A Hybrid Agent-based Design Methodology for Dynamic Cross-layer Reliability in Heterogeneous Embedded Systems

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

Akash Kumar
Technische Universität Dresden, Germany
akash.kumar@tu-dresden.de

ABSTRACT

Technology scaling and architectural innovations have led to increasing ubiquity of embedded systems across applications with widely varying and often constantly changing performance and reliability specifications. However, the increasing physical fault-rates in electronic systems have led to single-layer reliability approaches becoming infeasible for resource-constrained systems. Dynamic Cross-layer reliability (CLR) provides scope for efficient adaptation to such QoS variations and increasing unreliability. We propose a design methodology for enabling QoS-aware CLR-integrated run-time adaptation in heterogeneous MPSoC-based embedded systems. Specifically, we propose a combination of reconfiguration cost-aware optimization at design-time and an agent-based optimization at run-time. We report a reduction of up to 51% and 37% in average reconfiguration cost and average energy consumption respectively over state-of-the-art approaches.

KEYWORDS

Cross-layer Reliability, Run-time Resource Management, Embedded Systems, Reinforcement Learning

1 INTRODUCTION

Modern embedded systems are being used in an ever-increasing number of application areas with widely varying Quality of Service (QoS) requirements. The aggressive transistor scaling, enabling the cheaper design of Heterogeneous Multiprocessor System-on-Chip (HMPSoC)-based systems, has been the major driving force behind such ubiquity. However, the resulting increase in power density, manufacturing defects and Soft Error Rate (SER) [15] have also led to decreased hardware reliability. Traditional single-layer reliability-aware design methodologies, using methods such as uniform Triple Modular Redundancy (TMR), adopt an *other-layer-agnostic* approach and, with the resulting high costs (power/area/timing), are becoming increasingly infeasible for embedded systems with resource constraints.

In contrast, Cross-layer Reliability (CLR)-based design approach involves distributing fault-mitigation activities across multiple layers of the system stack. The joint optimization of reliability methods across multiple layers enables leveraging the implicit fault-masking at different layers [14]. Further, this approach can be used to modulate the CLR configuration to exploit the application-specific tolerances to degradation in one or more QoS metrics. However, to maximize the benefits from such an approach, the system needs to adapt to changes in the operating environment dynamically. As an example, consider the case of satellite-based surveillance where perpetual processing is of paramount importance. The varying battery levels, as a function of the exposure to sunlight and prior processing, can pose a significant challenge to the perpetuity of operation. Therefore, to ensure continuous processing, the system may need to conserve energy at the cost of higher application error rate. The variation in acceptable error rate can also be a function of the terrain under surveillance.

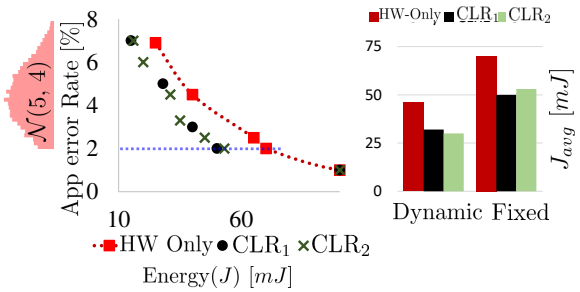


Figure 1: Motivation for Dynamic *CLR*

Figure 1 shows a case-study with Pareto-front of design points showing the trade-off between energy consumption and application error rate. These design points are shown for a system implementing task-mapping with cross-layer reliability (CLR_1 and CLR_2), and the one implementing only hardware reliability measures. As shown in the bar-graphs, the average energy consumption (J_{avg}) in the dynamic CLR systems with run-time adaptation is much lower than a fixed configuration-based approach. The additional reduction of J_{avg} in CLR_2 over CLR_1 is due to the increased number of design points (9 over 6) providing finer granularity for adaptation. The J_{avg} was estimated for a *normal* distribution of the QoS requirement of acceptable application error rate. A worst-case scenario for energy – ensuring lower than 2% error rate at all times – was used for the fixed configuration.

Most of the state-of-the-art methods for dynamic CLR do not consider the optimization across heterogeneous Processing Element (PE)s. The joint optimization across application-specific QoS requirements and CLR configuration results in a further explosion of the already vast design space for *usual* reliability-oriented task-remapping. Hence, a *purely* run-time Design Space Exploration (DSE) approach cannot provide efficient performance with guaranteed QoS. Similarly, purely design-time DSE approaches can result in degraded average performance. To this end, we propose a hybrid design methodology for enabling dynamic CLR in HMPSoC-based embedded systems.

Contributions: Our contributions are listed below:

- We propose a hybrid DSE methodology for CLR-integrated run-time task-remapping. Specifically, we use Genetic Algorithms (GA)-based multi-objective optimization at design-time and an agent-based dynamic adaptation at run-time.
- We propose a reconfiguration cost-aware DSE for enabling a user-modulated trade-off between performance and adaptation cost. The finer granularity of adaptations in CLR_2 than CLR_1 in Figure 1, may result in more reconfigurations. We integrate the cost of such dynamic adaptation in design-time DSE.
- We use an agent-based run-time adaptation to integrate any prior knowledge about the operating conditions. Specifically, a reinforcement learning-based approach is used to estimate the average performance of the stored design points. This enables the system to adapt to configurations with *optimal* average performance rather than the *best* instant improvement.

DAC '19, June2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317746

Table 1: Comparing Related Works

Related Work	Dynamic QoS-aware	Heterogeneous Multiprocessor	Multi-objective	Reconfiguration cost-aware
Rehman et al. [11]	✗	✗	✓	✗
Henkel et al. [7]	✗	✓	✗	✗
Cheng et al.[3]	✗	✗	✓	✗
Glaß et al. [6]	✗	✓	✓	✗
proposed	✓	✓	✓	✓

The rest of the paper is organized as follows – the relevant background and related research work are briefly discussed in Section 2. In Section 3, we specify the system model used for analysis and optimization. The proposed methodology is explained in detail in Section 4. The results from the experimental evaluation of the proposed methodology are discussed in Section 5. Finally, we conclude the paper in Section 6 the paper summary and discussions on the scope of related future research.

2 BACKGROUND AND RELATED WORK

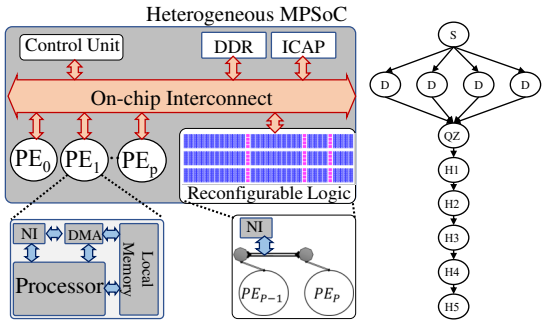
The variation in QoS requirements is not limited to just across different applications. The same embedded system may be expected to operate under different QoS requirements as well. Typical scenarios include – processing in minimal energy mode during *low-battery*, high throughput to guarantee quality under increased packet arrival rate and higher precision for some financial tasks. In the current article, we express the QoS requirements of an embedded system in terms of (1) *functional reliability* – the probability of correctness in computation results, and (2) *average makespan* – the average time taken for execution of the application. In addition to guarantees on these two QoS metrics, we express the performance in terms of the *average energy consumption*.

2.1 Cross-layer Reliability

Traditional single-layer reliability approaches focus on mitigating all physical faults at the hardware layer. A phenomenon-based approach is usually used, i.e., each fault mechanism (Negative-bias Temperature Instability (NBTI), Electro-migration (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. In contrast, 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 to the hardware layer, an appropriate combination of methods that meets the design goals and constraints can be implemented. As discussed in [8] and [12], implementing separate fault tolerance stages at different layers can result in reduced power and area overheads.

2.2 Run-time Task-remapping with CLR

Task-mapping of application tasks on MPSoCs involves assignment and ordering of the tasks and their communications on the platform resources given some optimization criteria. A survey of various task-mapping methodologies, both *design-time* and *run-time*, based on different optimization goals is presented in [16]. In [2], the authors outline a methodology for implementing dynamic 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-tolerance technique at the appropriate layer. However, the authors do not provide any information about the optimization of run-time adaptation to various operating scenarios. In [11], the authors provide a detailed methodology for hybrid CLR-based run-time adaptation for multiple objectives. However, the methodology is limited to single processor systems with fixed task-schedules. Similarly, the



(a) Architecture model **(b) Application model (*JPEG Encoder*)**

Figure 2: System model

methods proposed in [3, 6] are primarily suited for system synthesis at design-stage and are orthogonal to our proposed approach. In [7], the authors propose various cross-layer techniques – from *micro-architecture* to *application* level – for both general purpose processor-based and reconfigurable processor-based embedded systems. The dynamic task-remapping method outlined in [7] is primarily aimed at reducing the aging-related stress pro-actively, and reacting to permanent faults. However, the authors do not consider QoS-aware optimization and run-time adaptation costs.

A comparison of our proposed methodology with related research works is shown in Table 1. Most proposed approaches do not consider changing QoS requirements and the effect of run-time reconfiguration costs during design-time optimization in heterogeneous systems. To this end, we propose a novel method for integrating the consideration of the run-time reconfiguration costs during design-time optimization to provide more efficient design points for dynamic CLR.

3 SYSTEM MODEL

3.1 Architecture Model

For the architecture model, we assume an HMPSoC with a distributed shared memory architecture, similar to the one shown in Figure 2a, with centralized control of task-remapping and CLR implementation, and containing P processing elements (PEs). Each of the PEs, PE_p is characterized by the tuple: $(ID_p,\ PEType_p)$: the PE's index and type. We denote the heterogeneity among PEs by the term $PEType_p$ to represent the combinations of one or more of the factors – (1) the type of processor, such as general purpose embedded processors or accelerator on reconfigurable logic (2) aging-related fault profile of the PE (characterized by β_p) and (3) soft-error masking factor for the PE such as the Architectural Vulnerability Factor (AVF) [9].

3.2 Application Model

We model the application as a task-graph G_{app}, represented by a tuple $(T_{app}, E_{app}, P_{app})$, the set of task nodes, the directed connectivity of the nodes representing task dependencies, and the periodicity of the application respectively. Figure 2b shows a sample task-graph for an application with 11 tasks and 13 edges. Each edge, $E_e \in E_{app}$, is characterized by the tuple $(ID_e, Src_e, Dst_e, CommT_e)$: the edge index, source and sink task nodes, and the data transfer time. Similarly, each task, $T_t \in T_{app}$, in the task-graph is characterized by the tuple $(ID_t, Type_t, Impl_t)$: the task index, task type (functionality) and the set of implementations for the task. Each i^{th} implementation of T_t, $Impl_{(t,i)} \in Impl_t$, is characterized by the following: (1) the type of PE, (2) system software – *bare-metal* system

Table 2: CLR Model and Task-level performance metrics

Abstraction Layer	Redundancy Type	Sample Methods	Task-level Performance Metrics of $Impl_{(t,i)}$
Hardware	Spatial	**HWRel**: Partial TMR, Circuit Hardening	Scale parameter (stress indicator): $\eta_{(t,i)}$ Minimum execution time: $\mathbf{MinExT}_{(t,i)}$ Average execution time: $\mathbf{AvgExT}_{(t,i)}$ Probability of error during execution: $\mathbf{ErrProb}_{(t,i)}$ Mean time to failure: $\mathbf{MTTF}_{(t,i)}$ Average power: $\mathbf{W}_{(t,i)}$
System Software	Temporal	**SSWRel**: Retry, Checkpointing	
Application Software	Information	**ASWRel**: Code Tripling, Hamming Correction Checksum [10]	

Table 3: System-level QoS and performance Estimation

Metric	Estimation Method
Average Makespan ($\mathcal{S}_{app}$)	$\mathcal{S}_{app} = \max_{T_t \in T_{app}} \{\mathcal{S}_{ET_t}\}$ (1)
Functional Reliability ($\mathcal{F}_{app}$)	$\mathcal{F}_t = 1 - ErrProb_{(t,i)}$, where $Impl_{(t,i)}$ is used for T_t $\mathcal{F}_{app} = \sum_{T_t \in T} \mathcal{F}_t \times \zeta_t$ (2) where ζ_t = Normalized criticality of T_t
Power ($\mathcal{W}_{app}$), **Energy** ($\mathcal{J}_{app}$)	$\mathcal{W}_{app} = \max_{x \in (0, \mathcal{S}_{app}]} \sum_{T_t \in T_{app}} \mathcal{I}(x) \times W_t$ where, $\mathcal{I}(x) = \begin{cases} 1, & if\ x \in (\mathcal{S}_{ST_t}, \mathcal{S}_{ET_t}] \\ 0, & else \end{cases}$ (3) $J_{app} = \sum_{T_t \in T_{app}} AvgExT_t \times W_t$

or some operating system and (3) application software – algorithms and programming languages.

3.3 Cross-layer Reliability Model

For our current work, we consider fault-mitigation methods across three layers – Hardware (**HWRel**), System Software (**SSWRel**) and Application Software (**ASWRel**). Varying the selection and configuration of reliability methods for each layer leads to varying performance of the tasks' implementations. We follow the framework proposed in [13] to determine the task-level performance metrics. These metrics, as described in Table 2, are used for system-level analysis and design. The scale parameter $\eta_{(t,i)}$ is a function of the thermal profile of executing $Impl_{(t,i)}$, and can be used in estimating the system's lifetime. Similarly, the $ErrProb_{(t,i)}$ is a function of the $Impl_{(t,i)}$, the CLR configuration and the masking factor of the PE.

3.4 System-level QoS Metrics Estimation

CLR-integrated task scheduling involves executing any implementation, $Impl_{(t,i)}$, with any CLR configuration, say C_t, for every task, $T_t \in T_{app}$, on any of the available PEs of the hardware platform in some order. With the resulting execution schedule, the relevant system-level QoS and performance metrics are estimated as shown in Table 3. $\mathcal{S}_{ST_t}$ and $\mathcal{S}_{ET_t}$ refer to the average start and end execution time respectively for task T_t. We use a task criticality-based method for functional reliability estimation.

3.5 Reconfiguration Model

Dynamic CLR-based adaptation in HMPSoCs can involve any of the following modes – (1) re-ordering execution of tasks mapped on each PE, (2) changing the CLR configuration for the reliability methods for each layer, (3) changing the implementation used for a task (4) varying the task-to-PE binding for some tasks. We assume each PE has fixed amount of local memory to store the binaries for the tasks mapped on it. Therefore, the first two methods do not incur any reconfiguration cost due to task-migration. The latter two methods, however, involve copying of tasks' implementation binaries to appropriate PEs. Further, using different accelerators to the partially reconfigurable regions (PRRs) in the reconfigurable area involves changing the bit-stream mapped for the PRRs. We use the term d_{RC} to denote the reconfiguration cost between two CLR-integrated task-mapping configurations. The average reconfiguration cost can affect the system availability, communication energy of transferring the binaries and bit-streams and the reliability of on-chip interconnect. We use the reduction of average run-time adaptation costs as a generic method of improving such related metrics.

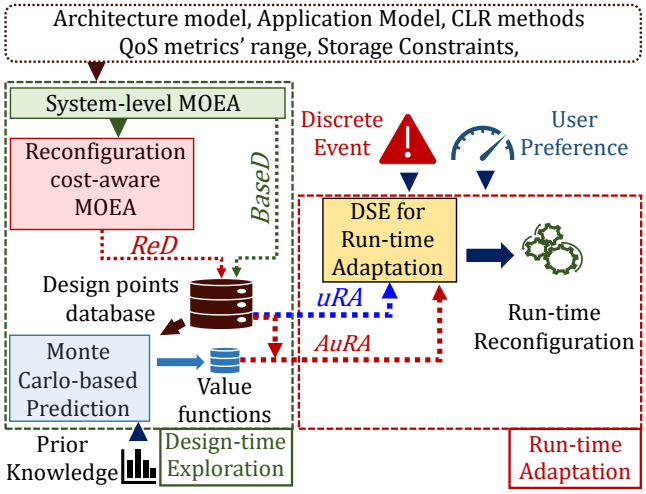


Figure 3: Hybrid Run-time Adaptation Methodology

4 HYBRID DSE METHODOLOGY

Dynamic CLR is used for adapting to some change in the system's operating scenario. The change could be internal: for example a permanent fault to one of the PEs resulting in reduced resource availability – or external: a change in QoS requirements or Single Event Upset (SEU) rate, λ_{SEU}. For our current work, we consider the case of run-time adaptation to varying QoS requirements at constant resource availability and λ_{SEU} as the working scenario. Variations in other factors can be considered as separate instances of this scenario with different values for λ_{SEU}, and the number of available PEs. As shown in Figure 3, the proposed hybrid methodology broadly consists of two stages – (1) During design/compile-time, a set of CLR configurations that form the Pareto-frontier w.r.t. the QoS and performance metrics are generated and (2) These set of points, along with a set of supporting data, are used during run-time to find the appropriate *next* configuration. The details of proposed methods during each stage are explained below.

4.1 Problem Statement

$$\underset{X_i \in X_{app}}{maximize}\ \mathcal{R}(X_i)\ where,\ X_{app} = \prod_{\forall T_t \in T_{app}} \Psi_t$$
$$\mathcal{R}(X_i) = -1 \times \mathcal{J}_{app}(X_i)$$
$$i, j \in \mathbb{R}\ ,\ \mathcal{S}_{app}(X_i) \leq \mathcal{S}_{SPEC}\ and\ \mathcal{F}_{app}(X_i) \geq \mathcal{F}_{SPEC} \quad (4)$$
$$\Psi_t = \begin{cases} \mathcal{M}_t, & for\ task - mapping\ only \\ C_t, & for\ CLR - Implementation\ only \\ \mathcal{M}_t \times C_t, & for\ CLR - Integrated\ task - mapping \end{cases}$$

We consider the maximization reduction in energy consumption under constrained maximum average makespan and application error rate as a representative example of a typical dynamic CLR problem. The set of all possible cross-layer reliability configurations for each task can be represented by:

$\mathcal{C}_t = \mathcal{HWR}el_t \times \mathcal{SSWR}el_t \times \mathcal{ASWR}el_t\ ,\ \forall T_t \in T_{app}$

It denotes the Cartesian product of the combinations of masking, detection and tolerance methods for each layer. Similarly, the possible task-to-PE binding and scheduling options the tasks can be represented by the set:

$\mathcal{M}_t = \mathcal{P}_t \times I_t \times Q_t\ ,\ \forall T_t \in T_{app}$,

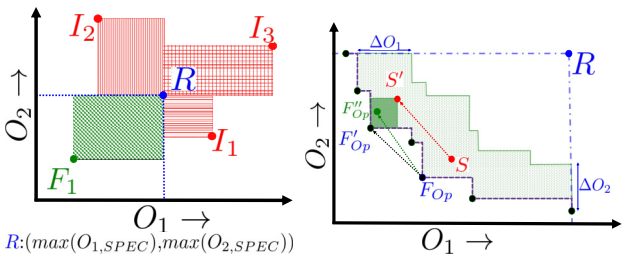
the product of the set of PEs that the task T_t can be executed on, the set of possible implementation choices for task T_t, and the set of possible *positions* of the task in the execution schedule respectively. Hence, for any arbitrary CLR-integrated task-mapping configuration of G_{app}, X_i, the objective function for the run-time optimization problem can be expressed as shown in Eq. (4). We use the term $\mathcal{R}(X_i)$ to denote the performance of a any arbitrary task mapping X_i. In our current work, we use the reduced energy consumption to signify improved performance. Other metrics such as Mean Time to Failure (MTTF) can be added to $\mathcal{R}(X_i)$ for optimization of *system lifetime.*

4.2 Design/Compile-time DSE

$$\begin{aligned} &\max_{p_i \subset \mathcal{P}_{app}} \mathcal{V}(p_i)\ where,\ \mathcal{P}_{app} = \{X_i\}, \forall X_i \in X_{app} \\ &\mathcal{V}(p_i) = \sum_{\forall X_i \in p_i} v(\mathcal{R}(X_i), \mathcal{F}_{app}(X_i), \mathcal{S}_{app}(X_i)) \\ &s.t.\ \forall X_r \in p_i, \mathcal{S}_{app}(X_r) \leq max(\mathcal{S}_{SPEC}), \\ &\ and\ \mathcal{F}_{app}(X_i) \leq min(\mathcal{F}_{SPEC}) \end{aligned} \tag{5}$$

To integrate the effect of varying QoS requirements of $\mathcal{S}_{app}$ and $\mathcal{F}_{app}$, Eq. (4) can be converted into a multi-objective optimization problem as shown in Eq. (5). It involves finding an optimal set of task-mapping design points which are not *dominated* by any other design point w.r.t. $\mathcal{S}_{app}$, $\mathcal{F}_{app}$ and $\mathcal{R}(X_i)$. The solution technique involves maximizing the sum of the hyper-volume, $\mathcal{V}(p_i)$, of all the non-dominated points in the collection p_i. Figure 4a shows an example of hyper-volumes of design points in the minimization of objectives O_1 and O_2. The reference point R denotes the constraints (maximum) for both the objectives. The hyper-volume of the feasible design point, F_1, represents it's *fitness* and is equal to the (green) area *swept* by the point relative to R. Similarly, the fitness of infeasible design points is represented by the negative of the (red) areas between R and those points.

4.2.1 Run-time Reconfiguration cost-aware DSE (ReD). In order to integrate the effect of run-time reconfiguration cost into the design-time optimization problem, we introduce an additional optimization stage that considers the reconfiguration distance between two arbitrary task-mapping configurations – d_{RC}. For each of the design points in the solution set of Eq. (5), we use the point as initial seeding to generate a set of points that are within some tolerance limit w.r.t the degradation of that point's QoS metrics and $\mathcal{R}(X_i)$. The new design point's average d_{RC} from the optimal set of design points is used as an additional objective in the new multi-objective optimization problem. Consider the typical scenario shown in Figure 4b, where the QoS requirements change from the point S to S'. Using the Pareto design points solely for reconfiguration would result in changing the operating point from F_{Op} to F'_{Op}. However, there might be some non-dominant point, F''_{Op}, that meets the QoS constraints of S' and incurs lesser task-migration and related reconfiguration costs. The additional optimization step aims at finding



(a) Hypervolume maximization **(b) Reconfiguration cost-aware DSE**

Figure 4: Design-time Exploration

Algorithm 1 User-modulated Run-time Adaptation (*uRA*)

Require: Stored design points:p_i ; User parameter:p_{RC}

1: **loop**
2: **if** discrete event: **then**
3: $FEAS \leftarrow Feasible\ design\ points\ in\ p_i$
4: **for all** $p \in FEAS$ **do**
5: {Estimate reconfiguration cost}
6: $d_{RC}(p) \leftarrow d_{RC}\ from\ current\ configuration$
7: {Estimate performance $(-1 \times \mathcal{J}_{app}(X_i)$}
8: $\mathcal{R}(p) \leftarrow \mathcal{R}(X_i)\ of\ p$
9: $RET(p) \leftarrow p_{RC} \times norm(\mathcal{R}(p)) - (1 - p_{RC}) \times norm(d_{RC}(p))$
10: **end for**
11: Select $\underset{p \in FEAS}{\text{argmax}}\ RET(p)$ for re-configuration.
12: **end if**
13: **end loop**

such design points for enabling efficient reconfiguration-aware run-time DSE.

4.3 Run-time Adaptation

The run-time decision making involves changing the operating point to a configuration that – (1) satisfies the new QoS requirements and (2) provides a user-defined balance between performance ($\mathcal{R}(X_i)$) and average reconfiguration cost.

4.3.1 User-modulated Run-time Adaptation (uRA). The proposed run-time DSE Algorithm is shown in Algorithm 1. The first DSE step involves filtering the feasible design points based on the new $\mathcal{S}_{SPEC}$ and $\mathcal{F}_{SPEC}$ constraints. The filtered set of points are evaluated based on the weighted costs of normalized d_{RC} from the current configuration and the normalized performance degradation of $\mathcal{R}$ (Eq. (4)). A parameter p_{RC} is used to enable user-specific importance to quality metric improvement and reconfiguration cost. The system is reconfigured to the lowest cost design point.

4.3.2 Agent-based uRA (AuRA). The approach in *uRA* is *myopic* and does not consider the uncertainties w.r.t. QoS requirements variations and other changes in the operating scenario. Further, it does not provide any way of integrating prior knowledge about the operating environment. To overcome these limitations, we use a reinforcement learning (RL) agent-based run-time adaptation method – *AuRA*. Due to space constraints, we list only the major features of the approach below.

- **State space:** Each of the stored design points is treated as a single state.
- **Policy:** We use a fixed policy, similar to that in *uRA* . However, the next state evaluation (lines 5-9 in Algorithm 1) is based on the *value functions* of the feasible states instead of their $\mathcal{R}(p)$ and $d_{RC}(p)$ values. It can be noted that the *uRA* method is subsumed into *AuRA* by setting the discount factor $\gamma = 0$ during policy evaluation.

Table 4: Percentage reduction in task-migration cost using using *ReD* over *BaseD* [11] for a CSP w.r.t. the QoS metrics

Number of Tasks	10	20	30	40	50	60	70	80	90	100
% Reduction over [11]	23	34	47	37	28	49	39	27	36	**56**

- **Value optimization**: With the fixed policy we use the returns from each *episode* (typically a thousand number of application execution cycles) to update the states' value functions.
- **Prior knowledge:** In the purely *online* approach the agent starts with uniform value functions for each state and learns about the operating environment from experience. However, to incorporate prior information about the distribution of QoS requirement variations we use a Monte Carlo simulation with the fixed policy for estimating the initial value functions of each state.

5 EXPERIMENTS AND RESULTS

5.1 Experiment Setup

The experiments were performed 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. Experiments were conducted for synthetic applications with the number of tasks varying from 10 to 100. The task-graphs and tasks' execution times for the synthetic applications were generated using Task Graphs For Free (TGFF) tool [4]. All the applications were mapped to an HMPSoC with 5 PEs of 3 different types that vary in masking factor. Additionally, 3 partially reconfigurable regions (PRRs) were used to execute accelerators for the tasks. The optimization methods were implemented in Python using the DEAP [5] and PYGMO [1] packages for GA. Probability parameters of 0.7 and 0.03 were used for crossover and mutation respectively. Tournament selection with 5 individuals was used for evolving the new generation. The experimental evaluation of the proposed DSE approaches involved estimating the effects of using different reliability methods at multiple layers. Models for the methods mentioned in Table 2 were used for the different layers. Bivariate Gaussian and exponential distributions, with a rate of 100 cycles, were used in the Monte Carlo simulation of run-time DSE for emulating changes in QoS specification and the time between discrete events respectively.

5.2 Evaluation of Reconfiguration cost-aware Dynamic Adaptation

Evaluation of the reconfiguration cost-aware optimization problem involved Monte Carlo simulations of the run-time DSE with two different stored database of design points, for a million application execution cycles. The first database contained a *purely* performance oriented set of only the Pareto-front design points (*BaseD*). This approach is similar to the ones used in [11] for hybrid task-remapping. The second database (*ReD*) contained additional non-dominant design points obtained from the additional multi-objective optimization problem explained in Section 4.2.1.

For a graphical explanation of the improvement in results, we use the results from solving a constraint satisfaction problem (CSP) for ensuring QoS w.r.t. average makespan and functional reliability. This is achieved by setting $\mathcal{R}(\mathcal{X}_i) = 0$ in Eq. (5). Figure 5 shows the design points used for dynamic adaptation in the application with 80 tasks. The additional non-dominant design points are marked with a '⁎'. As shown in Table 5, a reduction in average reconfiguration cost of up to 56% was observed using the additional design points. The task-migration costs incurred as a reaction to the first 50 instances of QoS variation, obtained from a section of the Monte Carlo simulation, are shown in Figure 6. The two sets of traces in the figure correspond to the results from using the two different databases for the application with 80 tasks. The higher number of

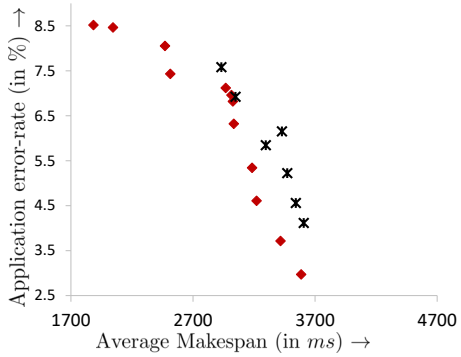


Figure 5: Pareto front and additional points from reconfiguration cost-aware optimization

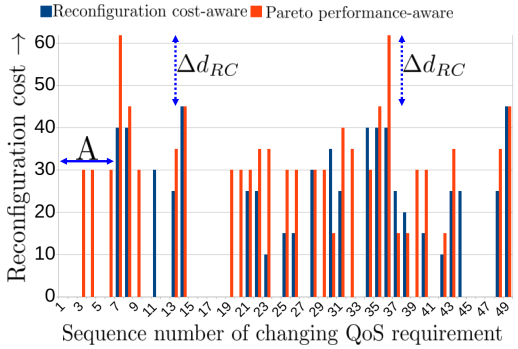


Figure 6: Comparing reconfiguration cost trace for a sequence of 50 QoS requirement changes

Table 5: Percentage increase in energy consumption and reduction in dynamic adaptation costs with reconfiguration-cost minimization on a single set of design points

Number of Tasks	10	20	30	40	50	60	70	80	90	100
% Reduction in Average Reconfiguration cost	38	45	28	8	**51**	44	30	49	43	39
% Increase in Average Energy Consumption	10	13	4	0	4	1	0	2	2	2

reconfigurations in this window, 31 compared to 24, for the Pareto performance-oriented approach – can be attributed to the search for the best hyper-volume design point for every change in QoS requirements. However, in the case of the other approach, the run-time adaptation is performed only during a violation of the QoS requirements. This is demonstrated clearly in the first few samples in Figure 6. While the reconfiguration cost-aware approach does not trigger any adaptations, the other approach results in continuous adaptations in the region **A** shown in the figure. Further, the maximum reconfiguration cost incurred with the set of Pareto design points, *BaseD*, shown as Δd_{RC}, is considerably higher than that using the alternative approach (*ReD*). All these factors contribute to the increased run-time reconfiguration cost in *BaseD*.

5.3 Evaluation of DSE methods

Using a reconfiguration cost minimization approach can result in reduced performance for a problem with additional objectives (similar to Eq. (5)). Table 5 shows the increase in average energy consumption in different tasks with solutions that minimize the average reconfiguration cost. These results are from simulations on a single

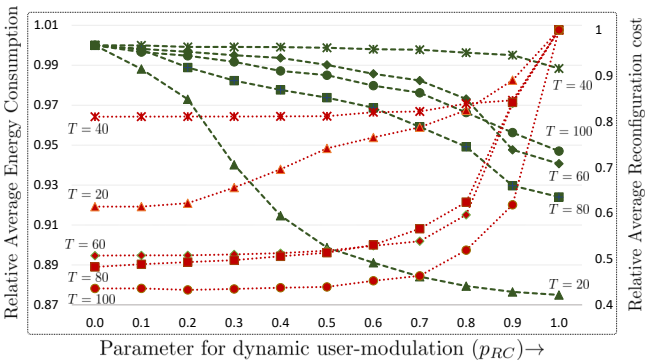


Figure 7: Relative variation of average energy (*in green*) and reconfiguration cost (*in red*) with p_{RC}

Table 6: Percentage improvements using *ReD* compared to *Based* with relevant values of p_{RC}

Number of Tasks	10	20	30	40	50	60	70	80	90	100
% Reduction in Average Reconfiguration cost (with $p_{RC} = 0$)	19.6	26.0	4.6	0.2	0.2	0.1	4.0	9.0	7.3	1.7
% Reduction in Average Energy Consumption (with $p_{RC} = 1$)	**36.8**	27.5	0.0	0.0	0.8	0.0	3.9	3.5	0.0	0.0

set of design points. The use of user modulation parameter p_{RC} enables the user to switch dynamically between varying priority modes. The effect of using p_{RC} is shown in Figure 7 for 5 applications with a varying number of tasks. The parameter p_{RC} can be used to tune between the two extremes – run-time reconfiguration cost minimization and performance maximization. Here, performance refers to the improvement in energy reduction. Figure 7 shows the variation of average reconfiguration cost (in red) and average energy consumption (in green) at different values of p_{RC}. For each application, maximum adaptation costs were observed for $p_{RC} = 1$, which also corresponds to the best gain in $\mathcal{R}$. The run-time adaptation costs do not decrease continuously as only a limited number of non-dominant points – the ones primarily responsible for the reduced costs – are present in the solution set.

Table 6 shows the percentage improvements with *ReD* over *BaseD* using appropriate values of p_{RC} for energy ($p_{RC} = 1$) and reconfiguration cost minimization ($p_{RC} = 0$) for different application sizes. The additional non-dominant points in *ReD* result in reduced average reconfiguration cost of up to 26% (average 7.3%). Similarly, the average energy consumption is also reduced by up to 37% (average 7.3%).

Similar comparison of the performance of *AuRA* over that of *uRA* are presented in Table 7. The use of prior knowledge about variations in QoS requirements with off-line Monte Carlo simulation-based *value function optimization* resulted in improved metrics for most of the cases. However, in some cases the value functions did not converge due to a large number of design points, resulting in slightly reduced performance.

6 CONCLUSION

With increasing susceptibility of hardware to physical faults and the increasing variability in the operating scenarios of embedded systems, cross-layer reliability-integrated run-time adaptation is a growing necessity. In order to ensure the development of low-cost embedded systems, such an approach is necessary to enable the usage of commercial off-the-shelf component-based systems in a wider variety of applications. However, the joint optimization of

Table 7: Percentage improvements using *AuRA* compared to *uRA* with relevant values of p_{RC}

Number of Tasks	10	20	30	40	50	60	70	80	90	100
% Reduction in Average Reconfiguration cost (with $p_{RC} = 0$)	-6.9	49.5	3.3	20.9	58.5	25.7	23.9	-1.2	0.6	7.2
% Reduction in Average Energy Consumption (with $p_{RC} = 1$)	1.2	7.0	-2.5	2.6	1.6	-1.0	-0.1	0.5	3.2	3.0

task-mapping, scheduling, QoS-awareness and cross-layer configuration during run-time form a major impediment to CLR-integrated design. In this paper, a hybrid methodology for enabling CLR-integrated runtime-adaptation is proposed. An user-modulation enabled method for dynamically assigning priority to reconfiguration cost and performance improvements was used for run-time DSE. Further, a technique for integrating the effect of run-time reconfiguration overheads into the design time optimization problem is evaluated. With our proposed methods, up to 51% reduction in average reconfiguration cost and 37% reduction in average energy consumption was observed. The frequently used hybrid approach of storing multiple design points for each possible operating scenario can lead to inadequate storage and longer run-time DSE. Future research would involve using cross-layer reliability to alleviate such issues in hybrid run-time adaptation.

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] F. Biscani and D. Izzo. 2018. esa/pagmo2: pagmo 2.9. https://doi.org/10.5281/zenodo.1406840
[2] N. P Carter, H. Naeimi, and D. S Gardner. 2010. Design techniques for cross-layer resilience. In *DATE*.
[3] E. Cheng, S. Mirkhani, L. G. Szafaryn, C. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra. 2016. CLEAR: Cross-Layer Exploration for Architecting Resilience - Combining Hardware and Software Techniques to Tolerate Soft Errors in Processor Cores. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*.
[4] R. P. Dick, D. L. Rhodes, and W. Wolf. 1998. TGFF: Task Graphs For Free. In *CODES*. IEEE Computer Society, 97–101.
[5] F. Fortin, F. De Rainville, M. Gardner, M. Parizeau, and C. Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* (July 2012).
[6] M. Glaß, H. Yu, F. Reimann, and J. Teich. 2012. Cross-Level compositional reliability analysis for embedded systems. *Computer Safety, Reliability, and Security* (2012), 111–124.
[7] J. Henkel, L. Bauer, H. Zhang, S. Rehman, and M. Shafique. 2014. Multi-Layer Dependability: From Microarchitecture to Application Level. In *DAC*.
[8] K. Lee, A. Shrivastava, M. Kim, N. Dutt, and N. Venkatasubramanian. 2008. Mitigating the impact of hardware defects on multimedia applications: a cross-layer approach. In *Proceedings of the 16th ACM international conference on Multimedia*.
[9] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. 2003. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[10] Michael Nicolaidis. 2010. *Soft Errors in Modern Electronic Systems* (1st ed.). Springer Publishing Company, Incorporated.
[11] S. Rehman, K. Chen, F. Kriebel, A. Toma, M. Shafique, J. Chen, and J. Henkel. 2016. Cross-Layer Software Dependability on Unreliable Hardware. *IEEE Trans. Comput.* 65 (Jan 2016).
[12] S. S. Sahoo, B. Veeravalli, and A. Kumar. 2016. Cross-layer fault-tolerant design of real-time systems. In *DFTS*.
[13] S. S. Sahoo, B. Veeravalli, and A. Kumar. 2018. CLRFrame: An Analysis Framework for Designing Cross-Layer Reliability in Embedded Systems. In *VLSID*.
[14] T. Santini, P. Rech, A. Sartor, U. B Corrêa, L. Carro, and F. R Wagner. 2015. Evaluation of Failures Masking Across the Software Stack. *MEDIAN* (2015).
[15] P. Shivakumar, M. Kistler, S. W Keckler, D. Burger, and L. Alvisi. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks,*.
[16] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. 2013. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*. ACM.