2018 International Conference on Field-Programmable Technology (FPT)

QoS-aware Cross-layer Reliability-integrated FPGA-based Dynamic Partially Reconfigurable System Partitioning

Siva Satyendra Sahoo[1], Tuan D. A. Nguyen[2], Bharadwaj Veeravalli[1], Akash Kumar[2]
[1]*Department of ECE*, National University of Singapore, Singapore
satyendra@u.nus.edu, elebv@nus.edu.sg
[2]*Center for Advancing Electronics Dresden*, Technische Universität Dresden, Germany
{tuan_duy_anh.nguyen1,akash.kumar}@tu-dresden.de

***Abstract*—Dynamic Partial Reconfiguration (DPR) can be used for time-sharing of computing resources within Partially Reconfigurable Regions (PRRs) in FPGA-based systems. The *heterogeneous* partitioning in such systems allows the user to exploit the application-specific mapping of Partially Reconfigurable Modules (PRMs) to PRRs to implement more efficient designs. It offers increased opportunities in optimizing the reliability of the system across multiple layers – from the low-level physical one to the higher application layer. This method, called cross-layer reliability, can potentially exploit the application-specific tolerances to the quality of service (QoS) to tackle the increasing device fault-rates more cost-effectively by distributing the fault-mitigation to different layers. In this work, we propose a QoS-aware cross-layer reliability-integrated design methodology for FPGA-based DPR systems. Specifically, our methodology analyzes the requirements of the applications in terms of *Functional Reliability*, *System Lifetime* and *Makespan* to determine the best possible combinations of reliability-oriented design choices in different layers. We report up to an average of 24% and 30% performance improvements for single and multi-objective optimization-based system partitioning.**

***Index Terms*—Cross-layer Reliability, Dynamic Partial Reconfiguration, Field Programmable Gate Arrays, Embedded Systems**

I. Introduction

The rapid developments in embedded technology have enabled the usage of smart devices in the majority of home and workspace appliances. Such systems are also expected to be flexible enough to adapt to changes in operating environments such as performance and dependability requirements. Reconfigurable systems, specifically FPGAs, along with *Dynamic Partial Reconfiguration* (DPR) [1], have emerged as a key concept to cope with such diverse application requirements. Technology scaling and architectural innovations have been the driving force behind this increasing ubiquity of FPGA-based embedded systems. However, these approaches have also led to an increased Soft Error Rate (SER) and aging-related fault-rates in logic circuits [2, 3].

In contrast to traditional phenomenon-based approach using hardware-only mitigation methods such as uniform Triple Modular Redundancy (TMR), Error Correcting Codes (ECC) etc., the cross-layer reliability (CLR) approach involves distributing fault-mitigation activities across multiple layers of the system stack [4]. Such an approach results in a reduced fault-mitigation effort at hardware layer leading to more cost effective designs. Implementing an optimal CLR-integrated system-design methodology requires efficient design space exploration (DSE) for design decisions related to *selection* and *configuration* of reliability methods across different layers of the system stack.

Further, designing for varied quality of service (QoS) requirements – *Makespan, Functional Reliability* and *System Lifetime* – can pose different and contrasting constraints on the system. The joint optimization of CLR and QoS-aware design can lead to a considerable increase in the design space of DPR-based systems.

In the reconfigurable platforms, DPR allows time-sharing of reconfigurable resources. It helps to improve the available parallelism which can be leveraged to reduce the makespan and/or increase the system lifetime by distributing the wear-out across different Partially Reconfigurable Regions (PRRs) [5]. However, the effectiveness of such an approach depends on the available redundancy in the system which is determined during the system partitioning at design-time. Most of the research into DPR-based system design has been focused on homogeneous PRRs where each PRR can accommodate any Partially Reconfigurable Module (PRM). However, due to the limited reconfigurable resources available in a system, a homogeneous PRR-based approach may not be feasible for every application.

Given the explosion in design space due to CLR, QoS-awareness, and PRR heterogeneity, we propose a novel design methodology for CLR-integrated DPR system design that leverages the application specific tolerances to degradation in one or more QoS metrics. Our contributions are listed below.

Contributions:

- We propose a design methodology for implementing CLR in FPGA-based DPR systems. Specifically, we provide a DSE approach that can incorporate the effect of distributing multiple reliability methods across different layers of the system stack – *Hardware, System Software*, and *Application Software*.
- We propose a methodology for system partitioning in heterogeneous PRR-based systems that integrates CLR implementation. Specifically, we partition the available reconfigurable resources into the application's QoS-aware heterogeneous PRRs.

The rest of the paper is organized as follows. In Section II we provide a background of CLR and DPR and briefly survey related work. We describe the system model and estimation methods for the QoS metrics in Section III. The proposed methodology for DPR-based system design is explained in Section IV. In Section V we discuss the results of the evaluation of the proposed methodology. We conclude the paper in Section VI with a brief summary and discussions on the scope for related future work.

978-1-7281-0214-6/18/$31.00 ©2018 IEEE
DOI 10.1109/FPT.2018.00041

II. Background and Related Work

For all intents and purposes in the current article, we express the QoS requirements of an embedded systems in terms of (1) *functional reliability* – the probability of correctness in computation results, (2) *average makespan* – the average time taken for execution of an application and (3) *system lifetime* – the expected duration of predictable fail-free system operation. Additional metrics, such as power and energy, can be easily integrated into our methodology.

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 QoS-aware system design. As discussed in [6], implementing separate fault tolerance stages at different layers can result in reduced power and area overheads. Further, distributing fault tolerance tasks to higher layers enable the designer to take advantage of the masking effects of more layers [7]. For reconfigurable systems, CLR-based runtime adaptation approaches were proposed in [8, 9]. While in [9], the authors propose a methodology to adapt to changes in the operating environment by switching between spatial and temporal redundancy-based methods, in [8] a reactive approach to detect and mitigate the effect of SEUs is proposed.

B. Dynamic Partial Reconfiguration

DPR offers a method for mitigating permanent faults [5, 10, 11]. Most of the proposed approaches for reliability improvement focus on increasing system lifetime by pro-actively reducing the stress on PRRs by both design-time and runtime methods. In [11], a cross-layer aging-aware placement method for accelerators in FPGA-based runtime reconfigurable architectures is proposed. The described methodology involves module diversification, as proposed in [10], during synthesis and stress-aware placement at runtime to reduce wear-out. Similarly, in [5], the authors use scheduling and system partitioning to improve system lifetime by wear-leveling and providing more redundancy to PRMs that induce faster aging.

Most of the state-of-the-art CLR techniques overlook the application-specific QoS requirements [12–14]. Further, most research works for CLR implementation in FPGA-based systems do not consider multiple QoS metrics and only focus on runtime adaptation. Similarly, the majority of the proactive approaches to system partitioning of DPR-based systems assume homogeneous PRRs. Such an assumption simplifies the PRM to PRR mapping and reduces the complexity of designing application-specific reliability. However, as shown in [5], if the PRMs have a large variation in their resource requirements, this approach can lead to degradation in results. Further, the effect of transient errors on the application and the corresponding effects of using different fault mitigation methods on system lifetime have not been investigated in related research.

III. System Model

A. Architecture model

We use a PR-HMPSoC architecture [15] architecture model similar to that used in related works [5, 16].

B. Application model

We model the application as a task-graph represented by the tuple G_{app} $(T_{app}, E_{app}, P_{app})$, the set of task nodes, the directed connectivity of the nodes representing task dependencies, and the periodicity of the application respectively. Each task $T_t \in T_{app}$ is represented by the tuple $(ID_t, Type_t, Impl_t)$: denoting the task index, the functionality and the set of implementations of the task respectively.

Each i^{th} implementation of T_t, $Impl_{(t,i)} \in Impl_t$, represents a possible choice of PRM for the task. These choices may result due to variations in the algorithm implemented for the task's functionality and the CLR configuration.

TABLE I: Cross-layer Reliability Model

Abstraction Layer	Redundancy Type	Sample Methods	Task-level Performance Metrics of $Impl_{(t,i)}$
Hardware	Spatial	**HWFM**	Resource requirements: $Rsrc_{(t,i)}$ $CLBs_{(t,i)}$, $BRAMs_{(t,i)}$,$DSPs_{(t,i)}$
		Partial TMR, Partial DMR	Minimum execution time: $MinExT_{(t,i)}$
System Software	Temporal	**SSWFM**	Average execution time: $AvgExT_{(t,i)}$
		Scrubbing, Checkpointing,	Probability of error during execution: $ErrProb_{(t,i)}$
Application Software	Information	**ASWFM**	
		Quantization, App-specific methods	Mean time to failure: $MTTF_{(t,i)}$ Average power dissipation:$W_{(t,i)}$

TABLE II: System-level QoS Metrics Estimation

Metric	Estimation Method
Average Makespan ($\mathcal{S}_{app}$)	$\mathcal{S}_{app} = \max_{T_t \in T_{app}} \{\mathcal{S}_{ET_t}\}$ (1)
Lifetime Reliability ($\mathcal{L}_{app}$)	$prrMTTF_r = \frac{P_{app}}{\sum_{T_r} \frac{AvgExecT_{(t,i)}}{MTTF_{(t,i)}}}$ $\mathcal{L}_{app} = MTTF_{sys} = \min_{all\ PRRs} (prrMTTF_r)$ (2)
Functional Reliability ($\mathcal{F}_{app}$)	$\mathcal{F}_t = 1 - ErrProb_{(t,i)},$ $where\ Impl_{(t,i)}\ is\ used\ for\ T_t$ $\mathcal{F}_{app} = \sum_{T_t \in T} \mathcal{F}_t \times \zeta_t$ $where\ \zeta_t = Normalized\ criticality\ of\ T_t$ (3)

C. Cross-layer Reliability Model

For our current work, we consider fault-mitigation methods across three layers – Hardware (**HWFM**), System Software (**SSWFM**) and Application Software (**ASWFM**).

1) Hardware-based Fault Mitigation (HWFM): Improving reliability at the hardware layer usually involves some form of **spatial redundancy**. *Partial Triple Modular Redundancy* for fault-mitigation and *Partial Dual Modular Redundancy* for fault-detection are examples of such methods.

2) System Software-based Fault Mitigation (SSWFM): One of the most common fault-mitigation approaches at the system software level is to utilize the **temporal redundancy** techniques such as *Checkpointing* with rollback/forward recovery [17]. Additionally, *Scrubbing* [18], with corresponding timing and power overheads, can affect the reliability of *SRAM* FPGA-based systems.

3) Application Software-based Fault Mitigation (ASWFM): For the application software layer, we consider methods that are usually application-specific and try to vary the levels of **information redundancy** to obtain different task-level performance metrics [19], as described in TABLE I.

D. System-level QoS Metrics Estimation

For any given execution schedule, the relevant system-level QoS metrics are estimated as shown in TABLE II.

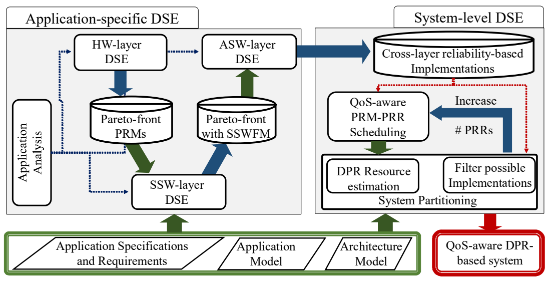


Fig. 1: Overall Design Methodology

IV. QoS-aware Cross-layer Reliability-based DPR System Design Methodology

The overall proposed design methodology is shown in Fig. 1. The design steps are categorized into two phases. The first phase, *Application-specific DSE*, involves task-level exploration to determine the set of Pareto-front design points w.r.t. the multiple task-level performance metrics. Each Pareto point represents a set of cross-layer reliability configuration applied to the implementation choices for the task. In the second phase, System-level DSE, the appropriate partitioning and scheduling strategies of PRMs to PRRs are explored for both heterogeneous (**HetDPR**) and homogeneous (**HomDPR**) DPR system. It also analyses the set of Pareto-front points generated after the Application-specific DSE phase to find the suitable PRMs for each task. The design steps associated with each phase are briefly described below.

A. Application-Specific DSE

1) Application Analysis: This step estimates the tolerable limits to task-level performance metrics.

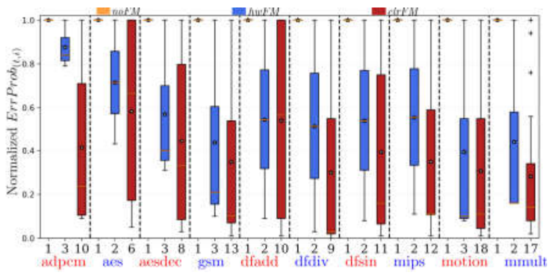
2) Layer-wise Pareto-filtering: This stage involves successive stages of DSE to find the Pareto-point implementations w.r.t. the task-level metrics, shown in TABLE I, for each task type. As shown in Fig. 1, the DSE for each layer involves using the Pareto-point implementations from the *previous* layer's DSE, the results from application analysis and the models of the reliability methods in the current layer as inputs.

B. System-Level DSE

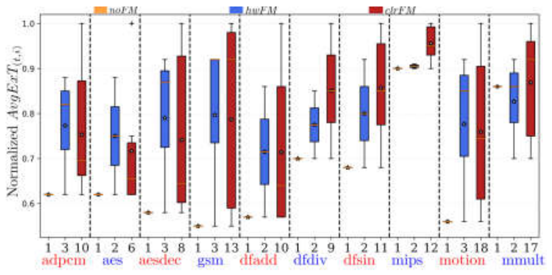
We design a QoS-aware scheduling methodology and add additional constraints for available reconfigurable resources in the system to implement system partitioning.

1) PRM-PRR Scheduling in DPR-based Systems: Optimization of CLR-integrated PRM-PRR scheduling involves executing the *appropriate* PRM with *optimal* CLR configurations on an *appropriate* PRR at the *right* time for each of the tasks in the application. Given the set of implementations for each task, obtained from the application-specific DSE phase, this translates to scheduling the right implementation for each task on the available PRRs. We use *Genetic Algorithms* (GA)-based QoS-aware multi-objective scheduling optimization for both *HetDPR* and *HomDPR* systems.

2) System Partitioning for DPR-based Systems: System partitioning of the DPR-based system involves creating an *optimal* number of PRRs with appropriate resources allocated to each of them. The various implementations for each task, $Impl_{(t,i)}$, obtained from application-specific DSE, may vary in their reconfigurable resource requirements, $Rsrc_{(t,i)}$. For a *HomDPR* system, resource allocation for each PRR must satisfy the resource requirements of every implementation. Hence, PRMs with higher $Rsrc_{(t,i)}$ than that available for creating the specified number of PRRs are filtered out. This may lead to degraded system-level performance. In contrast, for the *HetDPR* systems, the resource allocations for each PRR is allowed to vary from that of other PRRs, and hence all the implementations can be used during the optimization for PRM-PRR scheduling.



(a) Variation of $ErrProb_{(t,i)}$



(b) Variation of $AvgExT_{(t,i)}$

Fig. 2: Distribution of task-level performance metrics of $noFM$, $hwFM$ and $clrFM$ implementation sets for each IP.

V. Experiments and Results

A. Experiment Setup

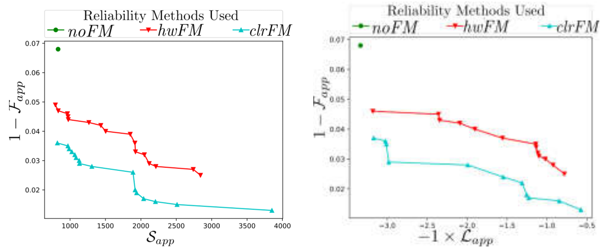
The experiments were run on a computer with two CPUs – Intel^{TM} Xeon^{TM} E5-2609 v2 @ 2.50GHz (each CPU is quad-core) and 32 GB of memory running Ubuntu 18.04 LTS 64-bit. The results are reported for FPGA-based systems implemented on Virtex-6 XC6VLX240T. However, the proposed methodology is generic for all Xilinx FPGAs. The DSE methodologies were implemented in Python using the DEAP [20] package for GA. Probability parameters of 0.7 and 0.1 were used for crossover and mutation respectively. Optimization experiments were conducted for synthetic task-graphs generated using Task Graphs For Free (TGFF) tool [21]. The number of tasks in the task-graphs was varied between 10 to 100 in increments of 10. Additionally, 10 real-world hardware accelerators from the *CHStone* benchmark [22] were used to represent the task type for the constituent tasks. Models of the methods mentioned in Section III-C, based on [17, 23] were used for CLR.

B. Layer-wise Pareto-filtering

The application-specific DSE phase of the proposed methodology involved generating a set of successively *Pareto-filtered* implementation set for each task type, $Type_t$. Three sets of implementations types were used in the experiments – $noFM$, $hwFM$, and $clrFM$ – representing the PRMs with no additional fault-mitigation, only *HWFM*-based and cross-layer fault-mitigation respectively. The relative distribution of the resulting metrics – $ErrProb_{(t,i)}$ and $AvgExT_{(t,i)}$– for the IPs of *CHStone* benchmark used in the experiments are shown in Fig. 2. The values on the horizontal axis represent the number of implementations in the corresponding set. Using CLR resulted in a higher number of Pareto implementations and a larger variation in the task-level metrics.

TABLE III: Optimization problems used for evaluation

Single Objective		Multi-objective	
Symbol	Description	Symbol	Description
$\mathcal{O}_{\mathcal{S}}$	$min\ \mathcal{S}_{app}$	$\mathcal{O}_{\mathcal{L},\mathcal{F}}$	$max\ \mathcal{L}_{app},\ max\ \mathcal{F}_{app}$
$\mathcal{O}_{\mathcal{L}}$	$max\ \mathcal{L}_{app}$	$\mathcal{O}_{\mathcal{S},\mathcal{F}}$	$min\ \mathcal{S}_{app},\ max\ \mathcal{F}_{app}$
		$\mathcal{O}_{All}$	$min\ \mathcal{S}_{app},\ max\ \mathcal{L}_{app},\ max\ \mathcal{F}_{app}$



(a) Pareto front for $\mathcal{O}_{\mathcal{S},\mathcal{F}}$ (b) Pareto front for $\mathcal{O}_{\mathcal{L},\mathcal{F}}$

Fig. 3: Comparing Pareto fronts in an application with 40 tasks

C. Evaluation of System Partitioning Methodology

The different set of implementations obtained from the application-specific DSE phase were used in the system partitioning experiments. We investigated the impact of – (1) Implementing cross-layer reliability (2) PRRs' heterogeneity – on design complexity and resulting performance of the proposed methodology. The experiments for evaluating the proposed QoS-aware system partitioning methodology involved comparing the results across 5 different optimization problems – 2 single-objective and 3 multi-objective – shown in TABLE III.

1) Effect of Cross-layer Reliability: The use of CLR-integrated implementations resulted in an average of up to 13.5%, 29.9% and 34.2% improvements in the hyper-volume in $\mathcal{O}_{\mathcal{S},\mathcal{F}}$, $\mathcal{O}_{\mathcal{L},\mathcal{F}}$ and $\mathcal{O}_{All}$ respectively. Fig. 3 shows the resulting Pareto-fronts for an application with 40 tasks. The $clrFM$ approach results in considerably improved metrics for both objectives in 2-objective optimization.

2) Effect of PRR Heterogeneity: The average performance improvement in the optimization problems using *HetDPR* systems over *HomDPR* ones are shown in TABLE IV. The subscript in single-objective optimization results denotes the corresponding average increase in the number of PRRs in the DPR system of the optimization solution. The experiments were performed for both $hwFM$ and $clrFM$-based optimization. The corresponding results for the 10 applications with an increasing number of tasks for all the optimization problems are shown in Fig. 4. The amount of improvement is positively correlated to the additional number of PRRs (ΔR) in the *HetDPR* system optimization over that in the *HomDPR* systems. These ΔR values for each application are shown in the bar-charts.

VI. Conclusion

CLR provides a cost-efficient approach for tolerating the increasing susceptibility of hardware to physical faults. Similarly, DPR provides an attractive methodology for time-sharing of reconfigurable resources for adapting to varying demands of embedded system applications. A QoS-aware design approach that can leverage both these techniques and exploit the application-specific tolerances to one or more QoS metrics can result in highly efficient system designs. A methodology for efficient joint optimization across all these factors in FPGA-based DPR systems was proposed. We report up to an average of 24% and 30% performance improvements for single and multi-objective optimization-based system partitioning.

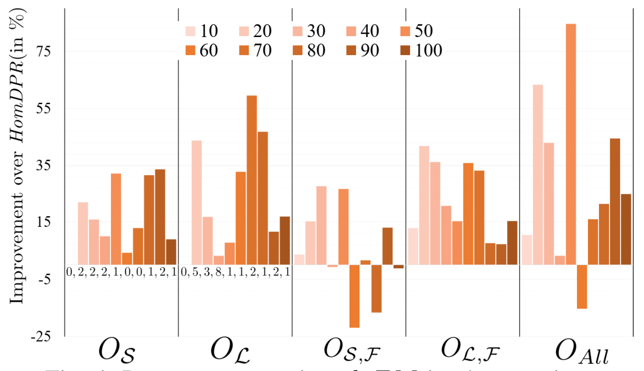


Fig. 4: Improvements using $clrFM$ implementations

TABLE IV: Average Performance improvements with *HetPRR* systems for $hwFM$ and $clrFM$ implementations

Problem Type	Single Objective (in average $\%\Delta q_{\Delta_R}$)		Multi-objective (in average $\%\Delta hyper-volume$)		
Impl Type	$O_{\mathcal{S}}$	$O_{\mathcal{L}}$	$O_{\mathcal{S},\mathcal{F}}$	$O_{\mathcal{L},\mathcal{F}}$	O_{All}
$hwFM$	$19.6_{1.7}$	$43.1_{2.1}$	11.5	35.4	48.0
$clrFM$	$17.1_{1.1}$	$\mathbf{23.9_{2.3}}$	4.7	22.6	**29.6**

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] Xilinx. Xilinx Partial Reconfiguration User Guide, 2013.
[2] A. Geist. Supercomputing's monster in the closet. *IEEE Spectrum*, 53(3):30–35, March 2016.
[3] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23:14–19, 07 2003.
[4] N. P. Carter, et al. Design Techniques for Cross-layer Resilience. In *Proceedings of DATE*, pages 1023–1028, 2010.
[5] S. S. Sahoo, et al. Lifetime-aware design methodology for dynamic partially reconfigurable systems. In *Proceedings of the 23rd ASPDAC*, pages 393–398. IEEE Press, 2018.
[6] S. S. Sahoo, et al. Cross-layer fault-tolerant design of real-time systems. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 63–68, Sept.
[7] T. Santini, et al. Evaluation of failures masking across the software stack. *MEDIAN*, 2015.
[8] J. Jin, et al. An adaptive cross-layer fault recovery solution for reconfigurable SoCs. In *Proceedings of FPT*, 2015.
[9] A. Jacobs, et al. Reconfigurable fault tolerance: A framework for environmentally adaptive fault mitigation in space. In *Proceedings of FPL*, 2009.
[10] H. Zhang, et al. Module diversification: Fault tolerance and aging mitigation for runtime reconfigurable architectures. In *ITC*, 2013.
[11] H. Zhang, et al. Strap: Stress-aware placement for aging mitigation in runtime reconfigurable architectures. In *Proceedings of ICCAD, 2015.*
[12] E. Cheng, et al. CLEAR: Cross-Layer Exploration for Architecting Resilience - Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores. In *Proceedings of the 53rd DAC*, 2016.
[13] J. Henkel, et al. Multi-layer dependability: From microarchitecture to application level. In *DAC*, 2014.
[14] K. Lee, et al. Mitigating the impact of hardware defects on multimedia applications: a cross-layer approach. In *Proceedings of the 16th ACM international conference on Multimedia*, 2008.
[15] T. D. Nguyen et al. PRFloor: An Automatic Floorplanner for Partially Reconfigurable FPGA Systems. *Proceedings of FPGA*, 2016.
[16] T. D. A. Nguyen et al. PR-HMPSoC: A versatile partially reconfigurable heterogeneous Multiprocessor System-on-Chip for dynamic FPGA-based embedded systems. In *Proceedings of FPL*, 2014.
[17] D. Koch, et al. Efficient hardware checkpointing: Concepts, overhead analysis, and implementation. In *Proceedings of FPGA*, 2007.
[18] R. Santos, et al. Scrubbing mechanism for heterogeneous applications in reconfigurable devices. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2017.
[19] S. S. Sahoo, et al. CLRFrame: An analysis framework for designing cross-layer reliability in embedded systems. In *Proceedings of VLSID*, 2018.
[20] F. Fortin, et al. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, July 2012.
[21] R. P. Dick, et al. TGFF: task graphs for free. In *CODES*, pages 97–101. IEEE Computer Society, 1998.
[22] Y. Hara, et al. Proposal and quantitative analysis of the CHStone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing*, 2009.
[23] G. Lee, et al. TLegUp: A TMR Code Generation Tool for SRAM-Based FPGA Applications Using HLS. In *Proceedings of FCCM*, 2017.