Using *Monte Carlo Tree Search* for EDA – A Case-study with Designing Cross-layer Reliability for Heterogeneous Embedded Systems

Siva Satyendra Sahoo, Akash Kumar
Chair of Processor Design, Center for Advancing Electronics Dresden (CfAED)
Technische Universität Dresden
Dresden, Germany
{siva_satyendra.sahoo,akash.kumar}@*tu-dresden.de*

***Abstract*—Continued transistor scaling and increasing power density have led to considerable increase in fault-rates in silicon nanotechnology-based real-time systems. Cross-layer fault tolerance techniques present a more cost-efficient methodology for adapting to such increased fault rates by distributing fault-tolerance to different layers. To this end, we propose a methodology for integrating the design space exploration (DSE) for task-mapping on heterogeneous hardware-platforms with designing cross-layer reliability. Specifically, we model the DSE for task-mapping with cross-layer reliability as a tree search problem and use Monte Carlo Tree Search for task-mapping and scheduling applications with specific reliability requirements. The proposed methodology results in considerable improvements over a *standalone* approach to task-mapping and implementing cross-layer reliability.**

***Index Terms*—Cross-layer System Design, Reliability, Randomized Algorithms, Design Space Exploration, System-level Design, Embedded Systems**

I. Introduction

Technology scaling and architectural innovations have been the driving force behind the increasing ubiquity of embedded systems. However, these approaches have also led to significant increase in Soft Error Rate (SER) of logic circuits [1], [2] in addition to reliability issues due to increased power density and manufacturing defects [3]. Traditional approaches to fault-aware design adopt a phenomenon-based approach that focuses on mitigating all physical faults at the hardware layer. However, hardware-only mitigation methods are becoming increasingly infeasible due to the increasing complexity, higher SER and tighter cost (power and area) constraints of embedded systems.

In contrast, the cross-layer design approach involves distributing fault-mitigation activities to several layers of the system stack [4] resulting in a reduced fault-mitigation effort at hardware layer leading to more cost-effective designs. Further, barring a few applications that require high reliability in all three reliability metrics – *Functional, Timing* and *Lifetime* – most applications, especially soft real-time systems, can tolerate some degradation in one or all of those metrics [5]. The design decisions w.r.t. selecting and configuring fault-mitigation methods at different layers in heterogeneous MPSoCs add to the already vast design space for task-mapping and scheduling. Therefore, *efficient* DSE techniques that can incorporate effects of distributing fault-tolerance activities to different layers, and leverage the application-specific tolerances in reliability metrics can enable the design of effective cross-layer reliability. To this end, we propose a novel system-level DSE methodology for implementing application-specific cross-layer reliability in embedded systems.

Contributions: Our contributions are listed below:

1) We model DSE for cross-layer reliability as a *tree traversal problem* (TTP) and use the *Monte Carlo Tree Search* (MCTS) method for determining the best selection of fault-mitigation activities for each layer. We use task-mapping on heterogeneous MPSoCs as a test case to evaluate the performance of the proposed DSE method. The proposed methodology considers application-specific requirements for each reliability metric.
2) We explore modifications to the *pure*, *problem-agnostic* MCTS method to adapt the method to the cross-layer DSE problem. Specifically, we investigate the implications of a *constrained* MCTS approach on the performance of the DSE methodology.

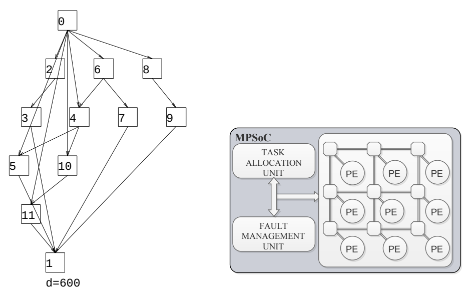
The rest of the paper is organized as follows: In Section II, we provide a brief background of cross-layer reliability and survey some state-of-the-art approaches to designing cross-layer reliability. In Section III, we describe the system model. We explain the representation of DSE for cross-layer fault-mitigation as a TTP and detail our proposed MCTS-based DSE approach in Section IV. The experiments and results for evaluation of the proposed approach are discussed in Section V. We conclude the paper in Section VI with the scope for related future work.

II. Cross-Layer Reliability and Related Work

Unlike the single-layer phenomenon-based design approach, the cross-layer approach provides a more application-specific and cost-efficient method for reliability-aware system design. Since the fault-mitigation activities are not limited the hardware layer, an appropriate combination of methods that meets the design goals and constraints can be implemented. As discussed in [6] and [5], 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].

In [4], the authors outline a methodology for implementing cross-layer resilience. Additional subsystems – *Error handler routine*, *Resource Map*, *Hardware Configuration Routine*, and *Task Scheduler* – in the operating system are used to trigger the appropriate fault-tolerance technique at the appropriate layer. In [8], the authors proposed new techniques – *Error-aware placement* and *Failure prediction* – for globally-optimized cross-layer resilience. Similarly, in [9], the authors propose various cross-layer techniques – from *microarchitecture* to *application* level – for both general purpose processor and reconfigurable processor based embedded systems. In all methods presented, every layer takes advantage of the information available at its adjacent layers. In [6], the authors present a cross-layer approach providing resilience in multimedia applications. Specifically, the proposed method uses hardware layer for error detection, middleware for Drop and Forward recovery and application layer for error resilient application design. In [10], the authors provide a heuristic-based methodology for combining several hardware and software techniques – *Circuit-level hardening*, *logic-level parity checking*, *microarchitectural recovery*, and Algorithm-based Fault Tolerance (*ABFT*) – to provide soft-error tolerance in processor cores.

Most of the state-of-the-art cross-layer reliability techniques lack a



(a) Application model (b) Architecture model

Fig. 1: System model

holistic approach and do not consider all application-specific reliability metrics – *Functional*, *Timing* and *Lifetime*. For instance, the approach described in [10] involves maximizing the fault-mitigation by software layers. Usually, software mitigation of hardware faults is based on *temporal redundancy* resulting in lesser area/power overheads. However, the increased execution time can lead to faster aging. In [11], the authors show the adverse effects of increasing checkpoints, a temporal redundancy-based method, on permanent fault tolerance. Therefore, systems that have design constraints of system lifetime have to use additional processing units. This can offset some of the area/cost advantages. In [12], the authors presented a cross-layer compositional analysis-based framework for reliability implementation in embedded systems. The proposed methodology involves estimating the interference of fault-mitigation approaches at different modeling levels. Our proposed approach for designing cross-layer reliability involves finding the right *selection* and *configuration* of fault-mitigation methods that should be implemented at each layer to meet the application-specific reliability objectives and constraints, and can be considered applicable to the system-level design in [12].

III. System Model

A. Application model

Mathematically, we model the application as a task-graph G_{app}, represented by a tuple (T, E, D), the set of task nodes, the directed connectivity of the nodes representing task dependencies, and the deadline of the application respectively. Each task, $Task_t$ in the task-graph is characterized by the tuple $(ID_t, Type_t, Impl_t)$: the task id, task type and the set of implementations for the task. Each i^{th} implementation of $Task_t$, $Impl_{(t,i)}$, is characterized by the minimum execution time $Tmin_{(t,i)}$ and scale parameter of Weibull aging model $\eta_{(t,i)}$. The scale parameter is a function of the thermal profile of executing $Impl_{(t,i)}$ of $Task_t$. Fig. 1a shows a sample task-graph for an application with 12 tasks and deadline of 600 time units.

B. Architecture Model

For the architecture model, we assume the hardware platform as a NoC-based MPSoC with a mesh architecture, similar to the one shown in Fig. 1b. The hardware platform has P processing elements (PEs), each PE_p characterized by the tuple $(ID_p, Type_p, \beta_p, MF_p)$: the PE's id, PE type, shape parameter and masking factor. $Type_p$ can be used to denote the different type of PEs, such as general purpose embedded processors, application specific instruction set processors etc. The β_p of Weibull model represents the aging-related fault profile of the PE. Similarly, MF_p represents the soft-error masking factor for the PE and can be characterized by the Architectural Vulnerability Factor (AVF) [13] of the PE. We consider heterogeneity among PEs w.r.t. variation in β_p and MF_p in addition to PE type.

C. Reliability Model

Functional reliability (Rel_{FR}) refers to the probability of an execution resulting in correct computation. The transient fault induced computational errors reduce the functional reliability of the system. We model Rel_{FR} as represented in Eq. 1. ER_{app}, ER_t ER_{spec} represents the error-rate of the application, a task and the system specifications respectively.

$$ER_{app} = \max_{Task_t \in T} ER_t \;\; ; \;\; Rel_{FR} \; = \; \frac{1 - ER_{app}}{1 - ER_{spec}} \tag{1}$$

Timing reliability (Rel_{TR}), of an application G_{app} can be expressed as the probability of execution of G_{app} completing within the certain specified threshold. Due to the usage of temporal redundancy based fault mitigation methods, Rel_{TR} can be a complex function of the timing reliability of the tasks in G_{app}. The dependencies among the tasks, mapping, and scheduling of tasks on the PEs, and the implemented fault-mitigation methods add to the complexity of determining Rel_{TR}. Such an analysis is beyond the scope of the current article. For the current article, we use the inverse of the average completion time as the measure of timing reliability [11], as shown in Eq. 2.

$$\begin{aligned} AvgEndT_{app} &= \max_{Task_t \in T} \;\; \{Average\ End\ time\ of\ Task_t\} \\ Rel_{TR} &= D / AvgEndT_{app} \end{aligned} \tag{2}$$

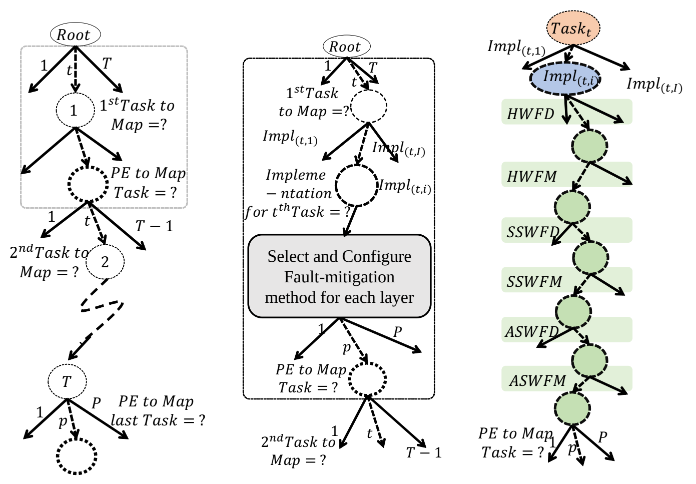
We represent the expected lifetime of the system, $SystemMTTF$, by the Mean Time To Failure (MTTF) of the system. The reliability model used is similar to that presented in [14]. Assuming a Weibull distribution of failures, and considering the temporal variation in aging effect, caused by the time multiplexing of different tasks on a PE, the effective MTTF of a PE, $MTTF_p$ and can be expressed as shown in Eq. 3. $MTTF_{(t,i,p)}$ refers to the MTTF for executing i^{th} implementation of $Task_t$ on PE_p. T_p is the set of tasks mapped on PE_p and $Tavg_t$ refers to the average execution time of the implementation of $Task_t$. P_{app} refers to the average makespan of the application. We define the **Lifetime Reliability** (Rel_{LR}) as the fraction of required MTTF for the system, $MTTF_{spec}$. The Rel_{TR} with MTTF considers the time to first failure. We assume that successive task mappings, in the event of permanent faults, that optimize the $MTTF_{SYS}$ leads to optimization of the overall system lifetime.

$$\begin{gathered} MTTF_{(t,i,p)} \; = \; \eta_{(t,i)} \times \Gamma\,(1 \; + \; 1/\beta_p) \;\;\; ; \;\;\; MTTF_p = \frac{P_{app}}{\sum\limits_{T_p} \frac{Tavg_t}{MTTF_{(t,p)}}} \\ SysMTTF = \min_{all\ PEs}(MTTF_p); Rel_{LR} = \frac{SystemMTTF}{MTTF_{spec}} \end{gathered} \tag{3}$$

IV. MCTS-based DSE

A. Modeling DSE as a TTP

Fault-mitigation involves implementing different fault-detection and fault-tolerance methods for masking the effect of hardware faults. The design of cross-layer reliability entails selecting and configuring such methods at different layers of the system. The *selection* and *configuration* can be represented as a sequence of design decisions. This *sequence* can be modeled as moves of a single-player game (like *Sudoku*). Consequently, the search for the optimal set of design decisions can be modeled as a TTP, searching for the leaf node with highest rewards. Figure 2a represents such a search tree for the task-mapping and scheduling problem for MPSoCs [15]. The dotted rectangular outline in the figure shows the design decisions associated with mapping each task on the available hardware platform. The first decision concerns selecting the appropriate task to be considered for mapping and is determined by the task-level dependencies in the application graph. Similarly, the second decision – selecting the PE to



(a) Task-mapping *only* (b) Task-mapping with fault-mitigation (c) Cross-layer fault-mitigation

Fig. 2: DSE for Cross-layer reliability as a TTP

map the selected task to – should consider the execution time of the selected task on each PE and the next earliest execution-slot available on the PE. Consequently, mapping T tasks on P processing elements results in a search tree of depth $2T$. The resulting branching factor of the tree is up to T, decreasing from T to 1 as we go down the tree from the $Root$ node, for the first design decision and P for the decision to select the PE. Hence, the number of *possible* design points on the exploration space, represented by the possible number of leaf nodes, is ($P^T \times T!$).

For cross-layer fault-mitigation, the design decisions for each task are shown in Figure 2b. The second decision refers to selecting among various implementations of the task ($Impl_t$), followed by a set of design decisions w.r.t. generic fault-mitigation methods at each layer, shown in the figure as a collection of selection and configuration decisions. The number of maximum possible leaf nodes can be expressed as ($P^T \times T! \times \prod_{t=1}^{t=T}(I_t \; FM_{CL})$), where I_t is the number of possible $Impl_t$ choices for task $Task_t$ and FM_{CL} is the product of the number of choices for generic fault-mitigation methods at each layer.

B. Monte Carlo Tree Search (MCTS)

Tree search-based optimization involves finding an optimal leaf node in the search tree. Exhaustive search approaches, such as Depth First Search (DFS) and Breadth First Search (BFS) expand the whole tree to find an optimal node and can be very costly for problem sizes described in the previous sub-section. Heuristics based methods, such as A-star [16] aim to expand fewer nodes in the tree while finding an optimal node. The search tree expansion (adding a new node) is usually guided by a heuristic measure that provides an estimation of the *reward* for expanding a node. MCTS [17] is one such guided search method where the reward for expanding a node, say $Node_X$, is estimated through Monte Carlo simulation of design decisions starting from $Node_X$. A brief description of the different stages in MCTS is outlined below:

- *Tree policy* involves selecting the appropriate $Node_X$ to perform Monte Carlo simulations from. At each iteration, a new node is added to the tree. The selection of $Node_X$ is based on a parameter, $XploreF$, that controls the balance between *exploitation* and *exploration*.
- *Default policy* refers to the Monte Carlo simulation for design decisions starting from $Node_X$ until a leaf node is reached.
- *Backpropagation* entails estimating the reward of the leaf node and updating the reward value and number of visits for all nodes along the path form $Node_X$ up to the $Root$ node.

Similar to most DSE approaches based on randomization and evolutionary algorithms, MCTS involves converting the NP-hard problem of finding the best task-mapping into iteratively, and *intelligently*, selecting and evaluating various feasible points on the design space to provide a *best-effort* solution that may be sub-optimal. The MCTS-based approach is scalable with the available computation budget. Additional time for DSE leads to expanding and evaluating more nodes, and with sufficient time, can lead to results similar to an exhaustive search. The parameter $XploreF$ can be used to control the computation budget spent in exploiting more rewarding paths or explore fewer rewarding ones.

C. Cross-layer DSE using MCTS

1) Implementing Cross-layer Reliability

Implementing a task involves designing cross-layer reliability for the task, mapping the task to a PE and scheduling the task's execution on the PE. We implement decision-making w.r.t. cross-layer reliability as a sequence of design decisions as shown in Fig. 2c. The fault/error mitigation at each abstraction layer $< X >$ can be modeled as a two-step process – detection ($< X > FD$) and mitigation ($< X > FM$). The reliability design for each layer is described below.

Task-specific Implementation (**IMPL**): In addition to selecting from a set of implementations (varying in latency, power etc.), we use the IMPL layer to represent the choices w.r.t. task-specific fault-mitigation approaches (e.g. partial TMR, algorithms, etc.).

Hardware Fault-mitigation (**HWFM**): For our current work, we limit the options for HWFM to the following:

- Using PEs with higher masking factor (MF_p) for improving the functional reliability of the task.
- Aging mitigation (**AgMit**) is implemented at the hardware layer by using PEs with lower β_p, signifying reduced aging effect. This is integrated into the search tree as additional HWFM choices, for each task type ($Type_t$), with increased $MTTF_{(t,i,p)}$.

System Software Fault-mitigation (**SSWFM**): We model the choices for fault-detection by system software (SSWFD) as a tuple (Cov_{FD}, TOv_{FD}), the coverage of the detection method and the timing overheads for implementing the SSWFD method respectively. TOv_{FD} is expressed as a percentage of the duration of *useful* computation time to which the detection applies. This lets us model application specific methods in addition to generic methods like dual temporal redundancy (DTR). The fault-tolerance step involves selecting and configuring the fault tolerance (SSWFM) method. For our current work, we limit SSWFM to checkpointing with rollback recovery. The design choices are denoted by N_{Chk}, the number of checkpoints and TOv_{chk}, the overhead associated with creating a checkpoint. TOv_{chk} is expressed as a percentage of the minimum execution time of the task. We denote the absence of SSWFM as $N_{Chk} = 0$, with no overhead. Similarly, $Retry$ with rollback can be represented with $N_{Chk} = 1$ and $TOv_{Chk} = 100$.

Application Software Fault-mitigation (**ASWFM**): We model the choices w.r.t. fault-mitigation at the application layer as additional implementation choices for the task. Typical examples of ASWFM are some algorithmic modifications (approximations, ABFT etc.) that can result in different execution times on the same PE.

2) Constrained Decision Making

The *pure* MCTS approach ($pMCTS$) is unconstrained and the nodes are expanded and simulated irrespective of the feasibility of the resulting system state. Only the leaf nodes are evaluated for correctness and reward. While this approach makes the method problem-agnostic, it can lead to wastage of computation resources by evaluating infeasible regions of the search tree. We implemented constrained MCTS ($cMCTS$) by constraining the decision-making while selecting the next task to implement on the MPSoC. The feasible options for selecting the task was based on the following criteria – a task is considered a feasible for implementing if either of the following conditions is satisfied:

- The task has no parent tasks defined by the application graph.
- All its parent tasks have already been implemented.

The constrained approach at an early stage of the design decision-making restricts the number of possible child nodes and hence fewer iterations are lost on simulating all nodes.

3) MCTS Node Reward

The rewards of leaf nodes of the search tree are used to guide the tree traversal. We model the reward of a leaf node in the cross-layer design tree as a weighted sum of the three reliability metrics, as shown in Eq. 4. The implemented MCTS methodology aims to maximize Rew_{CL}.

$$\begin{aligned} Rew_{CL} = Wt_{FR}\ Rel_{FR} + Wt_{TR}\ Rel_{TR} + Wt_{LR}\ Rel_{LR} \\ where,\ Wt_{FR}\ + Wt_{TR}\ + Wt_{LR}\ = 1, \\ and\ \ 0 \leq\ Wt_{FR}\ , Wt_{TR}\ , Wt_{LR}\ \leq 1 \end{aligned} \tag{4}$$

V. Results and Discussions

A. Experiment Setup

1) Cross-layer Fault-Mitigation

The experimental evaluation of integrating implementation of cross-layer reliability into task-mapping involved modeling the effects of fault-mitigation at different layers as described below. The design decisions are based on the methods and parameters described in Section IV-C1.

- **NoFM**: This is the base case of implementing the tasks on a homogeneous MPSoC without any fault-mitigation methods. The PEs of the MPSoC do not have any additional soft-error mitigation measures ($MF_p = 0$). Similarly, the homogeneity of the PEs w.r.t. aging is modeled by using a constant MTTF of the task implementations for each PE. Further, the *NoFM* method does not include any *SSWFM* measures and each of the task types has only one application software implementation. For our experiments, we use an MPSoC with 8 PEs with MTTF of 10 $years$ for each implementation.
- **SSWFM**: The coverage of SSWFD was varied form 0 to 100 in increments of 10. Similarly, the TOv_{FD} was varied from 0 to 100%. The SSWFT involved varying N_{Chk} form 0 to 10 and a fixed value for TOv_{Chk} of 2%.
- **ASWFM**: Additional implementation instances were added for half the task types with minimum execution time reduced by 10% of that of the original implementations.
- **HWFM**: Functional reliability improvement by HWFM was modeled by increasing the MF_p of half the PEs in the MPSoC to 20%, resulting in a heterogeneous hardware platform.
- **AgMit**: The heterogeneity of the MPSoC w.r.t. aging was modeled by changing 2 PEs to a new type, with the implementation of each task type on these PEs resulting in increased MTTF of 15 years.
- **AllFM**: The combination of all fault-mitigation methods described above.

2) DSE for Cross-layer Fault Mitigation

The evaluation of MCTS-based design space exploration for cross-layer reliability-based task-mapping involved comparison of the following approaches.

TABLE I: Comparison for $pMCTS$ ($M1$) and $cMCTS$ ($M2$)

Fat Task-graphs					$Slim$ Task-graphs				
Num Tasks	Normalized Makespan		Maximum Branching		Num Tasks	Normalized Makespan		Maximum Branching	
	$M1$	$M2$	$M1$	$M2$		$M1$	$M2$	$M1$	$M2$
10	1	1	10	8	13	1	1	13	8
22	1	1	22	12	24	1	1	24	8
34	1	1.03	34	24	35	1	1	35	8
42	1	1.06	42	30	46	1.36	1	46	8
50	1	1.07	50	38	54	1.74	1	54	8
110	1	1.07	110	88	106	2	1	106	8
212	1.29	1	212	85	206	1.9	1	206	8
317	1.59	1	317	57	306	1.99	1	306	8
407	1.89	1	407	111	406	2.17	1	406	8

1) **Integrated MCTS** (inMCTS): The *inMCTS*-based DSE involves integrating the design decisions w.r.t. implementing cross-layer reliability into the search tree for task-mapping, as shown in Fig. 2b. This approach provides complete freedom for exploration of all design decisions, including those for task to PE binding and task scheduling on the PEs.
2) **Binding-constrained MCTS** (bcMCTS): The *bcMCTS* approach constrains the design decision w.r.t. mapping tasks to PEs based on the already existing task-binding information. Hence, the DSE is limited to scheduling tasks on the PEs and cross-layer reliability design.
3) **GA-based parameter variation** (pvGA): The *pvGA* approach entails using *Genetic Algorithm* (GA) based DSE of design choices w.r.t. cross-layer reliability only. These choices were encoded as a sequence of parameters (one **individual** of a **generation**) for all tasks. The *pvGA* approach uses already existing task to PE bindings and task-schedules for each PE of the MPSoC.

The proposed MCTS-based approaches for DSE were implemented as a *C++* application. The *pvGA* approach was implemented in Python using DEAP [18] package.

3) Application-specific Reliability

The test applications for the experiments were generated using Task Graphs For Free (TGFF) tool [19]. The execution times for the tasks were also obtained from the TGFF tool. The proposed methodology was evaluated for application-specific cross-layer reliability design by varying the weights associated with each type of reliability. Specifically, the values used for the tuple $(Wt_{TR}, Wt_{FR}, Wt_{LR})$ are:

- Optimizing Timing Reliability (*forTR*): $(1, 0, 0)$
- Optimizing Functional Reliability (*forFR*): $(0, 1, 0)$
- Optimizing Lifetime Reliability (*forLR*): $(0, 0, 1)$
- Optimizing Balanced Reliability (*forBR*): $(0.33, 0.33, 0.33)$

B. Results and Discussions

1) Constrained MCTS

The effectiveness of the constrained MCTS ($cMCTS$) approach was evaluated based on the comparison of its performance against that of the *pure* MCTS ($pMCTS$) approach in minimizing the makespan of an application. The number of tasks in the applications was varied from 10 to 400 and two types of application task-graphs were used in the evaluation – Fat, ones with more task-level parallelism than the other type, $Slim$. The *pure* MCTS ($pMCTS$) approach involves list scheduling of tasks on reaching a leaf node by random simulation. The tasks are scheduled after all the tasks have been mapped to PEs. As a result, the number of maximum possible branches while selecting the task is always equal to the number of tasks in the application. This is evident from the performance results for both the approaches shown in Table I. For $cMCTS$, the possible choices while selecting the next task for implementing are based on the criteria discussed in Section IV-C2. This leads to a reduced maximum branching, more evident

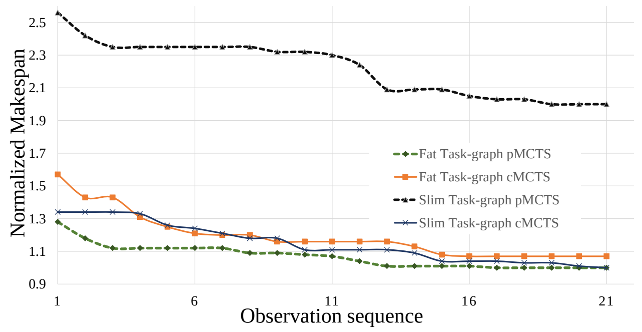


Fig. 3: Progress of MCTS-based makespan minimization

in the case of $Slim$ task-graphs, where the maximum branching is fixed by the number of PEs in the hardware platform (i.e. 8). The effect of the increased branching is clearly shown in Table I, where the $cMCTS$ performance is almost always better than the $pMCTS$. The results reported in the table are based on the best result for both approaches after 50000 iterations. In $cMCTS$, fewer nodes are evaluated at the *initial* portion of the search tree allowing the search path to go deeper down the tree and get better estimates of the leaf nodes. For $pMCTS$, most of the computations are wasted in evaluating all the possible branches closer to the $RootNode$. The progress of each approach for both $Slim$ and Fat application task-graphs with $\sim$ 100 tasks is shown in Fig. 3. For $Slim$ task-graphs, the $pMCTS$ approach does not let the search path go sufficiently deep down the tree to get better estimates of the leaf nodes. However, for Fat task-graphs, the $pMCTS$ approach performs marginally better than the $cMCTS$ approach for smaller graphs but does not scale beyond $\sim$ 100 tasks. For smaller task-graphs, the reduced depth of the search tree enables the $pMCTS$ approach to search the tree more effectively. However, with increasing depth of the search tree, the performance of the $cMCTS$ approach is markedly better than that of the $pMCTS$ approach.

2) *Comparing DSE Approaches*

For our experiments, a combination of the DSE approaches, as shown in Table II, were evaluated. *List* refers to the task to PE binding and task-scheduling information generated from an iterative list-based mapping of tasks to PEs and scheduling the tasks based on task-level dependencies of the application graph. Similarly, Int refers to the same information generated by an initial run of *inMCTS*. Fig. 4 shows the results of DSE for cross-layer reliability in an application with 100 tasks on a hardware platform with 8 PEs implementing *AllFM*. Each of the sub-figures shows the variation of a performance metric associated with one type of reliability – $AvgEndT_{app}$, ER_{app} and $SysMTTF$ for timing, functional and lifetime reliability respectively. The bar-graphs in the figures show the variation of these metrics with different DSE methods and optimization modes described in Section V-A3, after 1 $million$ design point evaluations. The *wall-clock* runtime is approximately 15 $minutes$ 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. The following points about the results are noteworthy:

- As intended, for all DSE methods the optimization of any specific type of reliability leads to most improved results for the metric associated with that type.
- In almost all cases, the constrained DSE methods perform better while using the constraints obtained from the *inMCTS* approach than those from the *List* method. Therefore, integrating the design of cross-layer reliability into task-mapping provides a better starting point for further improvements by either *bcMCTS* or *pvGA*.
- The values for ER_{app}, the maximum error rate, are an effect of the functional reliability modeling. The constant high values in Fig. 4b, except in *forFR*, signify the avoidance of using any *SSWFM* methods (with high timing overheads) while optimizing for other non-functional-reliability metrics. Similarly, in *forFR* mode, both *SSWFM* and masking effect of *HWFM* methods are utilized.
- The *forBR* results show better improvements in timing and lifetime than functional reliability. This can be attributed to the high timing overheads associated with improving functional reliability by *SSWFM*. Instead, better Rew_{CL} values are obtained by improving the average makespan and the resulting reduced aging leads to improved $SystemMTTF$.

TABLE II: DSE METHODS

Method	Task to PE binding	Task-scheduling	Cross-layer Fault-mitigation
inMCTS	inMCTS	inMCTS	inMCTS
bcMCTS(List)	List	bcMCTS	bcMCTS
pvGA(List)	List	List	GA
bcMCTS(Int)	inMCTS	bcMCTS	bcMCTS
pvGA(Int)	inMCTS	inMCTS	GA

The improved results of *bcMCTS*, and *pvGA* over *inMCTS* can be viewed as an extension of the *cMCTS* approach described in Section V-B1 (and demonstrated in Section V-B1) which lets the method explore more of the non-constrained design decisions in the available computational budget. Further, the *pvGA* approach provides better improvements than *bcMCTS* by *fine-tuning* the results from *inMCTS*.

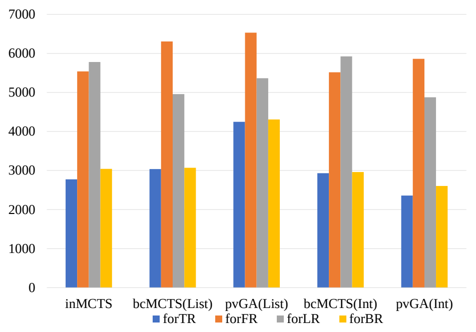
3) *Estimating Fault-mitigation Effects*

The proposed MCTS-based DSE methodology can be used to estimate the effect of fault-mitigation methods, implemented at different layers and their combinations, on system-level reliability. Fig. 5 shows the percentage improvements in the reliability metrics over *NoFM* implementation by using the fault-mitigation methods and optimization modes described in Section V-A. The hardware platform and the application are same as the one used for discussions in the previous sub-section. The following observations from the results are noteworthy:

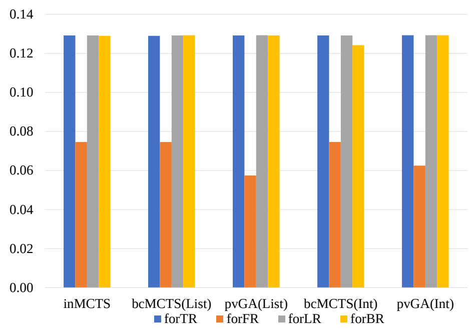
- As shown in Fig. 5a, *ASWFM* shows maximum improvements for Rel_{FR} as it reduces the execution time of some tasks by 10% without adversely affecting other reliability metrics. Higher makespan values for *SSWFM* can be attributed to the increased average execution time of tasks due to detection and checkpointing. Further, the increased average execution results in a reduction of $SysMTTF$; this effect is observed for all optimization modes.
- As shown in Fig. 5b, the maximum improvements to Rel_{FR} is observed for *AllFM*, where both *SSWFM* and masking effect of *HWFM* are utilized. However, as evident from the figure, unlike the case of *HWFM*-only, using *SSWFM*-only leads to increased average makespan for the application leading to a major reduction in Rel_{TR} and Rel_{LR}.
- The maximum Rel_{LR} improvement is observed with *AgMit*, as shown in Fig. 5c. The resulting reduced Rel_{TR} can be attributed to more tasks being mapped to the PEs with higher MTTF leading to increased average makespan.
- As shown in Fig. 5d, and described in the previous experiment's discussion, in the *forBR* mode, the improvements to Rew_{CL} is obtained by improving timing and lifetime reliability, while keeping the maximum error rate same as that of *NoFM*.

VI. Conclusion

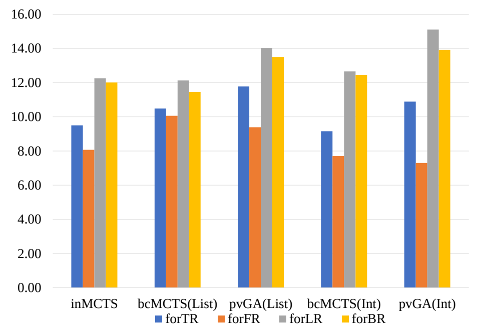
With increasing susceptibility of hardware to physical faults, a comprehensive fault-aware cross-layer design approach is necessary. To enable such an approach, a *Monte Carlo Tree Search*-based application-specific DSE methodology for designing cross-layer reliability in embedded systems is proposed. Further, a second-level *fine-tuning* by using *Genetic Algorithms* was introduced. Experimental evaluations show maximum improvements in reliability metrics using an integrated



(a) $AvgEndT_{app}$: Lower values for improved Rel_{TR}

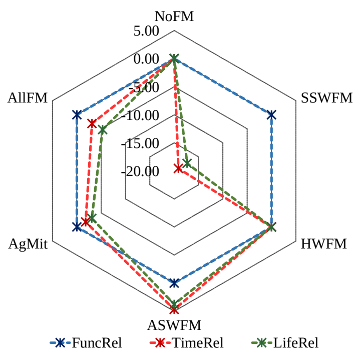


(b) ER_{app}: Lower values for improved Rel_{FR}

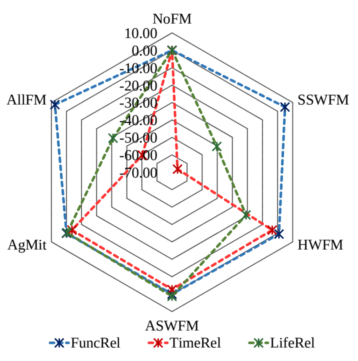


(c) $SysMTTF$: Higher values for improved Rel_{LR}

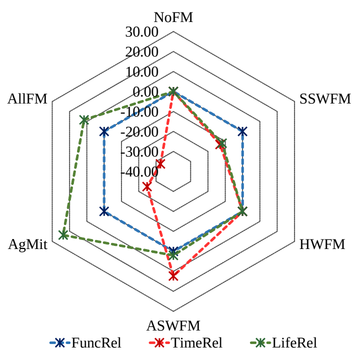
Fig. 4: Comparison of DSE methods for designing *Cross-Layer Reliability*



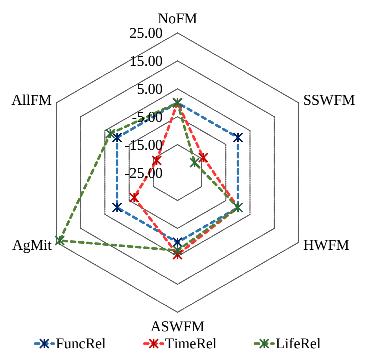
(a) $forTR$: Optimizing Rel_{TR}



(b) $forFR$: Optimizing Rel_{FR}



(c) $forLR$: Optimizing Rel_{LR}



(d) $forBR$: Optimizing with equal weightage to Rel_{TR}, Rel_{FR} and Rel_{LR}

Fig. 5: Percentage improvements over $NoFM$ reliability metrics for different fault-mitigation methods and their combination

MCTS approach followed by GA-based parameter variations. The proposed methodology provides a high-level DSE methodology for implementing application-specific cross-layer reliability in embedded systems and can be used for early-stage design space pruning by estimating system-level effects of introducing fault-mitigation at different layers. Further, the methodology can be coupled with more complex system models to obtain better realizable system designs.

References

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

[2] P. Shivakumar, et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks,*, 2002.

[3] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *IEEE Micro*, 2003.

[4] N. P. Carter, et al. Design techniques for cross-layer resilience. In *DATE*, 2010.

[5] S. S. Sahoo, et al. Cross-layer fault-tolerant design of real-time systems. In *DFTS*, 2016.

[6] 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.

[7] T. Santini, et al. Evaluation of failures masking across the software stack. *MEDIAN*, 2015.

[8] L. Leem, et al. Cross-layer error resilience for robust systems. In *ICCAD*, 2010.

[9] J. Henkel, et al. Multi-layer dependability: From microarchitecture to application level. In *DAC*, 2014.

[10] 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 Annual Design Automation Conference*, DAC, 2016.

[11] A. Das, et al. Aging-aware hardware-software task partitioning for reliable reconfigurable multiprocessor systems. In *CASES*, 2013.

[12] M. Glaß, et al. Cross-level compositional reliability analysis for embedded systems. *Computer Safety, Reliability, and Security*, pages 111–124, 2012.

[13] S. S. Mukherjee, et al. 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 36, 2003.

[14] Y. Xiang, et al. System-level reliability modeling for MPSoCs. In *CODES*, 2010.

[15] A. K. Singh, et al. Mapping on multi/many-core systems: Survey of current and emerging trends. DAC, 2013.

[16] P. E. Hart, et al. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[17] C. B. Browne, et al. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1), 2012.

[18] F. Fortin, et al. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, July 2012.

[19] R. P. Dick, et al. TGFF: task graphs for free. In *CODES*, pages 97–101. IEEE Computer Society, 1998.