CLRFrame: An Analysis Framework for Designing Cross-Layer Reliability in Embedded Systems

Siva Satyendra Sahoo, Bharadwaj Veeravalli
Department of ECE
National University of Singapore
Singapore
satyendra@u.nus.edu, elebv@nus.edu.sg

Akash Kumar
Center for Advancing Electronics Dresden
Technische Universität Dresden
Dresden, Germany
akash.kumar@tu-dresden.de

***Abstract*—Continued transistor scaling and increasing power density have led to considerable increase in fault rates in silicon nanotechnology-based real-time systems. Instead of fixing everything at the hardware layer, cross-layer fault tolerance techniques present a more cost-efficient methodology for adapting to such increased fault rates. The effectiveness (*Coverage*, *Fault-Masking*, and *Recovery*) and overheads (*Execution time*, *Energy* and *Cost*) of each fault-tolerance technique vary with the layer and the frequency at which it is implemented. Therefore, appropriate modeling of fault-mitigation methods is necessary for efficient cross-layer design space exploration (DSE). To this end, we propose a first-order framework for analyzing the effects of implementing fault-tolerance across multiple layers. We also propose a Markov-chain based methodology for analytical modeling of fault-mitigation methods and their inter-layer interaction. As a case-study, we model some generic fault-mitigation methods and provide detailed modeling of typical application execution involving fault-mitigation at different layers.**

***Keywords*-Cross-layer Resilience, Real-time systems, Fault-Tolerance**

I. Introduction

Technology scaling and architectural innovations have been the driving force behind the increasing ubiquity of embedded systems. However, these approaches have also led to significant increase in Soft Error Rate (SER) of logic circuits [1]. While the logical masking effect remains unaltered, electrical masking effect has reduced due to scaling down transistor size and supply voltage [2]. Similarly, deeper pipelines used for enabling higher clock speed have resulted in the reduction of latching-window masking, leading to even higher SER in microprocessors. Therefore, extracting increasing usable performance out of embedded systems requires building resilient systems out of increasingly unreliable hardware [3]. Traditional approaches to fault-aware design adopted a phenomenon-based approach that focuses on mitigating all physical faults at the hardware layer. However, such an approach is becoming increasing infeasible due to the increasing complexity, higher SER and tighter cost and energy constraints of embedded systems.

In contrast, cross-layer design approach involves distributing fault-mitigation activities to several layers of the system stack [4]. This entails utilizing the information and capabilities of each layer to provide adequate overall system resilience. Such an approach can reduce fault-mitigation effort at hardware layer leading to more cost-effective designs. Designing an effective cross-layer resilience strategy requires formulating efficient methods for cross-layer design space exploration (DSE)– both at design-time and run-time for finding the right *selection* and *configuration* of fault-mitigation methods that should be implemented at each layer to meet the system-level goals and constraints. Given the increasing design space due to the choices of fault-mitigation methods and configurations at each layer, application complexity, hardware variations etc., non-traditional DSE methods such as *Evolutionary Computing* and *Randomized Algorithms* are being increasingly used in related *Electronic Design Automation* (EDA). The effectiveness of such methods hinges on the efficient estimation of performance metrics and associated overheads for a set of design decisions. For example, in *Genetic Algorithms* (GA) the computation time and accuracy of evaluating the *fitness* of each *individual* of a *generation* can have a significant effect on the overall DSE performance. Therefore, efficient evaluation of design decisions from early stages of DSE can lead to the faster design of cost-effective error-resilient systems. To this end, we propose a methodology for designing cross-layer resilience for embedded systems.

Contributions: Our contributions can be listed as below:

- We propose a first-order framework for comparing different cross-layer designs. The framework includes the effect of implicit fault-masking at different layers along with explicitly implemented fault-mitigation methods to estimate system-level performance metrics.
- We propose a Markov-chain based methodology for analytical modeling of fault-mitigation methods. We demonstrate the usage of this model by modeling some application-agnostic fault-mitigation methods.
- We extend our proposed modeling methodology to account for inter-layer interactions of fault-mitigation implemented at different layers of the system stack.

The rest of the paper is organized as follows: In Section II we provide a brief background of cross-layer reliability and survey some state-of-the-art approaches to designing cross-layer reliability. The proposed framework for cross-layer reliability analysis is discussed in Section III. In Section IV we provide details of Markov-chain based analytical modeling methodology. We conclude the paper in Section V by providing the scope for future work.

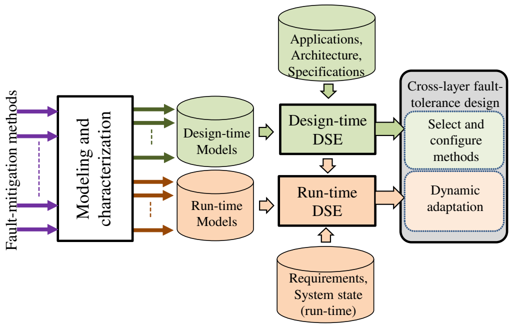
II. Background

Traditional single-layer reliability approaches focus on mitigating all physical faults at the circuit or the hardware layer. A phenomenon-based approach is usually used, i.e., each fault mechanism (NBTI, EM, Single Event Upsets

(SEU) etc.) is mitigated separately to provide an error-free hardware platform. Although it provides a convenient abstraction to the software developer, the rising costs – *area* and *power* – for mitigating the effects of increasing fault rates can make such an approach infeasible for many applications. Therefore, fault-mitigation needs to be implemented at multiple layers of the system stack. In [5], the authors provide a brief survey of such techniques at different abstraction layers. Designing cross-layer reliability involves finding the right *selection* and *configuration* of fault-mitigation methods that should be implemented at each layer to meet the system-level goals and constraints.

A. *Cross-layer Reliability*

In contrast to the single-layer phenomenon-based design approach, the cross-layer approach provides a more application-specific and cost-efficient method for reliability-aware system design. Since the fault-mitigation activities are not limited the hardware layer, an appropriate combination of methods that meets the design goals and constraints can be implemented. As discussed in [6], implementing separate fault tolerance stages at different layers can result in reduced power and area overheads. For example, TMR, which provides complete protection from single errors, has more than 200% area and power overheads. However, error/fault detection by Dual Modular Redundancy (DMR) usually has less (100%) overhead. Therefore, an implementation that uses DMR-based hardware error detection and software recovery methods can reduce overheads. Further, distributing fault tolerance tasks to higher layers enable the designer to take advantage of the masking effects of more layers [7].
In [4], the authors outline a methodology for implementing cross-layer resilience. Additional subsystems *Error handler routine*, *Resource Map*, *Hardware Configuration Routine*, and *Task Scheduler* in the operating system are used to trigger the appropriate fault-tolerant technique at the appropriate layer. In [8], the authors proposed new techniques – *Error-aware placement* and *Failure prediction* – for globally-optimized cross-layer resilience. Similarly, in [9], the authors propose various cross-layer techniques – from *microarchitecture* to *application* level – for both general purpose processor-based and reconfigurable processor-based embedded systems. In all methods presented, every layer takes advantage of the information available at its adjacent layers. In [6], the authors present a cross-layer approach providing resilience in multimedia applications. Specifically, the proposed method uses hardware layer for error detection, middleware for Drop and Forward recovery and application layer for error resilient application design. In [10], the authors provide a heuristic-based methodology for combining several hardware and software techniques – *Circuit-level hardening*, *logic-level parity checking*, *microarchitectural recovery*, and *ABFT* – to provide soft-error tolerance in processor cores.
Most of the state-of-the-art cross-layer reliability techniques lack a holistic approach and do not consider all reliability metrics – *Functional*, *Timing* and *Lifetime*. For instance, the approach described in [10] involves maximizing the fault-mitigation by software layers. Usually, software mitigation of hardware faults – based on *temporal redundancy* – incurs lesser area/power overheads. However, the increased execution time can lead to faster aging in the long run. Therefore, systems that have design constraints of system lifetime have to use additional processing units. This can offset some of the area/cost advantages. In [11], the authors show the adverse effects of increasing checkpoints, a temporal redundancy-based method, on permanent fault tolerance. Further, barring a few application areas that require high reliability in all three measures, most applications, especially soft real-time systems, can tolerate some degradation in each or all of the metrics. We envision a more application-specific and holistic methodology, as depicted in Figure 1, for incorporating cross-layer reliability into system design. Fault-mitigation methods need to be modeled for their characterization. Such models, along with a cross-layer DSE framework, can be used during design-time to determine the feasibility of the method for a particular application, estimate the range of variations in system state that the method can handle, compare different methods and determine the appropriate configuration of various methods at different layers from an early stage in the design phase. Similarly, run-time models of such mitigation methods can be used to adapt the methods for changing system state or switch to other fault-mitigation methods.


Applications, Architecture, Specifications
Fault-mitigation methods
Modeling and characterization
Design-time Models
Run-time Models
Design-time DSE
Run-time DSE
Cross-layer fault-tolerance design
Select and configure methods
Dynamic adaptation
Requirements, System state (run-time)
Figure 1: Cross-layer reliability design methodology

III. Cross-layer Fault-Mitigation Framework

Cross-layer fault-tolerance entails distributing fault mitigation activities among different layers, in order to achieve system-level objectives within the constraints. Fault mitigation can be broadly divided into two stages – *Detection/Validation* and *Tolerance*. Detection methods can be characterized by their latency (T_{FD}), coverage (Cov_{FD}) and the associated power and area overheads (Ov_{FD}). Similarly, a tolerance technique can be characterized by reconfiguration latency (T_{FT}), and associated overheads (Ov_{FT}). For instance, Dual Modular Redundancy (DMR) in hardware usually has a latency equal to that of the comparator, 100% power and area overheads and a coverage equivalent to *1 - (fraction of faults affecting both modules simultaneously)*. Similarly, Triple Modular Redundancy (TMR) in hardware has a reconfiguration latency equivalent to the latency of the voter module, and 200% power and area overheads. Each fault mitigation method, a combination of detection and

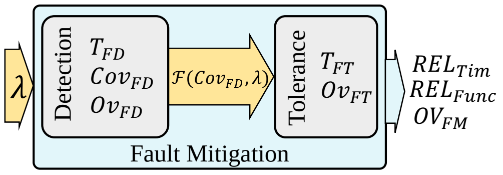


Figure 2: Characterization of fault-mitigation methods

tolerance methods can be characterized as shown in Figure 2. Masking factor (MF_{FM}) is the ratio of error rate with fault mitigation (λ') to that without (λ). CDF_{FM} denotes the cumulative distribution function (CDF) of execution latency and Ov_{FM} denotes overheads in area and power. These three quantities, as functions of λ and other characteristics shown inside the Fault Mitigation block in Figure 2, can be used to estimate the performance metrics of the method and also form the interface for plugging the method into the overall cross-layer analysis framework.

With a cross-layer approach, the convenient abstractions of single-layer design cannot be used. Designing for system-level fault-tolerance must incorporate effects of fault mitigation at all layers. Hence, appropriate resilience interfaces are required for effective transfer and usage of inter-layer information. Figure 3 shows a framework with such interfaces for cross-layer fault-tolerance design. Functional reliability, the rate of errors during application execution, can be determined by considering the masking factor, MF_X, of each layer (X). It consists of implicit fault masking at that layer, MF_{X0} and the masking effect due to fault mitigation methods implemented at that layer. Assuming all detected errors/faults are mitigated, the product of error rate at X, i.e. λ_X, and the coverage of detection methods at layers below X, i.e. $Cov_{\underline{X}}$ denotes the masking effect due to detection at layers below X. Similarly, $(1 - MF'_{X0}).Cov_X$ is the masking due to detection methods used at layer X. MF'_{X0} is the net effect of MF_{X0} and $\lambda_X.Cov_{\underline{X}}$. Please note that simple proportionality relations have been used among various quantities only as an illustration. Actual implementation will involve more complex method-specific relations among various quantities. Timing reliability of the system, quantified by the probability of execution completion before a deadline, can be obtained from the CDF of execution latency (CDF_X) due to mitigation methods implemented at each layer X. Similarly, the overhead information is essential for computing overall system performance and satisfying the design constraints. The average execution time, $T_{avg(X)}$, can be used to quantify metrics such as throughput, energy consumption, system lifetime etc. Similarly, area and power overheads information can be used to satisfy system constraints. The three interfaces described here provide methods to collect appropriate information for cross-layer analysis. The structures necessary for carrying the information vary with each layer. For instance, while λ_X provides a metric to quantify the effect of fault-mitigation at one layer on all upper layers, the granularity of structures for capturing the effect of λ_X depends on the designer and the intended application. In [12], Instruction Vulnerability Index and Function Vulnerability Index were used to incorporate

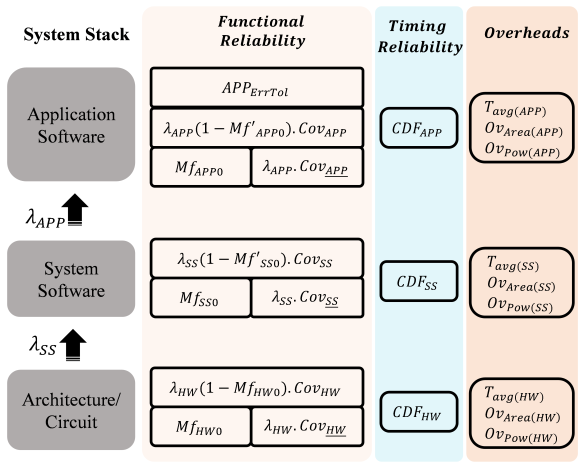


Figure 3: Framework for cross-layer fault-tolerance analysis

the effect of λ_X and implicit masking effects (MF_{X0}) at the architecture and software layers. Similarly, structures suitable for each interface can be used to enable inter-layer information.

IV. Modeling Cross-layer Fault-Mitigation

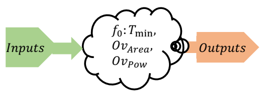
A. *Modeling Fault-Mitigation methods*

Designing cross-layer reliability involves implementing various fault-mitigation methods at different layers of the system stack. To select appropriate methods and their configurations for each layer, the effect of implementing each method needs to be estimated. To this end, we use analytical models for computing the following performance metrics:

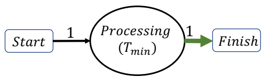
- **Functional Reliability:** It refers to the probability of getting functionally correct outputs in the presence of faults. As shown in Figure 2, the functional reliability is a function of the coverage of the detection method implemented and the fault/error rate.
- **Timing Reliability:** It refers to the probability of the execution completing within the specified deadline. Timing reliability can be determined from the cumulative distribution function of the execution time.
- **Overheads:** We limit our analysis to the determine the following overheads – cost (quantified by the physical area of implementation), power, energy and average execution time T_{avg}. Lifetime reliability can be estimated as a function of T_{avg}.

We model each application-agnostic fault-mitigation method discussed in this article as an absorbing Markov-chain. This enables the computation of T_{avg} and the cumulative distribution function (CDF) of the execution time analytically. For an absorbing Markov-chain, the transition matrix P can be represented as $P = \begin{bmatrix} I & 0 \\ R & Q \end{bmatrix}$, where I is the identity sub-matrix, R is the transition sub-matrix from transient to absorbing states and Q is a $n \times n$ transition sub-matrix for transient states only [13].

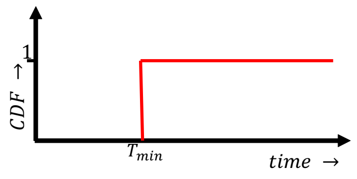
The fundamental matrix F can be obtained as $\mathrm{F} = (I - Q)^{-1}$. Each entry $\mathrm{F}_{i,j}$ represents the expected number of periods that the chain spends in transient state j given



(a) Block Diagram



(b) Markov-chain for base case



(c) CDF for base case

Figure 4: Base Case: No Fault Mitigation

that the chain began in transient state i. Further, the average execution time can be computed as $T_{avg} = T_{exec} \times (e_1^T \mathrm{F})$, where $e_1 = [1, 0, ...0]$ and T_{exec} is the state-residence time of the states in the Markov-chain.

Fault/Error Model: We assume the fault/error-mitigation methods discussed in this article are for mitigating the effects of operational transient faults and resulting errors and failures in the datapath, i.e. computational faults/errors. We assume the occurrence of faults/errors follows a Poisson distribution, λ being the fault/error-rate per module. Our analysis assumes that multiple faults/errors can occur during a module's execution. In the current article, we provide the modeling details of 2 methods in comparison to the Base case of not implementing any fault-mitigation.

1) Base Case (No Fault-Mitigation): We define the base case as– *Processing without implementing any fault-mitigation method*. The block diagram for the base case is shown in Figure 4a. The base case forms the reference point for modeling and comparing the various fault-mitigation methods. Also, the notations used in describing the base case will be followed for the rest of the methods modeled in the current article.

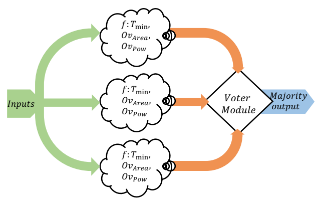
Functional Reliability: The probability of a fault-induced error occurring during processing is shown in Eq. 1. We assume no implicit masking of errors.

$$\begin{aligned} &Prob(Error\ without\ Fault - Mitigation) \\ &= 1 - Prob(No\ faults\ during\ processing) \\ &= (1 - e^{-\lambda T_{min}}) \end{aligned} \tag{1}$$

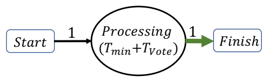
Timing Reliability: The Markov-chain for the base case is shown in Figure 4b. *Finish* signifies the absorbing state in the chain. The state-residence time is shown in the state in parenthesis. The CDF of the execution time is shown in Figure 4c.

Overheads: Since the base case does not involve any form of spatial or temporal redundancy, there are no extra overheads incurred. The resulting costs are shown in Eq. 2

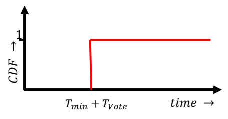
$$\begin{aligned} Average\ Execution\ time &: T_{avg} = T_{min} \\ Area\ cost &: Ov_{Area} \\ Power\ cost &: Ov_{Pow} \\ Energy\ Consumption &: Ov_{Energy} = T_{avg} \times Ov_{Pow} \end{aligned} \tag{2}$$


Inputs
f: T_{min}, Ov_{Area}, Ov_{Pow}
f: T_{min}, Ov_{Area}, Ov_{Pow}
f: T_{min}, Ov_{Area}, Ov_{Pow}
Voter Module
Majority output

(a) Block Diagram



(b) Markov-chain for $TMRVot$



(c) CDF for $TMRVot$

Figure 5: $TMRVot$: TMR with Voting

2) Triple Modular Redundancy with Voting (TMRVot): TMR is one of the most commonly used forms of spatial redundancy. TMR can mask a large proportion of faults/errors, at high area and power overheads. Figure 5a shows the block diagram for a TMR implementation. We assume the voter module to compute the majority of the 3 modules' outputs as the final output. T_{min} is the minimum execution time of each module. Similarly, Ov_{Area} and Ov_{Pow} are the area and power costs of implementing each module.

Functional Reliability: In TMR, the fault-masking is achieved by taking the majority of the outputs. Hence, no separate fault-detection method needs to be used. So, the coverage is equivalent to the coverage of the implicit fault-masking. The output can have a fault-induced errors in the case – A fault/error occurring in at least 2 of the three modules. So, the probability of a computational error can be estimated as shown in Eq. 3.

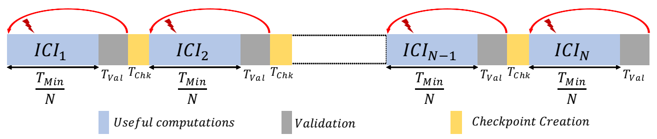
$$\begin{aligned} Prob(Error) = &\ 3 \times Prob(Error\ in\ any\ 2\ modules\ only) \\ &+ Prob(Error\ in\ all\ three\ modules) \\ = &\ 3(1 - e^{-\lambda T_{min}})^2 e^{-\lambda T_{min}} + (1 - e^{-\lambda T_{min}})^3 \\ = &\ 1 - 3e^{-2\lambda T_{min}} + 2e^{-3\lambda T_{min}} \end{aligned} \tag{3}$$

The probability of computational error in the absence of any fault-mitigation implementation and the resulting $Masking\ Factor$ is shown in Eq. 4.

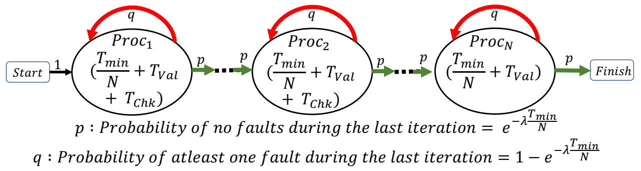
$$\begin{aligned} &Prob(Error\ without\ Fault - Mitigation) \\ &= Prob(Atleast\ one\ error\ during\ application\ execution) \\ &= (1 - e^{-\lambda T_{min}}) \\ &Masking\ factor:\ MF_{TMRVot} \\ &= \frac{Prob(Error\ without\ Fault - Mitigation)}{Prob(Error\ with\ Fault - Mitigation)} \\ &= (1 - e^{-\lambda T_{min}})/(1 - 3e^{-2\lambda T_{min}} + 2e^{-3\lambda T_{min}}) \end{aligned} \tag{4}$$

Timing Reliability: TMR does not involve any form of temporal redundancy. The resulting Markov-chain is shown in Figure 5b. $Finish$ signifies the absorbing state in the chain. The state-residence time includes the time taken by the $Voter\ Module$. The CDF of the execution time is shown in Figure 5c.

Overheads: Since TMR involves 3 parallel modules, the



(a) Typical execution pattern


Start
Finish
$Proc_1$ $(\frac{T_{min}}{N} + T_{Val} + T_{Chk})$
$Proc_2$ $(\frac{T_{min}}{N} + T_{Val} + T_{Chk})$
$Proc_N$ $(\frac{T_{min}}{N} + T_{Val})$
p : Probability of no faults during the last iteration $= e^{-\lambda\frac{T_{min}}{N}}$
q : Probability of atleast one fault during the last iteration $= 1 - e^{-\lambda\frac{T_{min}}{N}}$

(b) Markov-chain for $ChkEVal$

Figure 6: Checkpointing and rollback recovery with N ICIs

$States$	$Proc_1$	$Proc_2$	....	$Proc_i$	......	$Proc_N$	$Finish$
$Proc_1$	q	p	....	0	....	0	0
$Proc_2$	0	q	....	0	....	0	0
:	:	:	..:.	:		:	:
$Proc_i$	0	0	....	q	....	0	0
:	:	:	....	:	:	:	:
$Proc_N$	0	0	....	0		q	p
$Finish$	0	0	....	0	....	0	1

Figure 7: Transition Matrix for Markov-chain of $ChkEVal$

area and power costs incur around 200% increase over the base case. The resulting overheads are shown in Figure 5.

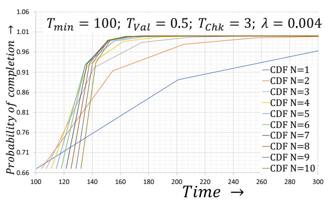
$$\begin{aligned} Average\ Execution\ time:\ & T_{avg(TMRVot)} = T_{min} + T_{Vote} \\ Area\ cost:\ & Ov_{Area(TMRVot)} = 3 \times Ov_{Area} \\ Power\ cost:\ & Ov_{Pow(TMRVot)} = 3 \times Ov_{Pow} \\ Energy\ Cost:\ & Ov_{Energy(TMRVot)} \\ & = T_{avg(TMRVot)} \times Ov_{Pow(TMRVot)} \end{aligned} \tag{5}$$

3) Checkpointing and Rollback Recovery with end-Validation ($ChkEVal$): Checkpointing with rollback recovery is one of the more prevalent methods used for fault-mitigation. It involves saving the program state at regular intervals and restoring the saved state in the event of a fault/error. The timing overheads associated with checkpoint creation and validation determine the effectiveness of this method. The typical execution pattern of processing with N inter-checkpoint intervals (ICIs) is shown in Figure 6a. Please note that our model assumes validating the computation results at the end of each ICI. Further, we assume fault-free validation and checkpoint creation. T_{Val} and T_{Chk} denote the time spent on performing each validation and creating each checkpoint respectively.

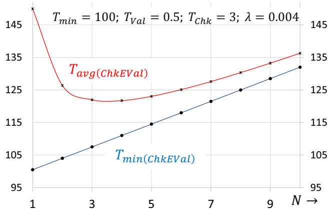
Functional Reliability: The fault-mitigation coverage is equal to the coverage of the validation method implemented.

Timing Reliability: The Markov-chain for $ChkEVal$ is shown in Figure 6b and its corresponding transition matrix ($P_{ChkEVal}$) is shown in Figure 7. The CDF and T_{avg} is a function of the number of ICIs, N. Figure 8a shows the varying CDF with increasing N for a test-case.

Overheads: The average time to completion can be obtained from the absorbing Markov-chain's transition matrix. Eq. 6 shows the overheads associated with $ChkEVal$. $\mathrm{F}_{\mathrm{ChkEVal}}$ corresponds to the $Fundamental\ Matrix$ derived from $P_{ChkEVal}$. Further, Figure 8b shows the variation of $T_{avg(ChkEVal)}$ with increasing N.



(a) Variation of CDF



(b) Variation of T_{avg} and T_{min}

Figure 8: $ChkEVal$: Variation of performance metrics with number of ICIs

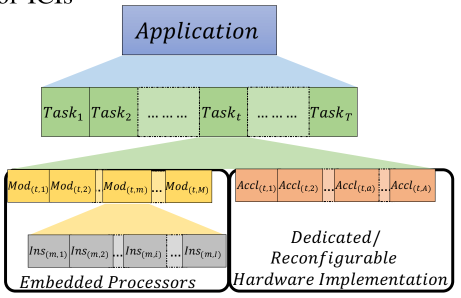

Application
$Task_1$ $Task_2$ ……… $Task_t$ ……… $Task_T$
$Mod_{(t,1)}$ $Mod_{(t,2)}$ $Mod_{(t,m)}$ $Mod_{(t,M)}$
$Ins_{(m,1)}$ $Ins_{(m,2)}$ $Ins_{(m,i)}$ $Ins_{(m,I)}$
Embedded Processors
$Accl_{(t,1)}$ $Accl_{(t,2)}$ $Accl_{(t,a)}$ $Accl_{(t,A)}$
Dedicated/ Reconfigurable Hardware Implementation

Figure 9: Multi-layer view of a typical hierarchy in application execution

$$\begin{aligned} & Average\ Execution\ time:\ T_{avg(ChkEVal)} \\ & = \left(\frac{T_{min}}{N} + T_{Val} + T_{Chk}\right) \times (e_1^T \mathrm{F}_{\mathrm{ChkEVal}}) - T_{Chk} \\ & Area\ cost:\ Ov_{Area(ChkEVal)} = Ov_{Area} \\ & Power\ cost:\ Ov_{Pow(ChkEVal)} = Ov_{Pow} \\ & Energy\ Cost:\ Ov_{Energy(ChkEVal)} \\ & = T_{avg(ChkEVal)} \times Ov_{Pow(ChkEVal)} \end{aligned} \tag{6}$$

B. Modeling Inter-layer interaction of Fault-Mitigation methods

The typical hierarchy involved in an application's execution on a heterogeneous hardware platform can be represented is shown in Figure 9. Any application can be represented as a set of tasks – $Task_t,\ where\ t\ \epsilon\ \{1, 2..., T\}$. Further, each task $Task_t$ can be modeled as composed of a set of modules (or function calls) – $Mod_{(t,m)},\ where\ m\ \epsilon\ \{1, 2..., M\}$ – that is executed on instruction-set based embedded processors as a sequence of instructions – $Ins_{(m,i)},\ where\ i\ \epsilon\ \{1, 2..., I\}$. Similarly, to satisfy the performance requirements of the application, some portion of the task may need to be processed on accelerators – $Accl_{(t,a)},\ where\ a\ \epsilon\ \{1, 2..., A\}$ – implemented on dedicated hardware and/or dynamically reconfigurable fabric. Please note that, as shown in Figure 9, we model the application, tasks, and modules as linear sequences of their constituent tasks, modules and accelerators, and instructions respectively. For the current article, we have not considered typically occurring branching and dependencies among the various tasks, modules, accelerators and instructions. However, our modeling methodology can be extended to consider such dependencies and is planned for future research.

Due to the randomness involved in the occurrence of faults

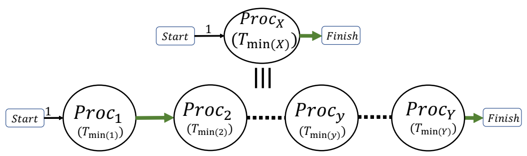


Figure 10: Modeling a process having sub-processes as an absorbing Markov-chain

and their mitigation, the execution of each application (task, module) can be modeled as a stochastic process comprising of a set of independent sub-processes. As shown in Eq. 7, each of the processes and sub-processes can be characterized by the set of parameters discussed in Section IV-A. We discus each of the parameters below.

$$\begin{aligned} &X(RelF_{(X)}\ ,T_{min(X)},\ CDF_{(X)},\ T_{avg(X)},\\ &\quad Ov_{Area(X)},\ Ov_{Pow(X)},\ Ov_{Energy(X)})\\ &X\epsilon\{Ins_{(m,i)},\ Mod_{(t,m)},\ Accl_{(t,a)},\ Task_t,\ \ Application\} \end{aligned} \tag{7}$$

Functional Reliability: $RelF_{(X)}$ refers to the probability of a fault-induced computational error occurring during the process X. It should be noted that the assumption of independence, w.r.t. functional reliability, of the sub-processes of a process holds iff the probability of an error occurring in one sub-process does not affect the error occurrence in any other. In the current article, we consider only fault-induced errors. Hence, $RelF_{(X)}$ can be defined as shown in Eq. 8.

$$RelF_{(X)} = \begin{cases} \quad if\ sub-processes\ exist,\ \max_{Y} RelF_{(y)}, \\ \quad where, Y: set\ of\ sub-processes\ of\ X \\ \\ else,\ Probability\ of\ error\ with\ fault-mitigation \end{cases} \tag{8}$$

Timing Reliability: The Markov-chain model used in Section IV-A can be extended to a combination of sub-processes as shown in Figure 10. Further, the estimation methods on an absorbing Markov-chain discussed earlier can be used to find the CDF of the overall execution time. However, a faster method to estimate the CDF is by taking the product of the CDFs of the sub-processes. Assuming independence of the sub-processes, the probability distribution function (pdf) of the complete process is the convolution of the pdfs of the sub-processes. Hence, the CDF of the process is the product of the CDFs of the sub-processes. So, $CDF_{(X)} = \prod_{y\epsilon Y} CDF_{(y)}$. Similarly, $T_{min(X)}$, the minimum execution time of process X is the sum of the minimum execution times of the sub-processes: $T_{min(X)} = \sum_{y\epsilon Y} T_{min(y)}$

Overheads: As mentioned before, the $T_{avg(X)}$ can be estimated from the expanded Markov-chain based representation of the process. However, under the independence assumption, the overheads can be computed as shown in Eq. 9. $Ov_{Area(X)}$ is a determined by the mapping of tasks/modules/accelerators. Similarly, $Ov_{Pow(X)}$ is a function of the scheduling of the tasks and fault-mitigation methods implemented.

$$T_{avg(X)} = \sum_{y\epsilon Y} T_{avg(y)};\ Ov_{Energy(X)} = \sum_{y\epsilon Y} Ov_{Energy(y)} \tag{9}$$

V. Conclusion

With increasing susceptibility of hardware to physical faults, a comprehensive fault-aware cross-layer design approach is necessary. To this end, a cross-layer analysis framework is proposed. The proposed framework considers the effect of using different fault-detection and fault-tolerance methods, their coverage and the layer of implementation to determine the overall system-level effect. While the proposed framework is used to estimate the effect of some combination of fault-mitigation methods, each fault-mitigation method needs to be modeled appropriately to fit into the framework. To this end, a Markov-chain based analytical modeling methodology to model methods and their cross-layer interaction was proposed. Further research is required towards integrating the effects of dependencies in tasks and non-linear, non-sequential execution of tasks on real hardware platforms.

Acknowledgment

This work is supported in part by the German Research Foundation (DFG) within the Cluster of Excellence “Center for Advancing Electronics Dresden” (cfaed) at the Technische Universität Dresden.

References

[1] A. Geist, “Supercomputing’s monster in the closet,” *IEEE Spectrum*, March 2016.

[2] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, “Modeling the effect of technology trends on the soft error rate of combinational logic,” in *Dependable Systems and Networks*, 2002.

[3] S. Borkar, “Designing reliable systems from unreliable components: the challenges of transistor variability and degradation,” *Micro, IEEE*, 2005.

[4] N. P. Carter, H. Naeimi, and D. S. Gardner, “Design techniques for cross-layer resilience,” in *DATE*, 2010.

[5] S. S. Sahoo, B. Veeravalli, and A. Kumar, “Cross-layer fault-tolerant design of real-time systems,” in *DFTS*, 2016.

[6] K. Lee, A. Shrivastava, M. Kim, N. Dutt, and N. Venkatasubramanian, “Mitigating the impact of hardware defects on multimedia applications: a cross-layer approach,” in *Proceedings of the 16th ACM international conference on Multimedia*, 2008.

[7] T. Santini, P. Rech, A. Sartor, U. B. Corrêa, L. Carro, and F. R. Wagner, “Evaluation of failures masking across the software stack,” *MEDIAN*, 2015.

[8] L. Leem, H. Cho, H. H. Lee, Y. M. Kim, Y. Li, and S. Mitra, “Cross-layer error resilience for robust systems,” in *ICCAD*, 2010.

[9] J. Henkel, L. Bauer, H. Zhang, S. Rehman, and M. Shafique, “Multi-layer dependability: From microarchitecture to application level,” in *DAC*, 2014.

[10] E. Cheng, S. Mirkhani, L. G. Szafaryn, C.-Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra, “CLEAR: Cross-Layer Exploration for Architecting Resilience - Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores,” ser. DAC, 2016.

[11] A. Das, A. Kumar, and B. Veeravalli, “Aging-aware hardware-software task partitioning for reliable reconfigurable multiprocessor systems,” in *CASES*, 2013.

[12] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel, “Reliable software for unreliable hardware: embedded code generation aiming at reliability,” in *ISSS+CODES*, 2011.

[13] J. G. Kemeny, J. L. Snell, and G. L. Thompson, *Introduction to Finite Mathematics*. Prentice Hall Inc, 1974.