Later, due to the decreased feature sizes, Static Random Access Memories (SRAMs) were affected as well. In SRAM cells, the information is stored electronically in a feedback loop. A particle hit can cause the cell to flip from one to the other stable operating point. Today, in nano-scale technologies, as already mentioned, also logic circuit reliability is affected by radiation-induced faults. Often, together with timing violation faults (caused e.g. due to temporarily lower supply voltage), radiation-induced faults are considered as a major source for transient errors and called soft errors.

- (ii) **Bias Temperature Instability** (BTI) [53], [5], [86], [51] called NBTI for pfet and PBTI for nfet devices, is a time-dependent degradation mechanism, which causes shifts in the device threshold voltage. It is caused by applying a bias voltage at the device at high temperature. The mechanism is based on so-called traps that are present at the oxide-substrate interface as well as in the gate oxide (e.g. due to impurities). The effect of traps is the creation of hole-electron pairs (capture effect) and their release again (emission effect), which causes temporal variations in the device properties. The variations in the device properties lead in turn to variations in the delay and noise-margin properties of logic circuits and stored electric charges in memory cells. During "no-stress" periods (with gate voltage grounded) devices can recover (at least partially); the time constants for capture and emission have a very wide span, ranging from nanoseconds up to days or longer. In smaller devices, the BTI impact is more significant as the effect of a trap charge is more significant on a tiny device.
- (iii) **Hot Carrier Injection** (HCI) [53] refers to a degradation effect, according to which, highly ("hot") accelerated (from the electric field of the gate) channel carriers cause impact ionization or can be injected into the oxide. In nfet devices Channel Hot Electron (CHE) and in pfet devices Channel Hot Hole (CHH) effects take place. Again, this effect causes variations of the device properties.
- (iv) **Time-Dependent Dielectric Breakdown** (TDDB) [53], [5], [55] is a degradation effect that occurs due to long lasting tunnel currents through the gate oxide caused by relatively small electric fields and forming filamentary shorts between the gate and substrate. Initially, the mechanism leads to variations of the device threshold voltage. In the end, the gate oxide breaks leading to permanent defects.

- (v) **Random Telegraph Noise** (RTN) [5] is based on similar mechanisms as BTI but it has much smaller time constants (from subsec down to the subns domain) and causes rapid threshold voltage and thereby current fluctuations. This effect is considered to have more of a statistical nature.
- (b) Mechanisms that concern wires include:
- (i) **Electromigration** (EM) [53], [5] refers to the physical movement of metal atoms in a wire due to high current density, resulting in an increased wire resistance or even in a permanent break. Increased resistances affect interconnect delay or IR drop (see below) in the supply network. The effect can be (partly) reversed by applying current in the opposite direction.
 - (ii) **Electromagnetic coupling** (crosstalk) [5] effects, are due to coupling capacitances between circuit nodes and mutual inductances between circuit loops. In the dominating case of capacitive coupling, a rapid voltage transition on a circuit node (called the aggressor) enforces an alteration of the electric potential of a neighboring node (called the victim). The result is a noise pulse on a quite victim node or, in case that the victim node simultaneously features a transition in opposite direction, an increased delay (the latter being called "noise on delay").
- (c) Mechanisms that are based on intra-die and inter-die correlations include:
- (i) The **IR drop** [5] is the Ohmic voltage drop caused by the interconnect wire resistance. It mainly affects the supply network where it causes dynamic "supply voltage droops" and contributes to the supply voltage noise. Supply voltage droops affect gate delay and supply voltage noise leads to noisy voltage levels at the gate outputs.
 - (ii) The **di/dt droop** [5] is the voltage drop over inductance during rapid transitions of the current flowing through it. It also mainly affects the supply network and causes supply voltage droops and contributes to the supply voltage noise as well.
 - (iii) **Temperature variation** [10] is caused due to heat flux, created at varying locations on the die depending on the activity (and indirectly the system blocks' functionality). Temperature variations are spatially and temporarily distributed and affect delay and noise margin.
- The effects of dynamic IR and di/dt drop on the supply network are similar and designers do take this effect into account and add a margin. Moreover, the supply voltage can be stabilized by applying properly dimensioned buffer capacitances (so-called decoupling capacitances or decaps) across the whole supply network.
- (d) Finally, significant enabling components for the aforementioned mechanisms include:

- (i) Process variations [10], [42], which take place at manufacturing time (e.g. Random Dopant Fluctuation (RDF)).
- (ii) Circuit topology and operating parameters (e.g. voltage, frequency).
- (iii) Input workload (e.g. duty cycle, bit pattern etc).
- (iv) Environmental conditions (e.g. heat coming from neighbouring systems).

Having discussed the basic mechanisms, it is important to underline the aspect of different vulnerability for the two logic states, which is relevant for the selection of an error model at the bit level.

Temporal deviations of the electric potential of circuit nodes can be caused due to noise pulses and/or incomplete previous logic level transitions (so-called timing violations). At strobe time, the electric potential at the input of a storage element may either be shifted to a higher (towards plus infinity) or lower (towards minus infinity) value. It can be concluded, that faults, which induce a positive potential shift, can *only* lead to a bit error in storage elements, whose input should be in the low-('0') state. In the same reasoning, faults, which induce a negative potential shift can *only* lead to a bit error on input nodes, which should be in the high-('1') state.

As it is known, chains of CMOS gates feature the capability of level regeneration due to a steep transition in their voltage transfer characteristics. This capability is quantitatively described by the so-called noise margin NM of a gate. Generally, the noise margin is not symmetric concerning the two logic levels, i.e. $NM_{1'} \neq NM_{0'}$. This can happen due to intentional device dimensioning ("skew"), device parameter variability ("mismatch"), temporal device parameter fluctuation (RTN), temporal or permanent device aging (e.g. due to HCI, BTI, EM) etc. As a consequence, the impact of a circuit node potential shift in general is different for the two logic states or the other way around: the two logic states feature different vulnerability to a potential shift.

Despite the difference in the vulnerability that the two logic states exhibit, the *bit-flip* error model is widely applied as an error injection model in reliability analysis of digital systems. It describes the probability that a bit is flipped from 1 to 0 or from 0 to 1 at a certain instance of time. This means that the separate probabilities for a flip-to-1- and flip-to-0-error somehow have to be combined (i.e. averaged) to a bit-flip probability. As this is the most well-established error model, we will use the term (functional) *bit-flip errors* to capture this class of errors, with the worst case manifestation toward the end user being a complete system *failure* on the expected system service.

The manifested bit-flip errors can be temporary or permanent [9], [10]. Temporary errors include transient and intermittent errors. Transient errors are non-deterministic (concerning time and location), e.g. as a result of a fault due to a particle strike. While the causing faults typically last for one cycle only, the resulting errors may persist longer due to propagation and storing. Transient

errors are considered "external fault"-induced errors and need to be characterized by their average rate of occurrence. Intermittent errors occur repeatedly but non-deterministically in time at the same location and last for one cycle or even for a long (but finite) period of time. Main causes for intermittent errors are design weaknesses, aging and wear-out (e.g. due to BTI, HCI, EM, silicon-wafer direct bonding (SDB) etc.), and parametric faults, not detected as errors during test (as errors due to them are triggered under rare conditions only). They are considered "internal fault"-induced errors and need to be statistically characterized by their (average) rate of occurrence and (average) time of duration. In contrast, permanent errors persist forever after their first occurrence. Causes for permanent errors are fabrication defects not detected during test, accelerated aging and wear-out. Moreover, the bit-flip errors may be spatially or temporarily correlated, forming *burst errors* [70], [79].

In a similar way that whole systems can be viewed using different abstraction layers with transistors and wires being the building components for gates, storage elements, memory and processing blocks, also reliability effects can manifest and be viewed across the system abstraction layers. Especially, the bit-flip allows the connection of the physical fault mechanisms with the higher system abstractions, where, often, more useful interpretations of the impact of faults on the system can take place. These interpretations can then be provided as input to the designer in order to derive appropriate countermeasures.

1.1 Thesis context and contributions

Undeniably, having a more robust system is the ultimate goal of studying the fault mechanisms and their propagation. Having quantitative information on how the faults manifest and propagate through the system allows the researcher and the designer to come up with system-specific, efficient mitigation solutions. So, *reliability assessment* and *resilience and mitigation techniques* form two complementary sub-domains within the domain of reliability of digital systems. Reliability assessment [72] includes the derivation of appropriate models for faults and error mechanisms, the various ways to intentionally introduce them in the system, the study of their propagation and impact and the derivation of appropriate metrics that quantify this impact. Resilience and mitigation techniques, as the name implies, include all those approaches that make systems more robust to errors. They cover a broad range of approaches, from static design-time up to demand-driven run-time techniques, that, for example, mask the effect of errors or prevent future occurrences [69].

Both reliability assessment and resilience/mitigation techniques can be performed at various abstraction levels of the system leading to different benefits and costs. The left-hand side of Fig. 1.1 shows some examples of how reliability assessment is performed across the abstraction layers. More information can be found in [72]. The right-hand side is complemented with examples of mitigation

actions at the various layers. In practice, the exact mitigation decision should be based on a careful trade-off analysis based on the demands of the system under consideration. In Fig. 1.1, the two contributions of this thesis are sketched.

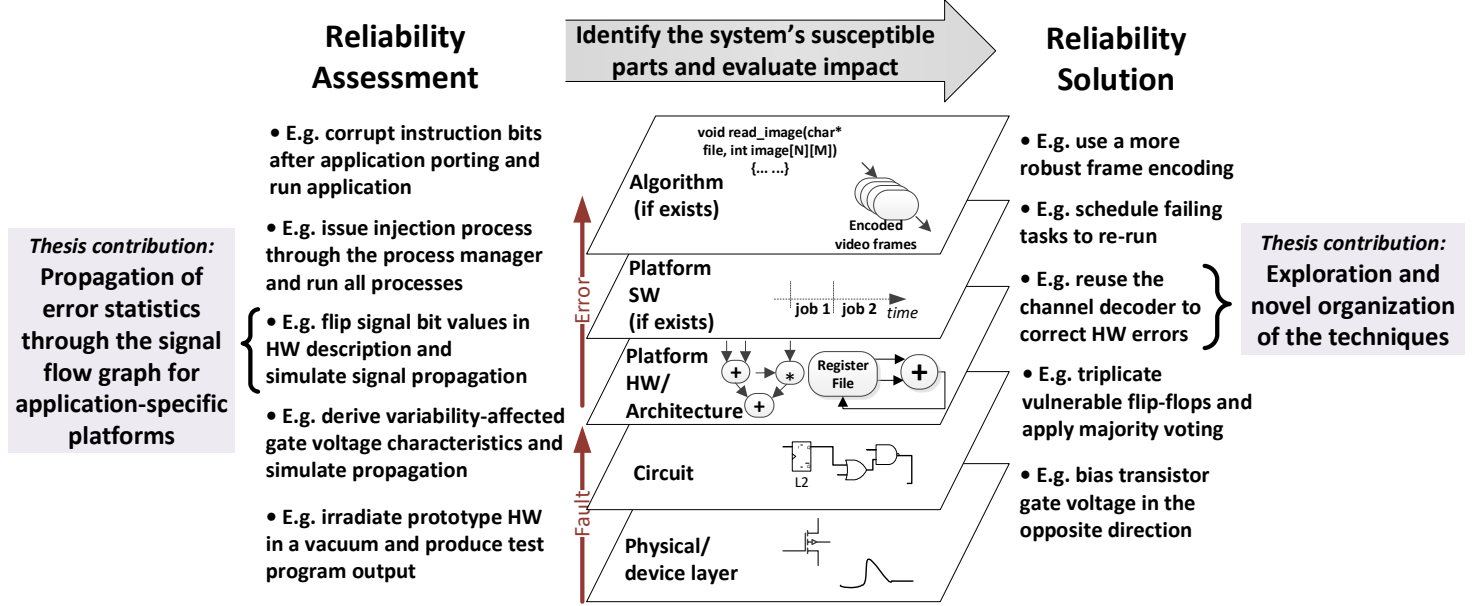


Figure 1.1: Thesis focus within the reliability assessment and mitigation flow

Reliability assessment sub-domain. Regarding the assessment part, the focus is on deriving efficient ways to propagate bit-flips through the system. Traditionally, at this abstraction layer, such a study has been performed through system simulation and/or emulation. There is a variety of tools with different characteristics that can be used for such a study. Typically, the hardware (HW) is provided in a descriptive form (i.e. using a hardware description language) and ported on the corresponding tool. Then, errors can be injected in different forms: through replacing hardware components (e.g. a NAND with a NOR gate) in the HW description [35], through altering data signals for a specific amount of time by using appropriate commands [35] (instead of altering hardware components), by reusing the scan-chains [23], through introducing extra components (e.g. multiplexers) that allow the injected signal to be introduced instead of the correct one [50] to name some examples. Sometimes, FPGAs [50] or HW emulator environments [16] are used to speed-up such a process. Despite the good accuracy provided by such tools, the execution time to perform repetitive experiments as well as the migration and general engineering effort to set-up the injection frameworks are issues to be avoided (whenever possible) for the designers. Therefore, there is an ongoing demand for improving such frameworks or finding alternatives. Exploiting the characteristics of one's specific context is a starting point for finding attractive alternatives.

The context of this work, in terms of target applications and platforms, is defined by the focus on communication systems. There are some common characteristics of such systems. The hardware is composed to a significant extent of non-flexible parts (ASIC-style). This is the case for example, for Personal Area Network (PAN) and Body Area Network (BAN) systems. Quite often, in communication systems, the application input signal is viewed as a stochastic signal with certain statistical characteristics. This gives the flexibility to make use of the signal statistics in order to find efficient ways to propagate the error signal. For this, the system signal flow is a sufficient system model and a detailed hardware description of the system is not required as the main objective is that the signal statistics at the output are accurate.

Having as starting point the bit-flip model, efficient ways are explored in this thesis to study the propagation of bit-flips through communication systems. Analytical and statistical approaches are employed to avoid the time-consuming system simulation. This is enabled by choosing appropriate models to represent errors as they propagate through the system, decomposing the system into blocks and exploiting linearity in a stochastic framework. On top, contrary to earlier works, the effect of non-linearities on the error propagation (due to operations like saturation and quantization in the block internals) are considered. More specifically, the core elements of this contribution are:

- (i) We discuss possibilities and limitations in employing analytical methods, like using statistical moments, to derive signal statistics in the context of hardware-induced bit-flips. The focus is on the so-called Linear-Time-Invariant (LTI) blocks. We show that purely analytical approaches can have limited use in the context of estimating the impact of hardware-induced bit errors, due to the correlation that exists between the errors and the signal.
- (ii) We discuss possibilities and limitations in using a mathematical transform, the Principal Component Analysis (PCA), to speed-up error injection experiments.
- (iii) We propose a framework that exploits the repetitive nature of fault injection experiments to speed-up experiments for LTI blocks. We distinguish between the cases that all the operators from the input to the output of the LTI block act in a linear time-invariant way and the case that non-linear operations due to finite wordlengths take place.

Reliability solution sub-domain. Once the error propagation has been studied and the impact is evaluated, an appropriate mitigation scheme has to be selected. The second contribution in this thesis regards mitigation. Given the larger overhead required to mitigate faults at the circuit level, it may be preferable to seek higher level solutions. Here, we provide an exploration and novel classification of hardware-based schemes at the higher system abstraction in the mitigation domain. More specifically, the core elements of this contribution are:

- (i) An comprehensive overview of the domain of functional reliability techniques at the (micro-)architectural level is presented, using a systematic, hierarchical top-down splitting into sub-classes. Pros and cons of each sub-class are identified.
- (ii) Multiple representative prior and state-of-the-art publications are mapped to these categories to illustrate the concepts involved.

1.2 Thesis outline

Chapters 2, 3, 4, 5 contain the first part of our contribution in the reliability assessment sub-domain. Chapter 2 gives the context of the error statistics propagation study including relevant definitions. Chapter 3 discusses our findings while considering the propagation of statistical moments of corrupted, uncorrelated signals through LTI blocks. Chapter 4 explores the possibilities in using the Principal Component Analysis to speed-up the error injection experiments. Chapter 5, illustrates the generic framework for propagating a corrupted signal through LTI blocks that include internally non-linear operations due to finite wordlengths. Chapter 6 is the contribution in the mitigation sub-domain and, more specifically, it presents a novel, systematic classification of hardware-based mitigation techniques along with discussing multiple examples per class. Chapter 7 summarizes and concludes the work.

Chapter 2

Background and Context

2.1 Outline

Chap. 2 provides the context of the current work, including the motivation to handle the system complexity for injection experiments and relevant definitions. Sec. 2.2 describes the target systems and characteristics of signals in such systems. Sec. 2.3 provides background information on the characteristics of error injection and propagation studies. In addition, the error model is introduced. Sec. 2.4 gives an overview of the stochastic framework being used, under which, the proposed approaches (presented in the following chapters) fall under. Sec. 2.5 illustrates the characteristics of the bit error model as compared to two other noise sources: quantization noise and channel noise. Sec. 2.6 summarizes the main elements that constitute the general framework.

2.2 Target systems and random processes

2.2.1 Target systems characteristics

Digital communication systems. In communication systems information is transmitted from a sender to a receiver over a channel. Namely, the information signal is not known beforehand to the receiver. The channel is exposed to noise and distortions. Both the information and noise signals can be represented using random processes. Especially digital communication has now become a standard due to its various advantages, among which, the fact that digital hardware is cheap, area-efficient and "reliable". Digital communication systems operate on signals that are specified at discrete time points (achieved through sampling) and discrete amplitude levels (achieved through quantization).

Signal processing and representation. The system transforms the input information signal in such a way so that the probability is in the best way

increased that the signal will be distinguished at the channel output. Among the issues that have to be handled during this transformation are the channel noise, symbol interference, synchronization between the transmitter and the receiver. One prominent transformation example is the processing of the signal by the scrambler before it gets transmitted. Through the scrambler, the data gets randomized so that long sequences of zeros and ones are avoided; this makes the received data pseudorandom, i.e. close to ideal binary random sequence with each signal taking the value 0 and 1 with a 0.5 probability [88]. This makes the *signal representation by a random process very fitting*. Interleaving and modulation are other processing steps in communication systems. Channel noise is an additive noise (does not depend on the transmitted signal). For wireline communication systems, this noise is characterized by a Gaussian process. For other systems, typically multiple paths exist and more complicated models are required [24].

Performance evaluation. As the signal communication cannot be error-free, the effort of designers is focused on reducing the error probability so that the accuracy of the received digital signal improves. To derive this probability, either the process is analytically characterized or statistics are gathered by repeating the experiment a sufficient amount of times. To illustrate this, the simple example of threshold detection in a binary channel is used, as shown in Figure 2.1. Symbols 0 and 1

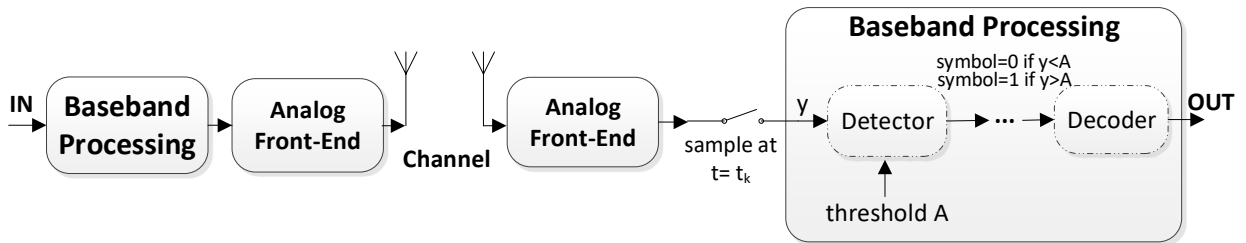


Figure 2.1: Detection operation in binary baseband communication system

are transmitted with equal probability over the channel using a negative and a positive pulse respectively. Each pulse is sampled at peak amplitude y (at time instance $t = t_k$, with k indicating the time index). If the sampled value y is bigger than the threshold A ($= 0$ here), the detector reads a 1; in the opposite case, it reads a 0. However, y contains the original pulse peak amplitude plus a noise amplitude, which ranges from $-\infty$ to $+\infty$. Therefore, a received 0 can be read as 1 and vice versa. To find this error probability, the experiments are repeated N times ($N \rightarrow \infty$) [44], so that the process can be characterized (unless the error probability can already be derived analytically). Often, and depending on the objectives, the baseband system model (i.e. the system without modulation) is only analyzed in such an error probability study. The results can be extended to the passband

(or bandpass) model.¹ To evaluate the performance of the communication system, typically metrics, such as the Bit Error Rate (BER), are used. BER indicates the number of erroneous bits out of the total number of received bits. The erroneous bits are found by comparing the error-free implementation—that is, in the absence of channel noise—with the implementation in the presence of noise. Sometimes, variations of this metric are used, like the Packet Error Rate (PER).

LTI blocks. A significant part of the system blocks are linear time-invariant (LTI) blocks. As the name implies, the characteristic properties of LTI blocks are homogeneity, additivity (necessary and sufficient properties for linearity) and shift-invariance. LTI blocks may have memory or be memoryless. They include arithmetic (rather than decision making) operators and more specifically, they are composed of linear operators such as adders, constant multipliers, constant shifters. Examples of LTI blocks include: memory blocks, interleavers, some types of multiplexers, Finite Impulse Response (FIR) filters, Discrete Fourier Transform (DFT) blocks.

A LTI block is fully characterized by its impulse response [57]. Here, the focus is on non-recursive, discrete-time and discrete-value (digital) LTI blocks that are causal. Assuming input samples x_k , output samples y_k and an impulse response c_n , the block output can be calculated by the superposition of the impulse response weighted and shifted according to the corresponding input samples. An alternative way to represent LTI blocks is by using their difference equation (here for non-recursive LTI blocks):

$$y_k = \sum_{n=1}^M c_n \cdot x_{k-(n-1)}. \quad (2.1)$$

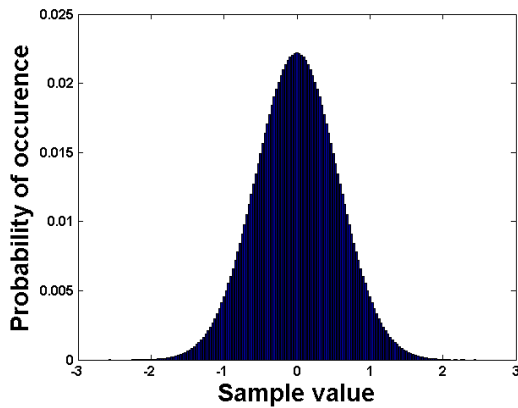
By appropriately describing the values of x, y, c_n, M (real- or complex-valued, vectors or matrices) in Eq. 2.1 all possible digital LTI blocks can be represented. For example, in a Discrete Fourier Transform (DFT) block, the x, y are vectors, $M = 1$ and c_1 is a matrix. In case of a direct-form FIR filter, c_n , with $1 \leq n \leq M$, are the filter coefficients and x, y are scalars.

Often, the random process at the output of LTI blocks can be derived analytically or at reduced cost using other mathematical methods for a given random process at the input of the system. LTI blocks are the focus of this study.

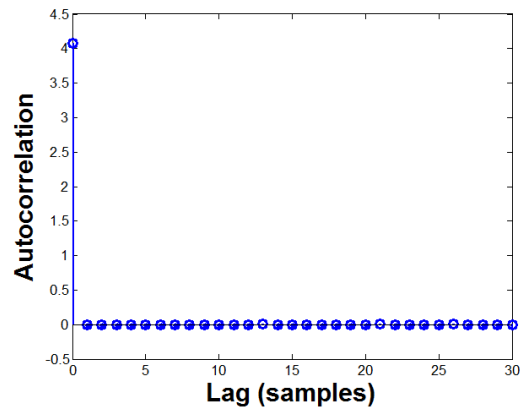
¹In radio-based communication systems before transmission, typically the signal has to be up-converted to a higher frequency band, the so-called *passband*. Consequently, at receive-side the received signal has to be down-converted. For the sake of simplicity and efficiency, in system simulation often the whole chain of up-converter, radio channel (i.e., the *passband-channel model* including amplifiers as well as filters etc. on transmit- and receive-side), and down-converter (including carrier recovery) is replaced by a typically complex-valued equivalent *baseband-channel model*. Moreover, in today's digital communication-system simulation frequently also the Digital-to-Analog (D/A) and Analog-to-Digital (A/D) converters (including clock recovery) are hidden in the equivalent *discrete baseband-channel model* [8, 15].

2.2.2 Relevant terms from statistics and probability theory

Signal statistical characteristics. As often done in digital signal processing, the digital signal x is modeled as a realization of a random process, with the signal samples x_k being discrete-state Random Variables (RVs) at discrete time instances k . The random process has certain statistical characteristics, like the statistical moments mean μ_X , variance σ_X^2 etc. and/or a Probability Mass Function (PMF). PMFs provide information on the probability distribution of the amplitudes of the discrete signal, like Probability Density Functions (PDFs) do for continuous variables. Fig. 2.2a shows a PMF example for normally-distributed discrete signals. In order to have a complete characterization of the random signal, information on how these amplitudes are interrelated over time is needed on top. For many applications, certain averages are used, like the autocorrelation [60]. For a time-discrete signal x the Autocorrelation Function (ACF) of lag l is defined as $r_{xx,l} = E[x_{k+l} \cdot x_k^*]$ with $E[\cdot]$ being the expected value and $(\cdot)^*$ being the complex conjugate of a complex-valued sample value. It is often visualized by plotting the sample autocorrelation values versus the time lags (or sample lags), as shown in Fig. 2.2b. In case of a so-called *wide-sense stationary process*, $r_{xx,l}$ does not depend on time index k . For a *white* random process the autocorrelation is zero (except at lag 0). In this case (see Fig. 2.2b), no identifiable relationship among the samples can be observed. In general, both types of information, probability mass functions (PMFs) and autocorrelation are required in order to fully characterize a stationary random signal. The relations among random variables can be described using functions other than the PMF and autocorrelation, like the characteristic function, the moment-generating function and others [60]. Covering an in-depth analysis of all the alternative ways is out of the thesis scope. A random signal after hardware-induced bit flips have been injected will be called **corrupted signal**. The error of a corrupted signal sample \tilde{x}_k is denoted by $e_k = \tilde{x}_k - x_k$ and will be called **error signal** sample.



(a) Probability Mass Function plot of normally distributed discrete signals



(b) Autocorrelation plot of a white process

Accuracy Evaluation. The accuracy between a precise model and an approximated one can be measured in different ways, depending on the objectives. Here, we present one well-established means, used for comparing probability distributions.

The *Kullback Leibler divergence* or *KL divergence* [48] is a measure of the difference between two probability distributions P and Q , when Q is used to approximate P . It represents the amount of information lost with this approximation; therefore, the smaller the KL value, the closer the distributions are. For discrete probability distributions P and Q , the Kullback Leibler divergence from Q to P is defined to be

$$D_{KL}(P \parallel Q) = \sum_i P(i) \cdot \log \frac{P(i)}{Q(i)}. \quad (2.2)$$

Namely, it is the expectation of the logarithmic difference between the probabilities P and Q , where the expectation is taken using the probabilities P . The quantity $\log \frac{1}{p}$ is a measure of the surprise when an event, which has probability p , actually occurs (e.g. a highly improbable outcome is very surprising). To acquire an estimate of the average value of the surprise involved with a whole set of events, that follow a probability distribution P , the previous is extended. For every possible outcome i associated with a probability $P(i)$, the $\log \frac{1}{P(i)}$ is weighted by $P(i)$ and all products are summed. This gives the entropy of the probability distribution P , which is defined as

$$H(P) = \sum_i P(i) \cdot \log \frac{1}{P(i)}. \quad (2.3)$$

Cross-entropy can be interpreted as a measure of the surprise when a wrong distribution Q is assumed while the data actually follows a distribution P . It is defined as

$$H(P, Q) = \sum_i P(i) \cdot \log \frac{1}{Q(i)}. \quad (2.4)$$

The KL divergence is then the difference of the cross-entropy $H(P, Q)$ and the entropy of P , $H(P)$. Namely, it gives a measure of the "additional" surprise when Q is wrongly assumed (rather than the actual P) compared to when P (the correct one) is assumed.

2.3 Error injection and propagation studies

Error injection campaigns (sometimes in literature called *fault injection* campaigns independent of the system abstraction) consist of repetitive simulation experiments (or executions on a prototype system), during which, values at selected points are modified to mimic a fault effect. An elementary error injection experiment corresponds to one simulation run of the target system [97], with one or more bits being corrupted. In general, a huge amount of elementary error injection

experiments are required in order to include all nodes of interest, all bit positions within a node and all system states. A node in this context designates a location in the system control- or data-path, at the input or output of its basic components, such arithmetic units, memories, registers. In such campaigns, the goal is to perform the following two actions efficiently: (1) *error injection*-selection of nodes, bits, system states to be corrupted and mimicking of the fault effect in a manner so that useful information regarding the vulnerability of the system is derived, and (2) *error propagation*-selection of the system representation and propagation of the corrupted signal to the system output (or to another point of interest, from which the impact on the system can be extrapolated). Efficiency, in this context, refers to minimizing the required time and/or the computational complexity to perform such an analysis.

2.3.1 Error injection

Injection points. Ideally, information on the impact of every bit in every node for every system state would be desired; this would lead to an *exhaustive* error injection campaign. In a system with n flip-flops, and without pruning any of the system states (i.e. without pruning non-reachable during normal execution system states), the state-space cardinality is 2^n . This makes an exhaustive campaign impossible and has made it imperative to look for alternatives.

A *selective* error injection campaign examines the impact of only a subset of the flip-flops. The selection of this subset is performed on several possible criteria, depending on the system and the designer’s objective. One direction is that the selected bit positions to be explored are *deterministically* chosen using some direct or indirect knowledge regarding their importance. This knowledge may concern their vulnerability to certain fault effects (for example flip-flops closer to nodes with specific interconnect distance, signal directions, switching times and driver strength can be more vulnerable to crosstalk effects) and is derived by lower-level fault models (such as transistor-level models) [66, 76]. Their importance may also be provided by the role they play in the system functionality (e.g. control bits) or that they are representative for a group of bit positions [52]. In the latter case, the designer’s deep understanding of the system functionality may be a decisive factor for the selection. A second direction is that the subset to be explored is *randomly* chosen, so that a desired confidence is achieved [16, 22]. The points in the sampling space (which consists of the total amount of flops and system states) may be equally probable or may be assigned a weight according to their relative probability of occurrence. Sometimes combinations of the injection approaches are implemented.

In this work it is assumed that all bits of a word can be potentially corrupted with equal probability, similar to [39]. The focus is on propagating efficiently the errors through system blocks (see below). So, the assumption is that the errors have occurred earlier at the internals of a block (within a system) and manifest at the input of the block of interest. For example, the errors occur in the memory buffer just before the filter block in the baseband model of a BPSK

communication line, as depicted in Fig. 2.2. We focus on propagating the errors through the filter block in this case.

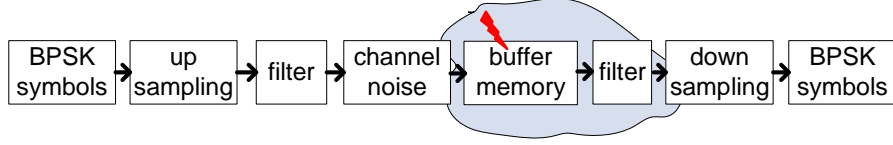


Figure 2.2: Example of BPSK-based communication line

Bit error model assumption. To simplify this discussion and without loss of generality, we assume integer signal samples represented by a non-redundant two's complement notation composed of n bits. We use ν to denote the bit position as used in the two's complement notation, i.e. the sample of the time discrete signal x at time instance k (subscript) is:

$$x_k = -x_k^{n-1} \cdot 2^{n-1} + \sum_{\nu=0}^{n-2} x_k^\nu \cdot 2^\nu, \quad (2.5)$$

where x_k^ν is the bit in position ν , with weight 2^ν , and x_k^{n-1} is the Most Significant Bit (MSB) sign bit ².

An erroneous bit-flip at time instance k in bit position ν is denoted by the flip-error bit ϵ_k^ν . The **bit-flip sample** $[\epsilon_k^{n-1}, \dots, \epsilon_k^1, \epsilon_k^0]$ can be described like a signal sample

$$\epsilon_k = -\epsilon_k^{n-1} \cdot 2^{n-1} + \sum_{\nu=0}^{n-2} \epsilon_k^\nu \cdot 2^\nu. \quad (2.7)$$

Clearly, after some bit-flips the erroneous signal sample \tilde{x}_k becomes

$$\begin{aligned} \tilde{x}_k &= -\tilde{x}_k^{n-1} \cdot 2^{n-1} + \sum_{\nu=0}^{n-2} \tilde{x}_k^\nu \cdot 2^\nu \\ &= -(x_k^{n-1} \oplus \epsilon_k^{n-1}) \cdot 2^{n-1} + \sum_{\nu=0}^{n-2} (x_k^\nu \oplus \epsilon_k^\nu) \cdot 2^\nu \end{aligned} \quad (2.8)$$

with $\tilde{x}_k^\nu = x_k^\nu \oplus \epsilon_k^\nu$ being the erroneous sample bit in bit position ν . The effect of the bit-flipping(s) can also be described by adding an error sample $e_k = -e_k^{n-1} \cdot 2^{n-1} + \sum_{\nu=0}^{n-2} e_k^\nu \cdot 2^\nu$ to the error-free signal sample in every bit position:

$$\tilde{x}_k = x_k + e_k = -(x_k^{n-1} + e_k^{n-1}) \cdot 2^{n-1} + \sum_{\nu=0}^{n-2} (x_k^\nu + e_k^\nu) \cdot 2^\nu. \quad (2.9)$$

From Eq. 2.9, 2.8, 2.5, it follows that:

$$\begin{aligned} e_k = \tilde{x}_k - x_k &= -(x_k^{n-1} \oplus \epsilon_k^{n-1}) \cdot 2^{n-1} + \\ &\quad \sum_{\nu=0}^{n-2} (x_k^\nu \oplus \epsilon_k^\nu) \cdot 2^\nu + x_k^{n-1} \cdot 2^{n-1} - \sum_{\nu=0}^{n-2} x_k^\nu \cdot 2^\nu \\ &= -(x_k^{n-1} \oplus \epsilon_k^{n-1} - x_k^{n-1}) \cdot 2^{n-1} + \sum_{\nu=0}^{n-2} (x_k^\nu \oplus \epsilon_k^\nu - x_k^\nu) \cdot 2^\nu \end{aligned} \quad (2.10)$$

²In the more general case of a signal with n integer bits (including the sign bit), and m fractional bits, the equation is equal to

$$x_k = -x_k^{n-1} \cdot 2^{n-1} + \sum_{\nu=-m}^{n-2} x_k^\nu \cdot 2^\nu. \quad (2.6)$$

Accordingly, the notation for a whole word is given by $n.m$, with the "." denoting the radix point.

with the error sample digit e_k^ν :

$$e_k^\nu = x_k^\nu \oplus \epsilon_k^\nu - x_k^\nu = (-1)^{x_k^\nu} \cdot \epsilon_k^\nu = (1 - 2 \cdot x_k^\nu) \cdot \epsilon_k^\nu \in \{-1, 0, 1\}. \quad (2.11)$$

It is worth noticing that due to the addition $\text{mod}_2(\oplus)$ (bit-wise exclusive OR) in Eq. 2.8 a bit-flip results in an error contribution being dependent on the signal sample value.

When the error-free signal sample gets corrupted by the bit-flip sample, the amplitude of the signal will be modified by a quantity as big as the error magnitude³. Fig. 2.3 depicts this effect for two different two's complement 8-bit samples. In both cases the bit-flip sample has the magnitude 16 (2^4). Corrupting the two samples with this bit-flip sample will change their amplitude by the quantity 16. In the first case, the value 16 will be subtracted. In the second case, the value 16 will be added.



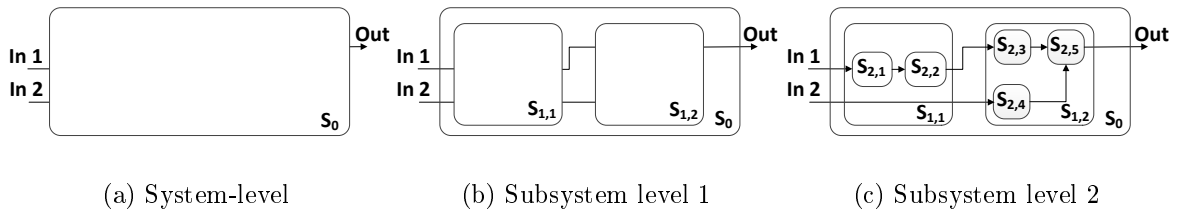
Figure 2.3: Both signal values change by the same magnitude but in different directions leading to different amplitudes.

2.3.2 Error propagation

System defined scope. Fig. 2.4 shows three views of the same system. First, in Fig. 2.4a the system is viewed as a black box with only inputs and outputs. Then, in Fig. 2.4b, the system is split into a number of subsystems, with each one having its own inputs and outputs. Fig. 2.4c shows a more fine-grained decomposition of the same system. Each of these views can be useful according to the context.

Fig. 2.5 shows the same system abstractions after an error has been injected. Several elements can help to identify a suitable scope for error injection and propagation experiments: There are two subsystems (or blocks) that are not affected by the injected error, i.e. the blocks $S_{2,1}$, $S_{2,4}$. Namely, in case the system is considered as in Fig. 2.5a or even Fig. 2.5b, these two subsystems will execute (and thus cost execution time) without contributing any new information compared

³magnitude of a variable is the measure of how far, regardless of direction, its quantity differs from zero while amplitude is the measure of how far, and in what direction, that variable differs from zero



to the information acquired by the error-free execution of these two blocks. In Fig. 2.5a (and potentially also Fig. 2.5b), a masked error cannot be identified until it reaches the system output. Having the information sooner allows the researcher to halt system execution and save computation time. In Fig. 2.5a, it does not become visible whether a specific block contributes significantly (asymmetrically) to the error propagation (for example, by forwarding it in multiple paths), and, thus, insight is missed. It becomes clear, that in the given context, there are several disadvantages in considering the system as a whole compared to examining the subsystems separately.

To efficiently handle complexity, instead of considering the complete chain of subsystems during error propagation, each subsystem can be studied separately. In the general case, this requires that no feedback loops are present among the subsystems under study. Then, by applying appropriate inputs, namely inputs with the same characteristics as in the case that the whole chain of system blocks would execute together, correct output results can be derived at a lower complexity cost. In cases that the information and/or error signal can be statistically characterized (and potentially represented by a random process) at the input of the block, new ways of efficiently handling the complexity of propagation become available. Fig. 2.6 shows a projection of the system decomposition concept onto communication systems, which constitute the focus application domain for this work.

When decomposing the system into subsystems, a trade-off is present between reducing complexity by considering stand-alone blocks and increasing

complexity by handling information in a very fine-grain manner and having to re-integrate fine-grain information in the system context [61]. Therefore, a compromise is required. In our context, working at the level of groups of primitive operators—operators like multiplications and additions—achieves a good compromise and is compatible with using as error model the bit-flip. An additional criterion for deciding on the level of the decomposition is that the function of the blocks can be potentially analytically or otherwise mathematically modeled; such is the case, for example, with LTI blocks.

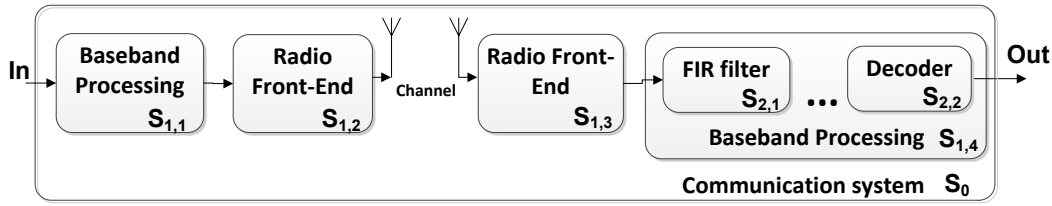


Figure 2.6: Example of splitting a communication system into subsystems

System functional model. *Simulation-based* techniques use a system model to propagate the errors. The model description can be closer to the hardware implementation (bit- and cycle-accurate e.g. using hardware description language models) or more abstract, depending on the type of information and accuracy required. The trade-offs involved among system models of different abstraction typically involve accuracy versus simulation time. As, in our case, the signal statistics are of interest, a bit-true functional system model is sufficient. A bit-true model uses the same number system and precision (like rounding or truncation) as the prototype hardware would. Simulations are a safe way to derive the necessary reliability data, as they can be applied to any type of system, but typically quite time-consuming; especially, due to the repetitive nature of error injection campaigns. In this work, simulation implementations constitute the reference implementations (as the most established means for such an analysis) and are performed in Matlab.

Analytical and other mathematical techniques are alternative-to-simulation approaches to calculate the signal statistics in a manner that the result is equivalent with the statistics that would be derived by performing system simulation(s); i.e. so that the signal statistical parameters are equal between the two approaches. For example, through analytical approaches, closed-form expressions can be used to derive certain statistical parameters when linear transformations take place. Using such an approach is typically much more time-efficient compared to performing system simulation, but can be only applied to systems with specific characteristics (i.e. in the aforementioned example, systems with linear operators); therefore, sometimes a combination between simulation-based and mathematical-transform-based methods can be the optimal solution. In this work we employ

analytical and other mathematical methods to speed-up the computations required in fault injection campaigns. Some of these techniques have been used since a long time in the domain of analyzing the effect of quantization noise [94, 54] but cannot be reused in a similar way due to the inherently different nature of hardware-induced errors (see Sec. 2.5 for more details).

2.4 Framework overview

To handle the simulation complexity in communication systems, stochastic analysis is a promising direction. According to the stochastic approach, the system's response is modeled in a statistical way. This can be achieved in any system by gathering sufficient data from the results of simulations based on random system states and input sequences. However, when, on top, the signal is considered as a stochastic process (i.e. a non-countable infinity of random variables), a number of tools become available which allow the representation and propagation of such data through system models. The signal then can be completely characterized by the PMF and ACF. Therefore, having these two functions available inbetween system blocks provides us with all the necessary information to derive the final system output metrics. If needed, random data can be generated according to the given PMF and ACF, using techniques, like the copulas [21].

Fig. 2.7 shows a conceptual illustration of the framework, within which, the proposed approaches, presented in the following chapters, operate. An example system with two inputs *In1*, *In2* and one output *Out* is composed by a mixture of LTI and non-LTI blocks. The error signal is generated and propagated through the various blocks until it reaches the system output. The error signal, characterized by the PMF and ACF, can be propagated through the non-LTI blocks by performing simulation (random data can be generated by using techniques like copulas) and through the LTI blocks using analytical and other mathematical methods. The latter is the focus of this study. When the error signal reaches the output, appropriate error metrics can be derived exploiting the PMF or the ACF. In summary, the components that synthesize the proposed error propagation framework are the following:

- The system is de-composed into subsystems and blocks. Each of these can then be examined as a stand-alone subsystem/block, assuming no closed loops. The propagation of the information can occur through a combination of simulation for non-LTI blocks and analytical and other mathematical techniques for LTI blocks (as a way to achieve computational speed-up).
- Statistics is an effective way to represent the information at the input and output of the block under investigation for communication systems as statistical metrics are of interest anyhow. On top, if the signal is represented as a stochastic process, analytical and other methods become available for the propagation through the LTI blocks. The error signal can be represented by

OVERVIEW OF THE ERROR PROPAGATION FRAMEWORK

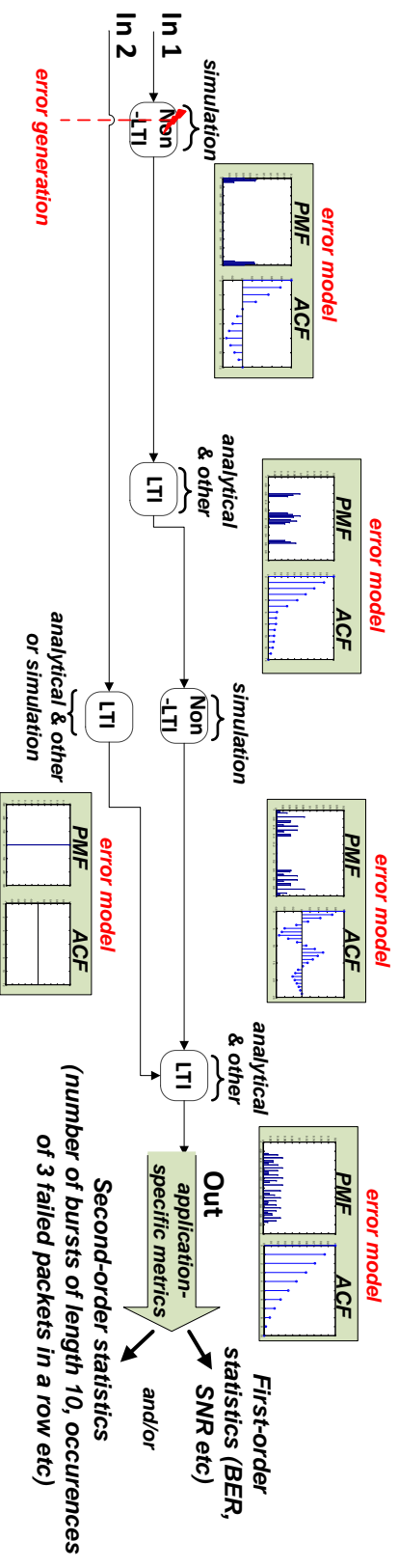


Figure 2.7: Error signal generation and propagation through the system blocks using a mixture of techniques

both the PMF and the ACF as these characterize fully a stochastic process.

- The error metrics of interest at the system output are application-specific, and typically include first-order statistical metrics, like the BER, and/or second-order statistical metrics, like a sequence of missed packets of a specific length. PMF and ACF are required to cover both types of metrics.
- To construct an error model at the input of a block, an increased error injection percentage may be assumed in order to accelerate the procedure of gathering sufficient statistics.

2.5 Correlation between the error and error-free signal and comparison with noise sources

As mostly done in literature, a bit-flip error model is assumed here. These models describe the probability of a bit-flip from 0 to 1 or 1 to 0 at a specific bit position in the digital signal samples. The bit-flip probability for bit position ν with weight $\pm 2^\nu$ is denoted by π^ν . In case of a 0-to-1 flip in bit position $0 \leq \nu < n - 2$ the sample value is corrupted by an error of $+2^\nu$ and a 1-to-0 flip yields an error of -2^ν . For bit position $\nu = n - 1$ (i.e. the sign bit in two's complement representation) the opposite signs have to be applied. The physically underlying mechanism leading to the bit-flips is also modeled as a random process, called *bit-flip mechanism* in this work.

Other sources of errors typically studied in signal processing and, more specifically, communication systems include the quantization noise (which manifests itself when the floating-point implementation of the system is turned into a fixed-point) and the noise that is introduced from the channel of the communication system. Both of these noise sources have been studied for a long time in the literature. All three sources of errors are additive and can be modeled as random processes; however, they have different characteristics. At this point the goal is especially to illustrate the differences of bit errors and the other two noise sources regarding their correlation with the information signal. For the illustration, a 20 kHz sinusoidal signal of amplitude 1 is used as reference; it represents the information signal before any errors are added.

Channel noise. Channel noise is the noise that is introduced on the information signal through the channel. In case the sources of noise are the electronic components and amplifiers at the receiver, the channel noise can be characterized statistically as a Gaussian process [93]. Actually, this model is the predominant one in communication system analysis and design. Information signal and channel noise are typically additive. Fig. 2.8 shows two periods of the reference error-free sinusoidal signal and the same signal after it has been corrupted by Gaussian channel noise with $\mu = 0$ and $\sigma^2 = 0.2509$. To improve the visibility, three red, vertical, dotted lines

split the graphs into four half-periods. The same splitting is performed in all figures of this section. Fig. 2.9 shows the channel noise for the corresponding two periods of

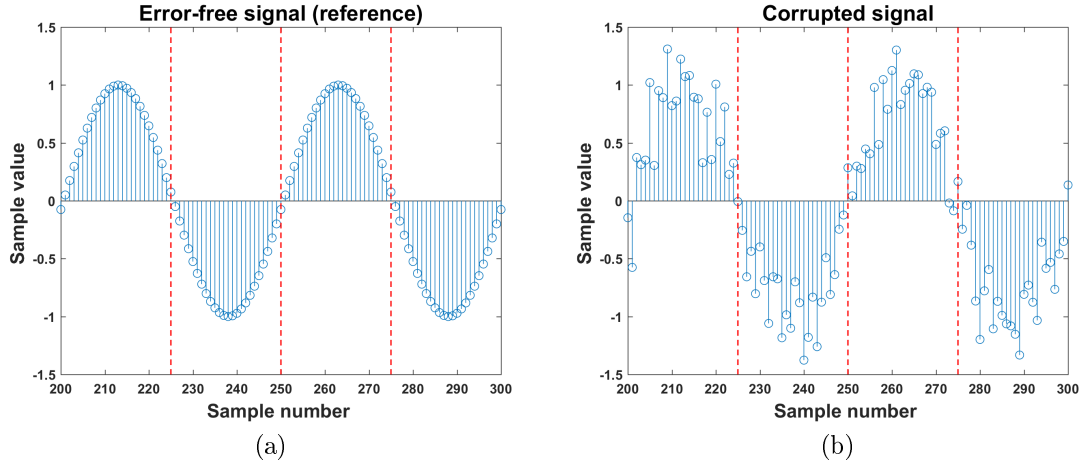


Figure 2.8: Two periods of the error-free (a) and the corrupted-by-channel-noise (b) sinusoidal signal

the sinusoidal signal, before and after the interaction with the sinusoidal signal. It is obvious that there is no difference between the two noise signals, i.e. the channel noise is independent of the information signal.

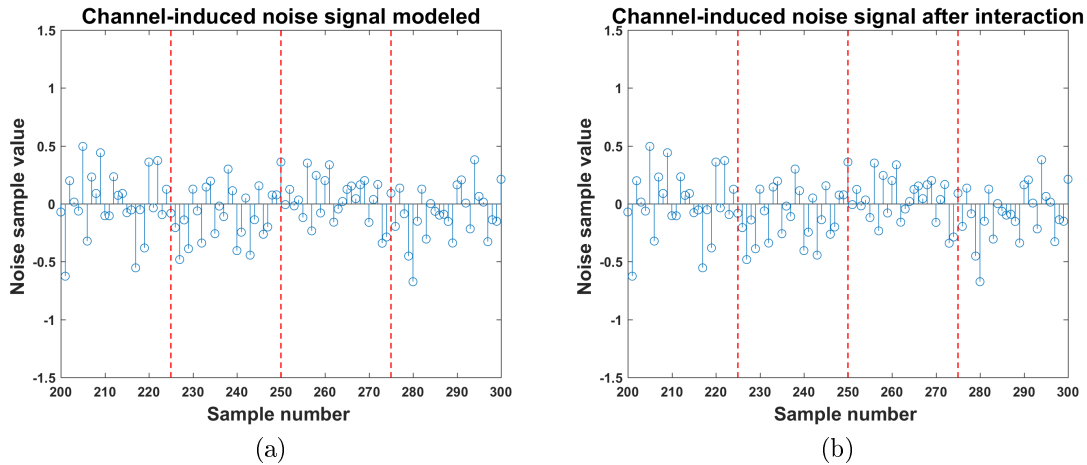


Figure 2.9: Channel-induced error signal (a) modeled and (b) resulting after the interaction with the error-free sinusoidal signal

Quantization noise. Quantization is the process of mapping values from a bigger (and possibly continuous) set of values to a smaller (discrete) set. In practice, for digital systems, this means either the mapping of analog values onto discrete

values (at the interface between analog and digital logic) or the mapping of words of a certain length onto words of a smaller length after some operation has taken place in the digital part of the system. Fig. 2.10 shows the discrete-time floating-point sinusoidal signal (in Fig. 2.10a) and the corresponding (quantized) fixed-point signal (in Fig. 2.10b). To derive the signal in Fig. 2.10b, the sample amplitudes of Fig. 2.10a have been rounded to the nearest level (more quantization modes are possible, but presenting those is unnecessary for the point being illustrated here) that can be represented by 8-bit words, with 1 bit representing the MSB and the other 7 being fractional bits. In Fig. 2.10b, the quantized values are represented with the red circles being printed on top of the non-quantized values (blue circles) so that the fine differences become more visible. The noise that is introduced during

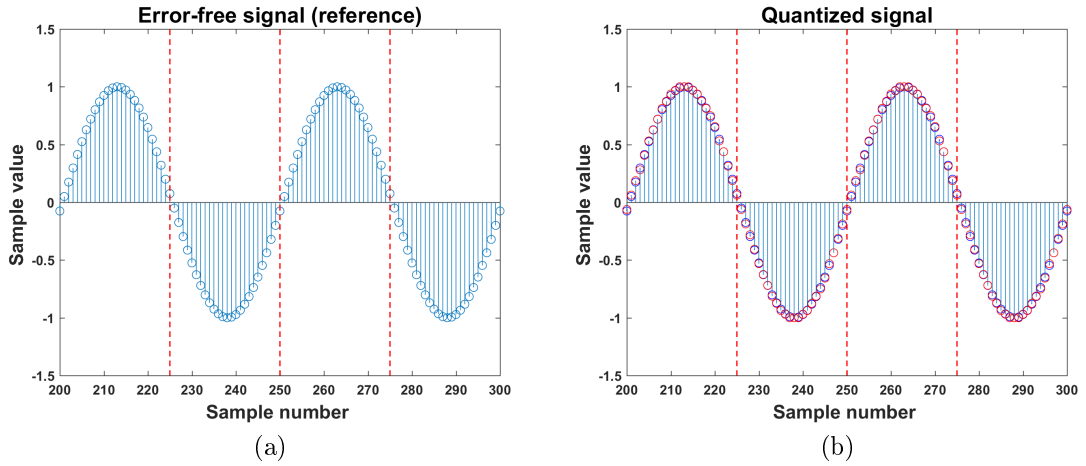
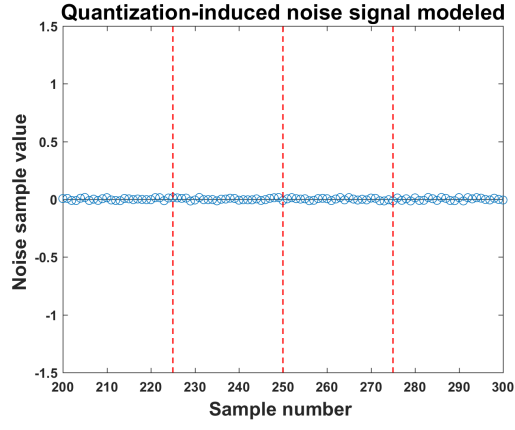


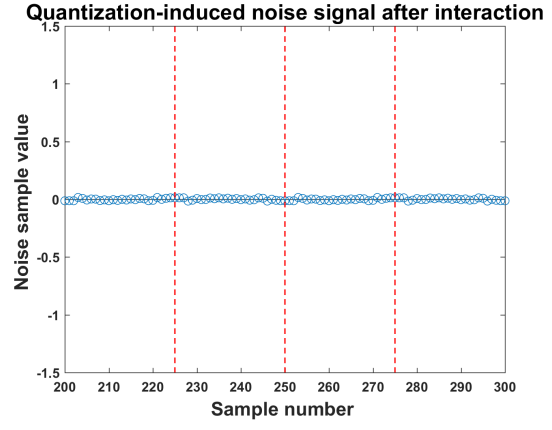
Figure 2.10: Two periods of the non-quantized (a) and the quantized (b) sinusoidal signal

this mapping can be seen in Fig. 2.11. The quantization error can be modeled as a random variable uniformly distributed in the interval $V = [-\frac{1}{2}2^{\nu_{min}}, +\frac{1}{2}2^{\nu_{min}})$, with $2^{\nu_{min}}$ being the weight of the Least Significant Bit (LSB) position [93]. Fig. 2.11a and Fig. 2.11b show the modeled and the actual noise values for the two periods of the sinusoidal signal under discussion.

Fig. 2.12a and Fig. 2.12b depict the same noise signal values with the *y-axis* magnified. In Fig. 2.12b it can be seen that in the actual noise values a small correlation with the information signal can be observed, in contrast to the modeled signal in Fig. 2.12a. This becomes visible by observing that the noise signal in the first and third half-periods of the graph (as divided by the red dotted lines) is identical. The same holds for the signal in the second and fourth half-periods of the graph. Very often, in quantization noise analysis, the noise signal is considered uncorrelated with the information signal and modeled by a uniform distribution. This holds in practice when the signal amplitude is large compared to the quantization step [49].

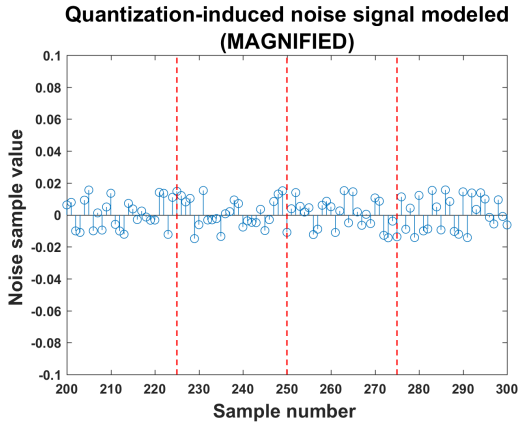


(a)

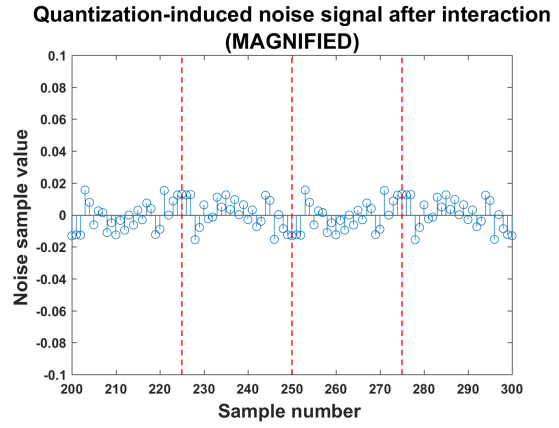


(b)

Figure 2.11: Quantization noise signal (a) modeled and (b) resulting after the interaction with the error-free sinusoidal signal



(a)



(b)

Figure 2.12: Quantization noise signal (a) modeled and (b) resulting after the interaction with the error-free sinusoidal signal

Bit errors. As a next step, the same reference signal is corrupted by bit errors. Once again, 8-bit words are considered with 7 fractional bits. To make the point more evident, an "exaggerated" corruption is performed. Every sample of the reference signal is being corrupted at one bit position, and every bit position has the same probability of being corrupted. Performing this process on the reference signal leads to the corrupted signal shown in Fig. 2.13b. In order to corrupt the

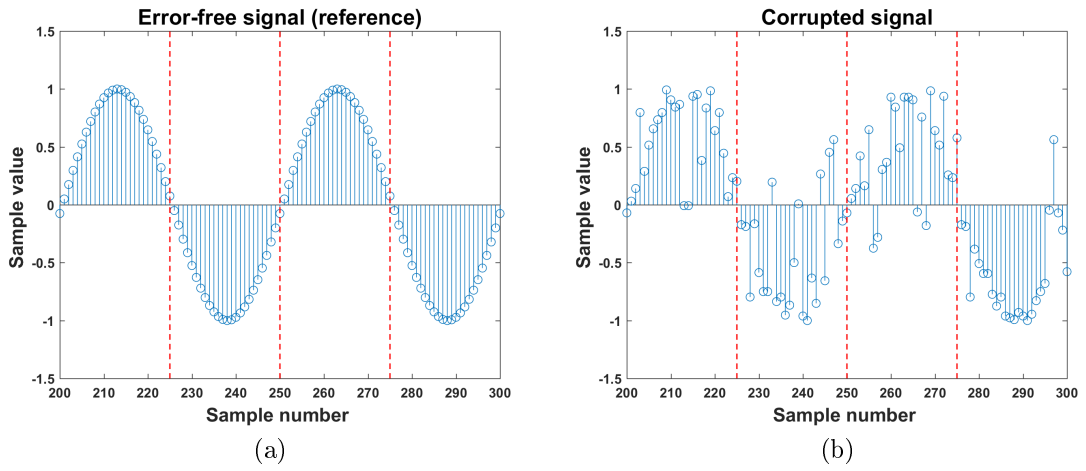


Figure 2.13: Two periods of the error-free (a) and the corrupted-by-bit-errors (b) sinusoidal signal

reference signal, a bit-flip signal is used (see Fig. 2.14a), as discussed in Sec. 2.3.1. After the bit-flip signal has interacted with the reference signal, the error signal is derived by subtracting the error-free signal from the corrupted signal, as discussed in Sec. 2.2.2. The result is shown in Fig. 2.14b. Due to the bit-flipping operation (see also example in Fig. 2.3), the error signal is, in general, correlated with the reference signal. In the example under discussion, this becomes especially visible by observing the values with the biggest magnitude (value equal to -1) in the bit-flip signal. These values correspond to bit-flips of the MSB in the samples. When these values are present in the first and third half-period, they keep their sign. However, when they are present in the second and fourth half-period, they change their sign.

As the error value depends on the bit value before flipping, in general, the error and the error-free signal are correlated when the errors occur due to hardware-induced bit-flips. This correlation is important because it has as a consequence that no well-known PMF can be expected for the error signal (as opposed, for example, to the PMF of quantization noise signal).

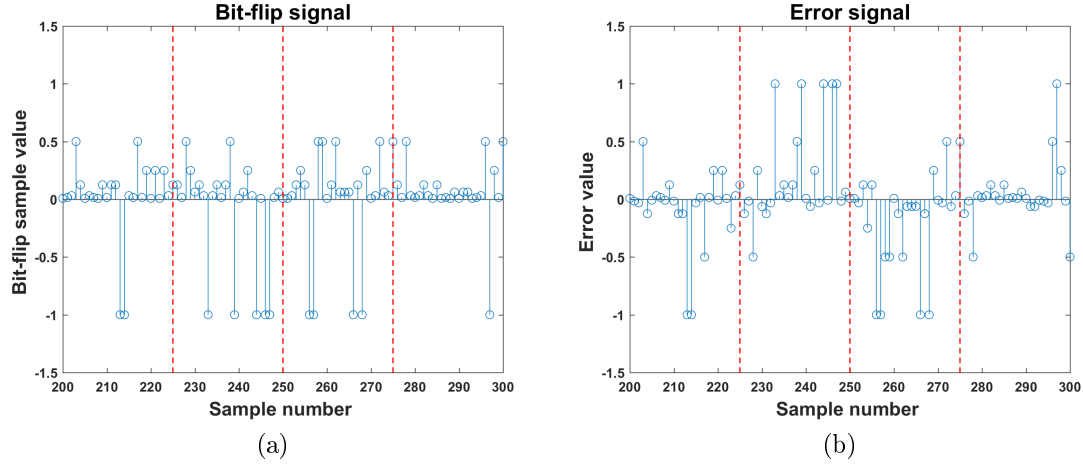


Figure 2.14: Bit-flip signal (a) and error signal (b)

2.6 Summary and link to next chapter

In this chapter, we presented the important elements that constitute the basis of our approach to address error injection and propagation experiments in an efficient way. System hierarchical decomposition allows to handle complexity and examine individual blocks separately. Using statistics and modeling the signal as a stochastic process provides analytical tools to propagate signals through systems in an efficient way. Especially LTI systems, which form an important class of modern systems, favor the use of analytical and other mathematical approaches for the propagation of statistical information and are the focus of the study. Information should ultimately be provided in a way that is compatible with the application-specific performance metrics, therefore, to cover all possible types of metrics, both PMF and ACF are required.

As a starting point and in order to build some insight, we considered the simplistic assumption that the error-free signal is modeled as a Gaussian distribution and errors are systematically injected. We discovered both possibilities and limitations. The results of the exploration are presented in the following chapter.

Chapter 3

Propagation of Corrupted, Uncorrelated Signal

3.1 Outline

Chap. 3 discusses the proposed approach for propagating a corrupted signal, when the underlying error-free signal features an uncorrelated PMF, using as driver the Gaussian PMF. In Sec. 3.2, preliminary information is provided on how Gaussian PMFs are affected by bit errors and how they behave under linear transformations. Based on this information, Sec. 3.3 illustrates the modeling concept and Sec. 3.4 presents experimental results and related work. Sec. 3.5 discusses limitations of the proposed approach. Sect. 3.6 concludes the chapter.

3.2 Motivation and preliminaries

To help designers realize more robust designs, insight of the system behavior in the presence of faults and errors is needed. As already discussed in previous chapters, this is typically achieved through system simulation, during which, errors are injected in the system and information regarding error statistics are derived. Lack of scalability is a major concern for such an analysis. An attractive alternative, especially for communication systems in which statistical metrics are being used, is to use analytical techniques in order to estimate the statistical properties of the signal after bit errors have been introduced and propagated through the rest of the system. This is especially true for LTI systems. Previous works have tried to address the challenge of scalability by propagating hardware-induced error statistics: In [59], the authors use closed-form expressions to propagate the signal power, causing a significant calculation speed-up but at the cost of accuracy. In [40, 39], the authors resort to more computationally demanding techniques, like the propagation of the characteristic function. As a potentially attractive option, in this chapter, the statistical moments are employed to propagate efficiently the signal statistics

after hardware-induced bit-flips have been injected. Possibilities and limitations are discussed. It is shown that such purely analytical approaches can have limited use in the context of estimating the impact of hardware-induced bit errors, due to the correlation that exists between the errors and the signal. To illustrate the concepts, the relatively simple case of the Gaussian PMF has been chosen but the approach is applicable to every type of uncorrelated signal. Special emphasis is given to the role of autocorrelation, since it limits the use of the analytical approaches.

3.2.1 Statistical characteristics of the considered signal

PMF. The target of the study is to generate a random signal at the output node of an LTI system, which accurately represents the propagation to the output of the input signal after it has been subjected to injection of bit errors (i.e. the produced corrupted output signal is statistically equivalent to the one that would be computed after performing the system simulation). Since no well-known PMF is to be expected for the error signal, the error and the error-free signal are explored together in this study. In order not to complicate the analysis further, simple PMF cases are chosen for the error-free signal (but not uniform PMF). It is known that the simplest PMFs are the Gaussian (or normal) processes as they are mathematically tractable and they are fully characterized by the mean μ and the standard deviation σ ; a random variable X that follows a Gaussian distribution is denoted as $X \sim \mathcal{N}(\mu_X, \sigma_X)$. Sometimes, in the notation, the variance σ_X^2 is used instead of the standard deviation; when that is the case, it will be clearly indicated. Especially for digital communication systems, in several cases the digital signal can be modeled by a Gaussian PMF or a PMF that can be decomposed into a number of Gaussian components : (i) For instance, in the case of M -ary amplitude shift keying (ASK) in the presence of Additive White Gaussian Noise (AWGN), the PMF of the received signal features M Gaussian components. Other examples occur in certain modulation schemes. (ii) In cases that processes are generated by the sum of multiple, independent RVs featuring PMFs of arbitrary shape. In this case, the Central Limit Theorem (CLT) applies [60]. The CLT states that if the RVs x_i are independent, under general conditions, the density of their sum $x_1 + x_2 + \dots x_n$, converges to a normal distribution as $n \rightarrow \infty$. The theorem does not hold if a small number m of the given densities are dominant. For example, the CLT applies at the output of specific digital signal processing blocks, like the Discrete Fourier Transform (DFT) [39].

Autocorrelation. In this context, autocorrelation can be present in the corrupted random signal in the following cases: (i) it is inherent due to nature of the physical process that produced the error-free random signal, (ii) it is acquired as the random signal propagates through the signal processing blocks (when these have memory), and (iii) it is acquired due to the error injection procedure. In this study, case (i) will not be considered (namely the inputs are considered uncorrelated), but the focus will be on cases (ii) and (iii).

3.2.2 Effects of error injection on Gaussian PMFs

As already discussed in Sec. 2.3.1, a bit-flip will increase or decrease the value of an error-free sample by a specific amount, which is decided by the position that the bit-flip occurs. When considering PMFs, it is however not appropriate to assume that all the sample values (amplitudes) within the PMF will behave the same way in the presence of bit-flips. That means calculating the new random variable after the bit-flip by simply adding or subtracting the error value to the mean μ of the distribution is not sufficient. This depends namely on whether the sample values have the same bit value (0 or 1) in the position where the bit-flip occurs.

Therefore, to create a reusable and accurate modeling basis, now the signal value and error interaction is analyzed, covering all realistic cases. Fig.3.1 illustrates the different effects of injecting a single bit-flip at different bit positions for $n = 8$ bits. Here, the PMFs are displayed through the use of histograms, which are created by experimental data; we operate under the assumption that the amount of data samples used to create the histograms is sufficient so that the presented histograms asymptotically approach the PMFs. For the error-free signal a

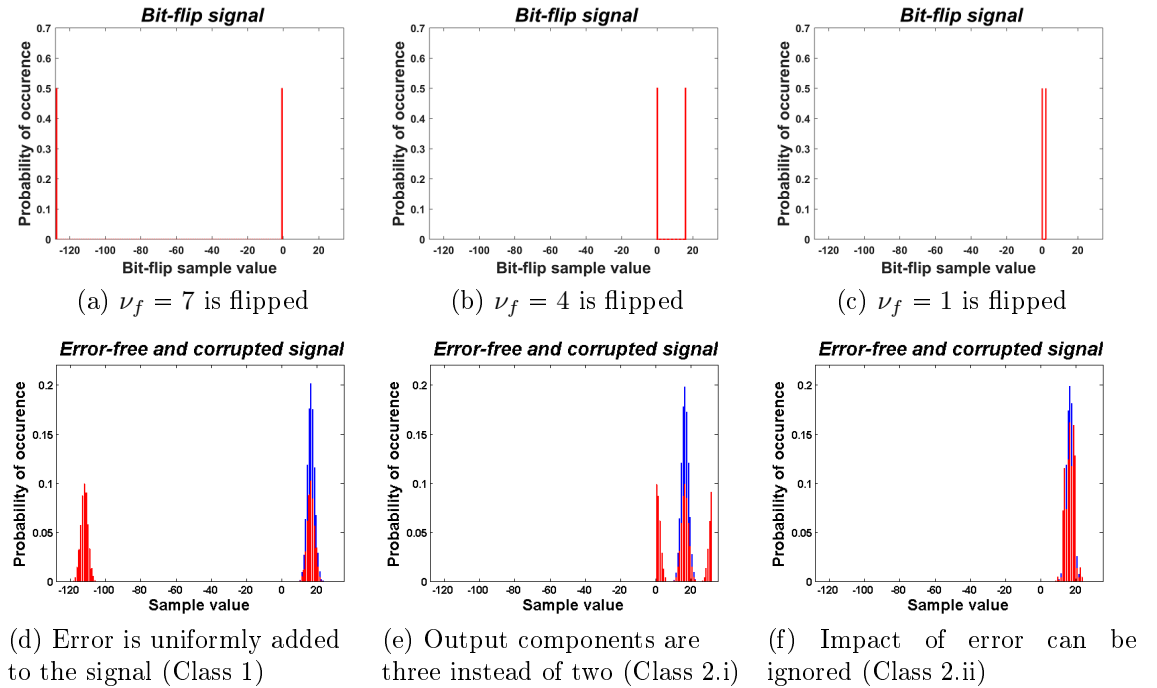


Figure 3.1: Bit-flip samples PMF and effect of single bit-flips on signal PMF.

distribution $X \sim \mathcal{N}(16, 2)$ is assumed. bit-flips are injected in $\pi^\nu = 50\%$ of the signal samples, i.e. π^ν denotes the percentage of corrupted samples. Fig.3.1a shows the PMF of the bit-flip signal for the bit-flip in bit position $\nu = \nu_f = n - 1 = 7$ (the MSB/sign bit). Fig.3.1b and Fig.3.1c show the bit-flip signal PMFs respectively for bit positions $\nu_f = 4$ and $\nu_f = 0$. In all three cases, 50% of the bit-flip samples

have the magnitude 0 since they correspond to the error-free signal samples. In Fig.3.1d-f, the blue histograms are the original data distribution, while the red ones show how the distribution is altered after the error injection. The following classes are identified, which span the complete range of cases:

- **Class 1: All sample values in the non-corrupted distribution feature the same bit in the bit-flip position ν_f so $x^{\nu_f} = \text{const.}$** This holds for the case depicted in Fig.3.1a and Fig.3.1d. As in the non-corrupted distribution (blue distribution in Fig.3.1d) all sample values are positive, and the MSB (sign bit) is flipped, $x^7 = 0 = \text{const.}$ holds. In case bit-flips would be injected in 100% of the samples, the whole distribution would be shifted by the weight of the sign bit $-2^7 = -128$. As here only 50% of the signal samples get corrupted, the distribution splits into two parts. One part (the 50% non-corrupted samples) is scaled by a factor of 0.5 (red distribution on the right-hand side in Fig.3.1d). The second part (the 50% corrupted signal samples) is scaled by a factor 0.5 and shifted by $-2^7 = -128$ (red distribution on the left-hand side in Fig.3.1d). In the case of an error-free sample distribution $X \sim \mathcal{N}(-16, 2)$ all sample values would be negative, so $x^7 = 1 = \text{const.}$ In this case a shift by $2^7 = +128$ would occur. Because for these examples the absolute values are relatively small, also bit flips in bit position $\nu_f = 6$ and $\nu_f = 5$ would belong to this class (as the bits in this positions are equal to the sign bit). Just the shift directions would be opposite, as the weights of these bit positions are positive.
- **Class 2: The sample values in the non-corrupted distribution do not all feature the same bit in the bit-flip position ν_f (i.e. $x^{\nu_f} \neq \text{const.}$).** This holds for the case depicted in Fig.3.1b and Fig.3.1e in which bit-flips in bit position $\nu_f = 4$ are applied (**Class 2.i**). Now the distribution has to be cut into two parts, one containing all sample values being smaller than $2^4 = 16$, i.e. $x^4 = 0 = \text{const.}$ and one with the larger sample values, i.e. $x^4 = 1 = \text{const.}$ Due to this, now the corrupted distribution consists of three parts. Again, one part is a scaled replica of the non-corrupted distribution (red one in the middle of Fig.3.1e), representing the 50% non-corrupted signal samples. For the corrupted signal samples, the scaled distribution is cut in one part representing all sample values smaller than 16, which gets shifted by $2^4 = +16$ (red distribution on the right-hand side in Fig.3.1e). The other part, representing all sample values equal or larger than 16 gets shifted by $-2^4 = -16$ (red distribution on the left-hand side in Fig.3.1e). The term *pseudo-Gaussians* will be used for these neighbouring parts. In the next lower bit-flip position $\nu_f = 3$, the corrupted part of the distribution is cut into four such pseudo-Gaussians. These are organized in two groups: one group featuring the bit pairs $x^4, x^3 = 0, 0 = \text{const.}$ and $x^4, x^3 = 1, 0 = \text{const.}$ which gets shifted by $2^3 = +8$ and one with bit pairs $x^4, x^3 = 0, 1 = \text{const.}$ and $x^4, x^3 = 1, 1 = \text{const.}$ getting shifted by $-2^3 = -8$. Finally, in the lowest bit-flip position $\nu_f = 0$ in the corrupted part of the distribution, the bit-wide bins get flipped pairwise.

A special subcase of the Class 2, (Class 2.ii) occurs when the error is

negligible compared to the signal value (what is acceptable can be case-specific decided by the designer). Such an example happens when the bit error occurs on the LSB of the word and the signal amplitude range is clearly larger. Then, signal and error can be considered uncorrelated and the errors are a small additive noise (similar to what quantization noise is typically considered [94]) despite the fact that signal amplitudes may not have the same bit value (0 or 1) in the position where the bit error occurs. So, again adding/subtracting the error magnitude to the mean μ of the input distribution is accurate enough (see Fig.3.1c and Fig.3.1f).

3.2.3 Linear transformation of normal random variables

Weighted sum of normally distributed random signals. Assume two random digital signals x_1 and x_2 with samples x_{1k} and x_{2k} being independent Normally or Gaussian distributed RVs (NRVs) with means μ_{X_1}, μ_{X_2} and variances $\sigma_{X_1}^2, \sigma_{X_2}^2$. Then, the weighted sum signal y with samples $y_k = c_1 \cdot x_{1k} + c_2 \cdot x_{2k} \forall k$ and constant coefficients c_1, c_2 again is a random digital signal. Its samples also are being NRVs with $Y \sim \mathcal{N}(\mu_Y, \sigma_Y^2)$ and $\mu_Y = c_1 \cdot \mu_{X_1} + c_2 \cdot \mu_{X_2}$, $\sigma_Y^2 = (c_1 \cdot \sigma_{X_1})^2 + (c_2 \cdot \sigma_{X_2})^2$. The PMF of y is a Gaussian "bell shape" resulting from the convolution of the bell-shaped PMFs of x_1 and x_2 .

In general, even when dependence or correlatedness is not absent, linear combinations of m normal distributions produce a normal distribution again. Consider m normally distributed RVs X_i with $i = 1, 2, \dots, m$, where $\mu_{X_i} = \mu_i$ and pairwise covariance $\sigma_{ij} = \text{cov}(X_i, X_j)$ with $i, j = 1, 2, \dots, m$. Then, the random variable $Z = \sum_{i=1}^m c_i \cdot X_i$ will have a mean $\mu_Z = \sum_{i=1}^m c_i \cdot \mu_i$, and a variance given by

$$\sigma_Z^2 = \sum_{i=1}^m \sum_{j=1}^m c_i \cdot c_j \cdot \sigma_{ij} \quad (3.1)$$

$$\sigma_Z^2 = \sum_{i=1}^m c_i^2 \cdot \sigma_{ii} + \sum_{i=2}^m \sum_{j=1}^{i-1} 2 \cdot c_i \cdot c_j \cdot \sigma_{ij} \quad (3.2)$$

where $\sigma_{ii} = \text{Var}(X_i)$ is the variance of X_i and $\sigma_{ij} = \text{Cov}(X_i, X_j)$. In case the pairwise covariances are not provided upfront or calculated analytically, the covariance of X_i, X_j can be estimated from n samples X_{ik}, X_{jk} (where $k = 1, 2, \dots, n$ is the time index) of the random digital signals X_i, X_j as $\text{Cov}(X_i, X_j) \approx \frac{1}{n} \cdot \sum_{k=1}^n (X_{ik} - E[X_i]) \cdot (X_{jk} - E[X_j])$. In case also the expectations $E[X_i], E[X_j]$ are not provided upfront or calculated analytically, they can be estimated by using the averages $E[X_i] \approx \frac{1}{n} \cdot \sum_{k=1}^n X_{ik}$, $E[X_j] \approx \frac{1}{n} \cdot \sum_{k=1}^n X_{jk}$ and to obtain an unbiased

estimation of the covariance, the formula can be adapted as $Cov(X_i, X_j) \approx \frac{1}{n-1} \cdot \sum_{k=1}^n (X_{ik} - E[X_i]) \cdot (X_{jk} - E[X_j])$ [37].

FIR filtering of normally distributed random signals. In the arithmetic part of a FIR filter such a weighted sum operation is performed on its Gaussian distributed input samples delayed by a tapped delay line. As long as there is no autocorrelation in the random input signal again all the samples at the output of the taps before the adder are independent NRVs (at lag 0 where the addition takes place). The output signal (after the sum) is composed of samples that are NRVs but its spectrum is not white anymore. The white input spectrum gets colored by the FIR's frequency transfer function and the output signal features autocorrelation (with N non-zero lags in case of a N -tap FIR). This autocorrelation is due to the fact that N consecutive output samples y_k, \dots, y_{k+N-1} share contributions from the same input sample x_k . FIR filtering of such a random signal with Gaussian distribution featuring autocorrelation again yields an output signal with Gaussian distribution and autocorrelated samples. This can be easily seen from combining the two filters into a larger filter realizing the total transfer function.

3.3 Propagating the corrupted signal using statistical moments

3.3.1 Set-up and observation

Summary of choices. In summary, to explore possibilities and limitations of using analytical techniques for the propagation of a corrupted signal through LTI blocks, the following choices have been made: The signal is characterized by a ***Gaussian*** distribution, since Gaussian distributions are mathematically tractable and can be completely ***characterized by the mean and the variance***. ***Linear combinations*** of Gaussian distributions produce Gaussian distributions. ***Propagating the mean and the variance*** through an LTI block can be easily implemented when knowing the block path function (which can be directly obtained by knowing the ***block signal flow***). Considering an ***uncorrelated*** signal allows the calculation to be simpler (the second term of Eq. 3.2 in Sec. 3.2.3 needs not be calculated). Finally, adding multiple independent random variables leads to a Gaussian distribution independent of their original PMF, due to the ***CLT***.

Effect of filtering on corrupted Gaussian signal. To illustrate the principles of the modeling a simple *2-tap* FIR filter will be used as a driver, with wordlength $n = 8$ (integers), and coefficients $c_1 = 1$ and $c_2 = 2$. As soon as an error signal exits the filter, the next error is injected (i.e. $\pi^\nu = 50\%$ in this case). Fig.3.2 illustrates the synchronization scheme and the parameters. The input error-free signal is a random variable $X \sim \mathcal{N}(\mu_X = 16, \sigma_X = 2)$:

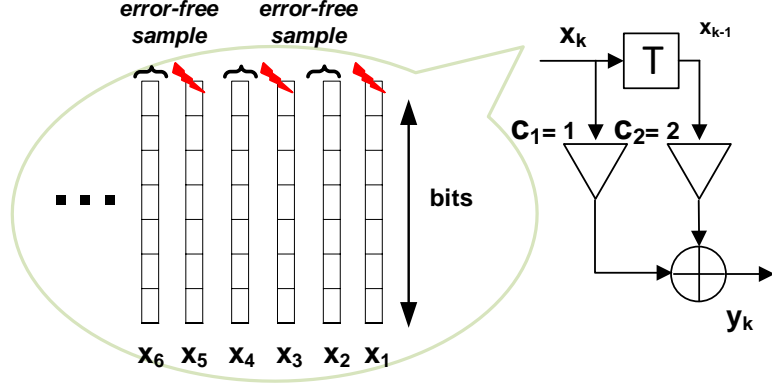


Figure 3.2: Error injection in simple signal-flow (2-tap FIR filter)

- As a start, only the MSB of the corrupted samples will be flipped ($\nu_f = 7$). For the distribution $\mathcal{N}(16, 2)$ all the fixed-point representations of the signal samples have the bit value 0 at the MSB position. So this case belongs to Class 1 (see Fig. 3.1). That means that the error amplitude can be added to the mean of the distribution. As the filter entails sequential logic, the corrupted sample will keep on corrupting the output for as long as the filter length, as illustrated in Table 3.1. When the MSB is flipped in the 8-bit word an error of -128 is obtained, on top of the error-free signal $\sim \mathcal{N}(16, 2)$, leading to $\sim \mathcal{N}(\mu_{\tilde{X}} = -128 + 16 = -112, \sigma_{\tilde{X}} = 2)$. When a random variable belonging to this distribution passes through the first tap, at the same time step, a random variable that belongs to the error-free distribution passes through the second tap. Their sum will appear at the filter output leading to random variable belonging to a distribution $\mathcal{N}(\mu_{\tilde{X}} = -112 + 2 \cdot 16 = -80, \sigma_{\tilde{X}} = \sqrt{2^2 + (2 \cdot 2)^2} = 4.47)$ (see first row of Table 3.1). In the table, the corrupted distribution is highlighted in grey and the error-free distribution in yellow. When, at the second time step, the random variable belonging to the corrupted distribution further propagates through the filter to the second tap, and after adding again the error-free signal centered around 16, the output distribution becomes $\sim \mathcal{N}(\mu_{\tilde{X}} = 16 + 2 \cdot (-112) = -208, \sigma_{\tilde{X}} = \sqrt{2^2 + (2 \cdot 2)^2} = 4.47)$ (see second row of Table 3.1). Fig. 3.3 shows the signal histogram (in blue) when Gaussian-distributed data have been generated, the bit position $\nu_f = 7$ has been corrupted and the data have been propagated through a 2-tap filter. Especially, in Fig. 3.3b, the theoretically expected PMF has been drawn (with a green line), as calculated in Table 3.1. By visual inspection, it can be observed that there is a very good match between the experimental and theoretical distributions.

- Now, the other representative case of Fig. 3.1 will be discussed. Namely, the input signal is corrupted at bit position $\nu_f = 4$ and, therefore, this case belongs to Class 2. As already seen in Fig. 3.1e, the corrupted input PMF has three

Table 3.1 Propagation of distribution $\mathcal{N}(16, 2)$ through 2-*tap* FIR filter when $\nu_f = 7$ is flipped

time step	x_k	x_{k-1}	y_k
k	$[\sim\mathcal{N}(-112, 2)]$	$2 \cdot [\sim\mathcal{N}(16, 2)]$	$[\sim\mathcal{N}(-80, 4.47)]$
$k + 1$	$[\sim\mathcal{N}(16, 2)]$	$2 \cdot [\sim\mathcal{N}(-112, 2)]$	$[\sim\mathcal{N}(-208, 4.47)]$

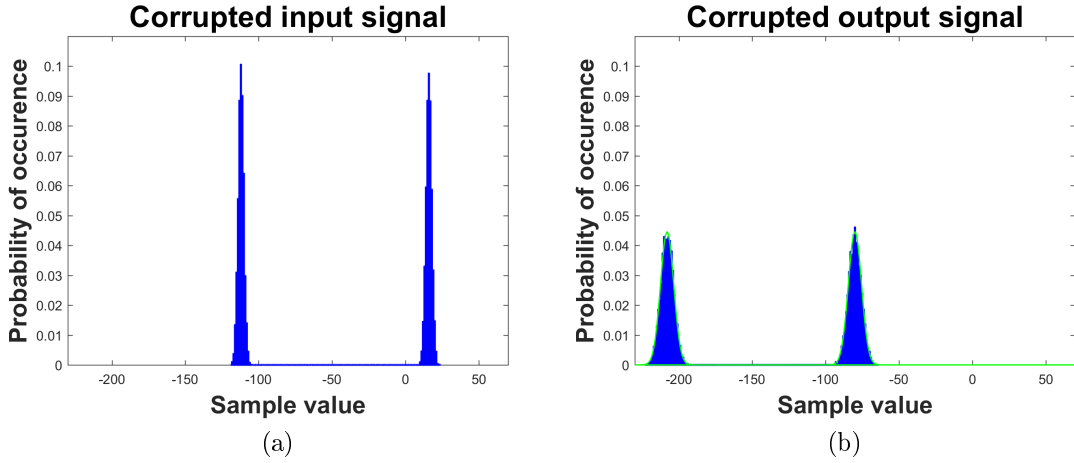


Figure 3.3: PMF at the filter input and output when $\nu_f = 7$ is corrupted at the input (2-*tap* filter)

Table 3.2 Propagation of distribution $\mathcal{N}(16, 2)$ through 2-tap FIR filter when $\nu_f = 4$ is flipped

time step	x_k	x_{k-1}	y_k
k	$[\sim\mathcal{N}(0, 2)]$	$2 \cdot [\sim\mathcal{N}(16, 2)]$	$[\sim\mathcal{N}(32, 4.47)]$
	$[\sim\mathcal{N}(31, 2)]$	$2 \cdot [\sim\mathcal{N}(16, 2)]$	$[\sim\mathcal{N}(63, 4.47)]$
$k + 1$	$[\sim\mathcal{N}(16, 2)]$	$2 \cdot [\sim\mathcal{N}(0, 2)]$	$[\sim\mathcal{N}(16, 4.47)]$
	$[\sim\mathcal{N}(16, 2)]$	$2 \cdot [\sim\mathcal{N}(31, 2)]$	$[\sim\mathcal{N}(78, 4.47)]$

components. One component that consists of error-free samples and is Gaussian-distributed, as earlier, and two more components appearing as pseudo-Gaussians, that have emerged due to the corruption of the signal. In the filter, at every time instant, when one error sample goes through one tap, an error-free sample goes through the other. In this case, the error sample can come from either of the two pseudo-Gaussians. Assuming the two pseudo-Gaussians were real Gaussians, then using the same reasoning as in the previous case, one would expect that the filter output components in this case would be four (two for each input combination of pseudo-Gaussian and error-free Gaussian combination that goes through the filter). These two parts can be approximately represented by a normal distribution and the mean of this pseudo-Gaussian can then be aligned with the position of their most frequently occurring values (in this case value 0 for the first part and value 31 for the second part). In addition, we have a weight value which indicates what portion of the original distribution they represent. Each of these two values will then create 2 distinct pseudo-Gaussians and because of the 2 taps, they will create 4 Gaussian components at the output. The moments of the Gaussians at the output are shown in Table 3.2. Implementing this approximation to model the output of the filter leads to the result shown in Fig. 3.4. The 4 components are approximated by Gaussians and shown in green while the simulated data is shown in blue. It is visible now that although the blue components have been smoothed out due to the filtering, there is some deviation from the modeled PMF. However, here is where the CLT can help. As in practice a filter has more taps, the filter output will seldomly be composed by components that are so far from each other.

As a matter of fact, Fig. 3.5 shows the same injection being performed but the filter is now composed of 5 taps instead of 2. The filter coefficients have the values: 1, 2, 3, 4, 5. The input PMF is slightly different than in the earlier case, due to the fact that now 1 out of 5 samples is corrupted instead of 1 out of 2. Due to the 5 taps, we have now in total 10 Gaussian components being created at the output (in green). The total output PMF will be computed by adding these individual PMF components (in the figure the y-axis has been scaled up to improve the visibility). The result is shown with the red line. We can see now there is a good match with the simulated data (in blue) despite the approximation with the pseudo-Gaussians. Due to the CLT, the approximation of the output components

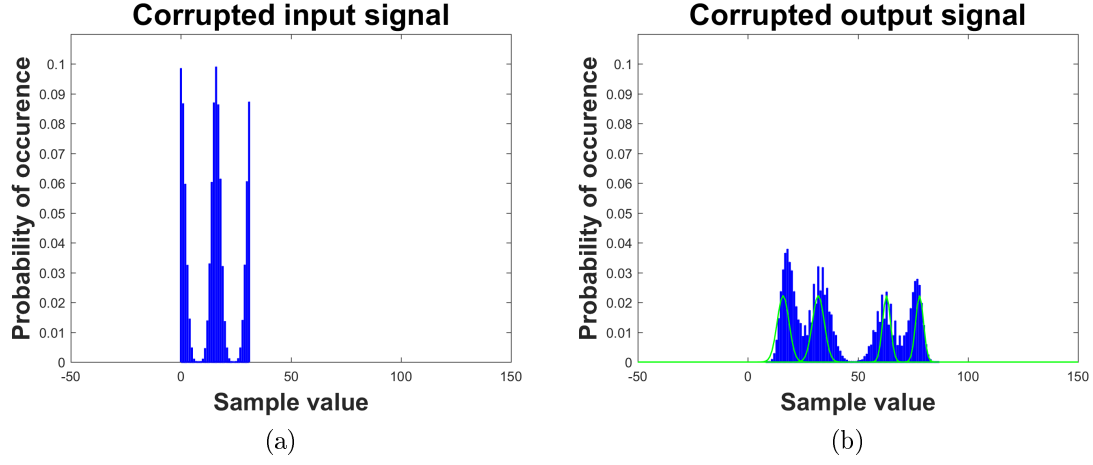


Figure 3.4: PMF at the filter input and output when $\nu_f = 4$ is corrupted at the input (2-tap filter)

as Gaussians is a good match. This observation forms the basis of our approach.

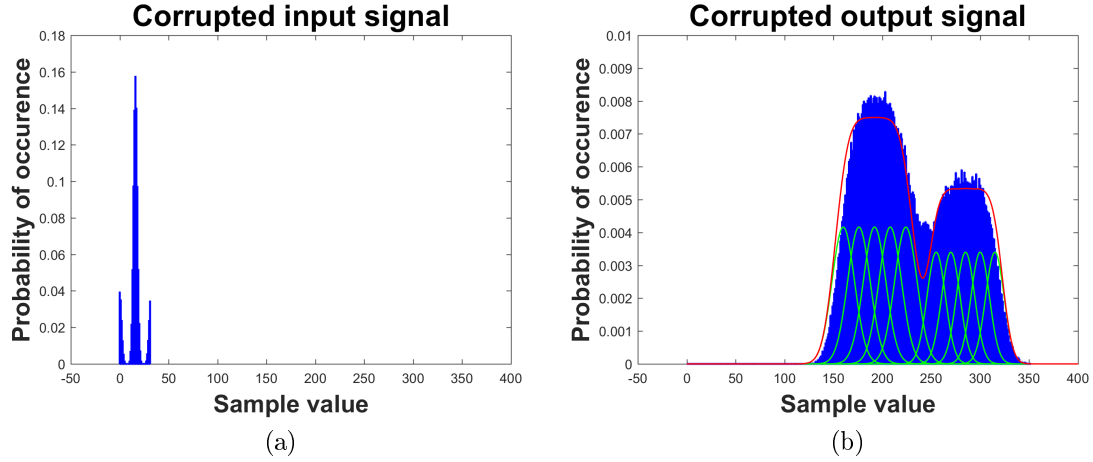


Figure 3.5: PMF at the filter input and output when $\nu_f = 4$ is corrupted at the input (5-tap filter)

3.3.2 Modeling approach

To summarize, in order to model the pseudo-Gaussians at the input of the LTI block, the following choices are made:

- Proportions w_i of a normal distribution (corrupted components) are

Procedure 1 Pseudocode of the proposed methodology for the propagation of corrupted uncorrelated signal through an LTI block

Input: error-free signal, bit-flip signal

Output: PMF at the output of the LTI block

```

1: initialize global vector to store final PMF
2: cut the input PMF into pseudo-Gaussians, i.e. input pseudo-Gaussians
3: for each input pseudo-Gaussian do
4:   for each error combination do
5:     bitwise xor the mean with the error sample
6:     replace corrupted value in the LTI signal-flow according to the injection scenario
7:     propagate means and standard deviations to the output according to the signal-flow
8:     create normal distribution (PMF) according to the derived mean, standard deviation, weight
9:     add (each) output PMF to the global vector
10:   end for
11: end for

```

approximated with different normal distributions (weighted with w_i), different means and standard deviations.

- Every mean in the new normal distribution is approximated by the most frequently occurring sample.
- The standard deviations are approximated by the standard deviation of the original normal distribution weighted with w_i .

Combining these approximations with the LTI signal flow graph allows us to produce Gaussian components at the output of the LTI block, which, when added, provide the total output PMF. The proposed overall flow is shown in Proc. 1. Step 1 initializes and step 9 accumulates the global vector which stores the values of the output PMF across the entire signal sample range. Step 2 contains the splitting of the initial PMF into separate components that will be propagated through the signal flow. The main loops span the core of the approach: Step 5 implements the corruption of the mean of the corresponding pseudo-Gaussian. Step 6 replaces the corrupted mean into the signal flow of the LTI block. In this step, it is made sure, for example, that the corrupted random variables pass through a single filter tap at one time step, while error-free random variables pass through the rest of the taps, as in Tables 3.1, 3.2. Step 7 implements the actual propagation through the signal flow. E.g. the corresponding multiplications and additions take place. Finally, step 8 produces the normal PMF distributions from the main parameters derived in the loops.

3.4 Experimental results and related work discussion

3.4.1 Experimental set-up

To illustrate the effectiveness of the approach we use as LTI driver a filter, as typically found in the receiver of a BPSK communication line. The assumption is that errors occur at the memory buffer in front of the filter (as has been illustrated in Fig. 2.2) and the objective is to accurately estimate the PMF at the output of the filter. In the general case, the hardware-induced errors are correlated with the input data signal (as discussed in Sec. 2.5) and that means that in such an analysis the effect of the bit-flips should be shown on the values of the received data signal after channel noise has been added. However, under certain conditions, only the channel noise can be considered. Such conditions include that the added bit-flips are not correlated in time with each other (no burst errors), the PMF of the data signal is generally symmetric around its mean (that will result in the signal being flipped in the positive and negative direction with equal probabilities and such is the case with the BPSK symbols), no non-linearities are considered in the filter internals (such as quantization effects). Such a framework has been used in [40, 39] (although these conditions have not been mentioned) and we adopt the same idea. Fig.3.6 roughly illustrates the idea: Instead of considering the whole input signal (as typically done in simulation) only the channel noise is considered, bit-flips are added and the result can be added to the BPSK symbols separately. This has the advantage that the input signal becomes an uncorrelated Gaussian distribution, which very often models the channel noise in communication systems.

The two main objectives are: the accuracy of the modeled distribution compared to the "ideal" distribution and the run-time of the modeling approach compared to using simulation. We use as reference for the accuracy checks the simulation results using the FIR filter signal-flow for 10 million samples. To compare the accuracy, we use the Kullback Leibler divergence (KL) [43], which is a measure of the difference between two probability distributions (also applied for the same purpose by [40]), as presented in Sec. 2.2.2. The smaller the KL value, the closer the distributions are. Regarding run-time, in absence of a single objective metric, we use the Matlab *timeit* function which counts the elapsed computation time. All experiments have been performed on a single machine (Intel Core i7, 2.40GHz, 8GB RAM) under the same conditions (e.g. no other tasks running in parallel). We adopt $\mu=0$, $\sigma=0.562$, wordlength $n = 8$ (with 5 fraction bits) as used in [40] for the input signal. The filter coefficients have been derived so that the filter's impulse response is a sinc function. The filters have been implemented using *for loops*.

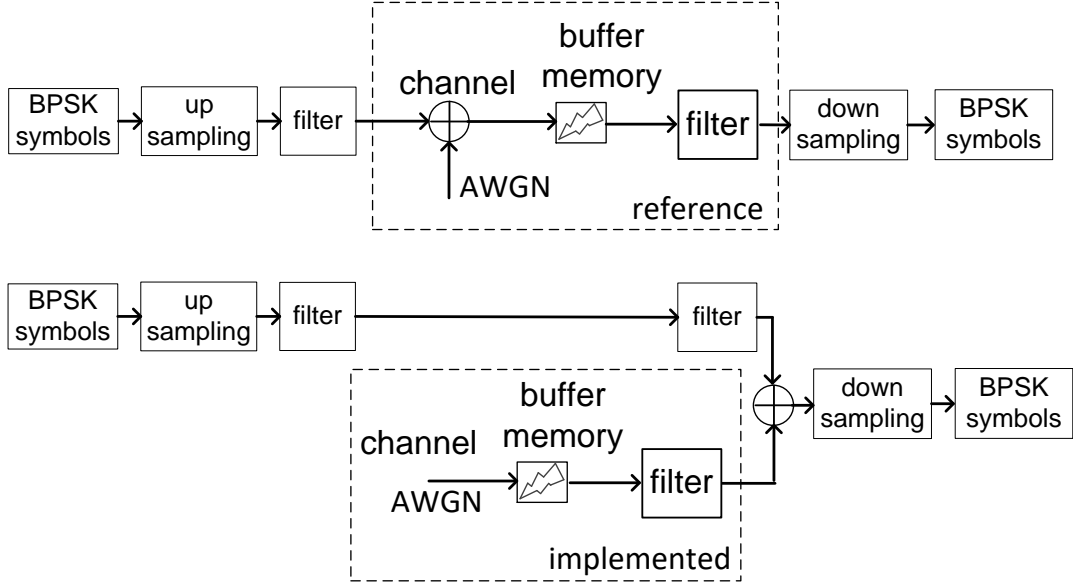


Figure 3.6: Illustration of reference set-up vs. the implemented set-up to efficiently propagate bit-errors on the noise distribution through the filter at the receiver

3.4.2 Results

Table 3.3 presents the accuracy and run-time results for three filters of different tap lengths: 16, 32, 64. As already discussed, the 10-million samples simulation results (can be seen at the fourth row) constitute our reference. The following three rows present results after using 3-, 2-, 1-million samples respectively (1-million samples is often considered as a representative simulation length). The last row presents the results from our model. Columnwise, first the accuracy results are presented. As KL compares the difference of PMFs, the results of the comparison of PMFs are shown, for the case when the MSB is corrupted and the LSB is corrupted (2 extremes). The case when the MSB is corrupted is expected to give the biggest differences since the disruption in the signal values maximizes in that case (the error magnitude the maximum possible). The last set of results regards the required execution time in seconds. In this case, the total time for performing the whole exploration (after corrupting all bit positions in the input data) is shown.

As it is to be expected, there is a trade-off between accuracy and execution time for the different simulation lengths. The 1-million-samples case is the fastest to execute but gives out the worse accuracy results, while the 3-million-samples case has the closest match with the reference but takes more time. Coming to the proposed work, the accuracy results are better than the simulation runs in the cases when the CLT can be exploited the most: (1) The first case concerns all the LSB corruptions as in this case the PMF stays closer together (becomes smoother)

instead of spreading out. That means that the initial PMFs after corruption look very much Gaussian-like. (2) The second case is the case when more filter coefficients are used. Then, there are more independent random variables being added at the filter output and CLT states that the density of the sum $x_1 + x_2 + ..x_n$ of random variables $x_1, x_2, ..x_n$ converges to a normal distribution as $n \rightarrow \infty$. This is the case for the 64-*tap* filter. The worst accuracy in the proposed work occurs for the MSB corruption of the 16-*tap* filter (as the CLT helps the least in that case).

Regarding timing, the proposed approach looks very efficient for the 16-, 32-*tap* filters but less so for the 64-*tap*. The latter is due to the bigger number of multiplications required to propagate the corrupted means through the filter signal flow. In contrast, the simulation approaches show almost no difference in the execution time among filter of different lengths for the same amount of samples. This can be explained by the vectorization in the *for-loops*, which counterbalances the extra memory costs of the longer vectors (in the bigger filter lengths) by the smaller number of iterations required for the same amount of samples. Overall, the proposed scheme looks the best compromise for the 32-*tap* filter, the best option for the 64-*tap* filter when accuracy is the objective (but at the cost of timing) and the best option for the 16-*tap* filter when timing is the objective (but at the cost of accuracy).

Table 3.3 Comparison for 16-, 32-, 64-*tap* filter (input signal: $\mu=0, \sigma=0.562$)

	<i>Accuracy (KL)</i>						<i>Exec.time (sec)</i>		
<i>Tap length</i>	<i>16</i>		<i>32</i>		<i>64</i>		<i>16</i>	<i>32</i>	<i>64</i>
<i>Corrupted bit</i>	<i>MSB</i>	<i>LSB</i>	<i>MSB</i>	<i>LSB</i>	<i>MSB</i>	<i>LSB</i>			
Ref.simul. ($10 \cdot 10^6$ samples)	-	-	-	-	-	-	121	121	125
Simul. ($3 \cdot 10^6$ samples)	44e-4	40e-4	54e-4	41e-4	226e-4	191e-4	35	35	37
Simul. ($2 \cdot 10^6$ samples)	71e-4	51e-4	69e-4	68e-4	325e-4	272e-4	24	24	25
Simul. ($1 \cdot 10^6$ samples)	130e-4	109e-4	158e-4	119e-4	728e-4	604e-4	12	12	13
This work	132e-4	3.5e-4	76e-4	3.8e-4	55e-4	16e-4	2	5	84

3.4.3 Related work

A limited number of contributions have developed analytical models to achieve a scalable but relatively accurate estimation of the impact of random hardware bit-flips on LTI system blocks [40, 59, 39]. It is important to note that none of these works is applicable for the case that non-linearities (such as quantization effects) are present in the filter internals.

More specifically, the authors in [40] propose a framework formalizing the case when the buffering memory introduces random uniform bit-flips (as a side-effect of reduced supply voltage) through shifting PMF values to the left and to the right (according to the bit error that is introduced). For the propagation of random bit-flips through the LTI block, the authors, instead of working directly with PMFs, utilize the Characteristic Functions (CFs), which are Fourier duals of PMFs. Then, scaled versions of the input CF (according to the filter impulse response) have to be multiplied with each other to get the output CF. Finally, the CF has to be transformed back to the original domain, so that the output PMF becomes available. This is an efficient way to propagate random bit-flips (that can occur at any bit location of the memory). This technique becomes less efficient for an exploration, such as the one implemented in this chapter. When the impact of a bit-flip at every bit position has to be explored separately, the technique implemented in [40] has to be adapted as follows:

For every bit position explored, a new corrupted CF has to be created. Random variables from each corrupted CF (there are as many as the wordlength of the input data) have to propagate through the signal flow of the filter. When a random variable from a corrupted CF enters the filter, there will be required as many time steps for it to leave the filter as the tap length. At each of these time steps, it will be scaled by one of the filter coefficients while random variables belonging to error-free CFs will go through the remaining of the filter taps. A new output CF will be created at each of these time steps by performing the corresponding multiplications. The PMF outputs (i.e. after transforming the CFs back to PMFs) have to be added to produce the final PMF. This process must be repeated for each corrupted CF. Essentially, this algorithm utilizes the same reasoning performed in the proposed technique of this chapter but instead of propagating only the statistical moments (means and variances), the whole CFs have to be propagated.

The work in [59] presents a technique to model the effect of single bit-flip errors (each bit position is examined separately similarly to our framework) in different nodes of an adaptive FIR filter. Objective of this investigation is to estimate the total noise power at the output of the filter. The noise power due to bit-flips is propagated through the rest of the LMS filter by multiplying it by the square of the magnitude of the frequency transfer function. In that sense, the spectral noise power is treated like a time sample and the frequency domain analysis is performed for the transient state of the system, rather than the steady state. Then, the calculated noise power is compared to the SNR that is required to receive bits at the receiver with a certain bit error rate plus the signal to noise ratio margin. If it is bigger, then the initially-flipped-node is identified as a vulnerable bit position. This analytical model provides a fast way to identify a set of vulnerable nodes in the system; however, it cannot be used for the estimation of the actual number of bit errors (since no PMF information can be obtained).

The work in [39] will be discussed later in the chapter.

3.5 Limitations with using the statistical moments

In the following, limitations of propagating analytically random signals through LTI signal flows for representative cases are discussed. An element that has not been discussed until now is the signal autocorrelation. Autocorrelation is introduced either by the signal processing block or by the error injection procedure. To illustrate this, we will use the easiest cases regarding the statistics of the error-free signal, as motivated earlier, by assuming: (i) a normal distribution for the error-free digital signal at the input of the LTI block and (ii) that the CLT applies due to signal processing at the output of the LTI block.

3.5.1 Propagating the corrupted signal through LTI blocks

3.5.1.1 Autocorrelation introduced by signal processing

- *Example 1: Linear transformation without memory*

In the following, a very simple, somewhat artificial data path (data path 1) example is used to illustrate the usage of analytical techniques. Fig. 3.7 shows the signal flow graph (SFG) consisting of two sections. In the first section, the input signal x is demultiplexed into two phases, each being decimated in time by a factor of two (which equals to a serial to parallel converter). One phase is composed of all input signal samples with even time index $x_{ek}; k = 2\kappa$ while the other one consists of all input signal samples with odd time index $x_{ok}; k = 2\kappa + 1$ with $\kappa \in \mathbf{Z}$. Let x be a random signal with samples being NRVs with $X \sim \mathcal{N}(\mu_X = 8, \sigma_X^2 = 4)$ and without any autocorrelation (i.e. "white").

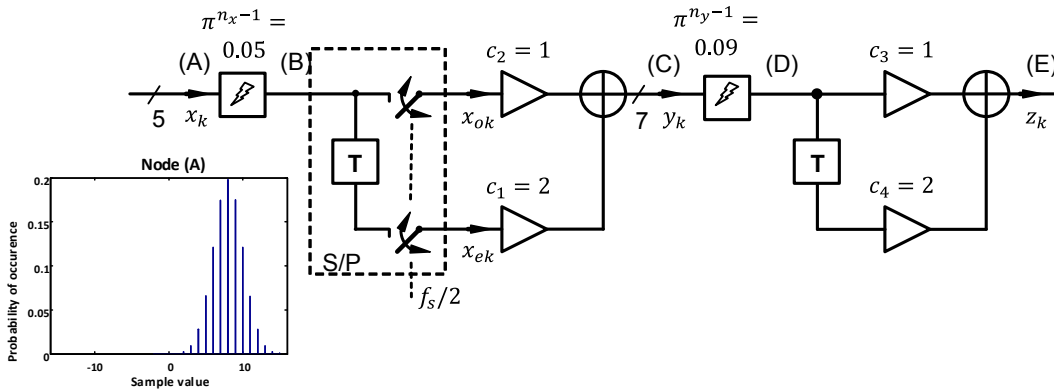


Figure 3.7: Data path 1

To facilitate the discussion, we generate this random signal in Matlab and derive the histograms (which correspond to the PMFs) at the different nodes as it propagates through the data path (bit-true, cycle-accurate simulation). The samples

of the two phases are independent NRVs with $X_e, X_o \sim \mathcal{N}(\mu_X = 8, \sigma_X^2 = 4)$, too. The rest of the first section performs a weighted sum operation $y_k = c_1 \cdot x_{ek} + c_2 \cdot x_{ok}$ with $c_1 = 2, c_2 = 1$ as described above. So, y_k is a white random signal with samples being NRVs with $Y \sim \mathcal{N}(\mu_Y, \sigma_Y^2)$ and $\mu_Y = c_1 \cdot \mu_X + c_2 \cdot \mu_X = 24, \sigma_Y^2 = (c_1 \cdot \sigma_X)^2 + (c_2 \cdot \sigma_X)^2 = 20$. The first section could implement the atomic part of a 2-point DFT (with the difference that in the DFT the coefficients are complex-valued). In the second section, y_k is 2-tap FIR filter according to the difference equation $z_k = c_3 \cdot y_k + c_4 \cdot y_{k-1}$ with $c_3 = 1, c_4 = 2$. The samples z_k are NRVs with $Z \sim \mathcal{N}(\mu_Z, \sigma_Z^2)$ and $\mu_Z = c_3 \cdot \mu_Y + c_4 \cdot \mu_Y = 72, \sigma_Z^2 = (c_3 \cdot \sigma_Y)^2 + (c_4 \cdot \sigma_Y)^2 = 100$. So far, only the error-free case has been investigated analytically. In the following this will be considered as **Case (I)**. Fig. 3.8 is composed of a number of graphs, organized as an array. They depict the PMFs of the signal at the different nodes of the data path (each row corresponds to one node). The PMFs for **Case (I)** can be seen in the first column.

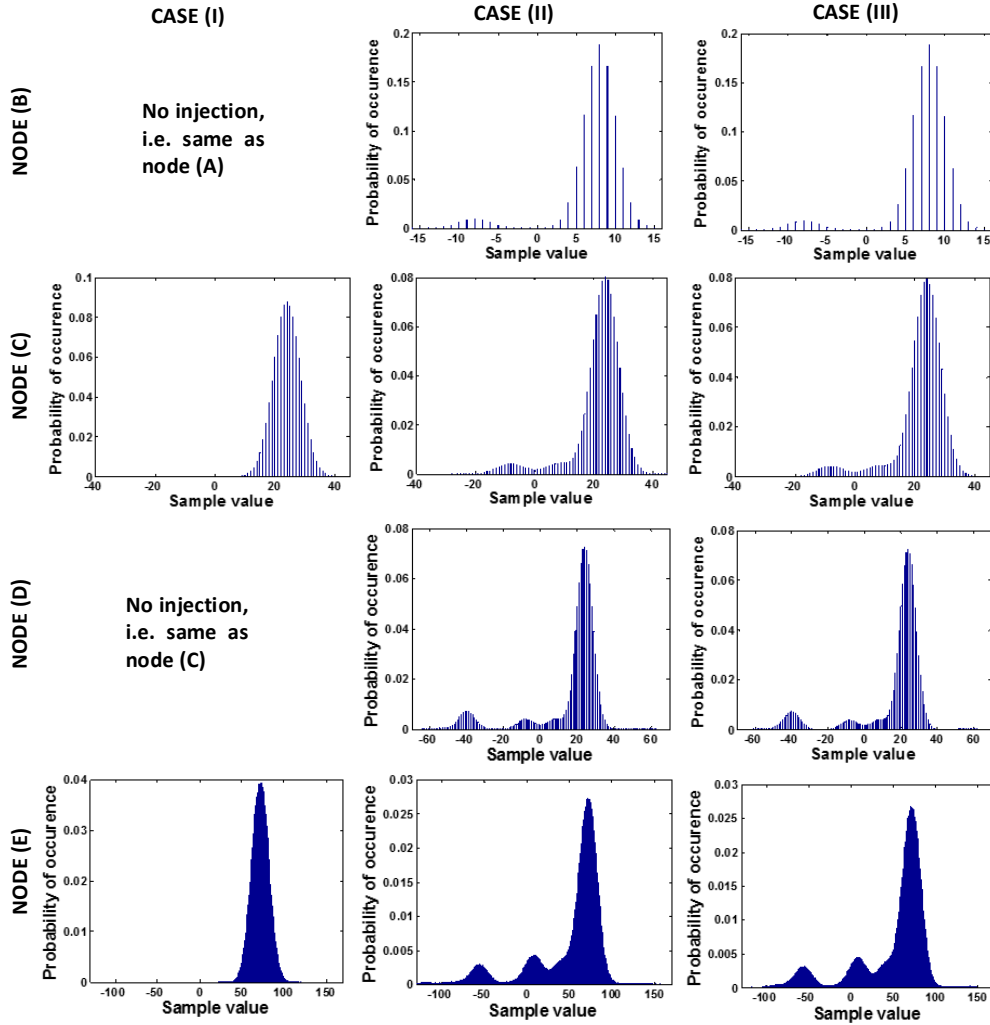


Figure 3.8: PMFs of the signal at the different nodes of the data path 1

Now, considered as **Case (II)**, the effects of error injection and propagation will be investigated analytically. In this case, we use Matlab to generate the random signal at the different nodes after calculating analytically the statistical moments and then we derive the corresponding histograms (which correspond to the PMFs). Namely, in this case *we do not simulate the propagation of the signal* through the data path. In the first section of the SFG, errors are assumed to be injected into the MSBs of the input signal samples x_k with bit-flip probability $\pi^{n_x-1} = 0.05$. With the assumption of $X \sim \mathcal{N}(\mu_X = 8, \sigma_X^2 = 4)$, the samples of the error-free input signal are positive, i.e. the sign-bit is equal to zero. For a wordlength of $n_x = 5$ bits this means that 5% of the samples are corrupted by an error of $-2^{n_x-1} = -16$. As described earlier, this results in a mixture distribution: Its first component represents the 95% error-free sample values with unchanged mean and variance $X \sim \mathcal{N}(\mu_X = 8, \sigma_X^2 = 4)$. In the PMF, the corresponding component is a scaled copy of the original PMF with a mixture weight determined by $(1 - \pi^{n_x-1})$. The second component represents the 5% corrupted sample values with modified mean and unchanged variance $\tilde{X} \sim \mathcal{N}(\mu_{\tilde{X}} = 8 - 16 = -8, \sigma_{\tilde{X}}^2 = 4)$. In the PMF, the according component is a scaled copy of the original PMF shifted by the (constant) error value -16 and a reduced mixture weight determined by the π^{n_x-1} . To facilitate the explanation, the creation of the different Gaussian components of the mixture distribution as the signal propagates through the flow is organized in a diagram, illustrated in Fig. 3.9. Every new level of splits corresponds to a different node. The percentages that are located on the branches correspond to the new components created by the injection procedure itself (and not due to the propagation). The different components are denoted by corresponding subscripts.

The two new components at node B can be seen at the first level of the graph. These components propagate through section one. Now, a new number of components will be created because of the different combinations at the input of section one. These combinations lead to different weighted sums: $(X, \tilde{X}), (\tilde{X}, X), (\tilde{X}, \tilde{X}), (X, X)$. As a result we have 4.75% \tilde{y} -samples with $\mu_{\tilde{y}_{1,1}} = c_1 \cdot \mu_{\tilde{X}} + c_2 \cdot \mu_X = 8$, 4.75% \tilde{y} -samples with $\mu_{\tilde{y}_{1,2}} = c_1 \cdot \mu_X + c_2 \cdot \mu_{\tilde{X}} = -8$, 0.25% with $\mu_{\tilde{y}_{1,1}} = c_1 \cdot \mu_{\tilde{X}} + c_2 \cdot \mu_{\tilde{X}} = -24$ (due to the low percentage this case can be omitted) and the remaining 90.25% y -samples with $\mu_Y = c_1 \cdot \mu_X + c_2 \cdot \mu_X = 24$. The corresponding percentages after removing the smallest, redistributing it proportionally to the others and rounding become 5%, 5%, 90%. The variances remain unchanged. These three new components which correspond to node C can be seen at the third level of Fig. 3.9.

Simultaneously to the error injections in section one, assume that errors are injected into the MSBs of the input signal samples y_k of section two with bit-flip probability $\pi^{n_y-1} = 0.09$. This time a case differentiation is required according to the three PMF components resulting from the first error injection:

For the biggest component, representing the yet error-free (i.e. positive) 90% of samples still all sign-bits are zero. Assuming a wordlength of $n_y = 7$ bits this means that 9% of the 90% yet error-free samples, i.e. in total $\pi^{n_y-1} \cdot (1 - \pi^{n_x-1}) \times 100\% = 8.1\%$, get corrupted now by an error of $-2^{n_y-1} = -64$.

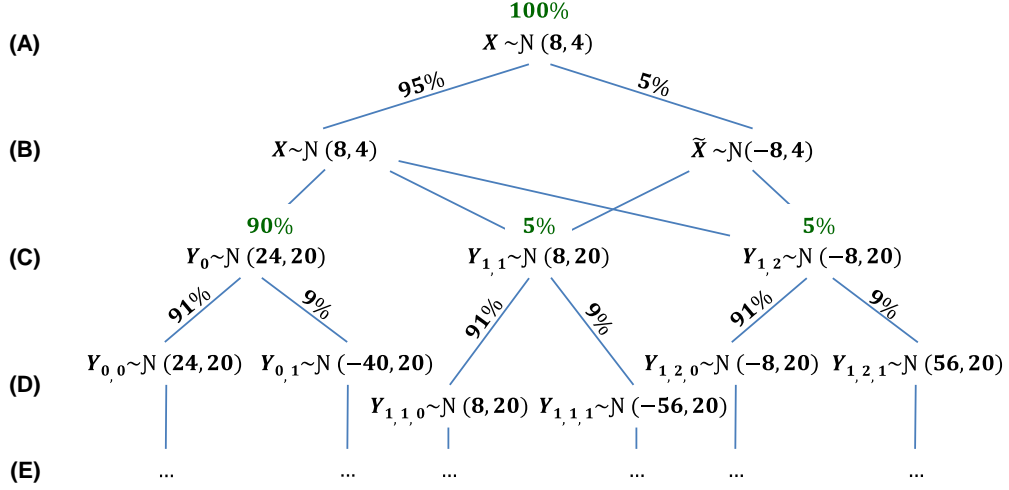
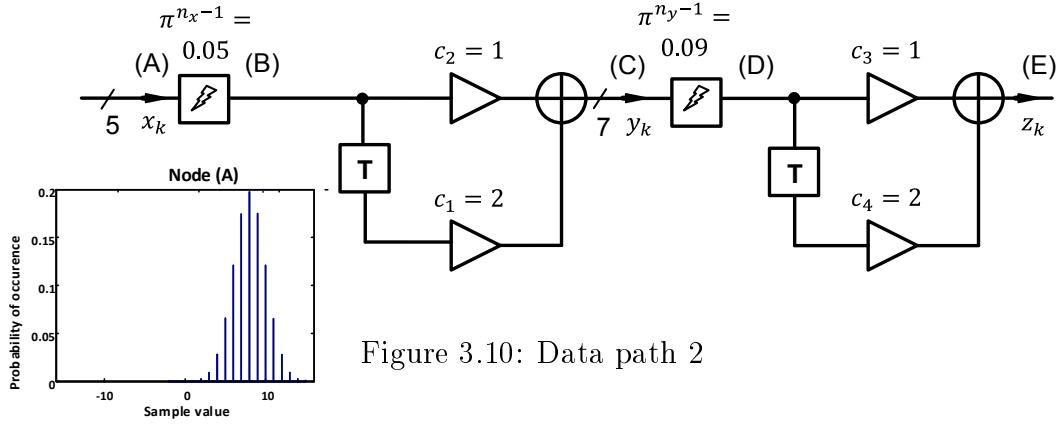


Figure 3.9: Creation of Gaussian components at data path 1

For the resulting new Gaussian component, we have $\tilde{Y}_{0,1} \sim \mathcal{N}(\mu_{\tilde{Y}_{0,1}}, \sigma_{\tilde{Y}_{0,1}}^2)$ with $\mu_{\tilde{Y}_{0,1}} = \mu_{Y_{0,1}} - 64 = 24 - 64 = -40$ while the variance remains $\sigma_{\tilde{Y}_{0,1}}^2 = \sigma_{Y_{0,1}}^2 = 20$. The other 91% of the samples represented by this component, i.e. in total $(1 - \pi^{n_y-1}) \cdot (1 - \pi^{n_x-1}) \times 100\% = 81.9\%$, are also not affected by the second error injection. So, component $Y_{0,0} \sim \mathcal{N}(\mu_{Y_{0,0}}, \sigma_{Y_{0,0}}^2)$ has the unchanged mean $\mu_{Y_{0,0}} = \mu_Y = 24$ and variance $\sigma_{Y_{0,0}}^2 = \sigma_Y^2 = 20$.

The same reasoning leads to the further four components ($Y_{1,1,0}, Y_{1,1,1}, Y_{1,2,0}, Y_{1,2,1}$) at node D, as illustrated in Fig. 3.9. The final step is the propagation of these six components through the second section, namely the 2-tap FIR filter. For this step, it is necessary that all combinations of the 6 components are taken 2 at a time in order to find the means of the components at the output. On top, 6 more combinations have to be considered for the cases that random variables from one component are combined with random variables from the same component, i.e. a random variable from $Y_{0,0}$ is combined with another random variable from $Y_{0,0}$ etc. This leads to a total of 36 possible components present at the output of the filter. Note that this number can be narrowed down without noticeable loss of accuracy because of the insignificant weight of certain components; however, this is not necessary for the current illustration. The variance will be common for all components and will equal $\sigma_Z^2 = (c_3 \cdot \sigma_Y^2) + (c_4 \cdot \sigma_Y)^2 = 100$. The corresponding PMF graphs can be seen in the second column of Fig. 3.8.

Finally, considered as **Case (III)**, the results of a *simulation-based*



investigation of the effects of error injection and propagation shall be presented for comparison. The results of the simulation are used to derive the histograms at the different nodes (without defining the signal statistics). It becomes obvious that the PMF results at node E fit closely between the analytical and simulation results. Despite the simplicity of the data-path in the specific case and the fact that we chose an input PMF with positive numbers, such an approach can be automated in order to produce the statistical moments of the Gaussian components at the desired nodes in the signal flow graph, speeding up in that way the statistical analysis. However, even this relatively straightforward approach, which is enabled by the analytical tractability of normal distributions in linear transformations, has quite some limitations, as it will be shown next.

- *Example 2: Linear transformation with memory*

To show the limits of analytical approaches, a second SFG is investigated in the following. This signal flow graph is very similar to the one discussed before. The difference lies in the first section. We replace the demultiplexer with a 2-tap filter, as shown in Fig. 3.10. We repeat the steps that correspond to Cases (I), (II), (III) in the previous example in the same way. Thus, we construct a similar set of PMF graphs that correspond to the signal PMFs at the different nodes of the SFG. These can be viewed in Fig. 3.11. By carefully observing the corresponding graphs, a certain element becomes clear: **the PMF results at node E look different.**

To identify the reason, we must go back to the output of the first section at node C. By carefully examining the time series of the signal, we can observe that the samples are not any more uncorrelated but they follow a certain pattern (see Fig. 3.12a): a sample that belongs to $\mathcal{N}(8, 20)$ (illustrated in the figure by a sample with red color) will be typically followed by a sample that belongs to $\mathcal{N}(-8, 20)$ (illustrated in the example with yellow color) and not by a sample that belongs to $\mathcal{N}(24, 20)$ (illustrated with green). The reason lies in the fact that the tapped delay line correlates the signal and excludes specific combinations from being present at the output.

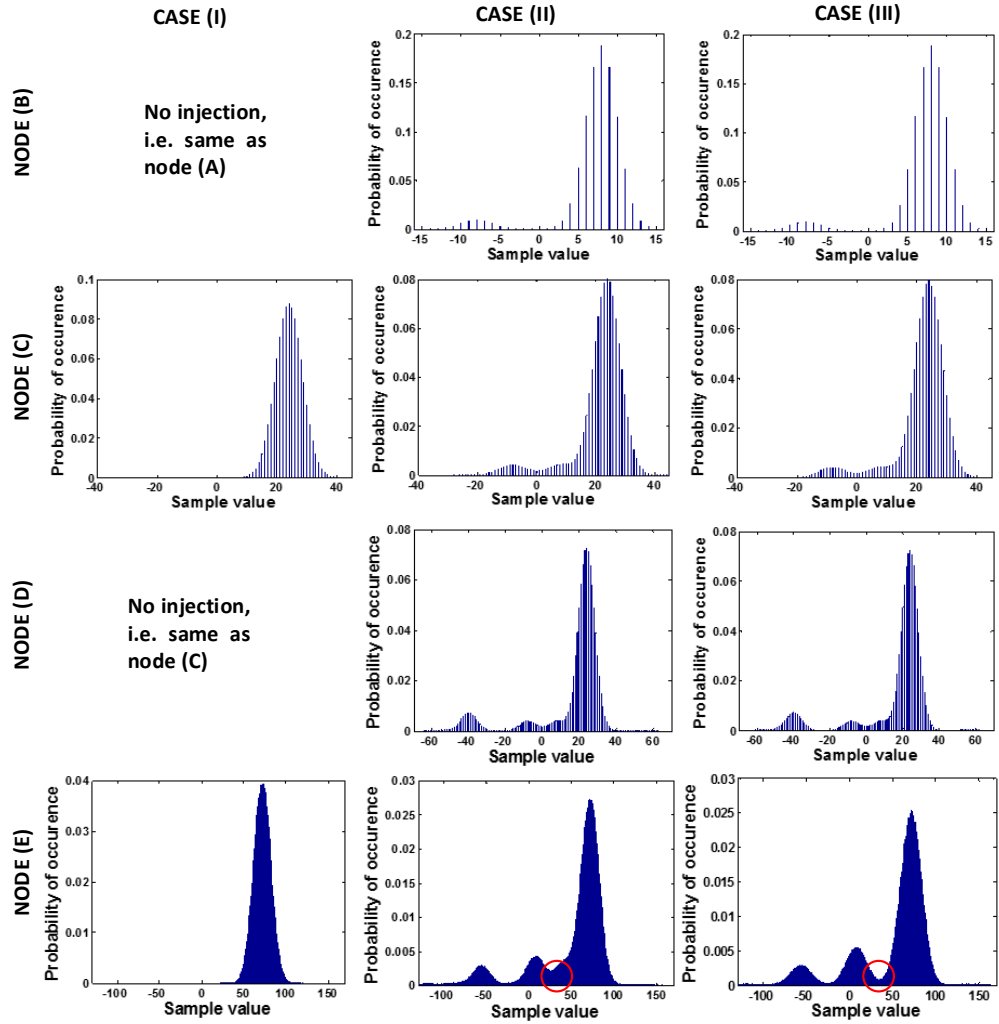


Figure 3.11: PMFs of the signal at the different nodes of the data path 2

In other words, in data path 1 there is no autocorrelation present at node C (all different combinations of samples have equal probability to occur), while in data path 2 an autocorrelation of lag 2 is present, as can be observed in the corresponding ACF plots in Fig. 3.13. As a consequence, analytical models have to be extended to accommodate specific sample patterns. This, in a realistic case, becomes soon computationally intractable.

These examples have demonstrated that as long as there is a white random Gaussian signal subjected to random bit-flips that is processed through a linear block without memory, analytical modeling of the error propagation is feasible, no matter how many stages of processing there are. However, as soon as blocks with memory are involved (combined with error injection at intermediate points), such techniques soon explode computationally.

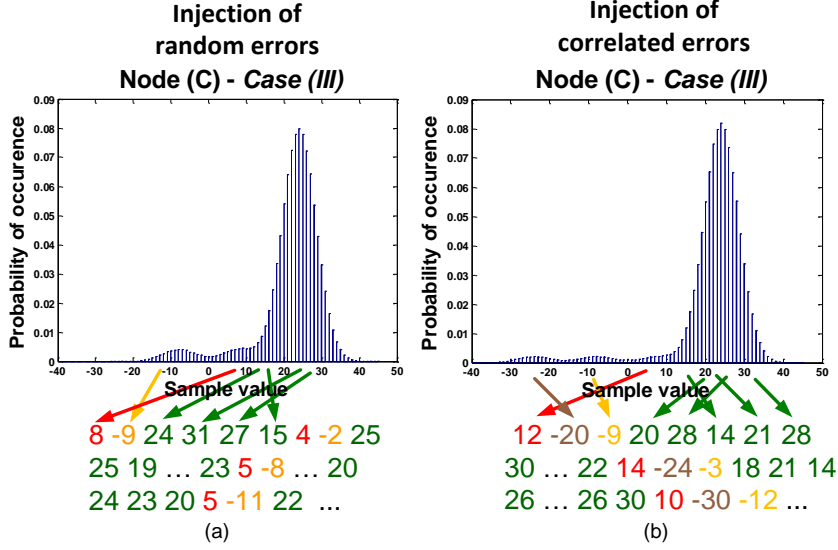


Figure 3.12: Internal correlation due to the filter in the first section of data path 2

3.5.1.2 Autocorrelation introduced by hardware-induced errors

In this subsection, we discuss an example in which the autocorrelation introduced by the injection procedure itself. We use a similar driver as the one used in [39], a DFT block. Fig. 3.14 shows for simplicity only the real (or imaginary) part of a N -point DFT for real-valued inputs (as customary for FFT SFGs open circles depict summation points and bold triangles depict multiplications). For the real-valued coefficients $w^{i,k} = \Re\{\underline{w}^{i,k}\}$ (or $w^{i,k} = \Im\{\underline{w}^{i,k}\}$) with $\underline{w}^{i,k} = \frac{1}{N} \cdot \exp(j \cdot i \cdot k \cdot 2\pi/N)$ holds (j being the imaginary unit). The output signal (spectral components) can be derived for both, the real and the imaginary parts, as a sum of N random variables. In case N gets large, by applying the CLT, we can approximate the output by a Gaussian distribution. In the context of bit-flip error injection and propagation, it has already been illustrated in [39] that the signal at the output of the DFT (FFT in the specific case) can be approximated by a Gaussian, after injecting random uniform bit-flips at a memory module in front of the FFT. The authors show that the larger the FFT size N (starting from 200), the closer the resulting distribution is to the Gaussian (due to the CLT). In the following, we will illustrate that not only the injection but also the way the injection takes place can play a significant role in the validity of such approximations. In the 64-point DFT block illustrated in Fig. 3.14, we perform experiments with an uncorrelated input signal which follows a multimodal PMF, as illustrated at the top left side of Fig. 3.15. We perform three types of simulation experiments: In the first case, we simulate the propagation through the DFT block of the aforementioned input. In the second case, we inject 1% random bit-flips at the input (node (A)) and, then, we let the signal propagate. In the third case, we inject again 1% bit-flips but not in a random way; in this case, the corruption always spans two consecutive samples (x_0, x_1) to illustrate the

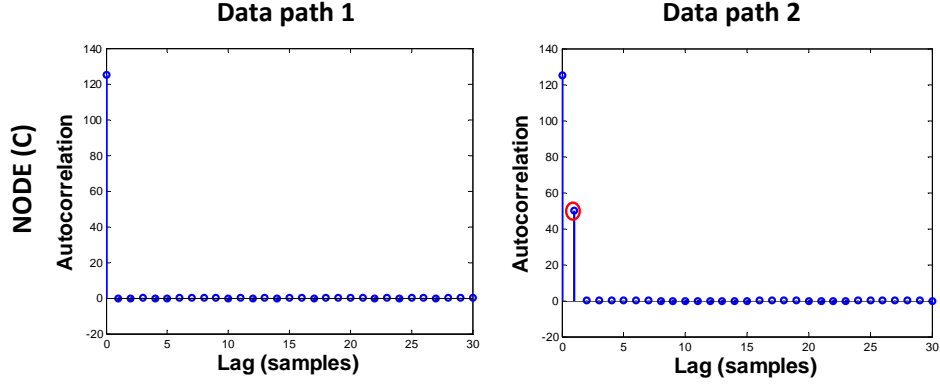


Figure 3.13: ACF plots at node (C) for data path 1 and 2

effect of error bursts. In both injection cases, we flip the corresponding bits in the MSB position. We will not go into discussing the details of the statistical moments, since it is not the point of interest in this example. The interested reader is referred to [39].

To determine how well the output signal can be modeled by a single normal distribution, the third and the fourth statistical moments are used: skewness and kurtosis. In a normal distribution, skewness equals to 0 while kurtosis equals to 3. What can be observed in Fig. 3.15 is that, despite the resemblance of the PMFs at the input of the two injection experiments, the PMFs of the output differ significantly. This can be shown also by the statistical moments of the output PMFs in Table 3.4. All three PMFs have a skewness value close to 0, while the first 2 PMFs (the error-free and the one produced after the injection of random errors) have a kurtosis value which is close to 3, and, thus, they are almost normal distributions. The last PMF significantly deviates from the normal distribution, as it can be seen in the figure. As the corrupted input signal features autocorrelation, in this case the CLT cannot be applied any more.

Table 3.4 Skewness and kurtosis at the DFT output

	No injection	Injection of random errors	Injection of correlated errors
Skewness	-1.3471e-04	-0.0047	-0.0297
Kurtosis	3.0718	3.0110	4.6412

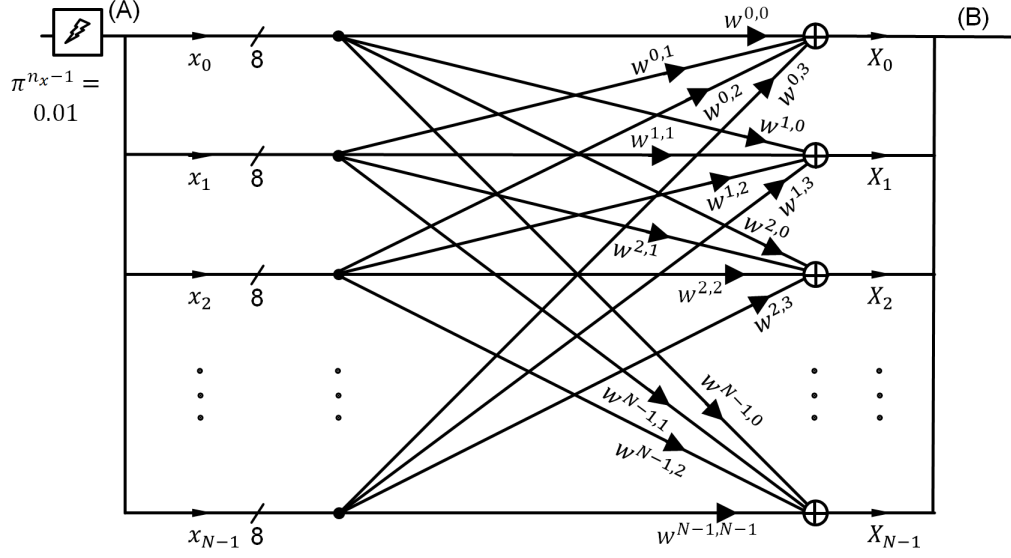


Figure 3.14: DFT signal flow

This example illustrates that the injection procedure itself can add autocorrelation to the corrupted signal (on top of the existing error-free signal autocorrelation). To illustrate this, again we plot the autocorrelation values at the different lags (see Fig. 3.16) for the two injection procedures. For the case of the injection of random errors (Fig. 3.16a), the autocorrelation in the corrupted input signal is distributed more or less uniformly across the different lags. However, in the other case (Fig. 3.16b), autocorrelation peaks can be observed at distinct lags: As the errors are injected at (x_0, x_1) , the time span between error pairs is an integer multiple of 64. The autocorrelation peaks occur often in pairs (but not with the same height as there are the different combinations of positive-positive, negative-negative, positive-negative, negative-positive samples), since two consecutive samples were corrupted. In this context, this means that the random variables are not any more fully uncorrelated so their sum does not converge to a normal distribution, due to the CLT. Autocorrelation introduced by the injection procedure itself should be examined when applying such modeling techniques.

As a side note, coming back to data path 2 of the previous example, the injection of 5% correlated errors (so that always two consecutive samples are corrupted) in the first section of Fig. 3.10 would lead to a different PMF compared to the one already shown for random injection. This can be seen in Fig. 3.12b. In this case, a different combination is present in the filter due to the injection procedure: a sample that belongs to $\mathcal{N}(8, 20)$ (shown in red) is followed by a sample that belongs to $\mathcal{N}(-24, 20)$ (shown in brown) and this by a sample that belongs to $\mathcal{N}(-8, 20)$ (shown in yellow) and then by a sample that belongs to $\mathcal{N}(24, 20)$ (shown in green). This leads to a different (longer) pattern, making the internal autocorrelation even stronger.

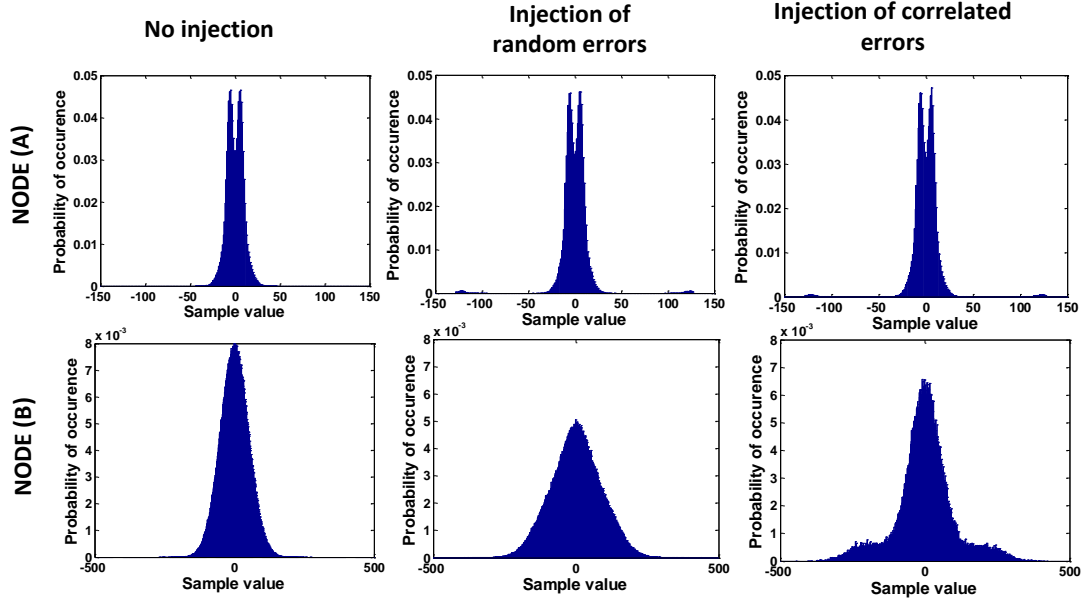


Figure 3.15: Input (node (A)) and output (node (B)) PMF after injection experiments in DFT block

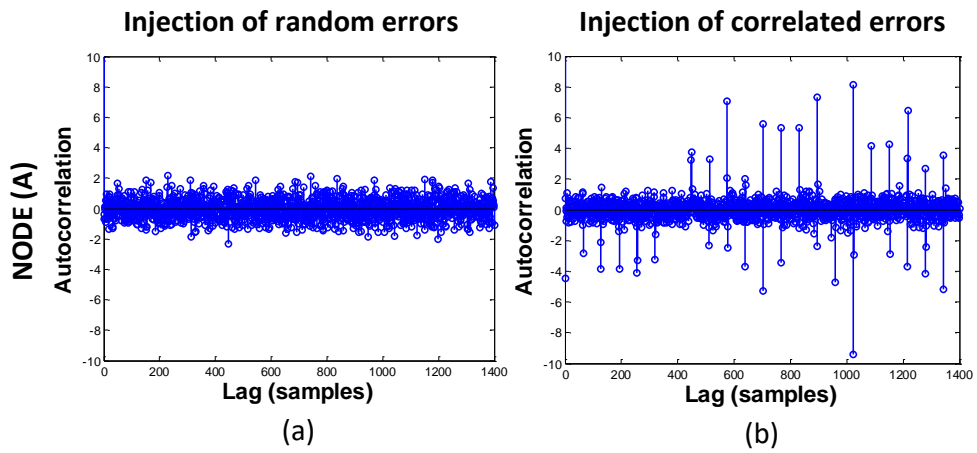


Figure 3.16: ACF plots at node (A) of the DFT block

3.5.2 Additional related work

In [39] the same group extends the work in [40] and propagates analytically bit errors (on the channel noise distribution) on several blocks of a typical OFDM-based communication system. These blocks include: a filter, a FFT block, an equalizer, a de-interleaver and a decoder. For the FFT block they make use of the CLT and calculate the variance of the output Gaussian distribution (this assumes the FFT is bigger than a certain length and the input data are not correlated). For the equalization they calculate the output distribution using integration tables. As the de-interleaver only permutes and shuffles the data, no new distribution is produced at its output. To prepare for the decoder, the authors find an equivalent Gaussian noise distribution that has the same area under the tail of the distribution (i.e. the same BER) as the corrupted data. After they derive this, they can use it to decide the output BER based on the decoder's SNR vs. BER characteristic (as the soft-input decoder assumes Gaussian noise). For the hard-input decoder (i.e. slicer), the BER of the distribution at the input has the same BER as the original system and this is sufficient to find the coded-BER.

This is an integrated framework that allows the fast computation of the impact of random bit-flips to the system BER. It operates under the assumption that the data are uncorrelated as they propagate through the communication line. Although this maybe true for some cases, very often it is not the case, as already illustrated in the previous section. Not neglecting autocorrelation becomes the focus of the next chapter.

3.6 Summary and link to next chapter

In this chapter, we presented possibilities and limitations of using analytical techniques in order to speed-up the execution time of error injection and propagation experiments in LTI operators of communication systems. It has been shown that under certain conditions, such techniques can be beneficial. However, due to the inherent correlation (in the general case) between the error and the error-free signal, the signal characteristics become very soon computationally intractable, even for simple PMF cases, when blocks with memory are present. Moreover, hardware-induced errors can appear in a correlated form which further complicates the analysis. Thus, there is need for novel techniques that enable the study of error propagation in a scalable way and take into account autocorrelation.

Chapter 4

Propagation of Corrupted, Autocorrelated Signal

4.1 Outline

Chap. 4 discusses the next explored option for efficiently propagating a corrupted (or an error) signal, which is autocorrelated, namely using a mathematical transform, called Principal Component Analysis (PCA). Sec. 4.2 introduces the motivation for exploring the PCA approach and outlines its steps. Sec. 4.3 presents the adaptation of the approach for performing error injection experiments. Results and limitations are presented in Sec. 4.4 while Sec. 4.5 summarizes the chapter.

4.2 Motivation and preliminaries

In the search to finding an efficient way to propagate corrupted data that are internally autocorrelated, an idea is to explore techniques that take advantage of this internal autocorrelation to reduce the amount of data required to be simulated without inducing significant information loss. Such a technique is the PCA [78, 18]. PCA is a well-established technique with many applications in data analysis, one important among which, is the dimensionality reduction. Namely, it performs a transformation on multidimensional data so that most of the data variance is preserved in the higher dimensions and, therefore, lower dimensions can be potentially removed while preserving all significant information; thus, it reduces the total amount of data to be processed with a small impact only on accuracy. The amount of dimensions that can be removed depends on the nature of the data correlation and the desired accuracy. If the data are fully uncorrelated, no dimensionality reduction is possible without significantly impacting the accuracy. PCA has been less explored for speeding-up data simulation in digital systems. One exception is found in [95]: There, the authors utilize a closely related mathematical transform, called Karhunen–Loève Expansion [60], to propagate input data with

quantization noise through LTI blocks while performing dynamic range estimation experiments in digital systems.

The way in which PCA could be exploited in our context is illustrated in Fig. 4.1. Especially, Fig. 4.1a is a generalization of the Fig. 3.10. The light blue boxes in Fig. 4.1a illustrate LTI blocks, which correspond to the filters in the first and second sections of Fig. 3.10. However, in this case, in order to cover the more general case, the LTI blocks are represented by the matrices $[H]$, $[G]$ with elements h_j, g_j , that are the matrices that represent the linear transformations that are performed by the blocks. The first block accepts the input samples z_k , produces the output samples x_k (which may now be autocorrelated). The samples x_k get bit-flipped and, after the corruption, the samples \tilde{x}_k propagate to the second LTI block to produce the output samples \tilde{y}_k (the vector representations are motivated and explained in Sec. 4.3.1). The objective here is to exploit PCA so that the amount of computations required during the information propagation through the second LTI block is reduced. In the end, the slightly altered (but with small impact on accuracy) output samples \tilde{y}'_k are produced, as illustrated in Fig. 4.1b.

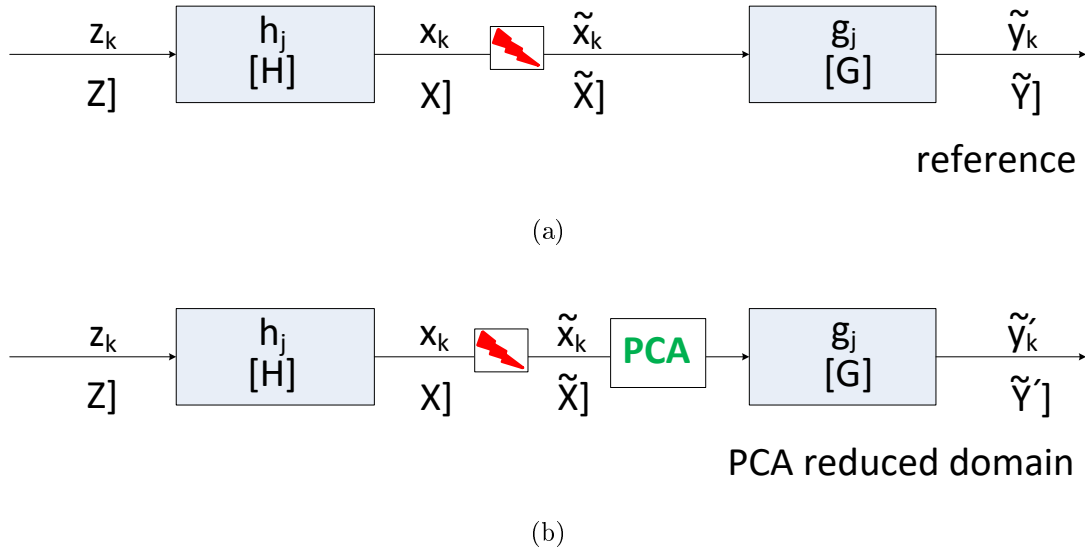


Figure 4.1: Set-up for propagating the corrupted, autocorrelated signal through the G operator in the reference domain (a) and in the reduced PCA domain (b)

Before discussing the way that PCA could be used for error injection experiments, first the PCA procedure is introduced along with the way it can be used to reduce the amount of dimensions in data analysis.

4.2.1 PCA steps and dimensionality reduction

Consider an m -dimensional random input vector or multivariate random variable (MRV) represented by the column vector X with mean μ_X and variance σ_X^2 . We acquire n measurements, called vector samples in this case, on the MRV. The data can be organized in a matrix. Each of the n columns represents a different repetition of the experiment, i.e. a different vector sample. Each of the m rows gives a particular kind of feature (or a dimension).

PCA is a procedure that transforms this set of measurements of possibly correlated variables of the MRV into a set of values of linearly uncorrelated variables, called principal components (PCs). According to the procedure, the PCs are ordered in such a way, so that the first PC has the largest possible variance of the original data, the second PC the second largest variance and so on. The PCs are orthonormal as they are the eigenvectors of the (symmetric) covariance matrix of the input data.

More specifically, assume the individual data samples of the matrix can be described by $[x_{1i}, x_{2i}, \dots, x_{mi}]^T$, with $i = 1, 2, \dots, n$. Using the estimated mean values $\hat{\mu}_1 = \frac{1}{n} \cdot \sum_{j=1}^n x_{1j}$ (with $\mu_{X_1} \approx \hat{\mu}_1$), $\hat{\mu}_2 = \frac{1}{n} \cdot \sum_{j=1}^n x_{2j}$, and so on, the centered data matrix $[R_X]$ with m rows and n columns is built:

$$[R_X] = \begin{bmatrix} x_{11} - \hat{\mu}_1 & x_{12} - \hat{\mu}_1 & \dots & x_{1n} - \hat{\mu}_1 \\ x_{21} - \hat{\mu}_2 & x_{22} - \hat{\mu}_2 & \dots & x_{2n} - \hat{\mu}_2 \\ \dots & \dots & \dots & \dots \\ x_{m1} - \hat{\mu}_m & x_{m2} - \hat{\mu}_m & \dots & x_{mn} - \hat{\mu}_m \end{bmatrix}. \quad (4.1)$$

The PCA transform is composed of the following steps [19]:

- Estimate covariance matrix $[\Sigma]$ of random vector X

$$[\Sigma_X] \approx [\widehat{\Sigma_X}] = \frac{1}{n-1} \cdot [R_X] \cdot [R_X]^T. \quad (4.2)$$

- Perform Eigenvalue Decomposition (EVD) of Σ_X

$$[\Sigma_X] = [V_X] \cdot [\Lambda_X] \cdot [V_X]^T, \text{ i.e. } [\Sigma_X] \cdot [V_X] = [V_X] \cdot [\Lambda_X]. \quad (4.3)$$

As the covariance matrix is symmetric and positive, $[\Lambda_X]$ is a diagonal matrix with entries the non-negative, real, ordered (biggest first) eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_m$ of $[\Sigma_X]$ and $[V_X]$ is an orthogonal matrix, having as columns the normalized eigenvectors of $[\Sigma_X]$, i.e. $[V_X] \cdot [V_X]^T = [I_m]$.

- Derive the principle components of X

$$[\Pi_X] = [V_X]^T \cdot \check{X}, \quad (4.4)$$

where $\tilde{X}]$ is the centered random vector $X] - \widehat{\mu_X}]$.¹

Dimensionality reduction. In case dimensionality reduction is desirable, the following step can be made:

- Crop eigenvector and eigenvalue matrices based on the dominant eigenvalues

$$\begin{aligned} [\Lambda_X] &\Rightarrow [\Lambda'_X] \\ [V_X] &\Rightarrow [V'_X], \end{aligned} \quad (4.5)$$

which yields

$$X] \approx X'] = \mu_X] + [V'_X] \cdot \Pi_X].^2 \quad (4.6)$$

Thus, $X]$ is approximated with a smaller number m' of dimensions.

The contribution of the j^{th} principal component Π_{X_j} to the total variance is given by

$$\sigma_{\Pi_{X_j}}^2 = \frac{\lambda_j}{\sum_{i=1}^m \lambda_i} \cdot \sum_{i=1}^m \sigma_{\Pi_{X_i}}^2. \quad (4.7)$$

The loss of information, which is determined by the number of components (or variables) m' to be kept, can be decided individually according to the desired SNR:

$$SNR = \frac{\sigma_{\text{signal}}^2}{\sigma_{\text{noise}}^2} = \frac{\sigma_{\Pi_{X_1}}^2 + \dots + \sigma_{\Pi_{X_{m'}}}^2}{\sigma_{\Pi_{X_{m'+1}}}^2 + \dots + \sigma_{\Pi_{X_m}}^2} \quad (4.8)$$

or the relative mean squared error from the truncation

$$e = 1 - \frac{\sum_{i=1}^{m'} \lambda_i}{\sum_{i=1}^m \lambda_i}. \quad (4.9)$$

If the variances of the components of the random vector $X]$ differ significantly, the PCA should be carried on the standardized input vector (the

¹It can be shown [19] that $X]$ can be represented by $X] = \mu_X] + [V_X] \cdot [\Lambda_X]^{\frac{1}{2}} \cdot \Psi]$, where $\Psi]$ is an m -dimensional uncorrelated random input vector with mean $\mu_\Psi] = 0$ and variance $\sigma_\Psi^2] = 1$. Then, $\Pi_X] = [\Lambda_X]^{\frac{1}{2}} \cdot \Psi]$.

Proof: $[\Sigma_X] = E\{([V_X] \cdot [\Lambda_X]^{\frac{1}{2}} \cdot \Psi]) \cdot ([V_X] \cdot [\Lambda_X]^{\frac{1}{2}} \cdot \Psi])^T\} = [V_X] \cdot [\Lambda_X]^{\frac{1}{2}} \cdot [\Lambda_X]^{\frac{1}{2}} \cdot [V_X]^T = [V_X] \cdot [\Lambda_X] \cdot [V_X]^T = [\Sigma_X]$ *q.e.d.*

²In addition, $X] \approx X'] = \mu_X] + [V'_X] \cdot [\Lambda'_X]^{\frac{1}{2}} \cdot \Psi]$.

original vector divided by its standard deviation). The covariance of the standardized input vector equals the correlation matrix $[P_X]$ of X . So, the correlation matrix is related to the covariance matrix as follows

$$[P_X] = [D]^{-1} \cdot [\Sigma_X] \cdot [D]^{-1}, \quad (4.10)$$

where $[D]$ is a diagonal matrix having as diagonal elements the standard deviations of the components of X .

It is important to note that in case the X data are processed through an LTI block and generate as output the Y (see Fig. 4.1), the above procedure can be adapted so that **the LTI processing takes place only on the reduced eigenvector set**. Namely,

$$Y \approx Y' = [G] \cdot X' = [G] \cdot (\mu_X) + [V_X'] \cdot \Pi_X, \quad (4.11)$$

where $[G]$ is the matrix that represents the linear transformation that is performed by the LTI block. For example, for a simple direct-form FIR filter, this matrix is the so-called Toeplitz matrix that performs the convolution. The Toeplitz matrix will be presented in detail in Sec. 5.3.2.

4.3 Propagating the corrupted signal using PCA

4.3.1 Data organization for injection experiments

As it becomes clear from the introduction of the PCA steps, the transform operates on multidimensional data. Therefore, as a first step in this section that discusses the adaptation of PCA for injection experiments, a different data organization is proposed. Two important components in discussing the data organization are the: injection rate in our experiments and the effect of the memory elements on the error propagation.

Injection rate. Although realistic error rates can be quite low, to obtain statistically relevant results about the potential impact of an error faster, higher injection rates can be chosen. When an error rate is very low and system simulation is performed, the system may need to process a lot of error-free data until the next error manifests, as illustrated in the upper part of Fig. 4.2. In the figure, the red lines illustrate error samples and the broken axis denotes a large time lapse. The error signals at the input and output are denoted by e_x and e_y . In the lower part of Fig. 4.2, the injection rate is scaled up by the scale factor π so that sufficient statistics are gathered faster at the output. The modified error signal at the input is denoted by \acute{e}_x and at the output by \acute{e}_y . To acquire the statistics for the *actual* injection rates, a downscale is required. For example, for the case that the error PMF at the output is of interest, the resulting PMF must be downscaled appropriately,

³Also: $Y \approx Y' = \mu_Y + [V_Y'] \cdot [\Lambda_Y']^{\frac{1}{2}} \cdot \Psi = [G] \cdot (\mu_X) + [V_X'] \cdot [\Lambda_X']^{\frac{1}{2}} \cdot \Psi = [G] \cdot X$ q.e.d. The steps for performing the LTI processing on the reduced eigenvector set for a random vector are shown in a systematic way in Table 7.1 in the Appendix for two LTI blocks in a row.

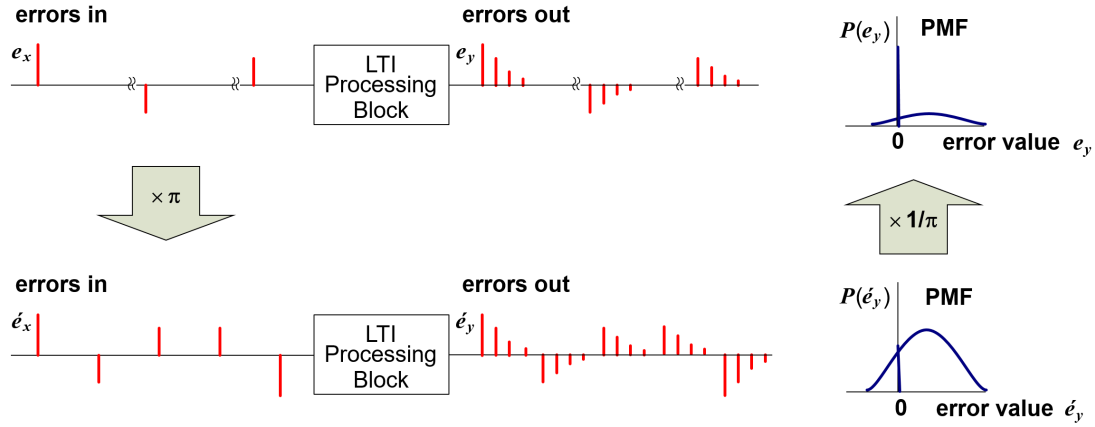


Figure 4.2: Modified injection rate in order to speed-up the error injection campaign

as shown in the figure. Notice that in the upscaled PMF, the height of the bar at 0 is low compared to the downscaled (realistic) case. That is because in the upscaled case, the error-free samples will be more seldom.

Effect of memory on error propagation. For a block with memory, a single error at the input will, in general, produce a number of consecutive errors at the output. The length of the sequence will be as big as the maximum number of consecutive memory elements in the block (assuming no feedback loops are present at the block). Fig. 4.3 shows how a single error produces (each red line indicates an erroneous sample) multiple errors at the output in a FIR filter example. In the figure, c_1, c_2, \dots, c_M denote the filter coefficients, and T are the filter delay elements. Assume we want to study the effect that single bit errors at the input of the block have on the output. It is intuitively understood that every consecutive error should be injected after the effect of the previous error is no longer present in the internals of the block. In that way, the effect of the next error does not interfere with the effect of the previous one. Therefore, the minimum distance in samples that should be preserved between consecutive error injections is equal to the largest number of memory elements that are sequentially interconnected in the block(s) under investigation.

Data organization. In order to acquire statistically representative results regarding the effect of a bit-flip at a specific bit position, many samples have to be corrupted in this position. For a block with memory, the error samples at the input need to have a minimum sample distance from each other as decided by the amount of the block's consecutive memory elements (as discussed above). For a block that accepts K samples in parallel as input, only one out of the K samples should be corrupted and the distance between consecutive corrupted samples should be K , so that the effect from one error does not interfere with the effect of the previous one. Since the distance among consecutive error samples in all cases is fixed, the input data can be organized in a different way. Instead of having one long input sequence for the injection experiments, an alternative approach is to use **sections** of the long sample sequence and process them separately. Each section can

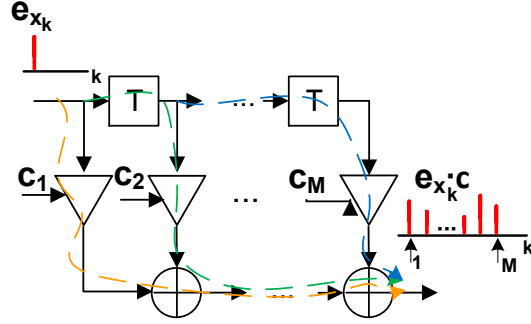


Figure 4.3: Filter output when a single error sample is placed at the input

be considered as a separate measurement (i.e. vector sample) of the random signal, each composed of S data samples. The minimum section length is decided by the minimum required distance between consecutive error samples. Fig. 4.4 depicts this alternative data organization. The long sample sequence x_1, x_2, \dots, x_L is split into $s + 1$ sections of length S . Section 1 is composed of samples x_1, x_2, \dots, x_S , section 2 is composed of $x_{S+1}, x_{S+2}, \dots, x_{2S}$ etc. Every sample is composed of a number of bits. In one injection scenario, a specific bit is corrupted. The thunderbolt in the figure indicates the (corrupted) bit position at which the error is injected according to the specific injection scenario. The sections are arranged columnwise and each section is processed (e.g. filtered) separately. So, the data matrix $[X]$ is:

$$[X] = \begin{bmatrix} x_1 & x_{S+1} & \dots & x_{s \cdot S+1} \\ x_2 & x_{S+2} & \dots & x_{s \cdot S+2} \\ \dots & \dots & \dots & \dots \\ x_S & x_{2 \cdot S} & \dots & x_{(s+1) \cdot S} \end{bmatrix}.$$

With this data organization the position of the corrupted sample in all sections is the same.

Discussion on the section length. The objective of splitting the long sample sequence into sections is to change the initial univariate RV into a multivariate RV so that the PCA tool can be applied. The statistical characteristics of the signal at the desired system node (in terms of the PMF and ACF) should be the same for the univariate and multivariate RV. To achieve this, the section length should be carefully chosen so that the autocorrelation length (number of lags used in the ACF) is sufficient to cover all lags for which correlation exists. For example, if an initially uncorrelated signal is autocorrelated through a 32-tap filter, we know that at the output of the filter the autocorrelation has a length of 32 lags. Splitting the long sequence into sections of 32 samples at the filter output will keep the correlation

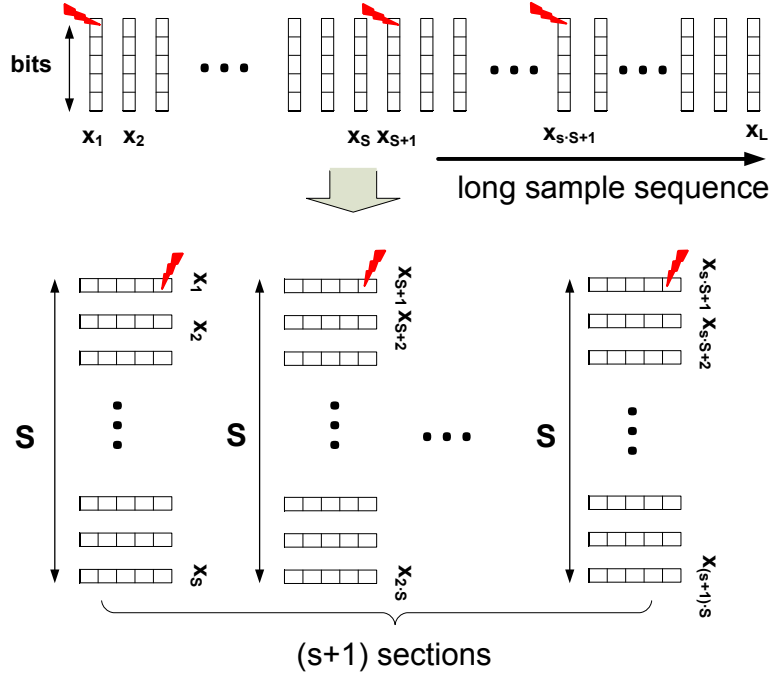


Figure 4.4: A long sample sequence is re-organized into sections of length S

information intact. However, the autocorrelation can be further changed as the signal propagates through the system; for example, by a second filter. Assume the second filter is also 32 taps, in order to have the whole relevant autocorrelation length at the output of this filter, the initial sequence should be split in sections of 64 samples instead.

If, at the desired system node, no information is provided regarding the inherent signal autocorrelation, a procedure to decide an appropriate section length is the following: First an arbitrary (small) section length is selected. Then, the ACF is calculated for all lags. In case the correlation value is non-negligible for all lags, the section length is increased until the earliest lag is found for which the correlation value becomes zero. To accommodate for autocorrelation that will be added by upcoming signal processing blocks an appropriate margin should be added as discussed in the example earlier.

For a given number of samples, a trade-off exists between the section length and the number of sections used to perform the calculations. A bigger section length gives a safety margin regarding the autocorrelation length (which changes as the signal propagated through the different blocks), but leads to a smaller number of sections. A smaller number of sections will give less representative correlation values for each lag (as an averaging takes place) compared to a bigger number of sections.

Having done this discussion and as the objective of the current exploration is to see if computational benefits can be acquired when using PCA for error injection experiments, we will use for the following experiments a simplified assumption: The section length is the maximum between the inherent signal autocorrelation length and the length of the autocorrelation that will be added in the upcoming block. If the computational benefits are sufficient, the exact implementation can be worked out according to the system specifics.

4.3.2 Illustration of the approach

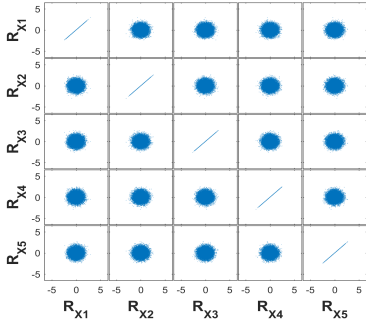
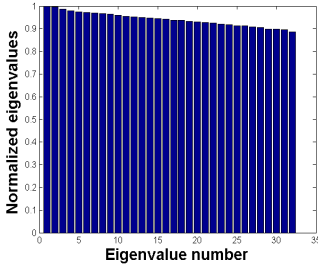
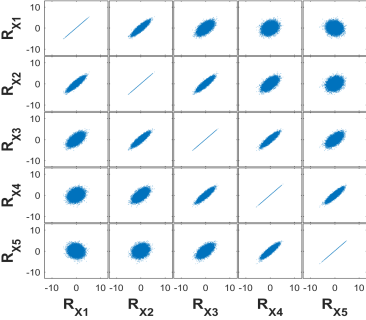
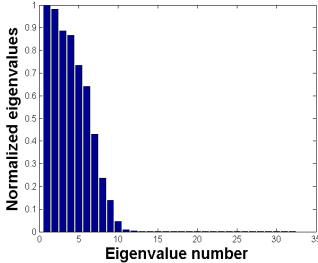
Set-up. To illustrate the effectiveness of the approach, we utilize the data representation that was presented above (see Fig. 4.4), to implement the PCA in order to speed-up error injection experiments. First, we illustrate the results of applying the PCA initial steps on both uncorrelated and autocorrelated data. Initially, we will use the correlation matrices of the two data sets to illustrate practically the possibilities and limitations of using PCA to reduce computational complexity in our context. Then, the focus remains on the autocorrelated data (since only with autocorrelated data benefits can be induced with this approach) and, as a last step, the error injection is introduced. Specifically, the autocorrelated data have been created through filtering uncorrelated data using a 32-tap low-pass filter. That defines also the section length in the data organization. Considering in total 10^6 samples, the data matrix is organized so that $m = 32$ and $n = 31,250$, i.e. we have 31,250 sections (or vector samples) with 32 dimensions each. These parameters are illustrated in the upper 4 rows of Table 4.1.

Table 4.1 Parameters used in the current illustration

<i>Parameters</i>	
number of samples	10^6
number of sections	31,250
number of samples per section	32
filter 1 length	32
filter 2 length	32
bit-flip position	MSB
truncation error	1%

Correlation and eigenvalues. The correlation matrix is chosen for the current illustration (instead of the covariance matrix) as it gives a more intuitive understanding of existing correlations due to the normalized values. Each element on the main diagonal of the correlation matrix is the correlation of an element (or vector) with itself, which always equals 1. Non-diagonal entries lie between -1 and

Table 4.2 Correlation among first 5 rows and eigenvalues for uncorrelated and autocorrelated data

	Correlation scatter plots (first 5 rows)	Correlation matrix $[P_X]$ (first 5 rows)	Eigenvalues $[\lambda_1, \lambda_2, \dots, \lambda_{32}]^T$
uncorrelated data		$[P_X] = \begin{bmatrix} 1.000 & 0.004 & -0.003 & -0.007 & -0.001 \\ 0.004 & 1.000 & 0.008 & -0.002 & -0.000 \\ -0.003 & 0.008 & 1.000 & 0.000 & -0.013 \\ -0.007 & -0.002 & 0.000 & 1.000 & -0.002 \\ -0.001 & -0.000 & -0.013 & -0.002 & 1.000 \end{bmatrix}$	
autocorrelated data		$[P_X] = \begin{bmatrix} 1.000 & 0.889 & 0.601 & 0.230 & -0.107 \\ 0.889 & 1.000 & 0.890 & 0.602 & 0.234 \\ 0.601 & 0.890 & 1.000 & 0.890 & 0.605 \\ 0.230 & 0.602 & 0.890 & 1.000 & 0.891 \\ -0.107 & 0.234 & 0.605 & 0.891 & 1.000 \end{bmatrix}$	

1, with values around 0 denoting no correlation. So, the correlation matrix is given by

$$[P_X] = \begin{bmatrix} Corr(R_{X1}, R_{X1}) & Corr(R_{X1}, R_{X2}) & \dots & Corr(R_{X1}, R_{Xm}) \\ Corr(R_{X2}, R_{X1}) & Corr(R_{X2}, R_{X2}) & \dots & Corr(R_{X2}, R_{Xm}) \\ \dots & \dots & \dots & \dots \\ Corr(R_{Xm}, R_{X1}) & Corr(R_{Xm}, R_{X2}) & \dots & Corr(R_{Xm}, R_{Xm}) \end{bmatrix}, \quad (4.12)$$

with $Corr(R_{X_i}, R_{X_j})$ being the correlation $\frac{E[R_{X_i} \cdot R_{X_j}^T]}{\sigma_{R_{X_i}} \cdot \sigma_{R_{X_j}}}$, where E is the expected value operator. Random variables R_{X_i}, R_{X_j} belong to the initial MRV X and the aforementioned expected value is calculated from the i^{th} and j^{th} rows (or dimensions) of the centered data matrix $[R_X]$, derived by centering $[X]$, as presented in Eq. 4.1. Table 4.2 shows the scatter plots, correlation matrix and eigenvalues for two data sets: a set of uncorrelated data and a set of autocorrelated data. Due to space limitations, scatter plots and correlation matrix are presented for only the first 5 data matrix rows, while the bar graph shows the eigenvalues for all 32 eigenvectors.

The derivation of the correlation (or covariance) matrix and the eigenvectors are the first steps in PCA as already shown in Eq. 4.2 and Eq. 4.3.

As it can be observed in Table 4.2 and is to be expected, for uncorrelated data, the non-diagonal elements of the correlation matrix are approximately 0, leading to eigenvectors, which have eigenvalues of closely-valued weight. The situation becomes different for autocorrelated data, for which, the non-diagonal elements of the correlation matrix differ (sometimes significantly) from 0. This leads to some eigenvalues having more weight than others.

Dimensionality reduction and accuracy evaluation. At this step, after having derived the eigenvectors and eigenvalues for the presented autocorrelated data set (the uncorrelated data will be ignored from now on), dimensionality reduction can take place. Setting the truncation error to 1% (see Eq. 4.9), allows 22 dimensions of the data set to be removed and, thus, only 10 dimensions need to be kept in the PCA domain ($m' = 10$). As the PMF and the ACF fully characterize a random signal, Fig. 4.5 and Fig. 4.6 show the PMF and the ACF plot of the same autocorrelated signal before truncation and after 1% truncation error has been applied using the PCA transform. As it can be seen, there is no observable difference.

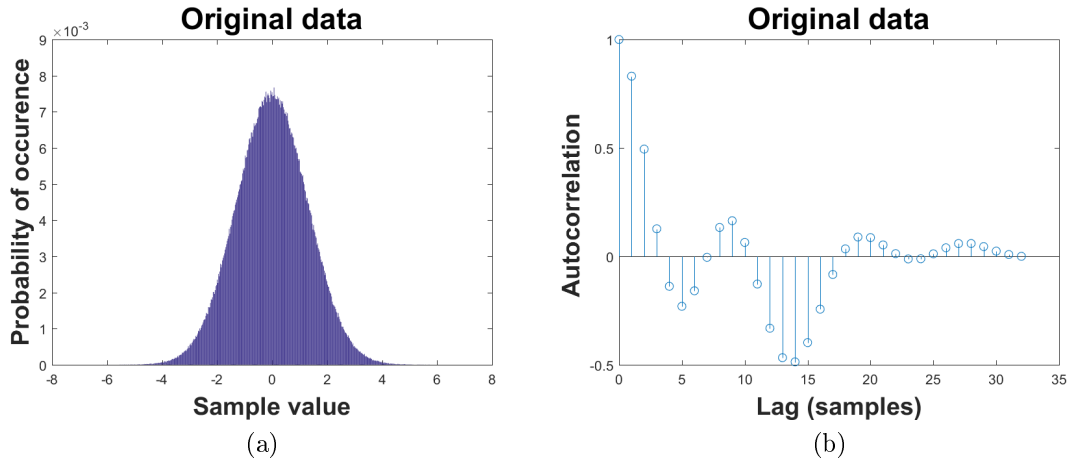


Figure 4.5: PMF (a) and ACF (b) plot of the original autocorrelated signal

4.3.2.1 Corrupted autocorrelated data propagation

To acquire benefits in terms of computational complexity in the context of error injection experiments, the following set-up is used. The above considered autocorrelated signal, indicated by $X]$ is corrupted before it enters the LTI operator G (see Fig. 4.1a). The errors are injected on the MSB of the first element of each section (as in Fig. 4.4) for this illustration. This is considered our reference experiment. According to Fig. 4.1b, the same experiment is conducted but with the following difference: After corruption, the corrupted signal $\tilde{X}]$ is transformed

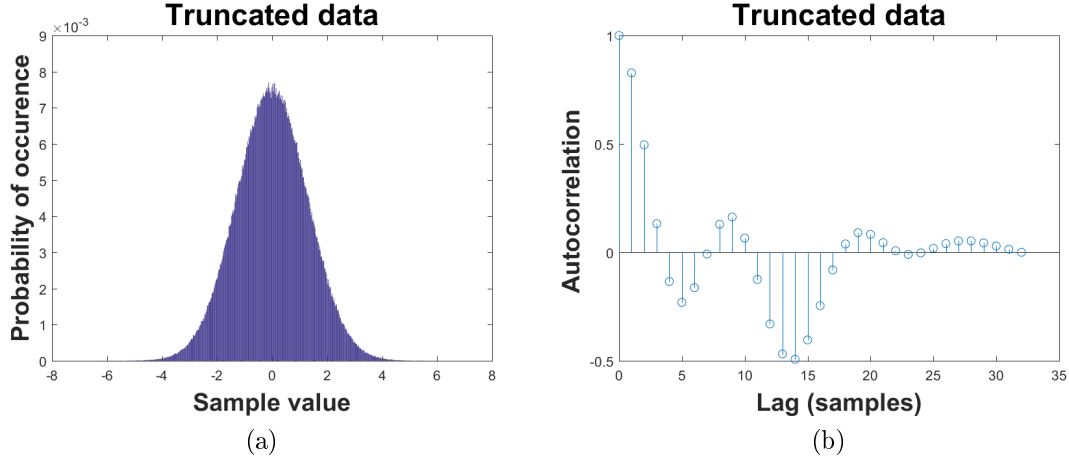


Figure 4.6: PMF (a) and ACF (b) plot of the autocorrelated signal after truncation (10 dimensions kept)

in the PCA domain. Then, to gain a computational benefit, 1% truncation error is applied. The parameters for the set-up can be found collectively in Table 4.1. This results in 10 dimensions being kept in the PCA domain (as earlier). That means that it is sufficient that the eigenvectors with the 10 biggest eigenvalues propagate through the operator G .

Computational benefit and accuracy evaluation. The computational complexity in terms of costly operations is as follows. For the reference experiments, $32 \cdot 32 \cdot 31,250$ multiplications are required. The convolution as such becomes much less costly in the (reduced) PCA domain, since the filtering of the eigenvectors requires $32 \cdot 32 \cdot 10$ multiplications. However, the data need on top to be projected from the original domain to the PCA domain (Eq. 4.4) and back (Eq. 4.6), which leads to $20 \cdot 32 \cdot 31,250$ additional multiplications (in these costs the calculation of the covariance matrix and the EVD have not been evaluated). The computational complexity for the reference and the PCA approach can be seen in Table 4.3. Note that although the G block operates on a reduced data set (i.e. on a subset of the original eigenvectors), the total number of elements at the output of the LTI chain does not change. Only the sample values are adapted due to the re-projection from the reduced PCA domain into the original domain.

Fig. 4.7 and Fig. 4.8 show the PMF and the ACF plot of the corrupted output signal before (\tilde{Y}) and after (\tilde{Y}') 1% truncation error has been applied, using the PCA transform. As already mentioned, in Table 7.1 in the Appendix, a systematic way to perform LTI Processing in the PCA domain is provided.

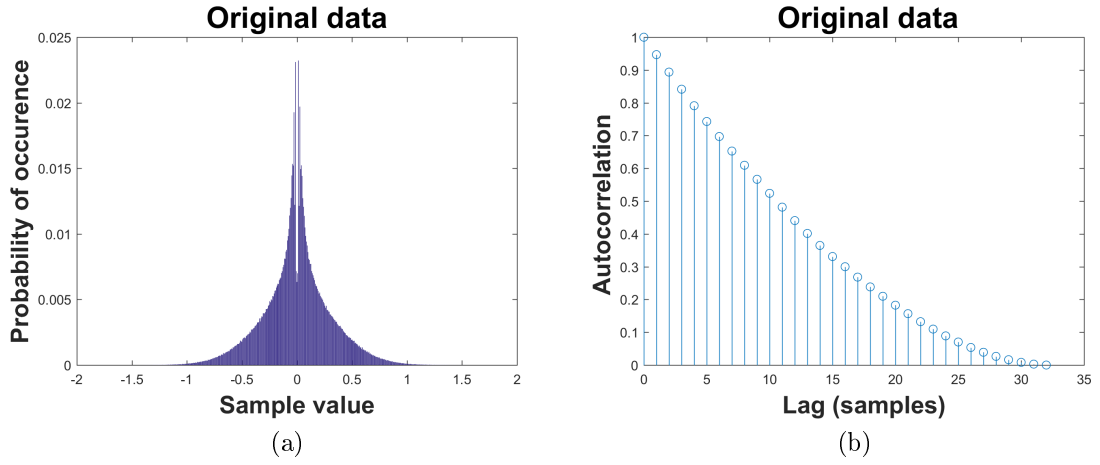


Figure 4.7: PMF (a) and ACF (b) plot of the reference corrupted signal \tilde{Y}

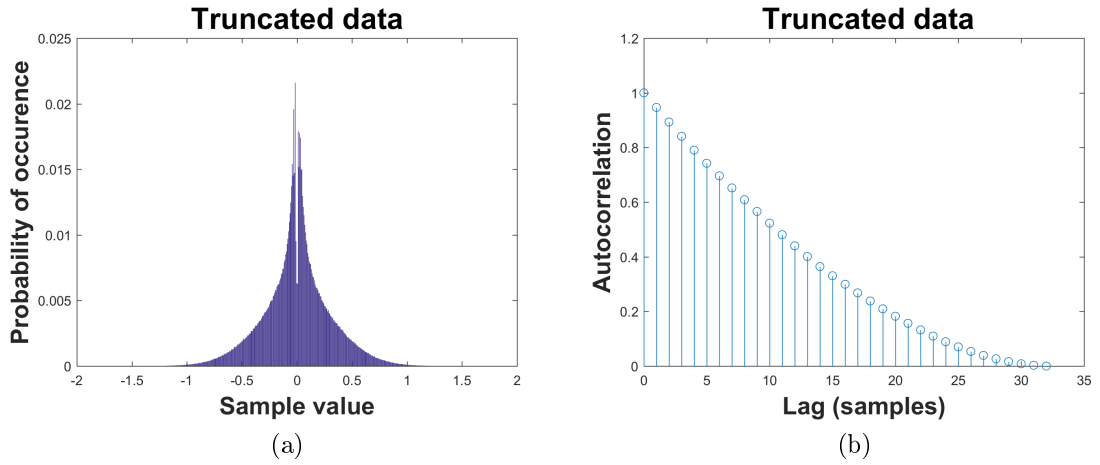


Figure 4.8: PMF (a) and ACF (b) plot of the re-projected (from PCA) corrupted signal $Y \approx \tilde{Y}'$

Table 4.3 Computational cost for filtering (through filter 2) the autocorrelated data (autocorrelated through filter 1)

<i>Comp. complexity (filter 2)</i>	
Original data	Truncated data
$32 \cdot 32 \cdot 31, 250$	$32 \cdot 32 \cdot 10$ (convolution)
-	$20 \cdot 32 \cdot 31, 250$ (projection to PCA domain and back)

4.4 Results and limitations with using the PCA for speeding-up the propagation of the corrupted, autocorrelated signal

Despite the potential benefits from reducing the dimensions for a given autocorrelated signal, whether such a reduction is possible and to what extent, it has to be explored individually for every autocorrelated signal. To make the previous statement more clear, we provide in Fig. 4.9, histograms that are produced for signals that have been autocorrelated through various filters. The types of filters span a wide range: low-pass, high-pass, band-pass and band-stop with 15, 16, 31, 32 filter taps and even-, odd- and non-symmetric coefficients. More specifically, these histograms show how many signal dimensions can be kept for 1% truncation error. The legend "Filter length" denotes the number of filter coefficients, while the legend "Total filter counts" illustrates the number of different filters that have been produced in order to create the histogram. The *x-axis* illustrates the amount of dimensions that need to be kept in order to remove only 1% of the signal energy. In this axis, as it is intuitive understandable, the maximum amount of dimensions coincides with the filter length. The *y-axis* represents the relative frequency of each amount of dimensions that should be kept. The gaps in the figure are there due to the fact that no filters could be produced under the specifications, that would correspond to the histograms in that position of the figure. As it becomes visible, for all histograms, the highest bar indicates that either all the dimensions of the signal should be kept or only a few dimensions can be removed. The cases, where significant savings are possible are the minority. For example, in the histogram at the top-left of the figure, we have the case where 30 filters have been explored with 15 taps each: in 47% of the cases all 15 dimensions must be kept, while only in 6% of the cases only 9 dimensions are sufficient.

Fig. 4.10 shows a small subset of the filters produced in order to create Fig. 4.9. The first and the third rows of the figure provide the impulse responses of the 10 filters illustrated (5 low-pass filters and 5 high-pass filters), while the second and fourth rows provide the corresponding frequency responses. The filters presented have 15 coefficients each (denoted by N in the figure title), specified by the index k and the corresponding value h_k in the graphs illustrating the impulse responses.

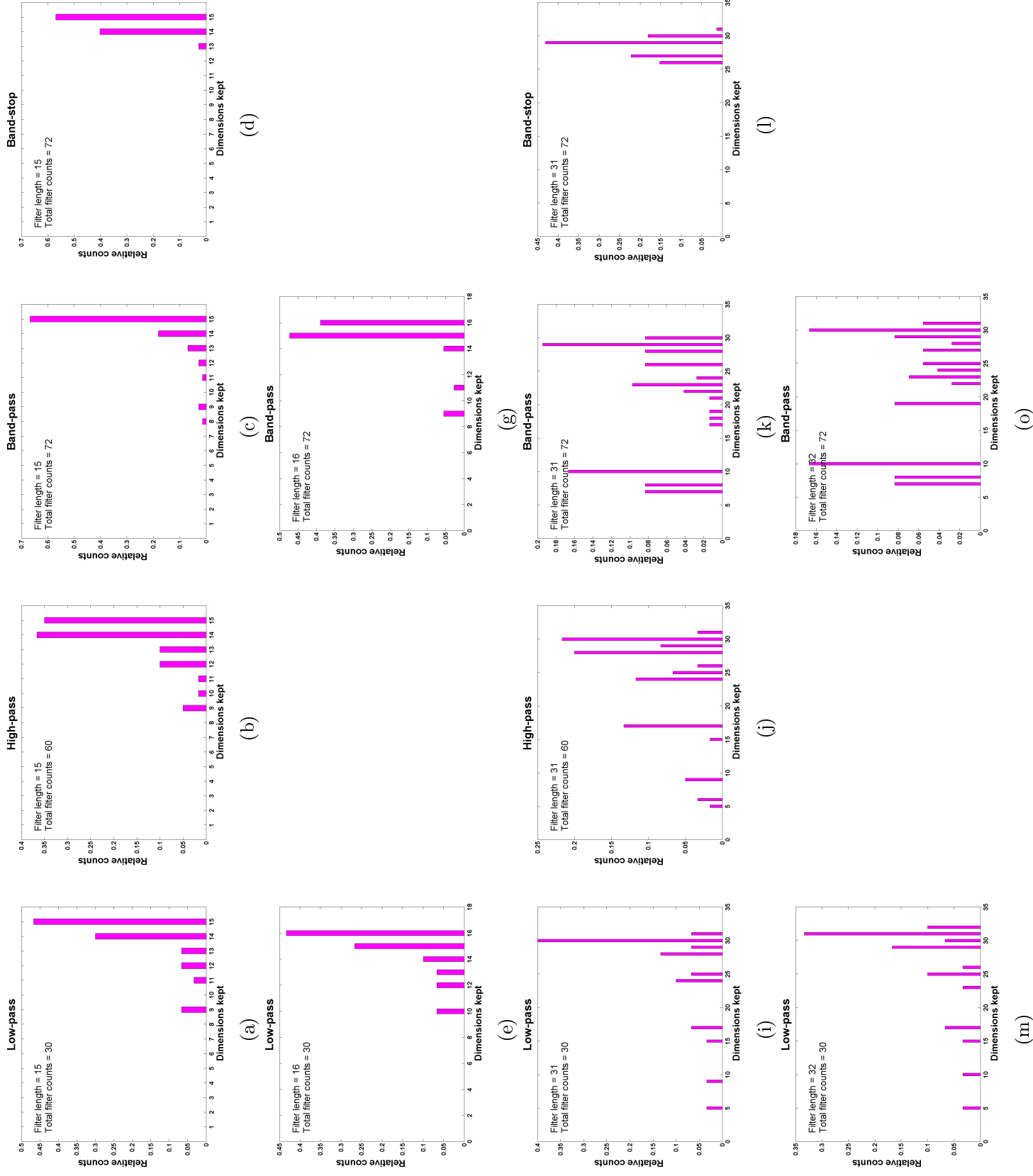


Figure 4.9: Histograms to illustrate gains in dimension reduction for multiple filters with various characteristics

The dotted lines constitute the lines of symmetry of the impulse responses.

In the graphs depicting the frequency responses (second and fourth rows of Fig. 4.10), the blue lines illustrate the magnitude response (or attenuation in this case) in logarithmic scale given by $a_{dB} = -20 \cdot \log_{10}(|H(f)|)$ and measured in decibels, where $H(f) = \frac{Y(f)}{X(f)}$, i.e. the ratio of the output signal spectrum to the input signal spectrum. The red lines give the output phase response ϕ , which can be calculated as $\phi = \arg(|H(f)|)$ and is measured in degrees. The dotted lines in these graphs illustrate the ideal low-pass and high-pass filter frequency responses respectively. The parameter f_{trans} designates the center of the transition band between frequencies that are allowed to pass and those that are cut-off in the ideal low-pass filters (located at the position where the attenuation turns from 0 to 3dB), and between the frequencies that are cut-off and those that pass in the high-pass filters respectively. The f_{slope} parameter denotes the width of the transition band (the between frequencies that are allowed to pass and those that are cut-off) and is equal to $0.01 \frac{f}{f_s}$, where f is the running frequency and f_s the sampling frequency.

Even-Symmetric Low / High Pass Filters: $N = 15$, $f_{slope} = 0.01 f / f_s$

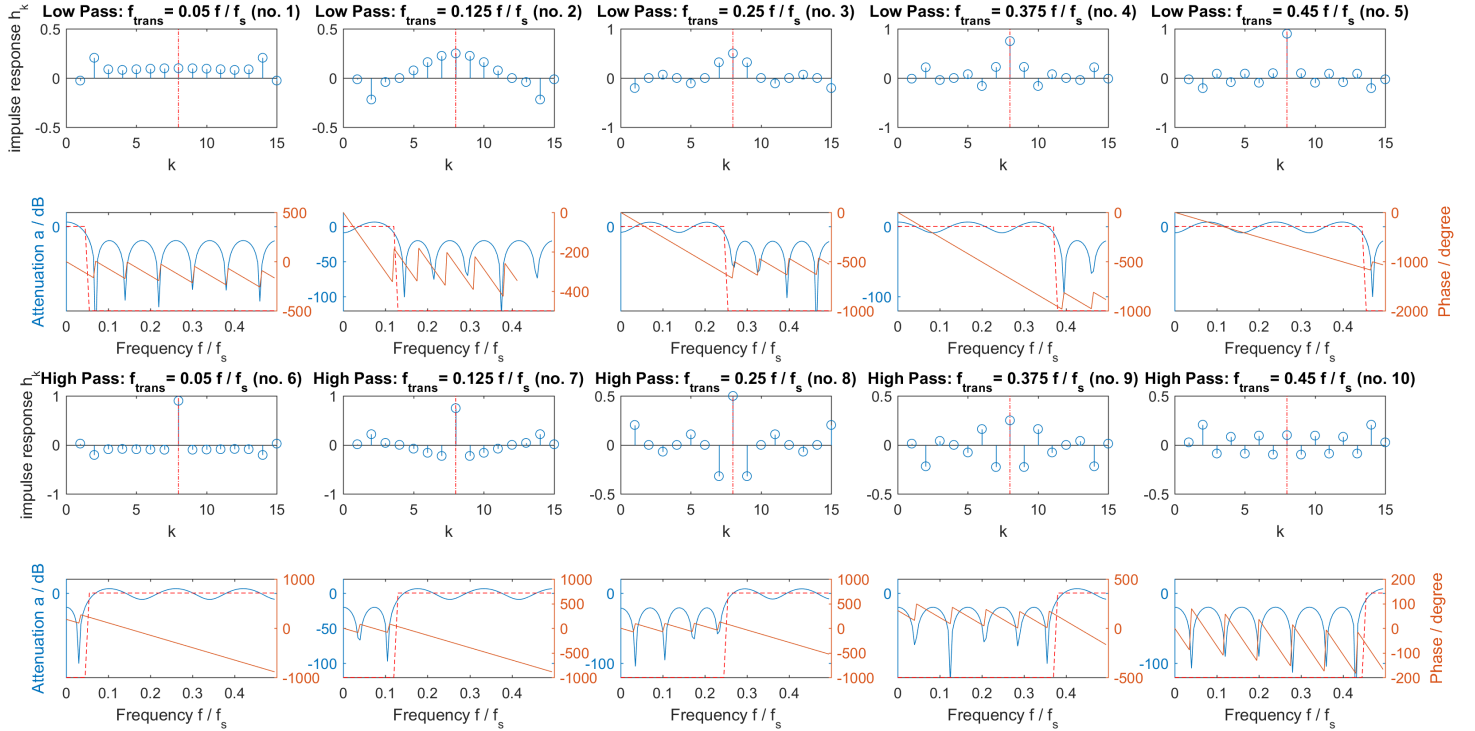


Figure 4.10: Impulse and frequency responses of 5 representative low-pass filters (two upper rows) and 5 representative high-pass filters (two lower rows).

4.5 Summary and link to next chapter

In this chapter possibilities to propagate a corrupted, autocorrelated signal with the help of PCA have been explored. It has been illustrated that although in some cases a reduction of the computational complexity is possible by exploiting the signal inherent autocorrelation, this is not possible in every case; namely, it depends on the type of autocorrelation that is present. This creates the need for a more generally applicable framework, that ideally takes also into account non-linear effects that are present at the filter internals (which is the typical case for digital filters). That is the focus of the next chapter.

Chapter 5

Propagation of Corrupted, Generic Signal

5.1 Outline

Chap. 5 discusses the proposed approaches for propagating a corrupted (or an error) signal, which is potentially autocorrelated. First, in Sec. 5.2 the need for the presented framework is motivated. Sec. 5.3 presents the proposed approach to reduce the computation cost in error-injection experiments and is composed of two parts: Sec. 5.3.1 proposes a framework for the propagation of the error signal when the LTI block is composed exclusively by linear operators. The proposed framework for LTI blocks that include non-linear operators, like the ones that are present due to finite wordlength effects, is presented in Sec. 5.3.2. Sec. 5.4 presents the results for different LTI blocks and Sec. 5.5 concludes the chapter.

5.2 Motivation and preliminaries

Error injection campaigns, as it has already been mentioned require repetitive experiments so that representative results are acquired. The amount of required experiments is based on the desired accuracy (confidence). Ideally, all bit positions should be explored separately. It has been shown earlier in Chap. 3 that when the input signal is uncorrelated, the propagation of the corrupted signal through an LTI block can be performed analytically through propagating the statistical moments of the input signal. However, such an approach explodes computationally when the signal is autocorrelated. Autocorrelation can be exploited by PCA in order to reduce the amount of data that should be processed by the LTI block, thus reducing the computational cost, as discussed in Chap. 4. This approach also has its limitations as it only works for specific types of autocorrelation. Therefore, a more generic and reusable approach is needed that can provide a computational speed-up and does not depend on the signal autocorrelation. Moreover, ideally, it should **take into**

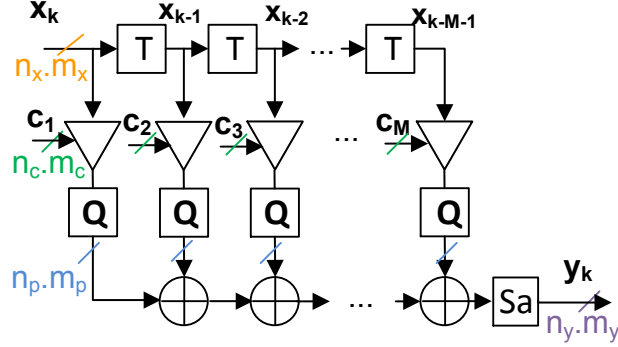


Figure 5.1: An example of non-linear operations due to the finite wordlengths in a direct-form filter

account non-linear effects, that are typically present in digital filters; the latter has not been addressed by related work in the domain (see Sec. 3.4.3).

5.2.1 Effect of finite wordlengths on error propagation

A digital system operates on data that have finite wordlengths and limited accuracy. As, in general, arithmetic operators extend the wordlengths, both on the MSB and the LSB side, additional operations are incorporated in the design to limit the wordlengths to desired accuracy. These are non-linear operations, such as the saturation that takes care of the MSBs and the quantization that takes care of the LSBs.

The properties of LTI blocks have been described in Sec. 2.2.1. In the following, without loss of generality, we will illustrate our concepts on a direct-form filter. Fig. 5.1 shows a signal flow graph (SFG) of a direct-form FIR filter, including non-linear operations. Every signal has a specific number of integer bits (including the sign bit), denoted by n , and fractional bits, denoted by m . Accordingly, the notation for a whole word is given by $n.m$, with the "." denoting the radix point. The indices x, c, p, y denote the input signal, coefficients, products and output signal, respectively. In this example, quantization is applied at the multipliers' output (denoted by Q), which removes LSBs by applying truncation, rounding or some other scheme. Saturation is applied at the final output (denoted by Sa), which provides an upper limit on the signal values. In practice, typically one additional quantization is applied at the final output in order to limit the accumulation error due to the quantized product values. In that case, a compromise is made regarding the removal of LSBs at the multipliers' output and the final output, but for the following discussion this is not significant. The exact effect of these non-linear operations depends on the individual data values. This has impact on the propagation of the error signal during error injection experiments: Assume the filter state variables are equal to 0 and consider an input sample $x_1 = 0.53125$ with coefficient c_1 equal to 2.

We use ν to denote the bit position as used in the two's complement notation for a discrete signal x , with

$$x = -x^{n_x-1} \cdot 2^{n_x-1} + \sum_{\nu=-m_x}^{n_x-2} x^\nu \cdot 2^\nu, \quad (5.1)$$

where x^ν is the bit in position ν with weight 2^ν , and x^{n_x-1} is the MSB sign bit. Table 5.1 shows the impact of quantization at the multiplier's output when the LSB is flipped ($\nu = -5$) and when the next bit is flipped ($\nu = -4$). In the first case, the error gets masked, in the second not.

Table 5.1 A small illustration of the effect of quantization on error propagation

added error	$\mathbf{x_1}$	product	quantized product	error effect
	$(\mathbf{n_x.m_x})$ (1.5)		$(\mathbf{n_p.m_p})$ (3.2)	
error-free	0.53125	1.06250	1	-
$\nu = -5$ flipped	0.50000	1	1	masked
$\nu = -4$ flipped	0.59375	1.18750	1.25	visible

More generally, the impact of quantization and saturation goes beyond masking. Fig. 5.2a shows the effect of quantization (in this case truncation) on the error value using the quantization characteristic. We will illustrate the effect for the cases that a bit-flip causes the error-free signal x to move to a larger value \tilde{x} but the concept can be extended for the case that the direction of the change is reversed. In the figure, the numbers 1-6 correspond to non-quantized x values and for the discussion we assume that the erroneous \tilde{x} value corresponds to the value with the next larger number. If the change due to the error happens from position 1 to 2, or 5 to 6 in the figure, which corresponds to points of discontinuity in the quantized values, the amplitude of the error increases. The difference in the amplitude due to the quantization is visualized by the difference in the length of the blue arrows for the case that the move is from 1 to 2. When the change occurs within a range that corresponds to a horizontal segment in the quantization curve, like changes from 2 to 3, 3 to 4, 4 to 5 then the error gets masked. For the saturation, the effect is different as it can be seen in Fig. 5.2b. When the error causes a move from position 1 to 2, which corresponds to the slope turning from positive to horizontal, the amplitude of the error decreases. In the linear part of the characteristic, obviously there is no effect. Changes in the subsequent positions (2-4) will be masked.

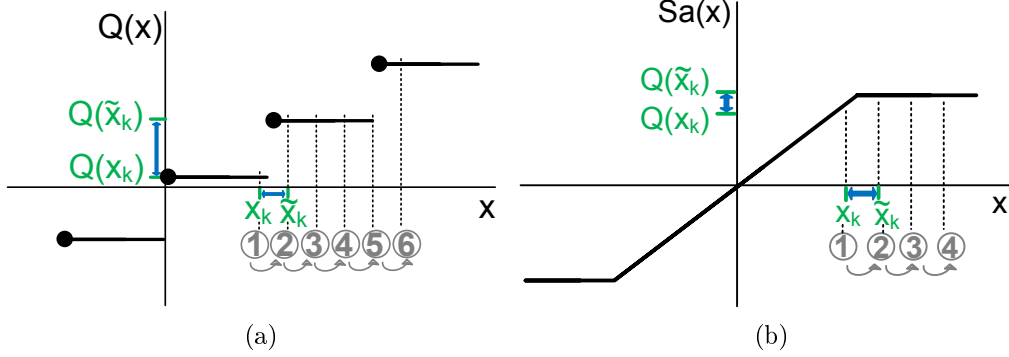


Figure 5.2: Quantization (truncation) and saturation characteristics and effect on error values

5.3 Error propagation approach

5.3.1 Error separation: propagation without finite wordlength effects

In case error propagation is performed without considering finite wordlength effects, the property of linearity can be exploited to reduce the computational complexity during error propagation of errors injected in error-free signals with any type of autocorrelation. The conventional approach to propagate errors (as shown in the upper part of Fig. 5.3) requires the following steps: perform error injection at the input signal x , perform convolution of the corrupted input signal \tilde{x} and extract the error at the output e_y .

However, linearity allows the error-free signal and the erroneous signal to be decoupled. Based on the superposition principle of linear blocks, considering \mathcal{C} as the impulse response of the block and "*" the convolution operator, the corrupted output signal can be derived as:

$$\tilde{y} = \mathcal{C} * \tilde{x} = \mathcal{C} * x + \mathcal{C} * e_x. \quad (5.2)$$

That means, for every injection experiment, it is sufficient that only the error signal at the input is derived and propagated to the output. For every error sample at the input, one weighted (by the error sample) impulse response is produced at the output (as already discussed in Sec. 4.3.1 and shown in Fig. 4.3). *In total, the calculation of $s + 1$ (as many as the number of sections) weighted impulse responses is required to get the full error signal at the output, as there is only one error sample per section. The processing of the error signal separately allows for a significant reduction of the computational complexity.* Computational complexity will be discussed in Sec. 5.3.3.

The lower part of Fig. 5.3 shows the principle. In case the corrupted

signal at the output is of interest, the error-free signal should be processed once and the output error signal should be added to the processed error-free signal.

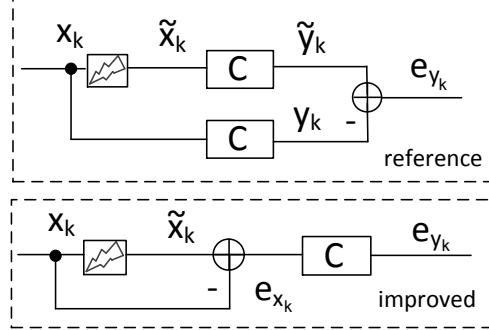


Figure 5.3: Two functionally equivalent approaches to calculate the error signal at the output

5.3.2 Error-free terms reuse: propagation combined with finite wordlength effects

This subsection discusses the case that non-linear effects due to finite wordlengths take place during the error propagation. In the specific case that the non-linear effects take place only at the block output, the previous approach with the error separation can be used. The non-linear operation is then applied on the derived corrupted output signal. However, in the more general case that non-linear effects take place after intermediate results in the SFG (as shown in Fig. 5.1 for the direct-form FIR filter), the previous approach is not applicable. In this case, intermediate results have to be accessed. The first step in the proposed methodology is to use the data organization, already introduced in Sec. 4.3.1. With this organization, as in every section the same sample gets corrupted over different injection scenarios, computations related to the error-free samples can be reused. Moreover, depending on the specific LTI block, parallel execution of the sections can be exploited to speed-up the experiments. In Sec. 5.3.2, we illustrate the concept in detail for a direct-form FIR filter and discuss briefly some other LTI structures in Sec. 5.3.3.

Illustration on direct-form FIR filter. Instead of performing scalar operations to implement the difference equation (see Eq. 2.1) of a direct-form FIR filter, a vector-vector operation can be used for calculating every output sample. To further improve the computation time, a vector of output samples can be calculated in parallel, using the banded Toeplitz matrix of the impulse response. The Toeplitz matrix is often used to describe convolutive relationships [45]. To construct the Toeplitz matrix for an LTI block, the difference equation is used. Assuming we have an input sample sequence of length L and we are interested in L samples of the output, by letting $k = 1, 2, \dots, L$ in Eq. 2.1, the ordering of the coefficients for

calculating each of the output samples y_1, y_2, \dots, y_L can be derived. The convolution operations become then a matrix-vector multiplication. For the simple case that $M=3$ and $L=3$, the Toeplitz matrix is:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} c_3 & c_2 & c_1 & 0 & 0 \\ 0 & c_3 & c_2 & c_1 & 0 \\ 0 & 0 & c_3 & c_2 & c_1 \end{bmatrix} \cdot \begin{bmatrix} x_{-1} \\ x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}. \quad (5.3)$$

The term *banded* is due to the fact that a finite number of diagonals have non-zero entries and the rest of the entries are zero. The values x_{-1}, x_0 perform the initialization of the filter state variables. In case these are zero, only the non-colored part of the matrix forms the Toeplitz matrix. In the following the banded Toeplitz matrix will be denoted by $[T_c]$. To further improve parallelism, instead of calculating one output vector at a time, multiple vectors can be calculated in parallel. That means, that instead of having a single input vector, multiple input vectors are applied. In our case, each section constitutes an input vector, and, then

$$[Y] = [T_c] \cdot [X], \quad (5.4)$$

provides the complete convolution operation for all sections in the form of a matrix-matrix multiplication, with $[Y]$ being the matrix of the filtered data.

It can be seen that every time an error injection takes place, only the first row of the data matrix changes (for a visual representation of the data matrix, see the earlier presented Fig. 4.4). Given our goal of incorporating the non-linear operations in the framework, the elements of the $[Y]$ matrix are now illustrated at the level of atomic multiplications and product additions in Eq. 5.5 below. This is the granularity that is required to implement the non-linear operations. For the illustration, we map the $S \cdot (s+1)$ elements of the $[X]$ matrix as follows: x_k corresponds to $x_{i,j}$ with $i = ((k-1) \bmod S) + 1$ and $j = \lfloor (k-1)/S \rfloor + 1$, so that x_1 becomes x_{11} , x_{S+1} becomes x_{12} etc. The quantization (as a representative example of non-linear operation) has been incorporated, assuming that it takes place right after the multiplication of the data samples with the filter coefficients as in Fig. 5.1. The highlighted part in the matrix in Eq. 5.5, illustrates the part that remains unaltered every time a new injection experiment takes place. Exactly this fact is exploited to speed-up the experiments by having these terms being reused for every new injection scenario.

In the explanation above, we have assumed that the initial state variables are zero for each section (we have considered only the non-colored part in Eq. 5.3). Actually, it is required that the filter state variables are initialized, i.e. data stored in the filter before the error sample enters the filter have to be taken into account in order to accurately account for non-linear effects. For example, to do

$$[Y] = \begin{bmatrix} Q(c_1 \cdot x_{11}) & Q(c_1 \cdot x_{12}) & \dots & Q(c_1 \cdot x_{1(s+1)}) \\ Q(c_2 \cdot x_{11}) + \sum_{n=1}^1 Q(c_n \cdot x_{3-n,1}) & Q(c_2 \cdot x_{12}) + \sum_{n=1}^1 Q(c_n \cdot x_{3-n,2}) & \dots & Q(c_2 \cdot x_{1(s+1)}) + \sum_{n=1}^1 Q(c_n \cdot x_{3-n,s+1}) \\ Q(c_3 \cdot x_{11}) + \sum_{n=1}^2 Q(c_n \cdot x_{4-n,1}) & Q(c_3 \cdot x_{12}) + \sum_{n=1}^2 Q(c_n \cdot x_{4-n,2}) & \dots & Q(c_3 \cdot x_{1(s+1)}) + \sum_{n=1}^2 Q(c_n \cdot x_{4-n,s+1}) \\ \vdots & \vdots & \ddots & \vdots \\ Q(c_M \cdot x_{11}) + \sum_{n=1}^{M-1} Q(c_n \cdot x_{S+1-n,1}) & Q(c_M \cdot x_{12}) + \sum_{n=1}^{M-1} Q(c_n \cdot x_{S+1-n,2}) & \dots & Q(c_M \cdot x_{1(s+1)}) + \sum_{n=1}^{M-1} Q(c_n \cdot x_{S+1-n,s+1}) \end{bmatrix} \quad (5.5)$$

the initialization for the direct-form filter, the $S - 1$ last elements of each section should be copied to the beginning of the subsequent section. At the beginning of the first section, $S - 1$ zeros should be added. When initialization of the sections is incorporated, the row in the $[X]$ matrix that changes at every new injection is not the 1^{st} but the S^{th} one. As in a matrix-matrix multiplication every entry in the output matrix is derived by combining the products of the elements of the rows and columns of the input matrices (and summing the results), we have to calculate S products for every entry of matrix $[Y]$. *By changing one row (at every new injection) in the $[X]$ matrix, only one product in every entry changes out of the S products that are required without exploiting the reuse. In total, $S \cdot (s + 1)$ multiplications (as many as the matrix entries) need to be calculated at every successive injection experiment instead of $M \cdot S \cdot (s + 1)$ multiplications (with M being the filter length), which would be required for the calculation of the whole convolution matrix.* More information is provided in Sec. 5.3.3.

Overall, in the proposed framework, the following steps are required (as shown in Proc. 2). As initialization steps, the banded Toeplitz matrix is created (including appropriate initialization) and the S^{th} row of the $[X]$ data matrix is zeroed-out, creating matrix $[X_{mod}]$ (steps 1 and 2). The non-linear operations are applied on the individual operations, while performing $[T_c] \cdot [X_{mod}]$, as step 3. The result is stored as the temporary matrix $[Y_{mod}]$. The main loop with steps 5, 6, 7, 8 is the core part of the algorithm, which: performs the injection of each bit-flip, multiplies the corrupted $[X]$ row with the Toeplitz matrix, applies the non-linear operation (like quantization) on the result and adds the result on the temporary matrix $[Y_{mod}]$. As a final step (step 6) the output error free signal is subtracted from the corrupted signal to derive the output error signal.

5.3.3 Reduction of computational complexity.

The input signal $[X]$ is a $S \cdot (s + 1)$ matrix filtered across the columns, using a filter with M coefficients. Here, we discuss the cost in terms of numbers of operations for the presented approaches. Especially the recurring cost of costly operations, i.e. multiplications, is highlighted in bold-italic font.

(i) **Reference convolution:** It follows that the computational complexity for deriving the output $[Y]$ for every injection scenario, using the conventional convolution operation, is $\mathbf{M \cdot S \cdot (s + 1)}$. The cost is linearly dependent on the number of the filter coefficients. On top, $(M - 1) \cdot S \cdot (s + 1)$ additions are required

Procedure 2 Pseudocode of the banded Toeplitz matrix error propagation methodology

Input: error-free signal, bit-flip position, LTI impulse response

Output: modified error signal after finite wordlength operations

- 1: create banded Toeplitz matrix
 - 2: zero-out S^{th} row in $[X]$ matrix, creating $[X_{mod}]$
 - 3: apply all required non-linear operations on each of the individual products (and/or sums) in $[Y_{mod}] = [T_c] \cdot [X_{mod}]$
 - 4: **for** each bit-flip position **do**
 - 5: bitwise xor each sample in the S^{th} row of $[X]$
 - 6: perform vector-matrix multiplication and apply all required non-linear operations
 - 7: add this matrix to the matrix $[Y_{mod}]$
 - 8: subtract from the error matrix the error-free matrix to acquire the error signal
 - 9: **end for**
-

to calculate the intermediate and final filter sums. Finally, a cost of $S \cdot (s + 1)$ subtractions is present in case the error signal is extracted at the output. This approach can be applied with or without taking into account non-linear effects. Actually this cost is the same, both in the case that we apply the filtering across the sections (including the initialization of the filter state) and the case where we perform the filtering on a long input sequence (without sectioning).

(ii)**Error separation:** In this case the error signal is extracted from the corrupted input and $(s + 1)$ subtractions are required. To derive the error signal at the output, every error sample is multiplied with the filter impulse response, leading to $M \cdot (s + 1)$ multiplications.

(iii)**Banded Toeplitz matrix:** In the proposed approach, the recurring cost during the injections is given by $S \cdot (s + 1)$ for the multiplications, and an equal number of sums is required. An additional cost of $S \cdot (s + 1)$ subtractions is present in case the error signal is extracted at the output. The preparation step incurs a one-time cost and requires the creation of the matrix $[X_{mod}]$, which includes $M \cdot S \cdot (s + 1)$ multiplications and additions. An additional cost of $S \cdot (s + 1)$ subtractions is present so that the error signal is extracted at the output. This approach can be applied with or without taking into account non-linear effects in the filter.

Other LTI blocks. As already discussed earlier, the benefits from organizing the data in sections to implement the error injection experiments for the non-linear case, depend on the specific LTI block. The potential benefits from reusing the error-free terms are interlinked with the LTI SFG. Here we discuss four additional blocks: the transposed FIR filter, the filter bit-plane implementation, the lattice filter and the DFT block. By observing the SFG of the transposed filter (see Fig. 5.4a), which is a frequent practical implementation, the following becomes apparent: for the typical case that the quantization takes place after the multipliers, the results

are the same with the direct-form. Non-linearities in the bit-plane implementation can be handled as in the fully linear case, by applying them at the final output. The lattice filter has a more complex structure, as it can be seen in Fig. 5.4b. An erroneous input sample will impact the result of $\frac{M}{2} \cdot (M + 1)$ multiplications from the upper branch and $\frac{M-1}{2} \cdot M$ from the lower branch for a filter of M stages. So, through reusing, only M^2 multiplications have to be implemented for one section, compared to $2 \cdot M^2 + M - 1$ that would be needed for the reference case. For a

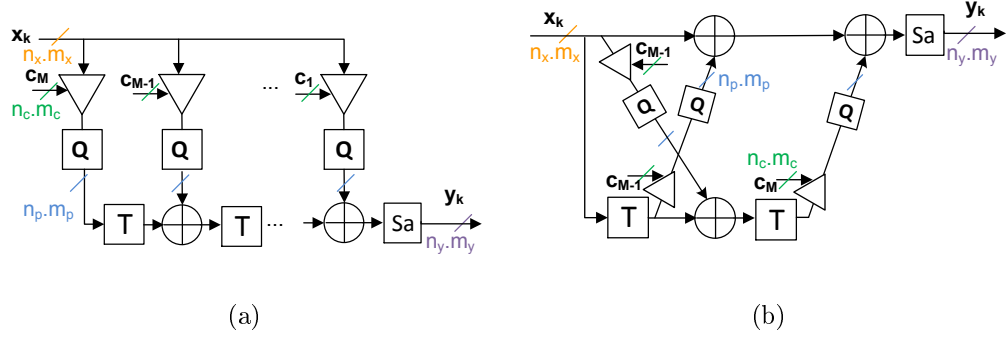


Figure 5.4: SFGs for transposed and lattice (only 2 stages) filters

DFT block of length N , the section length is N . At every new injection only a single sample changes, reducing the complexity from N^2 for a single section to N .

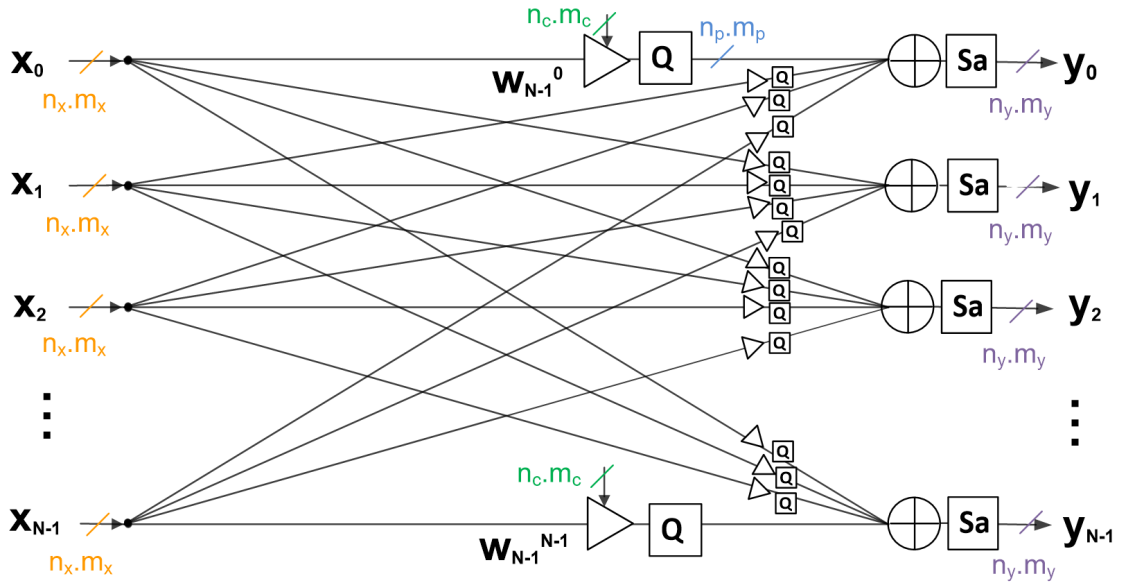


Figure 5.5: SFG for a DFT block of length N

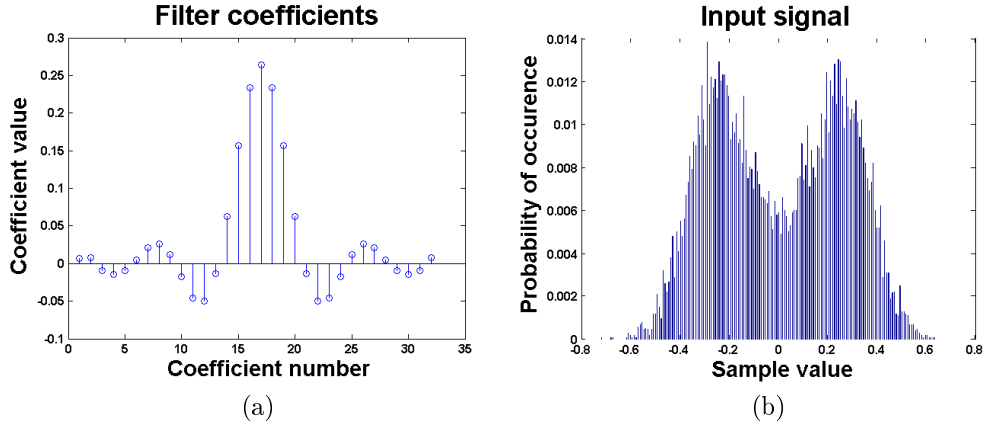


Figure 5.6: Coefficients for the 32-tap filter and PMF of the input signal

5.4 Results

5.4.1 Set-up

To illustrate our approach, we use the baseband model of a BPSK communication line, as has already been depicted in Fig. 2.2. The block under investigation is the filter at the receiver. The filter coefficients have been derived using the square root of the raised cosine filter function. Fig. 5.6a shows the coefficient values for a 32-tap filter. Fig. 5.6b shows the PMF of the signal for 10^6 samples, as it appears in the buffer memory before the filter. We explore the outcome of bit-flip injections in this signal, after it propagates through the filter, first without and then including non-linear effects at the filter internals. The results obtained after such an analysis with the proposed approach are identical with the ones obtained by simulation. Therefore the exact amount of channel noise that has been added is just a parameter and the BER at the end of the line is just the outcome of such an analysis. As the accuracy is exactly the same between using a simulator filter function and our approach, the two main objectives here are: the (quantitative) computational complexity and the execution time. The numbers for the complexity have been derived according to the formulas in Sec. 5.3.3 and Sec. 5.3.3. Regarding execution time, in absence of a single objective metric, for simplicity, instead of the Matlab *timeit* we use the *tic*, *toc* function which counts the elapsed computation time in seconds. Obviously, for multiple runs of the same code, a stochastic variation is present in the results of the computation time. Here, the focus is on the relative gain that the proposed approach offers rather than the absolute execution time. All experiments have been performed on an Intel Core i7 (2.40GHz, 8GB RAM) under the same conditions (e.g. no other tasks running in parallel). For the direct-form filter, we implement the filtering both on a long input sequence (under the name "ref. long") and on a sectioned input sequence (under the name "ref. sect."), using *for loops*, as a reference. When we consider also non-linear effects, we provide additionally results for a lattice filter and a DFT block. All experiments were performed for 10^6 samples.

5.4.2 Without non-linear effects

In the following, bit-flips are injected at each position of the input wordlength by separating the error signal (denoted by "error sep."). Table 5.2 shows the results for the cases, when the input wordlength $n_x + m_x$ (denoted as "input wl") has the values 8, 16, 24, 32. It is observed that as the wordlength increases, the results follow a linear trend both in computational complexity and in execution time.

Table 5.2 Results for 32-tap direct-form filter under different input wordlengths without non-linear effects

input wl	<i>Comp.complexity</i>		<i>Exec.time (sec)</i>		
	ref. long & ref. sect.	error sep.	ref. long	ref. sect.	error sep.
8	$8 \cdot 32 \cdot 10^6$	$8 \cdot 10^6$	13	16	0.09
16	$16 \cdot 32 \cdot 10^6$	$16 \cdot 10^6$	26	32	0.17
24	$24 \cdot 32 \cdot 10^6$	$24 \cdot 10^6$	38	48	0.26
32	$32 \cdot 32 \cdot 10^6$	$32 \cdot 10^6$	51	64	0.35

Table 5.3 illustrates the results for an 8-bit input wordlength but different filter lengths. Although computational complexity increases as the number of coefficients increase, the execution time remains stable for the presented filter lengths.

Table 5.3 Results for 8-bit input wordlength under different direct-form filter lengths without non-linear effects

M	<i>Comp.complexity</i>		<i>Exec.time (sec)</i>		
	ref. long & ref. sect.	error sep.	ref. long	ref. sect.	error sep.
16	$16 \cdot 8 \cdot 10^6$	$8 \cdot 10^6$	14	16	0.1
32	$32 \cdot 8 \cdot 10^6$	$8 \cdot 10^6$	13	16	0.09
64	$64 \cdot 8 \cdot 10^6$	$8 \cdot 10^6$	13	17	0.09

5.4.3 Including non-linear effects

Here, for the direct-form filter, the banded Toeplitz matrix approach is used, as proposed in Sec. 5.3.2, that allows the incorporation of non-linear effects due to the finite wordlengths. Regarding the wordlengths we select $n_x.m_x$ equal to 1.7 and $n_c.m_c$ equal to 1.14. Table 5.4 shows how the number of errors that are visible at the filter output changes depending on the quantization scheme applied at the

filter products (with wordlengths equal to $n_p.m_p$) for four different bit positions. The results are provided for a 32-tap filter, and the adopted quantization scheme is truncation. The 1st quantization scheme represents the case where no truncation takes place. As it is expected, in this case, for all presented bit positions, the injected errors corrupt all signal samples at the output. The same holds for the 2nd quantization scheme, according to which, 2 LSBs of the products are truncated. For quantization scheme 3, the results begin to differ as the number of errors that are visible at the output reduces when injection takes place at bit positions 0, -1. More errors are masked for quantization scheme 4, where injections at bit position -3 also lead to masking effects. For the last quantization scheme where only 8 fractional bits are kept at the word, error masking occurs for injection at every bit position except the MSB.

Table 5.4 Erroneous (non-masked) sample counts for 32-tap direct-form filter

bit-flip position	<i>Quantization schemes ($n_p.m_p$)</i>				
	1	2	3	4	5
	(2.21)	(2.17)	(2.12)	(2.10)	(2.8)
$\nu = -7$	10^6	10^6	622,841	305,532	100,447
$\nu = -5$	10^6	10^6	975,560	622,476	277,327
$\nu = -3$	10^6	10^6	10^6	986,873	570,133
$\nu = -1$	10^6	10^6	10^6	10^6	971,621
$\nu = 0$	10^6	10^6	10^6	10^6	10^6

Table 5.5 and Table 5.6 provide the computational complexity and execution time for injecting errors at all bit positions in the input signal, under a given quantization scheme at the filter products. Table 5.5 shows the trend when the input wordlength increases under a fixed-tap-length filter, while Table 5.6 shows the trend when the filter length increases under a fixed wordlength. The proposed approach is 7 to 14 times faster than the reference.

Table 5.7 shows the trend when an all-zero lattice filter length increases under a fixed wordlength. As reference, a non-vectorized and a vectorized implementation are used (denoted by "ref. non-vec." and "ref. vec."). As it can be seen, although complexity reduces by reusing error-free terms (denoted by "reuse"), the vectorized version is still better in terms of execution time. Table 5.8 shows the trend when a DFT block length increases under a fixed wordlength. As reference, a vector-matrix multiplication is implemented (denoted by "ref. sect."). By reusing error-free terms (denoted by "reuse"), both complexity and execution time improve significantly. The execution time depends on several parameters, among which, the number of loops and storage requirements (for the error-free terms). Therefore, it is observed that for lengths 4 and 256, the required time is almost equal.

Table 5.5 Results for 32-tap direct-form filter under different input wordlengths with non-linear effects

input wl	<i>Comp.complexity</i>		<i>Exec.time (sec)</i>		
	ref. long & ref. sect.	Toepl.	ref. long	ref. sect.	Toepl.
8	$8 \cdot 32 \cdot 10^6$	$8 \cdot 10^6$	971	1019	130
16	$16 \cdot 32 \cdot 10^6$	$16 \cdot 10^6$	1933	1065	140
24	$24 \cdot 32 \cdot 10^6$	$24 \cdot 10^6$	2904	1587	146
32	$32 \cdot 32 \cdot 10^6$	$32 \cdot 10^6$	3860	2123	148

Table 5.6 Results for 8-bit input wordlength under different direct-form filter lengths with non-linear effects

M	<i>Comp.complexity</i>		<i>Exec.time (sec)</i>		
	ref. long & ref. sect.	Toepl.	ref. long	ref. sect.	Toepl.
16	$16 \cdot 8 \cdot 10^6$	$8 \cdot 10^6$	962	1001	127
32	$32 \cdot 8 \cdot 10^6$	$8 \cdot 10^6$	971	1019	130
64	$64 \cdot 8 \cdot 10^6$	$8 \cdot 10^6$	997	1004	132

Table 5.7 Results for 16-bit input wordlength under different all-zero lattice filter lengths with non-linear effects

M	<i>Comp.complexity</i>		<i>Exec.time (sec)</i>		
	ref. non-vec. & ref. vec.	reuse	ref. non-vec.	ref. vec.	reuse
15	$16 \cdot 464 \cdot 10^6/15$	$16 \cdot 225 \cdot 10^6/15$	3405	288	461
31	$16 \cdot 1952 \cdot 10^6/31$	$16 \cdot 961 \cdot 10^6/31$	6934	293	464
63	$16 \cdot 8000 \cdot 10^6/63$	$16 \cdot 3969 \cdot 10^6/63$	14145	309	482

Table 5.8 Results for 16-bit input wordlength under different DFT block lengths with non-linear effects

N	<i>Comp.complexity</i>		<i>Exec.time (sec)</i>	
	ref. sect.	reuse	ref. sect.	reuse
4	$16 \cdot 4 \cdot 10^6/4$	$16 \cdot 10^6/(4 \cdot 4)$	299	58
32	$16 \cdot 32 \cdot 10^6/32$	$16 \cdot 10^6/(32 \cdot 32)$	148	15
256	$16 \cdot 256 \cdot 10^6/256$	$16 \cdot 10^6/(256 \cdot 256)$	917	59

5.5 Summary and link to next chapter

In this chapter a framework has been proposed in order to speed-up error injection experiments in LTI blocks that have no non-linear effects and for those that do. The approach was illustrated on various LTI blocks with varying benefits. The next chapter complements the contribution by presenting a survey of hardware-based mitigation approaches.

Chapter 6

A Classification of Hardware-Based Resilience Techniques at the Higher Abstraction of Digital Systems

6.1 Outline

Chap. 6 discusses the categorization and characterization of hardware-based mitigation techniques (at the higher hardware abstraction). Sec. 6.2 lays down the motivation and contribution of the chapter. Relevant definitions and the rationale of the proposed classification are discussed in Sec. 6.3. The core classification follows in Sec. 6.4. Sec. 6.5 summarizes the chapter.

6.2 Introduction

The current chapter presents a classification scheme for organizing the research domain on mitigation of functional errors at the higher hardware abstraction layers that manifest during the operational lifetime, and maps representative work for each category. Given the multitude of reliability issues in modern digital systems, it is vital to set the boundaries of the current survey: The survey discusses resilience schemes at the architectural/microarchitectural layer, which have increased in diversity during the last decades, following the evolution of computer architecture, parallel processing and general system design. Reliability-related errors that occur due to hardware-design errors, *insufficiently specified systems* or *malicious attacks* [7] or erroneous software interaction (i.e. manifestation of software bugs due to software of reduced quality [47]) are beyond the current scope. Techniques to mitigate permanent errors that have been detected during testing in order to improve yield or lifetime are not included. Techniques to tackle permanent errors due to device and wire wear-out are incorporated though.

The interested reader can find a corresponding classification and mapping for techniques at the software stack, in our publication found in [69]. Moreover, from this point on, the symbol [®] will be used to refer the reader to the supplementary material (see [69] and ACMCSUR website) for additional information. A discussion of the bulk of the mapped literature work takes place in the supplementary material.

6.3 Context and useful terminology

6.3.1 Resilient digital system design

Reliability is defined as the probability that over a specific period the system will satisfy its **specification**, i.e. the total set of requirements to be satisfied by the system. **Functional reliability** is defined as the probability that over a specific period of time the system will fulfill its **functionality**, i.e. the set of functions that the system should perform [34]. Functional reliability is related with correcting binary digits as opposed to parametric reliability that deals with aspects of variations in operation margins [72]. Functionality is one of the major elements of the specification set. Others may be minimum performance (e.g. throughput [ops/s], computational power [MIPS]), maximum costs (e.g. silicon area [mm²], power [W], energy [J/op], latency [s/op]). In the following, the term reliability will be used to denote the functional reliability. The term **resilience** describes the ability of a system to defer or avoid (functional) system failures in the presence of errors. When a system becomes more resilient, its reliability is increased. The terms reliable and resilient (system design) will be used interchangeably [®].

6.3.2 Computing terminology

6.3.2.1 Terminology on abstraction layers

This survey includes techniques implemented at the microarchitecture and architecture layers, as has been shown in Fig. 1.1. The term **platform** denotes a system composed of architectural and microarchitectural components together with the software required to run applications. When the system is not SW-programmable, like some small embedded systems are, the term platform denotes only the hardware part.

Platform HW. Microarchitecture describes how the HW constituent parts are connected and inter-operate to implement the operations that the HW supports. It includes the memory system, the memory interconnect and the internals of processors [31]. This applies both to very flexible SW-programmable processors, where an instruction-set is present to control the operation sequence, and to dedicated-HW processing components. Dedicated-HW processors feature minimum

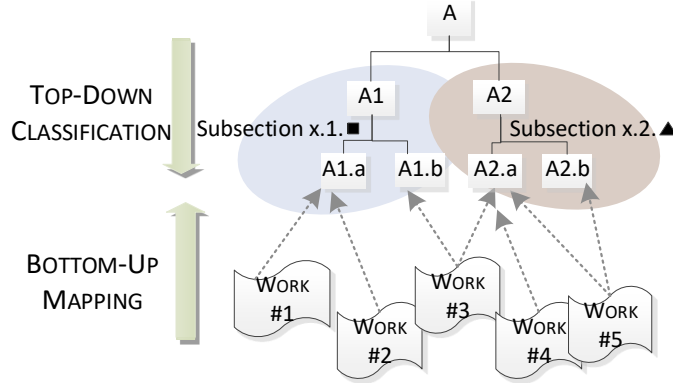


Figure 6.1: Top down splitting to create the classification tree and mapping of the related work

to limited flexibility. Both SW-programmable and dedicated components can be mapped on highly reconfigurable fabrics, like field-programmable gate arrays (FPGAs). The primary difference compared with the SW-programmable processors is that not only the control flow but also the data flow can be substantially changed/reconfigured. The microarchitecture together with the Instruction Set Architecture (ISA) constitute the computer **architecture** (although the term has been recently used to include also other aspects of the design [31]). In general, the term **HW module** denotes a subset of the digital system's HW, the internals of which cannot be observed (or it is chosen that they are not observed), correspondingly to the term *black box* [72]. To define a HW module, its functionality and its interface with the external world must be described. At the microarchitectural and architectural layer, examples of HW modules are a multiprocessor system, a single core, a functional unit, the row of a memory array, a pipeline stage, a register (without exposing the internal circuit implementation though). In the context of this survey, the term **platform HW** is an umbrella term, that encompasses the microarchitecture and architecture layers of a system.

6.3.3 Rationale of the classification and its presentation

The proposed classification tree is organized using a top-down splitting of the types of techniques that increase the system resilience. It is accompanied by a mapping of related work (see Fig. 6.1). The top-down splitting allows to reach a comprehensive list of types of techniques, which can always be expanded further on demand. Splits are created based on properties of the techniques, which allow them to be grouped together. More specifically, the properties in the proposed framework regard: (1) the effect that the techniques have on the execution and (2) the changes that are required on the system design for a technique to be implemented. The properties will be elaborated as the tree is being presented. Other organizations are also possible,

like organizing the splits around the system functionality, hardware components, types of errors (transient, intermittent, permanent), types of resilience metrics or the application domains. The aforementioned organization is chosen in order to stress the reusability of techniques but also to enable the better understanding of hybrid combinations. This is especially supported through the complementarity of the categories. It is important to note that many actual approaches that increase resilience typically represent hybrids and do not fall strictly into only one of the categories.

For the presentation of the classification tree, the following structure is followed. First, the main classes are presented for the different techniques. Within each class, subcategories are presented which are illustrated with the help of a figure. Groups of nodes are chosen to be discussed together. For the visualization of the groups, bubbles with different colors are used, along with the subsection number and a small geometrical shape (see Fig. 6.1). The colors and the geometrical shapes are used to enable a more explicit link with the corresponding subsections in the text. Especially the geometrical shapes are used for the facilitation of the reader in the black-white printed version. The order of the leaves, the colors and the geometrical shapes do not indicate the significance or the maturity of the techniques. For each of the classes, pros and cons are discussed, based on general properties bound to each class. Among the aspects considered are: area and power overhead, performance degradation (in terms of additional execution cycles), mitigation latency (delay until the scheme fulfils the intended mitigation function), error protection, general applicability, storage overhead. An overview of those for the different classes can be found in Tables 7.2, 7.3, 7.4 in the Appendix. In parallel, representative related work is discussed to further illustrate the subcategory concept and demonstrate the usefulness of the proposed classification scheme for classifying existing (and future) literature ^⑤. Moreover, in Tables 7.2, 7.3, 7.4 in the Appendix, a crude indication of the amount of literature for each of the classes is performed.

Finally, the notion of **non-determinism** is introduced and will be discussed whenever appropriate. A common technique to mask the effect of errors is by employing replication. During replication, an algorithmic function is executed again, often by using extra hardware or software. However, deterministic execution is required for replicas to work. Determinism ensures that different runs of the same function under the same input will produce identical outcomes. In practice, deterministic execution is challenged by a multitude of non-deterministic events [64], [82], [65]. Examples include non-predictable user or sensor inputs, timers, random numbers, system calls and interrupts ^⑤.

6.4 Platform hardware mitigation techniques

To make digital systems more robust, functional capabilities need to be provided that would be unnecessary in a fault-free environment. This section focuses on

techniques that modify the hardware capabilities for reliability purposes. The goal is to provide non-overlapping categories that cover the broad range of error mitigation and resilience techniques. The complete classification scheme is shown in Fig. 6.11 in Sec. 6.4.5. A high level split for the proposed classification tree is shown in Fig. 6.2. Techniques are first classified into techniques that continue the execution forward (**forward**) and those that move the execution to an earlier point (**backward**). Both categories are further split into techniques that require the addition of HW modules in the platform at design time (**additional HW modules provision**) and techniques that keep the amount of modules the same (**HW modules amount fixed**). In the latter case, only a HW or SW controller may be needed. These four classes are discussed in the following subsections, as shown in Fig. 6.2. Main criteria for further categorization include whether modifications are required in: existing functionalities, existing design implementations, resource allocation, operating conditions, the interaction with neighbouring modules, storage overhead. Leaves of the tree have an accompanying simple ordinal number for identification. The numbers (together with the leaves) are collectively shown in Fig. 6.11.

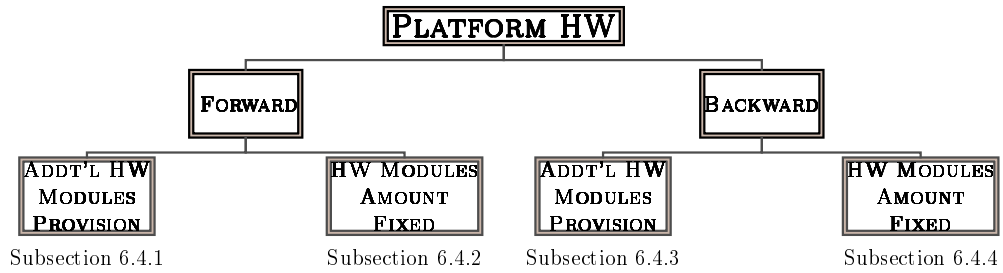


Figure 6.2: Basic classification¹ for techniques at the platform HW

6.4.1 Forward execution - Additional HW modules provision

This subsection discusses techniques that increase the resilience through adding HW modules on the platform. The added modules may have either the same (**same functionality**) or different (**different functionality**) functionality. The structure of this subtree along with the corresponding subsections is illustrated in Fig. 6.3.

6.4.1.1 Same functionality ■

This group includes techniques that add hardware modules of the same functionality as the one(s) that should be protected. Some of the most known and well-established fault tolerant techniques are found in this category. The **provision of additional**

¹The boxes in the classification figures include hyperlinks to the text. By clicking on each of the boxes, the reader will be transferred to the corresponding section in the text.

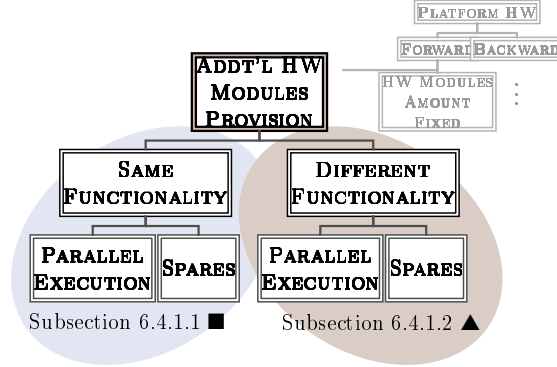


Figure 6.3: Classification for forward techniques that require additional HW modules

HW modules can be further categorized into modules that are used in parallel execution mode and modules used as spares. **Parallel execution** denotes that the modules are all active and processing operations (or hold/transfer data and instructions for processing). The term **spares** denotes that the added modules are not all executing in parallel with the default ones. They will only start executing upon certain conditions.

Parallel execution ①². In general, parallel execution implies that the modules are all actively used for the intended functionality, or at least potentially when the workload is very high^⑤. The term **lockstep** denotes a mode of operation, according to which, HW modules execute the same operations regarding the same program at the same time. Generally, lockstep processing can be “tight” or “loose” depending on whether the outputs of the modules are synchronized at the operation level or only selectively, for example at the I/O level [2]. Lockstep processing is used to make a system more robust either by masking an error, i.e. by allowing the correct output to be produced independent of which module caused the error, or by using explicit knowledge of the faulty module.

In the first case, multiple modules (N modules in general) with the same specification as the primary module are provided and majority voting is applied at their output. No error detection is required as the **error is masked** through the voting. This results in a well-known technique called *N-modular redundancy* or *NMR*. Typically N is an odd number to avoid uncertain output votes. Most often, the scheme has been employed in the form of *triple modular redundancy (TMR)* so that a correct output is produced with a two out of three vote.

Lockstep processing can be combined also with system **awareness of the faulty module**. In this case, a separate detection scheme is employed for the identification of the faulty module. Majority voting is not required, as after the detection, the faulty module is considered *not valid* any more. Only the output of

²The circled numbers refer to the corresponding leaves in the overall classification tree (in this case Fig. 6.11). By clicking on these numbers the reader will be transferred to this figure.

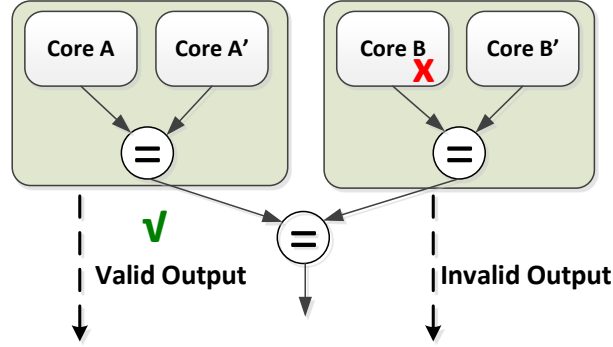


Figure 6.4: Lockstep execution in a *pair-and-spare* structure

the other module(s) is considered valid. So in this case, only two modules operating in lockstep suffice for producing a correct output.³ One technique commonly found in literature, belonging to this category, is the so-called *pair-and-spare*⁴ technique. In pair-and-spare, two pairs of replicas operate in lockstep, as illustrated in Fig. 6.4. Within each pair, error detection is performed through a comparison circuit. In presence of an error, the faulty pair declares itself as faulty. Then, the output of the other pair is selected as the valid one⁵. Replica **determinism** is not an issue here as the processors perform their operations simultaneously and they operate on identical inputs [64]. **Pros** in this class include the high error protection, the lack of latency and performance overhead and the general applicability. **Cons** include the very high area (e.g. 200% for TMR) and power overhead. Literature **examples** on the aforementioned concepts include: [17], [36], [52] on *TMR*, Stratus computers and the VAXft 3000 minicomputer [81] on *pair and spare*⁵.

Spares ②. In this category, the added modules, which deliver the same functionality as the original ones, act as spares. The role of spare modules can be potentially dual: The first use of spares is to remain in standby mode and **take over execution when the primary module fails**⁵. The second use of spares is to take over execution (or be included in the system operation) for part of the time, without the primary module experiencing some failure. That means that the **execution can potentially alternate between the spare and the primary** module. Several reasons can motivate the undertaking of such a scheme. One possibility is related to the benefits coming from sharing the workload (in time). For example, it is known [29] that the device stress, which contributes to the system aging, is increased when there is a full workload operation compared to when there is alteration of active and inactive periods. Through alternating the execution between a primary and

³However, detection of the faulty module can also be employed in *NMR* schemes [80]. Even though the output would be correct also without it, this knowledge can be used in order to have faulty module replaced.

⁴The part “spare” of the term is misleading as in fact all the modules involved operate in lockstep.

a spare the lifetime of the system could be expanded. Another possibility is that the modules (original and spare) have partially different internal implementation, which gives them characteristics that fit better for certain conditions. In this case, the execution may alternate depending on the changing application requirements, for example, due to changes in the input workload or in environmental parameters (e.g. noise or temperature) ^⑤. **Pros** include the high error protection and lack of performance overhead. **Cons** include the area overhead. The power overhead can be avoided depending on whether the spares are powered or not and this is a trade-off with latency (see supplementary material). The approach is generally applicable, except if spares are tailored to fit changing application requirements. Literature **examples** on the aforementioned concepts include: [13], [84] on spares with failing modules, [77], [56] on spares with working modules ^⑤.

6.4.1.2 Different functionality ▲

This group includes techniques that add hardware modules of different functionality than the one(s) that should be protected or become more robust. Again, a distinction can be made between modules that are in **parallel execution** mode and modules that act as **spares**.

Parallel execution ③. Here, the added module performs different functions than the original module. Several possibilities exist: A category includes **hybrid** schemes, according to which, the added modules that are **designed to be more robust** (by employing for example circuit-level techniques). The added module can perform only a subset of the operations of the original module for verification purposes, i.e. it is a module with **reduced functionality**. For example, the most crucial operations or the ones that cannot be performed (repeated) by any other of the already existing modules on the platform, maybe be performed by the added module. Since it is designed to be more robust, its output is assumed as the correct one. Another possibility is that the added module performs a superset of the operations of the original module, namely it is a module of **increased functionality**. That means that it performs the operations of the original module plus additional operations, which are normally performed by other modules on the platform. That would be the case when the added module would act like a supervisor for several modules. An additional possibility is that the added module performs **different types of functions**. For example, it may perform some error correction. Given that the HW module granularity can go down to a register, the error correction codes (ECC) are placed in this category. They are typically implemented in memory structures, but also in buses, state machines and arithmetic units. Fig. 6.5 shows an example of a single bit correction with the Hamming code [28]. Syndrome bits are created during the read operation. If a single error occurs, the syndrome identifies the erroneous bit. **Pros** include the flexibility to trade-off area, power, performance overhead and latency with the error protection by selecting a fitting functionality to be added. **Cons** include that this class generally requires system-specific solutions

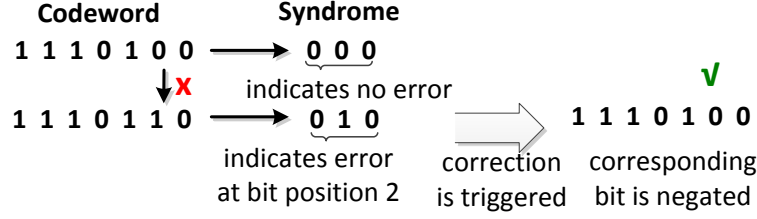


Figure 6.5: Read out (7,4) Hamming codeword and syndrome generation for zero and one error with correction

(although for ECC reusable concepts are typically applied). Literature **examples** on the aforementioned concepts include: the Algorithmic Noise Tolerance (ANT) [30] on modules with reduced functionality, [28], [20] on ECC [®].

Spare ④. As already discussed, spare modules can be present in order to take over execution in case the primary module fails or to take over execution for part of the time, even if no failure is present. A **reduced functionality** spare module is able to continue execution at a reduced power and area overhead but also at a degraded performance (since only part of the functionality is available). An **increased functionality** spare module is able to continue execution in an environment that the primary module has been shown to be not good enough. By using its additional functionality, it keeps or improves the reliability target (at extra area and power cost) [®]. Similarly to the earlier category, **pros** include the flexibility to trade-off area, power, latency and performance with error protection by selecting the appropriate solution. **Cons**, generally in this class, include that system-specific solutions are required. In the literature, techniques that employ spares with reduced functionality have been identified. **Examples** can be found in [87] [®].

6.4.2 Forward execution - HW modules amount fixed

This subsection discusses techniques that use only the same amount of modules on the platform as the original system (before reliability related countermeasures are added). Hardware modifications (like adding interconnects) may be required but no additional HW module is added. A HW or SW controller is often needed to coordinate the actions. These techniques are split into techniques that reuse the existing HW modules (**existing HW modules**) and those that replace one (or more) module with an alternate in order to make the system more robust (**alternate HW modules**). The first category is further split into techniques that either change the way of operation of the HW modules (**HW modules operation mode**) or leave the operation unaltered and change the way the workload is mapped on these HW modules (**resource allocation**). Changing the operation of the HW modules means

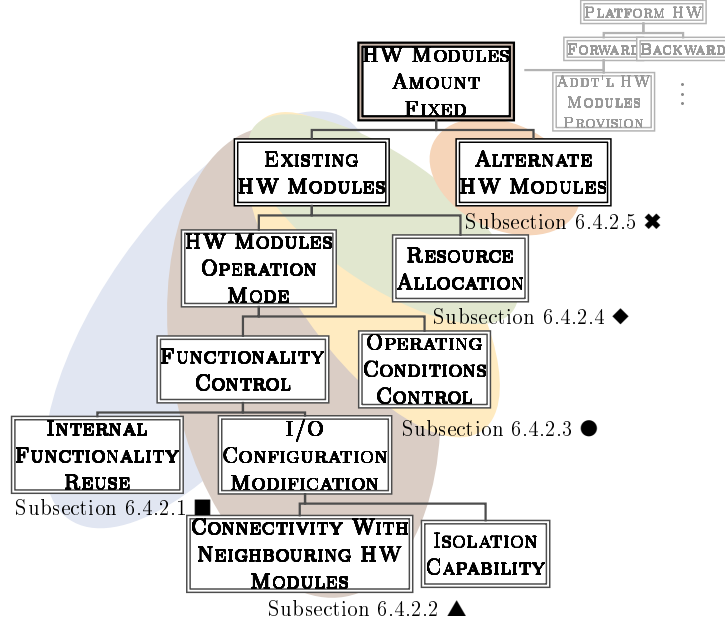


Figure 6.6: Classification for forward techniques that keep the amount of HW modules fixed

that the changes have as focus either the functionality (**functionality control**) or the operating conditions (**operating conditions control**). Functionality-oriented modifications either focus on the internals of a HW module so that the intended module usage is exploited for reliability purposes (**internal functionality reuse**) or on the input-output behavior of the module and how it interacts with the other modules (**I/O configuration modification**). Fig. 6.6 shows the proposed subtree and its division into subsections.

6.4.2.1 Internal functionality reuse ⑤ ■

Techniques belonging to this category are very **system/application dependent**. For example, communication or signal processing systems typically have blocks that perform channel or source coding. Channel decoders mitigate errors introduced by the channel and can be potentially reused in order to mitigate hardware-induced errors. **Pros** include the lowest possible area and power overhead due to the reuse. **Cons** include the lack of general applicability, latency, possible performance costs and the limited error protection. Literature **examples** that reuse the channel decoder include: [41], [12] ⑤.

6.4.2.2 I/O configuration modification ▲

This group of techniques re-organizes the interaction of a module with the other modules. This can potentially mean a different way of connecting or communicating (**connectivity with neighbouring HW modules**) or even an isolation action (**isolation capability**), during which, an erroneous module is bypassed from the system.

Connectivity with neighbouring HW modules ⑥. Inter-module techniques can exploit **inherent redundancy** typically present in regularly structured systems, like arrays of processing elements (PEs), to increase the masking and correction capability of the system. Nowadays, high-performance is achieved primarily by chip multiprocessors (CMPs). CMPs are composed of multiple cores located in a single die or on multiple dies in a single package. The types of cores may vary: from simple, in-order processors up to more complex, superscalar ones. They enable high performance through parallel computation. The CMPs are used here as driver, but the ideas can be applied to other regularly structured systems, where reuse is possible. The availability of the cores can be exploited to create masking capability by, for example, running a process in three cores in parallel in a *TMR* structure. Or the hardware itself can be built as **reconfigurable**, so that, the modules can be connected in a different way depending on run-time conditions. Typically, this last possibility is found in the form of a hybrid; for example, it is often found together with spare modules. **Pros** include the low area and power overhead (due to the reuse of existing modules but with additional cross-links), the general applicability (for systems with inherent redundancy), the potentially high error protection. **Cons** include the latency and blockage of resources for reliability that could be used to improve performance. A literature **example** that employs a modified connection network in CMPs is found in [3] ⑤.

Isolation capability ⑦. To prevent erroneous results from corrupting the system output, faulty components can be **bypassed (through a switch) or powered off**, in case such an isolation capability has been added in the system. The system continues operating but at a degraded performance. These schemes exploit inherent redundancy in regularly structured systems such as arrays of PEs, memories and interconnection networks or even processors.⁵ **Pros** include low area and power overhead, general applicability (for systems with inherent redundancy). **Cons** include latency, degraded performance, limited error protection. Literature **examples** of the concept include: [84], [11] on structures within processors, [27], [73] on pipeline stages in CMPs ⑤.

⁵Quite often, this isolation functionality is combined with techniques presented in *Additional HW modules provision/same functionality* or the previous category so that a different error-free module is used instead. In these cases, these are hybrid combinations. Note that techniques that employ additional modules that run in parallel or act as spares do not necessarily isolate the faulty component.

6.4.2.3 Operating conditions control ⑧ ●

Operating conditions represent the interference caused to a digital system by its environment [72]. This covers a broad range of effects like radiation, temperature, humidity but also electrical stimuli. This category includes all actions that influence the operating conditions of the digital system, beyond changing the system's functionality.

Typically, **operating parameters, such as the supply voltage and the clock frequency**, are controlled to manage the performance, power and reliability trade-offs. Scaling the voltage beyond a critical limit can lead to excessive error rates. On the other hand, using conservative guard bands for the voltage setting can lead to significant power overhead. **Pros** include lack of area overhead, general applicability (assuming that knobs are present in the system for power management). **Cons** include the latency and limited error protection. Power and/or performance will typically be affected depending on the knob being used. Here, works that implement control algorithms that change operational parameters are classified, like the **examples** of [38], [74] ⑤.

6.4.2.4 Resource allocation ⑨ ◆

Here, the way the hardware resources are assigned is modified without changing the way of operation of the HW modules. Simply the task is **migrated or swapped with another task**. **Pros** include the limited area, power, performance overhead due to the modules reuse (with the exception of adding specialized interconnects) and the rather general applicability (for systems with inherent redundancy). **Cons** include latency during migration and limited error protection. Literature **examples** on hardware-based task migration include: [67], [90] ⑤.

6.4.2.5 Alternate HW modules ⑩ ✕

This category includes schemes that replace an existing HW module with another **more robust implementation** for the system context (without employing circuit or lower layer techniques). **Pros** include the limited area and power, performance overhead as the new implementation will typically satisfy the system requirements, while minimizing additional cost. **Cons** include that system-specific solutions are required (if existing at all) and typically only limited error protection will be possible. Literature **examples** in this category include: [33], [32] on alternate channel decoders ⑤.

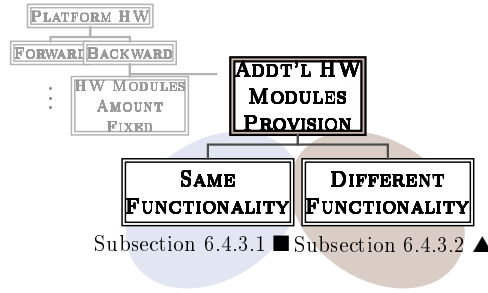


Figure 6.7: Classification for backward techniques that require the provision of additional HW modules

6.4.3 Backward execution - Additional HW modules provision

This subsection discusses techniques that increase the resilience of systems through rollback to an earlier point of execution and repetition of the execution. Just like in the *forward* execution category, the added modules can have either the same (**same functionality**) or different (**different functionality**) functionality. The corresponding categories and subsections are shown in Fig. 6.7.⁶

6.4.3.1 Same functionality (11) ■

This category discusses techniques that provide additional HW modules with the same functionality as the original ones. The recovery is achieved by **repeating part (or the whole) of the execution**, when an error is detected. In this category, the second module plays an active role in the recovery. For example, it can activate the execution repetition or provide necessary information to the first module so that the execution is repeated successfully. When the second module executes the same instruction sequence, **non-deterministic** execution is not a concern, as long as identical inputs can be provided to both modules. **Pros** include the potentially high error protection (at the expense then of performance and latency). Moreover, the technique is generally applicable. **Cons** include the high area and power overhead. A literature **example** in this category is [63] [Ⓢ].

⁶Note that lower level splits like a split between modules that are in **parallel execution** mode and modules that act as **spares** are also possible (like in the *forward* category). Spare modules would, for example, not only take over the execution after the primary module has failed but also repeat the failed execution. However, hardware-based techniques that retry the execution using spare modules have been less explored in the literature. So, this split is left as implied for this tree.

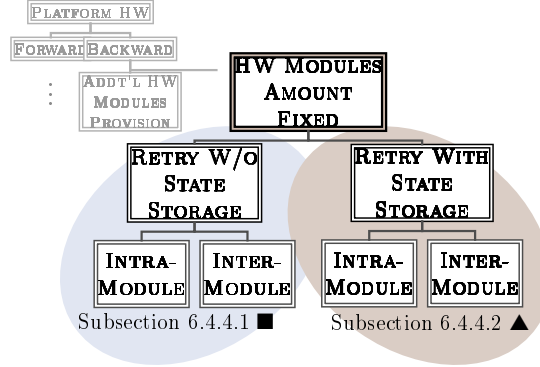


Figure 6.8: Classification for backward techniques that reuse existing HW modules

6.4.3.2 Different functionality (12) ▲

Instead of adding modules with the same functionality, modules with different functionality can be added; the added modules play an active role in the recovery as in the previous category. The added modules can be with **reduced or increased functionality** as in the corresponding *forward* category for similar reasons. **Pros** include the flexibility to trade-off area, power, performance, latency with error protection depending on the selected functionality. **Cons** include that the solutions are rather system-specific. A literature **example** in this category is [6] ^⑤.

6.4.4 Backward execution - HW modules amount fixed

The majority of the techniques proposed in the literature that employ backward execution, reuse the already existing HW modules as the additional area overhead of the previous category is avoided. This can be achieved by techniques that retry the execution without explicit storage (**retry without state storage**) and techniques that retry by storing some (redundant) system information at intermediate execution points to be used for system recovery (**retry with state storage**).⁷ *Checkpointing* is a term that refers to the intermediate storing of the application's state (or of part of it), like register and memory contents. Additional events may be registered as part of the state, which are called *logs*. The corresponding categories and subsections are shown in Fig. 6.8.

⁷Note that the *Backward/additional HW modules provision* subtree could also be split further in similar categories, depending on whether intermediate state storage is involved. However, in that case, because the biggest overhead comes from additional HW modules, this split is left as an implied lower level split.

6.4.4.1 Retry without state storage ■

This category includes the schemes that move back the execution to an earlier point and repeat it, upon error detection. The execution can be successfully repeated without explicitly storing the system state either because the **state information is not really needed** or because it is **provided indirectly** by executing another task, which produces the required information. In the degenerate case, a hardware-driven restart/reboot procedure can be triggered to remedy transient errors. The techniques can be further distinguished into techniques that take place within the boundaries of a single module, i.e. **intra-module** and techniques that operate across modules, i.e. **inter-module**, as shown in Fig. 6.8.

Intra-module (13). In this category belong schemes that either exploit **inherent features** of processors to retry a task execution, like instruction retry or cache refetch, or employ additional hardware-based tasks.

For example, Ray et al. [71] propose to use the pre-existing instruction rewind mechanism present in superscalar machines for branch mispredictions in order to handle error recovery. After detecting an error (by duplicating the instruction during the decode stage and comparing the results before committing), the contents of the ReOrder Buffer (ROB) are flushed and the instruction is re-executed, similarly to what happens upon a branch misprediction event (see Fig. 6.9a). In case the results agree after cross-checking, a single instruction retires and execution proceeds.

Hardware-based tasks in the literature are implemented by **simultaneous multithreading** to increase on-chip parallelism. Simultaneous multithreading (SMT) is a technique that allows multiple threads to issue multiple instructions each cycle on a superscalar processor [89]. The threads can be separate from each other or coupled to each other. Separate threads could be potentially created to execute the same program in a *TMR* structure, assuming that care is taken so that the threads use identical shared resources. The literature focuses on employing hardware-based threads in *coupled execution* mode. According to this mode of operation, the threads communicate with each other, i.e. one thread uses some knowledge from the other thread(s) in order to execute the program. Coupled execution has been used with processors in order to speed-up execution and the idea has been reused for fault tolerance [85]. The concept is as follows: Two streams of the same program run in parallel but with a time lag (see Fig. 6.9b). The first stream is a less accurate one as it processes less instructions than a complete stream would. It bypasses certain computations and branch instructions as indicated by a hardware monitor which has observed past instances. Thus, it can run faster than a complete stream. Its results are stored in a delay buffer. The second stream is an accurate one, as it executes all the instructions. However, it receives information from the first stream through a delay buffer, which allows it to run also faster. For example, it uses memory load values and thus it can avoid memory latencies. When the second thread commits (writes its results to the registers), the results

from both threads are compared. If they are not identical, the results of the second one are used to restore the system state. **Non-deterministic events** like traps and

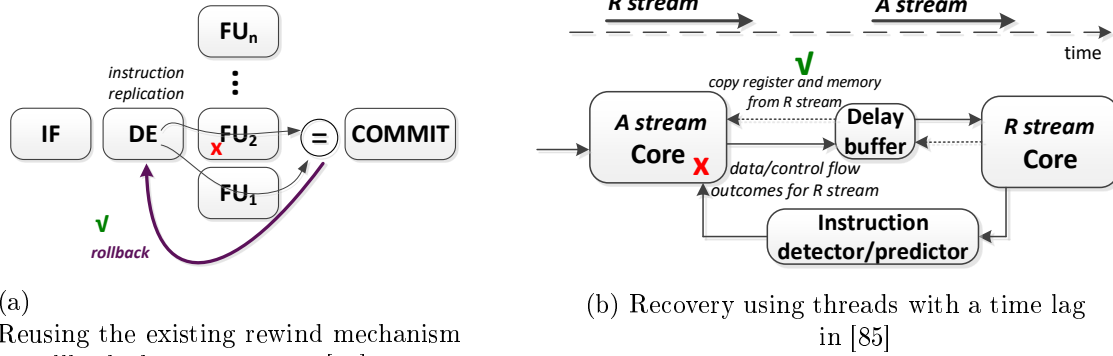


Figure 6.9: Illustration of concepts in the platform-HW *backward* category

exceptions are handled with some minimal support from the operating system. The first stream stalls until the delay buffer completely empties and the second stream is terminated. The first stream is serviced (by the operating system) and execution resumes. **Pros** include the low area and power overhead, potentially high error protection (but only for transient errors) and rather general applicability. **Cons** include the latency, performance overhead and the limitation to transient errors. Literature **examples** include: [71] on re-executing instructions, [75] on tasks with a time lag, [14] on restarting a core ^⑤.

Inter-module ^⑭. This category has similar properties with the earlier but requires the **cooperation** of modules. **Pros** and **cons** are similar with the previous category, but here also permanent errors can be handled and synchronization issues have to be addressed. In the literature, mainly **examples** that include tasks with time lag have been identified: [85], [25] ^⑤.

6.4.4.2 Retry with state storage ▲

This group of techniques employs the storage of a complete or partial error-free state and the rollback to that state upon detection of an error. Afterwards, the execution is repeated to acquire error-free results, assuming that the error was transient. The techniques are also distinguished into **intra-module** and **inter-module**. In the latter category issues that have to do with the state synchronization among several modules have to be addressed. Checkpointing/rollback refers to a widespread concept, according to which, the state of a process is proactively stored at certain intervals during execution so that the correct state is restored in case of an error. The majority of prior work realizes software-based checkpointing/rollback schemes.

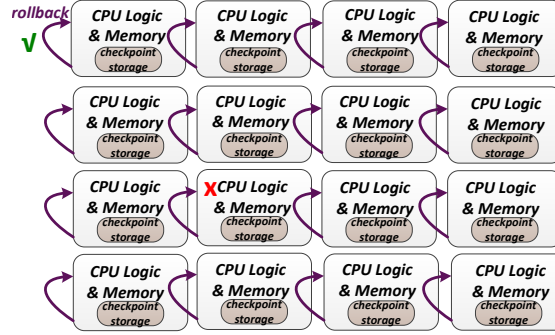


Figure 6.10: A local error can trigger all the CMP cores to roll-back in global checkpointing schemes

However, groups both in industry and academia have provided fully hardware-based implementations. Typically, when these schemes address **non-determinism**, this is done by synchronizing the checkpoints with the external events (e.g. interrupts). Namely, when an external event takes place, a checkpoint is forced.

Intra-module (15). This category includes schemes that **store the whole state or a subpart** of the state of a module in order to restart the execution from that stored point if an error occurs. The storage can take place in the main memory, hard disk, register file or memory buffers, and is often complemented by another error resilience technique, like ECC, in order to be more robust. A broad range of checkpointing techniques exist, from techniques that store checkpoints very rarely (every thousands up to billions of cycles) assuming low error rates up to techniques that perform checkpointing very often (every few cycles) assuming high error rates. **Pros** include the high error protection (for transient errors only), the general applicability. **Cons** include latency (depending on the checkpointing granularity), performance (depending also on whether checkpointing is overlapped with normal execution) and the limitation to transient errors. Area and power overhead is medium. Literature **examples** include: [4] on cache-based checkpoints, [92], [26], [46] on register-based checkpoints ^⑤.

Inter-module (16). Such schemes are typically found in multicore architectures. Here, on top of the external **non-deterministic events**, like interrupts, also internal events have to be taken care of, like the accesses to the shared memory. These checkpointing schemes can be characterized as global and local. In the **global** schemes, common checkpoints are created among all modules and upon detection all modules have to roll back to an earlier state (even when many of them are error-free). Fig. 6.10 illustrates the concept. A challenge with this approach is the scalability as the number of cores increases. On the other hand, **local** checkpointing schemes allow such actions to be made by a subset of the modules, performing only local synchronization and information storage. A taxonomy of hardware-based checkpointing schemes for CMPs can be found in [68]. **Pros** and

cons are similar with the previous category, but with extra synchronization costs. Global schemes induce more overhead during checkpointing, but have a simpler recovery, compared to local schemes. Literature **examples** include: [96], [1] on local and [83] on global schemes ^⑤.

6.4.5 Overall platform hardware classification

The sub-trees presented in the previous subsections are combined to form the overall classification tree for platform HW techniques, as shown in Fig. 6.11. Starting from the top-level split of Fig. 6.2, the intermediate nodes (colored by pale green) are followed when necessary, to reach the final classes (colored by darker green and numbered).

6.5 Summary

This chapter presented a novel top-down classification scheme for hardware-based mitigation schemes. The scheme is based on branches with complementary characteristics and, thus, different associated costs. Representative works have been mapped on each of the branches. Therefore, the framework not only allows the designer to get a more comprehensive view of the domain, but also, to identify fitting solutions depending on the application needs.

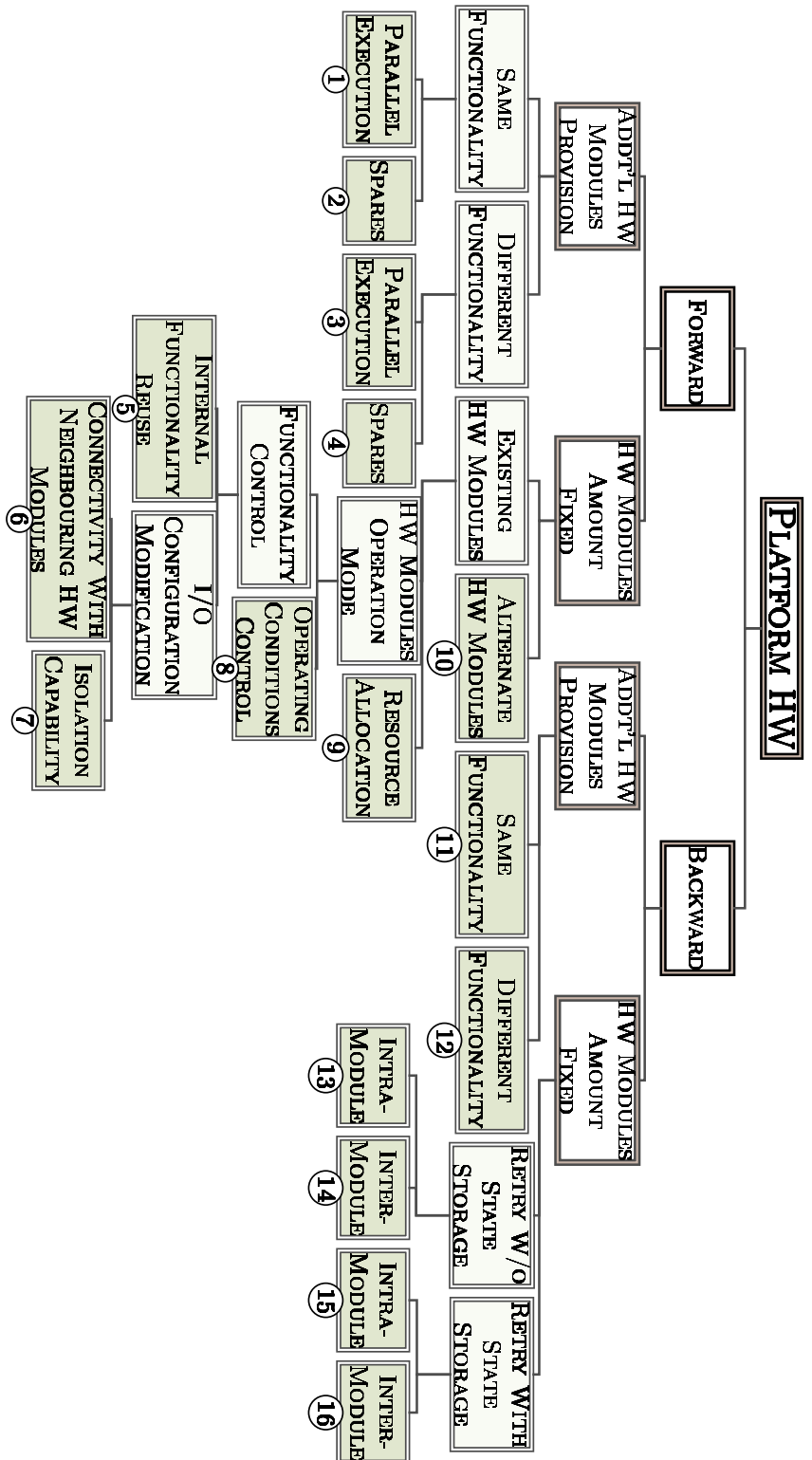


Figure 6.11: Overall proposed classification for techniques at the platform HW

Chapter 7

Conclusions

Reliability related to hardware-induced errors has become one of the important design issues in the design of modern digital systems, and is inherently in conflict with other goals, especially energy-efficiency. To derive reliability approaches that are cost-effective, accurate fault/error information should be provided for the system across the design layers, i.e. insight of the system behavior in the presence of faults and errors is needed. This information enables mitigation approaches, which aim either to prevent or to correct errors. The more fine-grained the information, the more targeted and, thus, cost-effective the mitigation approach can potentially be.

This information typically comes from system simulation, which is time-consuming due to the huge complexity of modern systems. The exploration space is very large given the numerous possible instances of error locations (in time and in space) while, in order to have satisfactory accuracy, a large amount of data needs to be processed. Given the huge injection space as well as the complexity of modern systems, evaluating the impact of bit errors on the system behavior using simulation can be very time consuming. Lack of scalability is a major concern for such an analysis.

An attractive alternative, especially for communication systems where statistical metrics are of interest anyhow, is to use analytical techniques, and more specifically the statistical moments, in order to estimate the statistical properties of the signal after bit errors have been introduced and propagated through the rest of the system. In this work, we present possibilities and limitations of using analytical techniques in order to speed-up the execution time of error injection and propagation experiments in LTI operators of communication systems. It has been shown that under certain conditions, such techniques can be beneficial. However, due to the inherent correlation (in the general case) between the error and the error-free signal, the signal characteristics become very soon computationally intractable. Inherent signal autocorrelation can be exploited in techniques, like PCA, to reduce the amount of data to be simulated, without compromising accuracy. Here, again, the benefits are present under certain conditions, leading to the need for more generally applicable solutions. To this end, we propose a framework that

offers a more computationally tractable way to perform repetitive fault injection experiments in signals that propagate through non-recursive LTI blocks. The proposed framework incorporates also the effects of non-linear operations, like saturation and quantization, on the error propagation. We provide results regarding computational complexity reduction and execution time reduction. We illustrate that for a given FIR filter block, the computational complexity is improved by a factor equal to the number of filter coefficients, while execution time improves 7 to 14 times for the given simulator.

As a complementary contribution in the reliability domain, hardware-based techniques that increase resilience and mitigate functional reliability errors have been classified in a novel way. This has been achieved through a framework with complementary splits, in which primitive mitigation concepts are defined. That allows every type of technique to be classified, by combining the appropriate components. The framework has been accompanied by a wide variety of sources from the published literature. In this way, insight can be provided to the designers and researchers about the nature of existing schemes, since every node has some unique properties. But also the development of efficient solutions in the future is facilitated, since the desired properties of a new technique, required to satisfy a certain need, can be more easily identified when they are presented in a structured way.

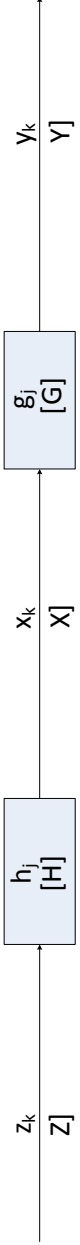
List of Publications

Some of the results presented in this thesis were published at various peer reviewed international conferences and journals. A list of the publications is given in the following (in reverse chronological order):

- G. Psychou, D. Rodopoulos, M. M. Sabry, T. Gemmeke, D. Atienza, T. G. Noll, and F. Catthoor, "Classification of resilience techniques against functional errors at higher abstraction layers of digital systems", ACM Comput. Surv., 50, 4, Article 50 (October 2017), 38 pages.
- G. Psychou, T. Gemmeke and T. G. Noll, "A framework for analyzing the propagation of hardware-induced errors in non-recursive LTI blocks with finite wordlength effects", 26th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), Bremen, Germany, 2016, pp. 147-154.
- G. Psychou, T. Gemmeke and T. G. Noll, "On the use of analytical techniques for reliability analysis in presence of hardware-induced errors", IEEE 13th International Conference on Industrial Informatics (INDIN), Cambridge, UK, 2015, pp. 1416-1423.
- D. Rodopoulos, G. Psychou, M. M. Sabry, F. Catthoor, A. Papanikolaou, D. Soudris, T. G. Noll, and D. Atienza, "Classification Framework for Analysis and Modeling of Physically Induced Reliability Violations", ACM Comput. Surv. 47, 3, Article 38 (February 2015), 33 pages.
- G. Psychou, J. Schleifer, J. Huisken, F. Catthoor, and T. G. Noll, "Cross-layer reliability exploration proposal for body area networks", 9th Workshop on Silicon Errors in Logic-System Effects-SELSE, Urbana-Champaign, IL USA, 2012.

APPENDIX

Table 7.1 LTI Processing of Stationary Random Process in the Principle Component Analysis (PCA) Domain



Uncorrelated Random Input Z with mean μ_Z and variance σ_Z^2

Autocorrelated Random Signal $X = [H] \cdot Z$

Autocorrelated Random Signal $Y = [G] \cdot X$

Centered Uncorrelated Random Input $\tilde{Z} = Z - \mu_Z$

Centered Autocorrelated Random Signal $\tilde{X} = X - \mu_X = [H] \cdot \tilde{Z}$

Centered Autocorrelated Random Signal $\tilde{Y} = Y - \mu_Y = [G] \cdot \tilde{X}$

Random Input Data Matrix (n columns of samples) $[R_Z]$

Random Data Matrix (n columns of samples) $[R_X] = [H] \cdot [R_Z]$

Random Data Matrix (n columns of samples) $[R_Y] = [G] \cdot [R_X]$

Covariance Matrix of Z

Covariance Matrix of X

Covariance Matrix of Y

$$[\Sigma_Z] = \frac{1}{(n-1)} \cdot [R_Z] \cdot [R_Z]^T, \text{ here } [\Sigma_Z] = \sigma_Z^2 \cdot [I]$$

$$[\Sigma_X] = \frac{1}{(n-1)} \cdot [R_X] \cdot [R_X]^T$$

$$[\Sigma_Y] = \frac{1}{(n-1)} \cdot [R_Y] \cdot [R_Y]^T$$

EVD of Covariance Matrix of Z

EVD of Covariance Matrix of X

EVD of Covariance Matrix of Y

$$[\Sigma_Z] = [V_Z] \cdot [\Lambda_Z] \cdot [V_Z]^T, \text{ i.e. } [\Sigma_Z] \cdot [V_Z] = [\Lambda_Z] \cdot [V_Z]$$

$$[\Sigma_X] = [V_X] \cdot [\Lambda_X] \cdot [V_X]^T, \text{ i.e. } [\Sigma_X] \cdot [V_X] = [\Lambda_X] \cdot [V_X]$$

$$[\Sigma_Y] = [V_Y] \cdot [\Lambda_Y] \cdot [V_Y]^T, \text{ i.e. } [\Sigma_Y] \cdot [V_Y] = [\Lambda_Y] \cdot [V_Y]$$

As the Covariance Matrix is symmetric and positive definite

As the Covariance Matrix is symmetric and positive definite

As the Covariance Matrix is symmetric and positive definite

$[\Lambda_Z]$ diagonal matrix with non-negative real entries (here $[\Lambda_Z] = [I]$)

$[\Lambda_Y]$ diagonal matrix with non-negative real entries

$[V_Z]$ orthogonal matrix with normalized eigenvectors (here $[V_Z] = [I]$)

$[V_Y]$ orthogonal matrix with normalized eigenvectors

Principle Components of Z

Principle Components of X

Principle Components of Y

$$\Pi_Z = [V_Z]^T \cdot \tilde{Z} = [\Lambda_Z]^{\frac{1}{2}} \cdot \Psi, \text{ i.e.}$$

$$\Pi_X = [V_X]^T \cdot \tilde{X} = [\Lambda_X]^{\frac{1}{2}} \cdot \Psi, \text{ i.e.}$$

$$\Pi_Y = [V_Y]^T \cdot \tilde{Y} = [\Lambda_Y]^{\frac{1}{2}} \cdot \Psi, \text{ i.e.}$$

$$Z = \mu_Z + [V_Z] \cdot [\Lambda_Z]^{\frac{1}{2}} \cdot \Psi$$

$$X = \mu_X + [V_X] \cdot [\Lambda_X]^{\frac{1}{2}} \cdot \Psi$$

$$Y = \mu_Y + [V_Y] \cdot [\Lambda_Y]^{\frac{1}{2}} \cdot \Psi$$

Approximation: Cropping of eigenvector and eigenvalue matrices for dominant eigenvalues

Approximation: Cropping of eigenvector and eigenvalue matrices for dominant eigenvalues

$$[\Lambda_X] \implies [\Lambda'_X]$$

$$[\Lambda_Y] \implies [\Lambda'_Y]$$

$$[V_X] \implies [V'_X]$$

$$[V_Y] \implies [V'_Y]$$

yields

yields

$$X \approx X' = \mu_X + [V'_X] \cdot [\Lambda'_X]^{\frac{1}{2}} \cdot \Psi$$

$$Y \approx Y' = \mu_Y + [V'_Y] \cdot [\Lambda'_Y]^{\frac{1}{2}} \cdot \Psi$$

$$= [H] \cdot (\mu_Z + [V'_Z] \cdot [\Lambda'_Z]^{\frac{1}{2}} \cdot \Psi) = [H] \cdot Z \quad q.e.d$$

$$= [G] \cdot (\mu_X + [V'_X] \cdot [\Lambda'_X]^{\frac{1}{2}} \cdot \Psi) = [G] \cdot X \quad q.e.d$$

Explanation of metrics used in the tables:

Power overhead is dependent very much on the exact implementation, including circuit-level aspects. So here, we perform a very relative assessment. Energy overhead will be commented when appropriate.

Performance overhead is considered in terms of number of additional execution cycles. More specific aspects like throughput, maximum frequency etc. are not considered here.

Mitigation latency refers to the induced delay until the scheme fulfils the intended mitigation function and should not be confused with the inherent system latency. For SW-mitigation techniques offline compilation time overhead is not considered.

HW Mitigation Techniques								
Class	Area	Overhead		Performance	Mitigation Latency	Error Protection	Applicability	Amount of Literature
		Power						
Forward								
	Different functionality Spares	Parallel execution	Same functionality Spares	Parallel execution				
	4 (3.1.2)	3 (3.1.2)	2 (3.1.1)	1 (3.1.1)				
	high to low (depends on whether increased/reduced functionality & similar aspects as in 2)	high to low (depends on whether increased/reduced functionality)	high to low (depends on whether increased/reduced functionality)	high to low (hot vs. cold spares)	medium to none (medium when not enough spares available for all defective modules)	high to low (cold vs. hot spares)	high to medium (medium when one spare for multiple modules)	general (system-specific solutions if spares tailored to fit changing requirements)
	very high to medium (e.g. 200+% for TMR, medium when one added module for multiple existing)	very high to medium (e.g. 200+% for TMR, medium when one added module for multiple existing)	medium to none (medium when not enough empty slots available for parallel execution)	medium to none (medium when not enough empty slots available for parallel execution)	high to medium (medium when not enough empty slots available for parallel execution)	general (lockstep requires system-specific solutions)	large (very large for TMR)	
	high to low (depends on whether increased/reduced functionality & similar aspects as in 2)	high to low (depends on whether increased/reduced functionality)	possibly (depends on function of the added module; accuracy may be affected)	high to low (depends on function of the added module, e.g. high for ECC)	high to low (depends on function of the added module)	system-specific (due to different functionality, exceptions like ECC)	medium (very large for ECC)	
	possibly (depends on function of the added module; accuracy may be affected)	high to low (depends on whether increased/reduced functionality & similar aspects as in 2)	high to low (depends on whether increased/reduced functionality)	high to low (depends on function of the added module; accuracy may be affected)	high to low (similar aspects as in 2)	high to low (depends on function of the added module)	system-specific (due to different functionality)	small

HW Mitigation Techniques						
Class	Overhead		Performance	Mitigation Latency	Error Protection	Applicability
	Area	Power				
Fixed Existing HW modules Forward	Internal reuse 5 (3.2.1)	min (due to pure reuse; potentially higher energy overhead)	possibly (depends on the reuse; accuracy may be affected)	medium to low (depends on the reuse, e.g. low when increased decoder iterations)	medium to low (depends on the reuse)	very system-specific (due to pure reuse)
				medium to low (e.g. due to reconnecting)	high to medium (typically limitations in re-connection options, but TMR structure is also possible)	rather general (for systems with inherent redundancy & regularity)
	I/O configuration 6 (3.2.2)	low (added cross-links; potentially higher energy overhead)	possibly (e.g. due to resource blockage ¹)	medium to low (medium e.g. if data have to be moved before isolation takes place)	medium to low (typically limitations in isolation possibilities)	rather general (for systems with inherent redundancy & regularity)
				medium to low (depends on knobs used, medium e.g. due to increased supply voltage; potentially higher energy overhead)	medium to low (only transient; depends on implementation)	rather general (similar to DVFS)
	Operating conditions contr 8 (3.2.3)	possibly (depends on knobs used, medium e.g. due to increased supply voltage; potentially higher energy overhead)	possibly (depends on knobs used, medium e.g. due to reduced clock frequency)	high to medium (depends on re- allocation options, high e.g. when scheduling conflicts)	medium to low (typically limitations in re-allocation possibilities)	rather general (for systems with inhe- rent redundancy but also system-specific as- pects e.g. re-allocate on ISA-compatible cores)
Resource Allocation 9 (3.2.4)	possibly (e.g. added cross-links for migration)	low (e.g. during migration; potentially higher energy overhead)	possibly (depends on re- allocation options)	possibly (depends on alternate implementation)	medium to low (typically limitations due to replacement of existing module)	system-specific (limited availability of alternate robust implementations)
Alternate 10 (3.2.5)	medium to low (depends on alternate implementation)	medium to low (depends on alternate implementation)	possibly (depends on alternate implementation)			very small

¹ Resource blockage refers to the fact that the HW resources are consumed e.g. for replicating functions. In the system without reliability capabilities, the same resources could be used for different functions and, thus, performance would be higher.

Table 7.3 Trade-offs in HW-based resilience techniques-Part 2

HW Mitigation Techniques								Class						
Backward				Additional										
Fixed		Retry w/o state storage		Diff func		Same func		Area	Overhead Power	Performance	Mitigation Latency	Error Protection	Applicability	Amount of Literature
Inter module	Intra module	Inter module	Intra module											
16 (3.4.2)	15 (3.4.2)	14 (3.4.1)	13 (3.4.1)	12 (3.3.2)	11 (3.3.1)									
(to enable state storage, rollback and synchronization)	(to enable state storage and rollback)	(only to enable re-execution and synchronization)	(only to enable re-execution)	(depends on function of the added module)	(medium e.g. when one added module for multiple existing)									
medium	medium	medium to low	low	high to low	high to medium									
		(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on function of the added module)	(medium e.g. when one added module for multiple existing)									
medium	medium	medium to low	medium to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead due to re-run)	(medium e.g. for tasks with time lag; energy overhead potentially due to re-run)	(depends on the implementation)	(depends on the implementation)									
medium	medium	medium to low	high to low	high to low	high to low									
(due to regular checkpointing; energy overhead due to re-run)	(due to regular checkpointing; energy overhead due to re-run)	(medium e.g												

Table 7.4 Trade-offs in HW-based resilience techniques-Part 3

List of Terms and Symbols

$[\Sigma_X]$	Covariance matrix of X], page 55
$[P_X]$	Correlation matrix of X], page 57
μ	mean, page 29
π^ν	percentage of corrupted samples, page 29
\underline{X}	Row vector X , page 55
\widetilde{X}	Centered X , page 56
$\hat{\mu}$	Estimated mean, page 55
<i>EVD</i>	Eigenvalue Decomposition, page 55
<i>MRV</i>	Multivariate Random Variable, page 55
<i>PCA</i>	Principal Component Analysis, page 53
$X]$	Column vector X , page 55
ACF	Autocorrelation Function, page 12
Bit-flip sample $[\epsilon_k^{n-1}, \dots, \epsilon_k^1, \epsilon_k^0]$	A (signal) sample ϵ_k used to introduce bit-flips to the error-free sample x_k , so that $\tilde{x}_k = -(x_k^{n-1} \oplus \epsilon_k^{n-1}) \cdot 2^{n-1} + \sum_{\nu=0}^{n-2} (x_k^\nu \oplus \epsilon_k^\nu) \cdot 2^\nu$, page 15

Corrupted (signal) sample \tilde{x}_k	A (signal) sample after hardware-induced bit flips have been injected , page 12
Error (signal) sample e_k	The error of a corrupted (signal) sample \tilde{x}_k , with $e_k = \tilde{x}_k - x_k$, page 12
PDF	Probability Density Function, page 12
PMF	Probability Mass Function, page 12
RV	Random Variable, page 12

Bibliography

- [1] Rishi Agarwal, Pranav Garg, and Josep Torrellas. *Rebound: scalable checkpointing for coherent shared memory*, volume 39. ACM, 2011.
- [2] Nidhi Aggarwal. *Achieving high availability with commodity hardware and software*. ProQuest, 2008.
- [3] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P Jouppi, and James E Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 470–481. ACM, 2007.
- [4] Rana Ejaz Ahmed, Robert C Frazier, and Peter N Marinos. Cache-aided rollback error recovery (carer) algorithm for shared-memory multiprocessor systems. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 82–88. IEEE, 1990.
- [5] Robert Aitken, Görschwin Fey, Zbigniew T Kalbarczyk, Frank Reichenbach, and Matteo Sonza Reorda. Reliability analysis reloaded: how will we survive? In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 358–367. EDA Consortium, 2013.
- [6] T.M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 196 –207, 1999.
- [7] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.
- [8] Amitabha Bhattacharya. *Introduction to Digital Communication. In: Digital Communication*. Tata McGraw-Hill, 2006.
- [9] Andrea Bondavalli, Silvano Chiaradonna, Felicita Di Giandomenico, and Fabrizio Grandoni. Threshold-based mechanisms to discriminate transient from intermittent faults. *Computers, IEEE Transactions on*, 49(3):230–245, 2000.

- [10] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov 2005.
- [11] Fred A Bower, Daniel J Sorin, and Sule Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 197–208. IEEE Computer Society, 2005.
- [12] Christian Brehm, Matthias May, Christina Gimmmler, and Norbert Wehn. A case study on error resilient architectures for wireless communication. In *Proceedings of the 25th international conference on Architecture of Computing Systems*, ARCS’12, pages 13–24, 2012.
- [13] Mengly Chean and Jose AB Fortes. A taxonomy of reconfiguration techniques for fault-tolerant processor arrays. *Computer*, 23(1):55–69, 1990.
- [14] Hyungmin Cho, Larkhoon Leem, and Subhasish Mitra. Ersas: Error resilient system architecture for probabilistic applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(4):546–558, 2012.
- [15] John M. Cioffi. Lecture notes on digital communication - signal processing, "Ch02: Passband systems and analysis", 2007.
- [16] Jean-Marc Daveau, Alexandre Blampey, Gilles Gasiot, Joseph Bulone, and Philippe Roche. An industrial fault injection platform for soft-error dependability analysis and hardening of complex system-on-a-chip. In *2009 IEEE International Reliability Physics Symposium*, pages 212–220. IEEE, 2009.
- [17] M. M. Dickinson, J. B. Jackson, and G. C. Randa. Saturn v launch vehicle digital computer and data adapter. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part I*, AFIPS ’64 (Fall, part I), pages 501–516, New York, NY, USA, 1964. ACM.
- [18] Manfred Dietrich and Joachim Haase. *Process Variations and Probabilistic Integrated Circuit Design*. Springer Science & Business Media, 2011.
- [19] Manfred Dietrich and Joachim Haase. *Process Variations and Probabilistic Integrated Circuit Design*. Springer Publishing Company, Incorporated, 2011.
- [20] Nikil Dutt, Puneet Gupta, Alex Nicolau, Abbas BanaiyanMofrad, Mark Gottscho, and Majid Shoushtari. Multi-layer memory resiliency. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, pages 1–6. IEEE, 2014.
- [21] Paul Embrechts, Filip Lindskog, and Alexander McNeil. Modelling dependence with copulas. 2001.

- [22] Adrian Evans, Shi-Jie Wen, and Michael Nicolaidis. Case study of seu effects in a network processor. In *Proc. of IEEE Workshop on Silicon Errors in Logic-System Effects*, 2012.
- [23] Peter Folkesson, Sven Svensson, and Johan Karlsson. A comparison of simulation based and scan chain implemented fault injection. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 284–293. IEEE, 1998.
- [24] Robert Gallager. Course material for 6.450 principles of digital communications i, mit opencourseware. downloaded on [01 02 2017], 2006, Fall.
- [25] Mohamed Gomaa, Chad Scarbrough, TN Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 98–109. IEEE, 2003.
- [26] Meeta S Gupta, Jude A Rivers, Pradip Bose, Gu-Yeon Wei, and David Brooks. Tribeca: design for pvt variations with local recovery and fine-grained adaptation. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 435–446. ACM, 2009.
- [27] S. Gupta, Shuguang Feng, A. Ansari, J. Blome, and S. Mahlke. The stagenet fabric for constructing resilient multicore systems. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 141–151, 2008.
- [28] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.
- [29] Haibo He, Sheng Chen, Kang Li, and Xin Xu. Incremental learning from stream data. *Neural Networks, IEEE Transactions on*, 22(12):1901–1914, 2011.
- [30] Rajamohana Hegde and Naresh R Shanbhag. Soft digital signal processing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 9(6):813–823, 2001.
- [31] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [32] Amr Hussien, Muhammad S Khairy, Amin Khajeh, Ahmed M Eltawil, and Fadi J Kurdahi. A class of low power error compensation iterative decoders. In *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, pages 1–6. IEEE, 2011.
- [33] Amr Hussien, Muhammed S Khairy, Amin Khajeh, Kiarash Amiri, Ahmed M Eltawil, and Fadi J Kurdahi. A combined channel and hardware noise resilient

- viterbi decoder. In *Signals, Systems and Computers (ASILOMAR)*, pages 395–399. IEEE, 2010.
- [34] IEEE_Std. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.
- [35] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into vhdl models: the mefisto tool. In *Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing*, pages 66–75, June 1994.
- [36] Doug Jewett. Integrity s2: A fault-tolerant unix platform. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 512–519. IEEE, 1991.
- [37] R. A. Johnson and D. W. Wichern, editors. *Applied Multivariate Statistical Analysis*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
- [38] Eric Karl, David Blaauw, Dennis Sylvester, and Trevor Mudge. Reliability modeling and management in dynamic microprocessor-based systems. In *Proceedings of the 43rd Annual Design Automation Conference, DAC '06*, pages 1057–1060, New York, NY, USA, 2006. ACM.
- [39] Muhammad S. Khairy, Amin Khajeh, Ahmed M. Eltawil, and Fadi J. Kurdahi. Equi-noise: A statistical model that combines embedded memory failures and channel noise. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(2):407 – 419, 2014.
- [40] A. Khajeh, K. Amiri, M.S. Khairy, A.M. Eltawil, and F.J. Kurdahi. A unified hardware and channel noise model for communication systems. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–5, Dec 2010.
- [41] Amin Khajeh, Minyoung Kim, Nikil Dutt, Ahmed M Eltawil, and Fadi J Kurdahi. Error-aware algorithm/architecture coexploration for video over wireless applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(1):15, 2012.
- [42] K. J. Kuhn, M. D. Giles, D. Becher, P. Kolar, A. Kornfeld, R. Kotlyar, S. T. Ma, A. Maheshwari, and S. Mudanai. Process technology variation. *IEEE Trans. on Electron Devices*, pages 2197–2208, 2011.
- [43] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.
- [44] Bhagwandas P Lathi. *Modern digital and analog communication systems*. Oxford University Press, Inc., 1990.

- [45] Bhagwandas P. Lathi and Zhi Ding. *Modern Digital and Analog Communication Systems*. Oxford University Press, Inc., New York, NY, USA, 4th edition, 2010.
- [46] Tuo Li, Muhammad Shafique, Semeen Rehman, Swarnalatha Radhakrishnan, Roshan Ragel, Jude Angelo Ambrose, Jörg Henkel, and Sri Parameswaran. Cser: Hw/sw configurable soft-error resiliency for application specific instruction-set processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 707–712. EDA Consortium, 2013.
- [47] Klaus Lochmann and Andreas Goeb. A unifying model for software quality. In *Proceedings of the 8th international workshop on Software quality, WoSQ '11*, pages 3–10, New York, NY, USA, 2011. ACM.
- [48] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.
- [49] Dimitris G. Manolakis and Vinay K. Ingle. *Finite Wordlength Effects*. In: *Applied Digital Signal Processing: Theory and Practice*. Cambridge University Press, 2011.
- [50] W. Mansour and R. Velazco. An automated seu fault-injection method and tool for hdl-based designs. *IEEE Transactions on Nuclear Science*, 60(4):2728–2733, Aug 2013.
- [51] Elie Maricaud and Georges Gielen. *CMOS Reliability Overview*. In: *Analog IC Reliability in Nanometer CMOS*. Springer, New York, NY, USA, 2013.
- [52] Matthias May, Matthias Alles, and Norbert Wehn. A case study in reliability-aware design: A resilient ldpc code decoder. In *Design, Automation and Test in Europe, 2008. DATE'08*, pages 456–461. IEEE, 2008.
- [53] J. W. McPherson. Reliability challenges for 45nm and beyond. In *Proceedings of the 43rd Annual Design Automation Conference, DAC '06*, pages 176–181, New York, NY, USA, 2006. ACM.
- [54] Daniel Menard, Romuald Rocher, and Olivier Sentieys. Analytical fixed-point accuracy evaluation in linear time-invariant systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 55(10):3197–3208, 2008.
- [55] R. Muralidhar, T. Shaw, F. Chen, P. Oldiges, D. Edelstein, S. Cohen, R. Achanta, G. Bonilla, and M. Bazant. Tddb at low voltages: An electrochemical perspective. In *2014 IEEE International Reliability Physics Symposium*, pages BD.3.1–BD.3.7, June 2014.
- [56] Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L. Jones. Scalable stochastic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '10*, pages 335–338, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

- [57] Alan V. Oppenheim and Ronald W. Schaffer. *Discrete-time Signal Processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [58] Krishna Palem and Avinash Lingamneni. What to do about the end of moore’s law, probably! In *Proceedings of the 49th Annual Design Automation Conference*, pages 924–929. ACM, 2012.
- [59] S. Pandey and B. Vermeulen. Transient errors resiliency analysis technique for automotive safety critical applications. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–4, March 2014.
- [60] Athanasios Papoulis and S Unnikrishna Pillai. *Probability, random variables, and stochastic processes*. Tata McGraw-Hill Education, 2002.
- [61] Karthick Parashar. *System-level approaches for fixed-point refinement of signal processing algorithms*. PhD thesis, Université Rennes 1, 2012.
- [62] Andrei Pavlov and Manoj Sachdev. *CMOS SRAM Circuit Design and Parametric Test in Nano-Scaled Technologies: Process-Aware SRAM Design and Test*. Springer Publishing Company, Incorporated, 1st edition, 2008.
- [63] Matthias Pflanz and Heinrich Theodor Vierhaus. Online check and recovery techniques for dependable embedded processors. *IEEE Micro*, (5):24–40, 2001.
- [64] Stefan Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.
- [65] Stefan Poledna. Lecture notes on dependable computer systems, "System aspects of dependable systems", 2007.
- [66] Ilia Polian, John P Hayes, Sudhakar M Reddy, and Bernd Becker. Modeling and mitigating transient errors in logic circuits. *IEEE Transactions on Dependable and Secure Computing*, 8(4):537–547, 2011.
- [67] Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA ’09, pages 93 –104, 2009.
- [68] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 111–122. IEEE, 2002.
- [69] Georgia Psychou, Dimitrios Rodopoulos, Mohamed M. Sabry, Tobias Gemmeke, David Atienza, Tobias G. Noll, and Francky Catthoor. Classification of resilience techniques against functional errors at higher abstraction layers of digital systems. *ACM Comput. Surv.*, 50(4):50:1–50:38, October 2017.

- [70] Martin Radetzki, Chaochao Feng, Xueqian Zhao, and Axel Jantsch. Methods for fault tolerance in networks-on-chip. *ACM Comput. Surv.*, 46(1):8:1–8:38, July 2013.
- [71] Joydeep Ray, James C Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 214–224. IEEE Computer Society, 2001.
- [72] Dimitrios Rodopoulos, Georgia Psychou, Mohamed M. Sabry, Francky Catthoor, Antonis Papanikolaou, Dimitrios Soudris, Tobias G. Noll, and David Atienza. Classification framework for analysis and modeling of physically induced reliability violations. *ACM Comput. Surv.*, 47(3):38:1–38:33, February 2015.
- [73] Bogdan F Romanescu and Daniel J Sorin. Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 43–51. ACM, 2008.
- [74] Tajana Simunic Rosing, Kresimir Mihic, and Giovanni De Micheli. Power and reliability management of socs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(4):391–403, 2007.
- [75] E. Rotenberg. Ar-smt: a microarchitectural approach to fault tolerance in microprocessors. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 84 –91, june 1999.
- [76] Jochen Schleifer, Thomas Coenen, and Tobias G Noll. Statistical modeling of reliability in logic devices. *Microelectronics Reliability*, 51(9):1469–1473, 2011.
- [77] Jeonghee Shin, Victor Zyuban, Pradip Bose, and Timothy M Pinkston. A proactive wearout recovery approach for exploiting microarchitectural redundancy to extend cache sram lifetime. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 353–362. IEEE Computer Society, 2008.
- [78] Jonathon Shlens. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*, 2014.
- [79] M. Short and J. Proenza. Towards efficient probabilistic scheduling guarantees for real-time systems subject to random errors and random bursts of errors. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 259–268, July 2013.
- [80] D. Siewiorek and R. Swarz. *The Theory and Practice of Reliable System Design*. Digital Press, 1982.

- [81] D.P. Siewiorek. Fault tolerance in commercial computers. *Computer*, 23(7):26–37, July 1990.
- [82] Joseph Slember and Priya Narasimhan. Living with nondeterminism in replicated middleware applications. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, pages 81–100. Springer-Verlag New York, Inc., 2006.
- [83] D.J. Sorin, M.M.K. Martin, M.D. Hill, and D.A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 123–134, 2002.
- [84] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 520–531, Washington, DC, USA, 2005. IEEE Computer Society.
- [85] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 257–268, 2000.
- [86] Yuan Taur and Tak H. Ning. *Fundamentals of Modern VLSI Devices*. Cambridge University Press, New York, NY, USA, 2nd edition, 2009.
- [87] James E Tomayko. Lessons learned in creating spacecraft computer systems: Implications for using ada (r) for the space station. 1986.
- [88] Steven A Tretter. *Communication System Design Using DSP Algorithms: With Laboratory Experiments for the TMS320C6713TM DSK*. Springer Science & Business Media, 2008.
- [89] Dean M Tullsen, Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, and Rebecca L Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *ACM SIGARCH Computer Architecture News*, volume 24, pages 191–202. ACM, 1996.
- [90] Shyamsundar Venkataraman, Rui Santos, Akash Kumar, and Jasper Kuijsten. Hardware task migration module for improved fault tolerance and predictability. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 197–202. IEEE, 2015.
- [91] John Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34:43–98, 1956.

- [92] Nicholas J Wang and Sanjay J Patel. Restore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):188–201, 2006.
- [93] Dr. Darren Ward. Lecture notes on EE2/ISE2 communications II, "Part I, Communications principles", 2004.
- [94] Bernard Widrow, Istvan Kollar, and Ming-Chang Liu. Statistical theory of quantization. *IEEE Transactions on Instrumentation and Measurement*, 45(2):353–361, 1996.
- [95] Bin Wu, Jianwen Zhu, and Farid N Najm. An analytical approach for dynamic range estimation. In *Proceedings of the 41st annual Design Automation Conference*, pages 472–477. ACM, 2004.
- [96] Kun-Lung Wu, W Kent Fuchs, and Janak H Patel. Error recovery in shared memory multiprocessors using private caches. *Parallel and Distributed Systems, IEEE Transactions on*, 1(2):231–240, 1990.
- [97] H Ziade, R Ayoubi, and R Velazco. A survey on fault injection techniques. *International Arab Journal of Information Technology*, 1:171–186, 2004.