Cross-Layer Fault-Tolerant Design of Real-Time Systems

Siva Satyendra Sahoo, Bharadwaj Veeravalli
National University of Singapore
Department of Electrical and Computer Engineering
Email: satyendra@u.nus.edu, elebv@nus.edu.sg

Akash Kumar
Technische Universität Dresden
Center for Advancing Electronics Dresden (cfaed)
Email: akash.kumar@tu-dresden.de

***Abstract*—Continued transistor scaling and increasing power density has resulted in considerable increase in fault rates of nano-technology systems. Cross-layer fault tolerance techniques present a more cost-efficient methodology for adapting to such increased fault rates as opposed to fixing everything at the hardware layer. The effectiveness (*Coverage, Fault-Masking and Recovery*) and overheads (*Execution time, Energy and Cost*) of each fault tolerance technique varies with the layer and frequency at which it is applied. The choice of appropriate fault-aware design should also account for the application specific design goals and constraints of real-time systems. To this end, we provide a brief survey of fault-tolerance methods and discuss their suitability to cross-layer design. We also provide a few case studies that motivate the need for effective design space exploration (DSE) for cross-layer fault-aware design of real-time systems and discuss a few factors that have a major impact on such DSE.**

I. Introduction

We are in the middle of an exciting technology transformation. Most classes of electronic devices are already Internet-enabled and inter-connected. With the growing interest in Internet of Things (IoT) and *wearables*, even the smallest of devices are expected to have some form of connectivity. Consequently, real-time systems are going to be used in an even wider range of applications. This broad variety of application areas signifies varying demands on the system regarding both performance and dependability [1]. A multimedia application like portable gaming console, for instance, has more stringent throughput and energy requirements than a finance related application like an ATM, which has stricter demands on correctness of computation. Varying relevance of execution deadline in different real-time systems (*Soft, Firm* and *Hard*) add to the system design complexity. Even for similar applications, the performance metrics may vary with different usage environments. It is impossible for a single system architecture to cater to this wide variety of applications. Therefore, appropriate computation platforms and design techniques are necessary to meet application-specific performance goals. Increasing physical fault rates pose a major challenge to meeting those goals without exceeding the system's resource constraints.

Technology scaling and architectural innovations have led to the design of denser and more complex systems. However, breakdown of Dennard Scaling [2] has led to increased power density, which along with reduced transistor size, has introduced reliability issues [3]. With smaller transistors, the number of faults due to manufacturing defects such as imperfect lithographic patterning, has also increased. Moreover, increased temperature due to higher power density leads to faster aging. Consequently, aging related fault mechanisms like Negative-Bias Temperature Instability (NBTI), Time-Dependent Dielectric Breakdown (TDDB), Hot Carrier Injection (HCI) and Electro Migration (EM) are accelerated and result in higher intermittent and permanent fault rates, eventually leading to a reduced system lifetime. Smaller transistor size also results in an increase in Soft Error Rates (SER) of logic circuits. While the logical masking effect remains unaltered, electrical masking effect is reduced by smaller and faster transistors [4], [5]. Further, deeper pipelines used for enabling higher clock speed have resulted in reduction of latching-window masking, leading to even higher SER in microprocessors. Therefore, extracting increasing *usable-performance* out of real-time systems requires building resilient systems out of increasingly unreliable hardware [6].

Traditional fault tolerance methods focus on mitigating all physical faults at the hardware layer. Although this approach provides freedom from dealing with physical faults to the software designer, the area and power overhead costs may be infeasible for most applications. Increased fault rate mitigation and achieving application-specific performance targets require innovative approaches beyond such *single-layer* approaches. In Section II we review some fault tolerance methods implemented at each layer of system stack for both memory and computation. In contrast to the single-layer methods, a cross-layer design approach involves utilizing the information and capabilities of each layer to provide adequate overall system resilience with a design that does not exceed the application's design constraints. In Section III we compare cross-layer approach with single-layer fault tolerance and review some recent projects that implement cross-layer resilience. Cross-layer fault tolerance introduces the added complexity of selecting appropriate resilience methods for an application and also optimizing these methods for system-level design goals. In Section IV we provide a first order framework for comparing different cross-layer designs towards an effective Design Space Exploration (DSE). In Section V, we conclude the paper with discussions about important aspects of cross-layer DSE and provide directions and scope of our future work.

II. Design for Fault Tolerance

Any fault tolerance technique involves either masking the effect of the fault or detecting and recovering from the errors caused due to the fault. While system recovery may be sufficient for resilience against transient faults or soft errors, permanent fault tolerance usually requires additional step of

isolating the fault by some form of system reconfiguration. We provide a brief overview of some fault tolerance techniques for faults in memory and computation. While some of the techniques are applicable to both transient and intermittent faults, we focus more on transient fault tolerance.

A. Fault tolerance in Memory

Information redundancy in the form of additional bits for Error Checking and Correcting (ECC) is commonly used for both SRAM-based caches and DRAM-based main memory. Hamming [7] or Hsiao [8] code based Single-bit-Error-Correcting and Double-bit-Error-Detecting (SEC-DED) codes are usually sufficient for most systems. More robust methods like Double-bit-Error-Correcting and Triple-bit-Error-Detecting (DEC-TED) codes can be used for higher resilience against random bit errors. Reed Solomon [9] codes and Single-Nibble-error-Correcting and Double-Nibble-error-Detecting (SNC-DND) codes [10] are usually used for protection against multiple-bit burst errors. Granularity of ECC implementation provides the trade-off between resilience and storage overhead. Table I shows the storage overheads associated with some ECC implementations [11].

In caches, the write-policy determines the amount of correction capabilities that can be implemented. For a write-through policy, the access granularity at Last Level Cache (LLC) from Level 1 (L1) cache is a word and hence the ECC granularity at LLC should be a single word. However, for a write-back policy of L1 cache, the LLC access is a full L1 cache line and therefore allows higher granularity of ECC with reduced overheads. Additional tolerance can be provided by interleaving more ECC codes within the cache line, albeit with more overheads. Most systems use commodity DRAM devices for main memory. Therefore, there is an additional cost in terms of I/O pins over and above the usual storage overheads. ECC DIMMs (dual in-line-memory module) provide SEC-DED for each DRAM rank and have higher overheads compared to non-ECC DIMMs. More recent methods have been developed to provide fault tolerance against permanent faults in one or more chips on a DIMM. Such Chipkill-correct [12] techniques spread the DRAM access over multiple chips and use single-symbol-error-correcting and double-symbol-error-detecting codes for error-correction. Adaptive methods of ECC like Virtualized ECC [13] and Bamboo codes [14] provide flexible and tunable approaches to main memory fault tolerance. Such techniques can be used to find appropriate trade-offs in cross-layer design approaches.

Summarizing, ECC granularity and fault-coverage provide the tunable parameters, and memory controller, the tuning knob for varying error protection levels based on system requirements.

TABLE I: ECC storage overheads [11]

Data Bits	SEC-DED		SNC-DND		DEC-TED	
	Check bits	Overhead	Check bits	Overhead	Check bits	Overhead
16	6	38%	12	75%	11	69%
32	7	22%	12	38%	13	41%
64	8	13%	14	22%	15	23%
128	9	7%	16	13%	17	13%

B. Fault tolerance in Computation

Tolerance techniques for soft-errors in computation circuits primarily involve some form of execution redundancy – either spatial and/or temporal. Implemented at any level of the system stack, Dual Modular Redundancy (DMR) provides only fault/error detection and Triple Modular Redundancy (TMR) provides masking of any single fault/error. In terms of cost, TMR can result in more than 200% area and power overheads. Area and power overheads in a LEON3 core when introduced with varying levels of TMR in pipeline, cache and register file are reported in [15]. Corresponding results for an FPGA-based implementation of LEON3 are presented in [16]. In both cases, power and area overheads of more than 200% are observed. We briefly describe a few fault tolerance techniques for each layer.

Circuit level fault masking usually involves circuit hardening by using multiple flip-flops [17] or by gate resizing [18]. Multiple flip-flop based design uses scan-flops already present in the circuit to provide error tolerance. Gate resizing involves using bigger transistors to provide better tolerance against radiation induced soft errors. More flexible methods [19]–[21] use partial replication based on profiling results to obtain reduced coverage at lower power and area overheads. Similarly, low overhead methods based on circuit monitoring enable low cost [22] and configurable fault detection [23].

At the architecture level, the granularity of execution replication provides the trade-off in error resilience and associated overheads. The granularity may vary from a single module like the pipeline [24], [25] to an entire core in chip multiprocessors [26]. Time redundancy-based techniques like redundant multithreading [27], [28] are also used. Some fault detection methods involve manipulating the pipeline [29], [30] to detect both transient and intermittent faults. Similar to circuit level, symptom or assertion monitoring based detection methods [31] provide incomplete coverage at very low overheads. These symptoms could be exceptions, control flow mis-speculations and cache or translation look-aside buffer misses. Some code-based methods are also used at architecture level to provide concurrent fault detection and masking at lower area overheads. These methods are based on *AN codes* [32] and are based on the principle of providing a redundant representation of numbers such that the results of some operation on them can be analyzed to detect and correct errors. As shown in Equation 1 some operations preserve certain properties of the operands. Such operations are performed on both operands as well as the result and the results can be used for detection and sometimes correction. However, such methods are very application specific and require high design effort.

$$\begin{aligned} &A \times N_1 \pm A \times N_2 = A \times (N_1 \pm N_2) \\ &(N_1 \otimes N_2)\ mod\ A = ((N_1\ mod\ A) \otimes (N_2\ mod\ A))\ mod\ A \end{aligned} \tag{1}$$

C. Software Fault Tolerance

Decreasing masking effects in logic circuits was discussed in Section I. However, logical masking does not depend on the transistor size and on-chip variations. Therefore, every logic circuit has some masking effect for SEUs. This masking is not limited to logic circuits only. Program level error masking and its propagation are discussed in [33], where the

authors exploit program-level masking to perform reliability driven prioritization of instructions. Experimental studies into the masking effects of software stack [34] show that lower application failure rates are observed if more abstraction layers exist between hardware and application. So, the error rates seen by each layer decreases as we abstract away from the hardware layer. However, this benefit is obtained at higher performance overheads. We now discuss a few fault tolerance techniques in system and application software.

Similar to circuit and architecture levels, replication and re-execution are the most common methods employed for fault tolerance by system software as well. The replication may be full re-execution [35] of the application or compiler insertion of replicated instructions and checks [36], [37]. More recent approaches use reliability oriented compilation to generate executables with required tolerance levels [38], [39]. Symptom monitoring based approaches are also used at system software level. These symptoms include fatal traps or application aborts and can be used for detecting all kinds of faults.

Application level techniques can take advantage of the information about applications requirements and characteristics to provide customized error tolerance. However, designing such methods might not always be possible for every application and require high design effort. All such tunable methods depend on appropriate profiling of the application [19], [33]. Algorithm based fault tolerance [40], [41] techniques implement specific error checking and correcting in the algorithm to be executed. Some of these methods allow for trade-offs between accuracy and cost. Checkpointing [42] with backward/forward recovery is a more commonly used technique. Whereas backward recovery is not application-specific, forward recovery depends on the application's error-resilience. Higher error resilience of the application can enable more forward recovery and hence reduce the average execution time of real-time tasks. Therefore, accurate application profiling and models that relate application properties to real-time system level objectives and constraints are necessary for selecting and customizing software fault tolerance methods.

III. Cross-Layer Fault Tolerance

Traditional single layer approaches address almost all hardware reliability issues at circuit and architecture level. A phenomenon-based approach – where each fault mechanism (NBTI, EM, Single Event Upsets (SEU) etc.) is mitigated separately to provide an error-free hardware platform– is usually used for physical fault tolerance. This is depicted to the left of system stack in Fig. 1. Barring a few application areas that require high resilience in terms of both functionality and execution time, most applications, especially soft real-time systems, can tolerate some degradation in either or both. We provide a few examples of such applications as case-studies in Section IV.

Building cross-layer resilience into the system involves inter-layer information exchange in the system stack. This information exchange during run-time is depicted to the right of system stack in Fig. 1. Dots and arrows in the figure show the origin and direction respectively for reliability related information. As shown in the figure, fault mitigation activities are distributed across different layers. Inter-layer information

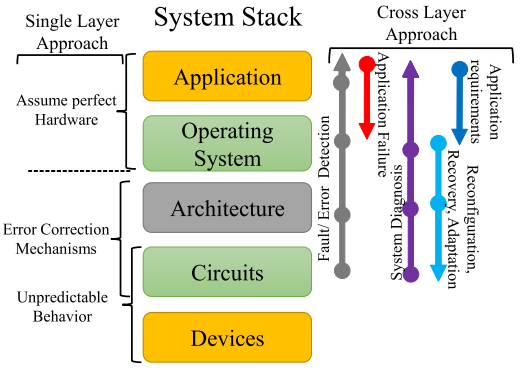

Fig. 1: Comparison of Single-layer and Cross-layer Resilience approaches [43]

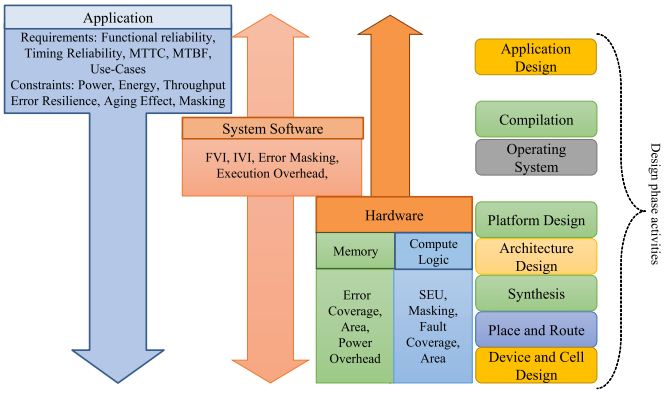

Fig. 2: Cross layer Fault aware system design

transfer enables better knowledge of the current system state and application requirements and hence results in more effective run-time response/reconfiguration to physical faults. Similarly, Fig. 2 shows the activities during design phase of real-time system design and the information origin and transfer from the three layers – *Hardware, System software* and *Application*. This information is essential to evaluate the suitability of a fault tolerance method for an application. Further, each of these selected methods need to be tailored to the application-specific performance goals and design constraints. Some projects that implement some form of cross-layer design approach are discussed below.

ROAR [44] is one of the earliest projects that used interaction among different system layers to provide a demand specific switch between high performance and high resilience on the same hardware platform. Relax [45] provides a framework for achieving effective software recovery from hardware faults. It leverages the decreasing checkpointing information of emerging applications to provide energy-efficient fault tolerance. In [46], 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. For example, authors use Instruction Vulnerability Index (IVI) to quantify the robustness of each instruction executing

on a specific processor. IVI is in turn used to compute the Function Vulnerability Index (FVI) of a function and used for reliability oriented compilation to generate executables based on application specific reliability requirements. In [47], 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.

IV. Cross-Layer Design Case Study

As discussed in Section II, fault tolerance measures can be implemented at different layers with different trade-off between performance and overheads. Therefore, system-level metrics are necessary to evaluate the usefulness of these various measures. System resilience of real-time applications can be expressed in terms of *functional reliability* – correctness of the results and *timing reliability* – frequency of deadline misses. Circuit and architecture level spatial redundancy provide the best reliability values for both these metrics. However, this comes at a very high cost. Although temporal redundancy can reduce the area overheads, it results in decreased timing reliability. System and application software resilience methods have less area overhead, but suffer from decreased timing reliability. A single reliability-oriented design cannot serve every application. Choosing appropriate methods and customizing them for system-level optimization is essential for system-wide cross-layer resilience. In this Section, we provide few example applications that motivate the need for cross-layer DSE. Initially we define a simplified system-level framework that is used for back-of-the-envelope calculations in the case study. We use rough estimates for the analysis, using values based on the results reported in state-of-the-art techniques reviewed in Section II.

We characterize the hardware, system software and the application with parameters as shown in Table II. The masking factor of a layer is the ratio of fault and/or error rate observed at the current layer to those observed at the layer above it. We differentiate this from coverage for error detection, as masking refers to all those faults/errors mitigated by a layer and not just detected. For the hardware layer, masking factor is the ratio of SEU rate to SER and the factors contributing to a higher value are logical, electrical and latch-window based masking. Additional measures (TMR, gate resizing, partial replication, information redundancy in memory etc.) can be implemented to increase Mf_{HW}. Masking factor of system software can be increased by Operating System (OS) level error mitigation, compiler directed replication etc. The overheads in execution due to additional reliability oriented measures implemented at system software and application layer are represented by Ov_{SS} and Ov_{APP} respectively.

Table III shows the performance metrics that can be considered applicable for most real-time systems. λ refers to the SEU rate and is an indicator of the usage environment of the system. Deadline D refers to the time by when the application should complete the execution in order to avoid considerable loss in the quality or catastrophic results in hard real-time systems. Rel_{TIME} is expressed as the $Prob(execution\ time \leq D)$. Rel_{FUNC} can be quantified by either Mean Time To Failure (MTTF) or Mean Time Between Failures (MTBF), depending upon the repairable nature of the system and faults under study. Based on these parameters, the performance metrics are expressed as shown in Equation 2.

TABLE II: Layer Parameters

System Layer	Variable	Description
Hardware	$Area_{HW}$	Area of hardware components
	Pow_{HW}	Power of hardware components
	Mf_{HW}	Masking factor of hardware components
System Software	Ov_{SS}	Execution time overhead factor
	Mf_{SS}	Masking factor of system software
Application	T	Typical execution time
	Ov_{APP}	Execution time overhead factor
	Mf_{APP}	Masking factor of application

TABLE III: Application requirements and performance metrics

Symbol	Description
λ	SEU rate
D	Application deadline
Rel_{TIME}	Timing reliability
Rel_{FUNC}	Functional reliability
Pow_{SYS}	System power dissipation
$Area_{SYS}$	Core and main memory area
$Energy_{SYS}$	Energy usage per cycle

$$\begin{aligned} System\ failure\ rate: \lambda_{FAIL} &= \lambda \div (Mf_{HW} \times Mf_{SS} \times Mf_{APP}) \\ Rel_{FUNC} &= 1 \div (\lambda_{FAIL}\ \ in\ hours) \\ Avg\ execution\ time: T_{avg} &= T \times (1 + Ov_{SS}) \times (1 + Ov_{APP}) \\ Rel_{TIME} &= Probability(Execution\ time \leq D) \end{aligned} \tag{2}$$

We define a baseline system configuration, a generic processor core without any additional built-in reliability methods and using non-ECC main memory. We assume that the application runs on a real-time OS without any explicit fault/error mitigation methods, neither in OS nor in application code. The layer parameters for this baseline system are: $Area_{HW0}$, Pow_{HW0}, Mf_{HW0}, Ov_{SS0}, Mf_{SS0}, Ov_{APP0}, Mf_{APP0}, λ_{FAIL0}, Rel_{FUNC0}, Rel_{TIME0} and $T_{avg} = T$. Based on this we estimate the performance metrics with a hardware-only (HW-only) and a software-only (SW-only) fault tolerance implementation. Note that we consider application failures only due to incorrect end result and do not consider other failures like *system crash* and *system hang* for this analysis.

Hardware-only fault tolerance: Assuming a uniform TMR for the processor core, we can estimate the performance metrics for the system. As per [16], uniform TMR can result in 10x better performance against radiation induced SEUs. So, masking factor for hardware in such system can be estimated as $Mf'_{HW} = 10 \times Mf_{HW0}$. Therefore, Rel_{FUNC} of such a system is $10 \times Rel_{FUNC0}$. Further, the overheads on timing due to any of the layers is minimal i.e. $Ov'_{APP} = Ov'_{SS} = 0$. So, Rel_{TIME} remains unchanged at Rel_{HW0}. Other parameters remain unchanged. Area and power figures are three times that of the baseline system.

Software-only fault tolerance: Checkpointing and rollback recovery is a commonly used fault-tolerance method which can be implemented completely in software. Assuming an application-dependent error detection with coverage Cov_{APP} and only re-execution based recovery, we deter-

mine the performance metrics of such an implementation on our baseline system. We assume an execution overhead of C $(in\ time\ units)$ for error detection and checkpointing and no overheads for recovery or resumption. We use the results from [48] to derive the performance metrics. The application fail rate depends on the coverage of error detection method implemented. Rel_{FUNC} equals $Rel_{FUNC0} \times (1 - Cov_{APP})$. Rel_{TIME} depends on the deadline of the application and is expressed as an $N - fold$ convolution of a probability density function, where N is the number of checkpoints. We use Markov-chain Monte-Carlo simulations to calculate Rel_{TIME} for an example application as shown in Table IV. We fix N that gives the best value of N as that which provides the maximum timing reliability assuming a Poisson distribution of the fault/error inter-arrival times. Area and power values are same as that of the baseline system.

Cross-layer fault tolerance: The increased functional reliability in a SW-only approach is proportional to coverage of the error detection method. An effort to increase the error detection coverage at application layer can lead to a higher execution overhead and hence reduce timing reliability. On the other hand, a HW-only approach results in improved functional reliability with negligible reduction in timing reliability. However, the impact on cost (silicon area) and power is very high. Additionally, the increased energy usage can be infeasible for a portable application. As an example, we consider the *next-frame* computation in a portable gaming device, a soft real-time system. To maintain a frame rate of 30 $frames/sec$, the average execution time has to be around 33.33 ms. If we consider an ideal execution time of 30 ms, a HW-only implementation of uniform TMR gives better guarantees of avoiding *bad-frames* (due to computation errors and incomplete processing) and getting higher frame rates. Increased silicon area and up to three times higher energy-usage can, however, increase the cost and reduce portability of the product. If the functional reliability criteria is relaxed, i.e. tolerating a few *bad-frames* once in a while, a lower Mf'_{HW} may be sufficient. The lower Mf'_{HW} can be achieved by application-specific partial replication with reduced area and power overheads. In a SW-only implementation, if the cost C is high, it may not be able to provide the requisite frame rate for the application. A reduced error rate at application layer, obtained by better masking at all layers can, however, increase timing reliability on account of reduced η.

Table IV shows five different implementations and their performance metrics on different hardware platforms. We assume a linear scaling between the reduction in application fail rate and percentage of hardware area over which TMR was implemented, with a maximum 10x for 300%. Similarly, we assume a linear scaling of error detection time with Cov_{APP} with 0.1 msec for 30% coverage. Since the results are based on hardware replication, we assume the pattern of overheads in power is same as that of area. The first two columns show the HW-only and SW-only implementation discussed earlier. Note the Rel_{TIME} of 0.9995 in SW-only implementation translates to 1 next-frame computation missing deadline in about every 67 seconds. The usage environment of the system can add additional complexity to cross-layer DSE. Note the increased

TABLE IV: Implementation comparisons for equal reliability targets

Parameters	Baseline	HW-only	SW-only	Multimedia (ground)	Multimedia (at altitude)	Financial
Area	A_0	$3 \times A_0$	A_0	$2 \times A_0$	$3 \times A_0$	$2.5 \times A_0$
SEU rate	λ_0	λ_0	λ_0	λ_0	$10 \times \lambda_0$	λ_0
T time (in ms)	30	30	30	30	30	30
D (in ms)	33	33	33	33	33	33
Fault mitigation method	None	Full TMR	Checkpointing (30% coverage)	Partial TMR + Checkpointing (30% coverage)	Full TMR + Checkpointing (10% coverage)	Partial TMR + Checkpointing (60% coverage)
Mf_{HW}	Mf_{HW0}	$10 \times Mf_{HW0}$	Mf_{HW0}	$5 \times Mf_{HW0}$	$10 \times Mf_{HW0}$	$7.5 \times Mf_{HW0}$
Rel_{FUNC} $(errors/sec)$	1.5×10^{-2}	1.5×10^{-3}	1.05×10^{-2}	2.1×10^{-3}	1.35×10^{-2}	0.85×10^{-3}
Rel_{TIME} $(probability)$	1	1	0.9995	0.9999	0.9995	0.9993
$Energy_{SYS}$	E_{SYS0}	$3 \times E_{SYS0}$	$1.075 \times E_{SYS0}$	$2.196 \times E_{SYS0}$	$3.127 \times E_{SYS0}$	$2.744 \times E_{SYS0}$

hardware redundancy along with checkpointing needed to achieve similar results at an altitude. This is needed to counter the effect of increased rate of radiation-induced SEUs. The third implementation shows a miss rate of about once every 6 minutes. This shows the implementation requirements of typical multimedia applications where a higher Rel_{TIME} is required for better QoS at the expense of some errors in some frames which may not be discernible. The last column shows an implementation that is more suitable for finance-related applications, where a higher Rel_{FUNC} i.e. correctness of results is much more desirable than finishing within the deadline. Note the reduced power and area overheads in all cross-layer implementations. Here we have shown only the result of distribution of fault-tolerance methods over different layers. Additional methods to improve masking of each layer can be used to further improve results. Varying application requirements and usage scenarios need different implementation of resilience methods. Effective DSE is necessary to achieve such results within tolerable overheads. In [49], the authors present a framework for analysis of multibit error protection overheads. This enables better architecture-level design choices. However, the analysis is limited to the scope of design of processor core and does not consider cross-layer fault tolerance. Effective cross-layer DSE needs consideration of several other factors like multilayer masking, run-time reconfiguration and above all system-level design optimization.

V. Discussions and Conclusions

A common theme among all the discussed cross-layer implementations is the decoupling of fault tolerance into its constituent stages – *Error detection* and *Recovery* – and distribution of these activities across different layers of the system stack. As discussed in [47], implementing separate fault tolerance stages at different layers can result in reduced power and area overheads. TMR, which provides complete protection from single errors, has more than 200% area and power overheads. However, error/fault detection by 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 masking effects of more layers.

As discussed in Section IV, all fault tolerance methods have to consider system-level objectives. Usually software mitigation of hardware faults suffers less overheads. However, the increased execution time can lead to faster aging in the

long run. Therefore, systems that have design constraints about system lifetime have to use additional spare processing units. This offsets some of the area/cost advantages. In [50], the authors show the effect of increasing checkpoints on permanent fault tolerance and provide hardware-software co-design approaches to counter such effects. Hence, system level optimizations should consider all system-level requirements.

Therefore, a comprehensive fault-aware design approach is necessary to sustain and improve usability of real-time embedded systems. Traditional approaches to resilience may not provide us feasible solutions for this. To this end, cross-layer fault-aware system design provides a more effective alternative. However, given the large number of fault tolerance methods and diverse system requirements, cross-layer DSE is imperative. In this paper, we provided an overview of fault-tolerant design and cross-layer resilience. We provided examples of some applications as a case study to show the importance of cross-layer DSE. We also discussed a few important factors that are crucial to such DSE. We have considered only design stage aspects of cross-layer system design. We aim to include run-time methods into the framework to enable effective run-time reconfiguration of cross-layer resilience based systems.

ACKNOWLEDGMENTS

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] P. Marwedel, *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer Science & Business Media, 2010.
[2] R. H. Dennard, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, 1974.
[3] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *IEEE Micro*, 2003.
[4] 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.
[5] A. Geist, "Supercomputing's monster in the closet," *IEEE Spectrum*, March 2016.
[6] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, 2005.
[7] R. W. Hamming, "Error detecting and error correcting codes," *Bell System technical journal*, 1950.
[8] M.-Y. Hsiao, "A class of optimal minimum odd-weight-column sec-ded codes," *IBM Journal of Research and Development*, 1970.
[9] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, 1960.
[10] C.-L. Chen and M. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM Journal of Research and Development*, 1984.
[11] C. W. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *Device and Materials Reliability, IEEE Transactions on*, 2005.
[12] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," *IBM Microelectronics Division*, 1997.
[13] D. H. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," in *ACM SIGARCH Computer Architecture News*, 2010.
[14] J. Kim, M. Sullivan, and M. Erez, "Bamboo ecc: Strong, safe, and flexible codes for reliable computer memory," in *High Performance Computer Architecture (HPCA)*, 2015.
[15] F. Kriebel, S. Rehman, D. Sun, M. Shafique, and J. Henkel, "Aser: Adaptive soft error resilience for reliability-heterogeneous processors in the dark silicon era," in *Design Automation Conference (DAC)*, 2014.
[16] M. J. Wirthlin, A. M. Keller, C. McCloskey, P. Ridd, D. Lee, and J. Draper, "SEU mitigation and validation of the leon3 soft processor using triple modular redundancy for space processing," in *FPGA*, 2016.
[17] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft-error resilience," *Computer*, 2005.
[18] D. Lunardini, B. Narasimham, V. Ramachandran, V. Srinivasan, R. D. Schrimpf, and W. H. Robinson, "A performance comparison between hardened-by-design and conventional-design standard cells," in *2004 workshop on radiation effects on components and systems, radiation hardening techniques and new developments*, 2004.
[19] K. Mohanram and N. A. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits," in *ITC*, 2003.
[20] ——, "Partial error masking to reduce soft error failure rate in logic circuits," in *Defect and Fault Tolerance in VLSI Systems*, 2003.
[21] R. R. Rao, D. Blaauw, and D. Sylvester, "Soft error reduction in combinational logic using gate resizing and flipflop selection," in *ICCAD*, 2006.
[22] A. Narsale and M. C. Huang, *Variation-tolerant hierarchical voltage monitoring circuit for soft error detection*. IEEE, 2009.
[23] P. Ndai, A. Agarwal, Q. Chen, and K. Roy, "A soft error monitor using switching current detection," in *ICCD*, 2005.
[24] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Microarchitecture. MICRO-32.*, 1999.
[25] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson *et al.*, "IBM's S/390 G5 microprocessor design," *Micro, IEEE*, 1999.
[26] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *ISCA*, 2002.
[27] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," in *Fault-Tolerant Computing. Digest of Papers. Twenty-Ninth Annual International Symposium on*. IEEE, 1999.
[28] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, 2001.
[29] S. Das, C. Tokunaga, S. Pant, W.-H. Ma, S. Kalaiselvan, K. Lai, D. M. Bull, and D. T. Blaauw, "RazorII: In situ error detection and correction for PVT and SER tolerance," *Solid-State Circuits, IEEE Journal of*, 2009.
[30] E. Mizan, T. Amimeur, and M. F. Jacome, "Self-imposed temporal redundancy: An efficient technique to enhance the reliability of pipelined functional units," in *SBAC-PAD*, 2007.
[31] N. J. Wang and S. J. Patel, "ReStore: Symptom-based soft error detection in microprocessors," *Dependable and Secure Computing, IEEE Transactions on*, 2006.
[32] T. R. Rao, *Error coding for arithmetic processors*. Elsevier, 1974.
[33] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel, "Exploiting program-level masking and error propagation for constrained reliability optimization," in *DAC*, 2013.
[34] 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.
[35] J. H. Wensley, M. W. Green, K. N. Levitt, and R. E. Shostak, "The design, analysis, and verification of the SIFT fault tolerant system," in *International conference on Software engineering*, 1976.
[36] B. Nicolescu, R. Velazco, M. Sonza-Reorda, M. Rebaudengo, and M. Violante, "A software fault tolerance method for safety-critical systems: Effectiveness and drawbacks," in *ICSD*, 2002.
[37] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005.
[38] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel, "Reliable software for unreliable hardware: embedded code generation aiming at reliability," in *ISSS+CODES*, 2011.
[39] S. Rehman, M. Shafique, and J. Henkel, "Instruction scheduling for reliability-aware compilation," in *DAC*, 2012.
[40] V. Balasubramanian and P. Banerjee, "Compiler-assisted synthesis of algorithm-based checking in multiprocessors," *IEEE Transactions on Computers*, 1990.
[41] A. A. Al-Yamani, N. Oh, and E. J. McCluskey, "Performance evaluation of checksum-based abft," in *DFT*, 2001.
[42] G. Aupy, A. Benoit, T. Hérault, Y. Robert, F. Vivien, and D. Zaidouni, "On the combination of silent error detection and checkpointing," in *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*, 2013.
[43] N. P. Carter, H. Naeimi, and D. S. Gardner, "Design techniques for cross-layer resilience," in *DATE*, 2010.
[44] N. R. Saxena and E. J. McCluskey, "Dependable adaptive computing systems-the roar project," in *Systems, Man, and Cybernetics,*, 1998.
[45] M. De Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *ACM SIGARCH Computer Architecture News*, 2010.
[46] J. Henkel, L. Bauer, H. Zhang, S. Rehman, and M. Shafique, "Multi-layer dependability: From microarchitecture to application level," in *DAC*, 2014.
[47] 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.
[48] A. Duda, "The effects of checkpointing on program execution time," *Information Processing Letters*, 1983.
[49] L. G. Szafaryn, B. H. Meyer, and K. Skadron, "Evaluating overheads of multibit soft-error protection in the processor core," *IEEE Micro*, 2013.
[50] A. Das, A. Kumar, and B. Veeravalli, "Aging-aware hardware-software task partitioning for reliable reconfigurable multiprocessor systems," in *CASES*, 2013.