Reliability Optimization on Multi-Core Systems with Multi-Tasking and Redundant Multi-Threading

Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen
Department of Informatics, TU Dortmund University, Germany
E-mail: {kuan-hsun.chen, georg.von-der-brueggen, jian-jia.chen@tu-dortmund.de}

***Abstract*—Using Redundant Multithreading (RMT) for error detection and recovery is a prominent technique to mitigate soft-error effects in multi-core systems. Simultaneous Redundant Threading (SRT) on the same core or Chip-level Redundant Multithreading (CRT) on different cores can be adopted to implement RMT. However, only a few previously proposed approaches use adaptive CRT managements on the system level and none of them considers both SRT and CRT on the task level. In this paper, we propose to use a combination of SRT and CRT, called Mixed Redundant Threading (MRT), as an additional option on the task level. In our coarse-grained approach, we consider SRT, CRT, and MRT on the system level simultaneously, while the existing results only apply either SRT or CRT on the system level, but not simultaneously. In addition, we consider further fine-grained task level optimizations to improve the system reliability under hard real-time constraints. To optimize the system reliability, we develop several dynamic programming approaches to select the redundancy levels under Federated Scheduling. The simulation results illustrate that our approaches can significantly improve the system reliability compared to the state-of-the-art techniques.**

I. Introduction

Due to aggressive technology downscaling, memory and logic components are now vulnerable to electromagnetic inference and radiation, leading to transient faults in the underlying hardware [4, 32]. These transient faults may jeopardize the correctness of software execution, so called soft errors, or even lead to a system crash. Redundant Multithreading (RMT) techniques [21, 30, 35] provide fault detection and recovery by replicating a task into multiple identical threads and comparing the produced results. Especially dual modular redundancy (DMR) and triple modular redundancy (TMR) [20, 21, 30, 34] are widely used, where DMR only provides fault detection while TMR can possibly correct the faulty instance as well. Simultaneous Redundant Threading (SRT) approaches [30, 35] provide transient fault coverage by running identical copies of the task on the same processor. However, due to the additional computation time [21, 30, 35], applying SRT directly on high utilization tasks may lead to a system overload.

In multi-core systems, redundant cores can be used for RMT, using approaches like Intel's Chip-level Redundant Multithreading (CRT) [7, 21, 25], which provide alternatives to mitigate soft-error effects. Redundant replicas of a given task are executed on different cores in parallel and error detection and recovery approaches are performed by comparing the threads' output. Nevertheless, solely adopting CRT with redundant cores is not sufficient, since the number of redundant cores is limited. Even if the number of cores is adequate to activate CRT for all tasks, the utilization of the dedicated cores may be unnecessarily low due to low utilization tasks.

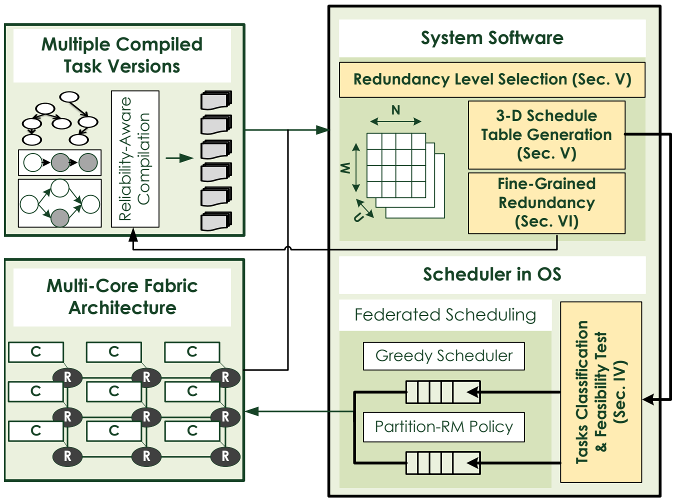

Multiple Compiled Task Versions
Reliability-Aware Compilation
System Software
Redundancy Level Selection (Sec. V)
N
M
C
3-D Schedule Table Generation (Sec. V)
Fine-Grained Redundancy (Sec. VI)
Multi-Core Fabric Architecture
C
R
Scheduler in OS
Federated Scheduling
Greedy Scheduler
Partition-RM Policy
Tasks Classification & Feasibility Test (Sec. IV)

Fig. 1: Overview of the considered system, illustrating interactions between different components for dependable system execution. Our contributions are highlighted in yellow and are linked to the related section of the paper.

Satisfying given design constraints, i.e., limited time and cores, motivates us to explore if the aforementioned techniques, i.e., SRT and CRT, can be efficiently applied at the same time in order to fully exploit the system resources. Existing results typically assume that TMR-based RMT has to be applied using three cores in parallel, i.e., CRT [7, 25], or that three replicas are executed on one core sequentially, i.e., SRT [30]. In this paper, we propose the alternative that TMR-based RMT executions can also be a mixture of CRT and SRT, where two threads are executed on the same core and one is executed on another core in parallel, termed as Mixed Redundant Threading (MRT). Our scheme minimizes the *overall reliability penalty* [7, 25, 29] for a given hard real-time system by considering RMT with SRT, CRT, and MRT on multi-core systems at the same time. [1] Some more concrete motivational examples are given in Section III, in which MRT provides a better system reliability and applicability comparing to SRT and CRT.

[1]Whenever we mention the system reliability, the considered metric is from the penalty perspective. If the reliability penalty is high, the system is less reliable. If the reliability penalty is low, the system is more reliable. Details can be found in Section IV-C.

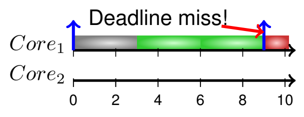
Penalty Value	ϕ	SRT	MRT	CRT
τ_1	R_1	∞	$\epsilon+\Delta$	ϵ
τ_2	R_2	∞	$\epsilon+\Delta$	ϵ

(a) Tasks reliability penalty on different modes

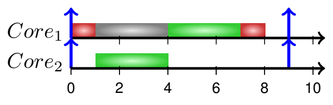
Mappings	Total Penalty	Max Penalty
SRT(τ_1) + $\phi(\tau_2)$ and SRT(τ_2) + $\phi(\tau_1)$	∞	∞
$\phi(\tau_1)+\phi(\tau_2)$	R_1+R_2	$\max\{R_1, R_2\}$
CRT(τ_1) + $\phi(\tau_2)$ and CRT(τ_2) + $\phi(\tau_1)$	$R_1+\epsilon$ or $R_2+\epsilon$	$\max\{R_1+\epsilon, R_2+\epsilon\}$
MRT(τ_1)+MRT(τ_2)	$2(\epsilon+\Delta)$	$\epsilon+\Delta$

(b) Possible selections and overall penalty

TABLE I: Corresponding penalty values and possible mappings for the motivational example in Section III.



(a) SRT is not feasible due to a deadline miss.



(b) MRT can activate TMR with two cores.

Fig. 2: TMR-based RMT on two cores. Grey blocks are the original execution, green blocks are replicas, and red blocks represent the workload necessary for forking and joining the original execution and replicas. CRT is not possible since only two cores are available. While SRT is not feasible due to a deadline miss as shown in subfigure (a), MRT can feasible schedule the task with TMR as shown in subfigure (b).

In this paper, we aim to maximize the system's reliability while satisfying the given hard real-time constraint. In this reliability optimization problem we consider different RMT options on a multi-core system, i.e., DMR/TMR with SRT, CRT, and MRT. This reliability optimization problem is tackled by adopting Federated Scheduling [18] and R-BOUND-MP-NFR [2] as the backbone. We propose several dynamic programming algorithms to optimize the system's reliability while guaranteeing the system's schedulability. Through the simulation results, we can see that the proposed approaches can significantly decrease the system reliability penalty up to 54% compared to the state-of-the-art techniques.

Our contributions: Figure 1 illustrates an overview of our contributions, detailed as follows:

- Typically, SRT and CRT are studied individually in the literature. In this paper, we propose Mixed Redundant Threading (MRT) as a mixture of SRT and CRT, i.e., the task and one replica are executed on one core and a second replica is executed on a different core. We take SRT, CRT, and MRT into consideration at the same time to improve the system reliability while satisfying the timing constraints in multi-core systems.
- In Section V we show how Federated Scheduling can be adopted to assign each task either to dedicated cores, i.e., cores only used by one task, or to multi-tasking cores, under the assumption that all execution levels of all tasks are given.
- If the execution levels of tasks are not given, we propose a dynamic programming to find the optimal selection of redundancy levels while satisfying the timing constraints in Section VI.
- If different task stages can use different levels of redundancy, we show how to adapt the dynamic programming algorithm to provide a fine-grained optimization in Section VII.
- We evaluate our algorithms in Section VIII by comparing with a greedy approach used in the two state-of-the-art approaches [7] and [25] under different execution scenarios.

II. Related Work

RMT techniques are well-known solutions to provide fault detection and recovery [21, 30, 35] on uniprocessor or multi-core systems. The state-of-the-art techniques in [7, 25] assume that TMR-based RMT should be applied using three cores in parallel with CRT, or that three replicas are executed on one core sequentially with SRT [30] . In [8, 9], the main objective is energy efficiency. However, these works only consider fault tolerance or the optimization of system reliability without any real-time constraints.

Scenarios in general similar to our studied problem are considered in [5, 10, 16], i.e., the objective of these papers is to schedule tasks with real-time constraints while concerning fault detection/tolerance feature. However, these papers consider frame-based task systems. Even in such a simplified setting, the problem explored in this paper is still very different from the problems explored in [5, 10, 16]. In [5, 10] the energy efficiency and mean-time to failure under lifetime constraints is optimized. In [16] the authors present a runtime schedule strategy to adapt to the occurrence of faults at runtime to reduce the overhead due to fault tolerance. Those algorithms are not applicable for our studied problem.

Some related researches for the schedulability problem of parallel workloads in fork-join and DAG task models are known, i.e., [3, 17, 18]. Axer et al. [3] focus on worst-case response time analysis with a fork-join task model. Li et al. [18] propose a Federated Scheduling strategy to deal with parallel workloads in partitioned scheduling. However, most existing research assumes that the parallel workload distribution is known, i.e., the number of threads per task is given beforehand, and only analyze the feasibility problem in terms of timeliness. Kwon et al. [17] tackle the scheduling problem with the global scheduling algorithm PD^2 [33] and determine the parallel executing options by trivially testing all possible combinations. The above work focuses on the scheduling of given parallel workloads, whereas our work aims to optimize the system reliability by parallelizing the workloads, scheduling them under given timing constraints on a given multicore platform.

III. Motivation

Assume that we want to activate TMR-based RMT to enable detection and correction for soft errors. In the literature, either SRT [7, 21] or CRT [7, 25, 30, 35] is used to provide TMR but these techniques are not combined. This means, the task and the two replicas are either executed sequentially on the same core (SRT) or on three different cores (CRT). The proposed mixture of SRT and CRT, called Mixed Redundant Threading (MRT), can balance the amount of used cores and the additional computation as MRT enables TMR with two available cores. Furthermore, MRT can exploit the available cores more efficiently to improve the reliability under the given time constraints.

First, consider one task and two available cores. Since SRT-TMR imposes a higher computation demand on one processor, e.g., 3x the execution time, it may lead to a deadline miss as shown in Figure 2(a). CRT-TMR is also not possible, as only two cores are available. However, if we execute the task and one replica on $Core_1$ sequentially and one on $Core_2$ in parallel like in Figure 2(b), TMR-based RMT is possible without sacrificing timeliness.

However, the main focus of this paper is not the schedulability problem. We consider that for a given task set the system reliability should be increased as much as possible by activating TMR for some (or at best all) tasks. We assume that the activation of TMR for a task τ_i has a different impact on the system reliability for each task, i.e, a reliability penalty is given for each task that is smaller when TMR is activated and larger when TMR is not activated. The total system reliability can be defined by any applicable metric, e.g., the sum of reliability penalty or the the maximum reliability penalty (see Section IV-C), and the goal is to minimize the systems reliability penalty under the given metric.

Suppose we can only use four homogeneous cores to execute two tasks τ_1 and τ_2 and that the corresponding reliability penalties under different RMT modes for τ_1 and τ_2 are given as shown in Table Ia. The penalties of task τ_1 and τ_2 without any redundancy (ϕ) are R_1 and R_2, respectively. The penalty values of τ_1 and τ_2 for SRT are ∞ due to deadline misses caused by the additional computing time. Δ is the reliability penalty induced by MRT and ϵ is the negligible reliability penalty of TMR-based RMT. Typically Δ and ϵ should be much smaller than R_1 and R_2, i.e., $R \gg \Delta > \epsilon$, $R \in \{R_1, R_2\}$. The mappings with two SRT-TMRs and two CRT-TMRs are not feasible. The mapping with one CRT-TMR does not provide the optimal reliability penalty under the given reliability penalty metrics, as only one task can be protected by CRT. However, using MRT for both tasks allows them to have redundant executions concurrently, by which the system penalties in both metrics are lower than the CRT mapping, if the reliability penalty Δ is much smaller than R_1 and R_2.

This shows that MRT provides additional options for balancing the usage of cores and for reliability optimization.

IV. System Model and Problem Definitions

In this section we introduce the application model, describe the different redundant multithreading schemes used in the paper, explain how we quantify the task reliability, and state the problem definition.

A. Application Model

We consider a hard real-time task system with N sporadic tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_N\}$. Each task τ_i is associated with a minimum inter-arrival time T_i, also called period, a relative deadline D_i, and a worst case execution time (WCET) C_i. This means an instance of task τ_i arriving at time t must be completed not later than the absolute deadline $t + D_i$ and the next task instance must be released not earlier than $t + T_i$. We focus on implicit-deadline task systems throughout the paper, i.e., $D_i = T_i \; \forall \tau_i \in \Gamma$, and consider the schedulability on a multi-core platform with M homogeneous cores that are connected by a communication fabric, e.g., a network on chip (NoC). We assume preemptive fixed-priority scheduling, i.e., a unique priority level is assigned to each task, as this model is widely used in the industrial practice and also supported in most real-time operating systems. Furthermore, we assume preemption overheads to be negligible compared to the execution time of one task instance.[2] Throughout this paper, we assume that all additional communication overhead can be integrated into the execution time of tasks, e.g., NoC overhead, cache coherency traffic, or shared resource accesses.

B. Redundant Multithreading (RMT)

We consider the two well known RMT modes Dual Modular Redundancy (DMR) [21, 30] and Triple Modular Redundancy (TMR) [26, 34]. In addition, we propose the mixed usage of CRT and SRT, called Mixed Redundant Threading (MRT), in which TMR can also be activated in two parallel cores with two replicas, i.e., the original execution and one replica are executed on one core while a second replica is executed on a second core. Using all possible combinations defines the following six redundancy levels:

- NON-RMT (ϕ): The task without any redundancy.
- SRT-DMR : original and 1 replica executed sequentially.
- SRT-TMR : original and 2 replicas executed sequentially.
- CRT-DMR : original and 1 replica executed in parallel.
- CRT-TMR : original and 2 replicas executed in parallel.
- MRT-TMR : original and 1 replica executed sequentially, 1 replica executed in parallel, i.e., original and 2 replicas in total.

These redundancy levels can be characterized as a set of directed acyclic graphs (DAGs) [18]. Figure 3 illustrates the possible DAGs, where each node (sub-task) represents a sequence of instructions and each edge represents execution dependencies between nodes. Each node is characterized by the WCET of the corresponding sub-task. A node is ready to be executed if all of its predecessors have been executed. Each box represents a processor, i.e., nodes in the same block are assigned to the same processor. In Section VI we assume that each task τ_i has K_i given levels generated by the reliability-aware compilation [15, 28], each matching one of the above six

[2] If the overheads are not negligible they can be integrated into the worst-case execution time (WCET) of tasks using standard approaches in literature.

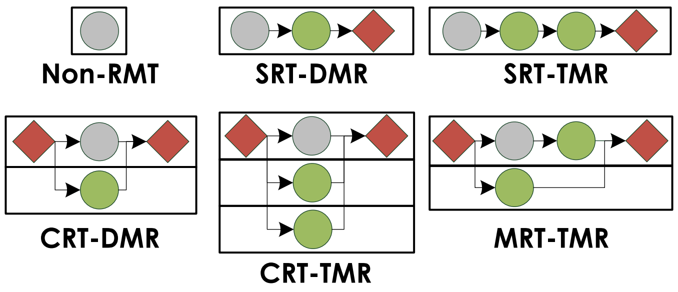


Fig. 3: DAG abstractions of the different redundancy levels, where the gray nodes are original executions and the green nodes are replicas. The red nodes represent the workload due to the necessary steps for forking the original executions and replicas, joining, and comparing the delivered results from DMR/TMR at the end of redundant multi-threading. The directed edges represent the dependencies between nodes. Each block represents one core, i.e., the number of cores differs depending on the redundancy level.

redundancy levels, i.e., $\tau_i = \{\tau_{i,1}, \tau_{i,2}, \cdots, \tau_{i,K_i}\}$. For each task $\tau_i \in \Gamma$, one of the levels is chosen to be executed, denoted as $\theta = \{\theta_1, \theta_2, \ldots, \theta_N\}$. For each level $\tau_{i,j}$, two parameters are assumed to be given:

- Total execution time $C_{i,j}$: The sum of the WCETs of all the sub-tasks of $\tau_{i,j}$.
- Critical-path length $L_{i,j}$: The WCET of the critical-path in the given DAG, i.e., sum of the WCETs of the path with the longest total WCET.

We assume that the correctness of the execution result is affected by soft-errors (faults). The WCET is profiled in fault-free system offline and protected by the adoption of watchdog approaches. Each level $\tau_{i,j}$ is associated with two cost values: the utilization $u_{i,j}$ and the number of required cores $H_{i,j}$. The utilization $u_{i,j}$ is given by $\frac{C_{i,j}}{T_i}$ and $H_{i,j}$ is determined by the scheduling policy detailed in Section V. We assume each task τ_i has at least a level $\tau_{i,\phi}$ without any redundancy and that its total execution time is not larger than its period.

C. Quantification of Task Reliability

We assume that the reliability penalty of each task level $\tau_{i,j}$ is given as $R_{i,j}$. It describes the probability that a fault during the execution of level $\tau_{i,j}$ leads to a visible error when executing. The probability of failure for each instruction is estimated by using Instruction Vulnerability Index and Function Vulnerability Index metrics [7, 27, 28]. The task vulnerability can be characterized/estimated by the composition of its instructions. The task level with lower vulnerability has a smaller reliability penalty, i.e., it has a better reliability. We use the Overall Reliability Penalty $\Psi_\Gamma(\theta)$ as the objective function in this paper to demonstrate our approaches, which can be defined as:

Definition 1 (Overall Reliability Penalty $\Psi_\Gamma(\theta)$): The overall reliability penalty of task set Γ is $\Psi_\Gamma(\theta) = \sum_{\tau_i \in \Gamma} R_{i,\theta_i}$, where R_{i,θ_i} is the reliability penalty of task τ_i executing at redundancy level θ_i and the set θ contains the redundancy levels θ_i of all tasks $\tau_i \in \Gamma$.

Please note that the proposed approaches are applicable to any system reliability metrics, if the optimization can be solved by a dynamic programming algorithm, i.e., $R_{i,j}$ can be set to any reliability penalty. In this work, we directly use the same metric of reliability penalty as in [7], which is also adopted in [25, 29] as a part of the linear combination for functional and timing reliabilities.

D. Problem Definition

Assume a multi-core system with M homogeneous RISC cores and a task set Γ to be given. The problem we are analyzing is to get a maximized system reliability, i.e., a minimized reliability penalty, under the given hard real-time constraints and a given number of homogeneous cores.

E. Structure of the Paper

As we use Federated Scheduling as backbone for our approach, it is briefly reviewed in Section V. In Section VI we provide a dynamic programming algorithm to find the optimal selection of redundancy levels θ, assuming the different redundancy levels to be given. After that, we explain how to determine the optimal redundancy level for each task stage in a fine-grained manner without assuming any given K_i task levels in Section VII. The approaches presented in Section VI and Section VII are evaluated in Section VIII.

V. Federated Scheduling

As we use it as backbone to solve the multi-task scheduling problem on M cores, we give a short, general overview of *Federated Scheduling* [18] that includes a short example.

In Federated Scheduling, the tasks in Γ are partitioned into subsets that are scheduled individually on a multi-core system with M homogeneous cores. To simplify the presentation, we assume that the execution levels of all tasks have been determined, i.e., θ_i is given $\forall \tau_i \in \Gamma$, and use the notation presented in this paper. Obviously a schedule for Γ on M uniform frequency cores can only exist, if the following two necessary conditions are met:

- The sum of all task utilizations is not greater than the number of processors, i.e., $\sum_{\tau_i \in \Gamma} u_{i,\theta_i} \leq M$.
- No task has a critical path greater than its period, i.e., $\forall \tau_i \in \Gamma$, $L_{i,\theta_i} \leq T_i = D_i$.

Federated Scheduling [18] partitions the tasks into two disjoint subsets according to their utilization: τ_{BIG} contains all high-utilization tasks, i.e., $u_{i,j} \geq 1$, and τ_{LITTLE} contains all low-utilization tasks, i.e., $u_{i,j} < 1$. We denote the executing levels of tasks in τ_{BIG} with θ_{BIG} and the executing levels of tasks in τ_{LITTLE} as θ_{LITTLE}. First, the number of cores necessary to schedule τ_{BIG} is determined while τ_{LITTLE} will be scheduled on the remaining cores if possible. Please note that we present *Federated Scheduling* in the general case here. Some remarks regarding our studied problem and some properties that arise due to the structure of that problem are given in the next section.

- **High-utilization tasks (τ_i in τ_{BIG}):** For each task $\tau_i \in \tau_{\text{BIG}}$, the parallel sub-tasks are scheduled on cores dedicated to the task, called list scheduling in the literature [12], by any work-conserving parallel scheduler. A work-conserving list scheduler is a scheduler that never lets a core idle if there is any sub-task ready to be executed. As shown in Theorem 2 in [18], the required number of dedicated cores H_{i,θ_i} for τ_{i,θ_i} is at most:

$$H_{i,\theta_i} = \left\lceil \frac{C_{i,\theta_i} - L_{i,\theta_i}}{T_i - L_{i,\theta_i}} \right\rceil \tag{1}$$

For the task set τ_{BIG}, we denote the sum of dedicated cores as $H_{\text{BIG}} = \sum_{\tau_i \in \tau_{\text{BIG}}} H_{i,\theta_i}$.

- **Low-utilization tasks (τ_i in τ_{LITTLE}):** We adopt R-BOUND-MP-NFR, developed by Andersson et al. [2], to schedule the tasks in τ_{LITTLE} on the number of remaining cores $H_{\text{LITTLE}} = M - H_{\text{BIG}}$. On each core of H_{LITTLE} rate-monotonic priority assignment is used. According to Theorem 7 in [2], R-BOUND-MP-NFR feasibly schedules the tasks in τ_{LITTLE} using partioned scheduling if

$$\sum_{\tau_j \in \tau_{\text{LITTLE}}} u_{j,\theta_j} \leq H_{\text{LITTLE}}/2 \tag{2}$$

This directly leads to the following sufficient schedulability test. A prove is therefore omitted.

Lemma 1: Federated Scheduling can schedule the tasks $\tau_{\text{BIG}} \cup \tau_{\text{LITTLE}}$ in Γ on M cores, if the following condition holds:

$$\sum_{\tau_i \in \tau_{\text{BIG}}} H_{i,\theta_i} + \sum_{\tau_j \in \tau_{\text{LITTLE}}} u_{j,\theta_j} \cdot 2 \leq M \tag{3}$$

Example for Federated Scheduling: We assume the redundancy level of the tasks to be given and thus drop the level indexes for C_i and L_i. Consider 5 tasks with $\{(T_1 = 10, C_1 = 12, L_1 = 5)$, $(T_2 = 2, C_2 = 4, L_2 = 1)$, $(T_3 = 20, C_3 = 2, L_3 = 1)$, $(T_4 = 30, C_4 = 6, L_4 = 3)$, $(T_5 = 12, C_5 = 6, L_5 = 4)\}$, i.e., utilization values of $u_1 = 1.2$, $u_2 = 2$, $u_3 = 0.1$, $u_4 = 0.2$, and $u_5 = 0.5$, to be scheduled on 7 processors using Federated Scheduling. In Federated Scheduling the tasks are first classified into τ_{BIG} and τ_{LITTLE} according to their utilization. Therefore, $\tau_{\text{BIG}} = \{\tau_1, \tau_2\}$ and $\tau_{\text{LITTLE}} = \{\tau_3, \tau_4, \tau_5\}$. We determine the number of dedicated cores for each task $\tau_i \in \tau_{\text{BIG}}$, i.e., $H_1 = \left\lceil \frac{12-5}{10-5} \right\rceil = 2$ and $H_2 = \left\lceil \frac{4-1}{2-1} \right\rceil = 3$.

Since $H_{\text{BIG}} = 2 + 3 = 5$, $H_{\text{LITTLE}} = M - H_{\text{BIG}} = 7 - 5 = 2$. According to Theorem 7 in [2], the tasks in τ_{LITTLE} are schedulable as Eq. (2) holds, i.e., $(0.1+0.2+0.5) \leq 2/2$. The tasks are sorted in an ascending order of their periods, i.e., $\{\tau_5, \tau_3, \tau_4\}$, and assigned to the remaining 2 cores in this order according to R-BOUND-MP-NFR. As a result, τ_5 is assigned to core 1, and $\{\tau_3, \tau_4\}$ are assigned to core 2.

VI. Redundancy Levels Selection

In this section we show how the overall reliability penalty can be optimized, assuming that we can choose from different given redundancy levels for each task. Obviously it is possible to determine the optimal selection of redundancy levels by checking all possible combinations of redundancy levels and core assignments, and choosing the combination that yields the minimum reliability penalty while satisfying the timing constraints. However this straightforward method has exponential time complexity. Instead, we propose a dynamic programming algorithm to determine the optimal redundancy level for each task while satisfying the feasibility under Federated Scheduling. We start by calculating the possible solution for $\{\tau_1\}$, use those results to calculate the possible solution for $\{\tau_1, \tau_2\}$, use those results to calculate the possible solution for $\{\tau_1, \tau_2, \tau_3\}$ and so on until we calculate the possible solution for $\{\tau_1, \tau_2, \ldots, \tau_N\}$. From the results for $\{\tau_1, \tau_2, \ldots, \tau_N\}$ we choose the one with the minimum reliability penalty.

When we calculate the possible solutions for $\{\tau_1, \tau_2, \ldots, \tau_i\}$, those solutions depend on:

- The selection of the redundancy levels θ_j for the tasks $\tau_j \in \{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$.
- The number of total required cores $m = \sum_{j=1}^{i-1} H_{j,\theta_j}$ for $\tau_j \in \{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$ where $u_{j,\theta_j} \geq 1$, i.e., τ_j that are in H_{BIG} for there selected redundancy level θ_j.
- The sum of utilizations $k = \sum_{j=1}^{i-1} u_{j,\theta_j}$ for tasks $\tau_j \in \{\tau_1, \tau_2, \ldots, \tau_{i-1}\}$ with $u_{j,\theta_j} < 1$, i.e., τ_j that are in H_{LITTLE} for there selected redundancy level θ_j.
- The redundancy level θ_i chosen for τ_i and the resulting increase of either m or k.

The necessary values are stored in two 3-dimensional tables G and j^* to record the sub-optimal reliability values and the selected redundancy levels, respectively. This means, $G(i, m, k)$ stores the minimum reliability penalty for the first i tasks, using m cores for tasks in H_{BIG} and with a total utilization of k for tasks in H_{LITTLE}, while $j^*(i, m, k)$ stores the selected redundancy level for task i. Using these values, the chosen redundancy levels for all other tasks can be traced back step by step. Those calculations have to be done for all possible combinations of m and k, i.e., m is an integer with $0 \leq m \leq M$ and k is in the range of $[0, 0.5 \cdot M]$.

When building $G(i, m, k)$ and $j^*(i, m, k)$, we assume $u_{i,j}$ and $H_{i,j}$ to be given for all redundancy levels. If this is not the case, they can be calculated in a preprocessing step.

In the initial step, i.e., when we only consider τ_1, for given values of m and k we calculate $j^*(1, m, k)$ as

$$j^*(1, m, k) = \arg \min_{j \in \{1,2,\ldots,K_1\}} \begin{cases} R_{1,j} & \text{if } u_{1,j} \geq 1 \text{ and } m \geq H_{1,j} \\ R_{1,j} & \text{if } u_{1,j} < 1 \text{ and } k \geq u_{1,j} \\ \infty & otherwise \end{cases} \tag{4}$$

leading to minimum reliability penalties of

$$G(1, m, k) = R_{1,j^*(1,m,k)}. \tag{5}$$

For the following steps we calculate the values of $G(i, m, k)$ assuming $G(i-1, m, k)$ to be given, where $i = 2, 3, \ldots, N$. This means, when we select level j for τ_i, we know that the minimum reliability penalty for task $\tau_1, \tau_2, \ldots, \tau_{i-1}$ has been calculated and stored in

- $G(i-1, m - H_{i,j}, k)$ when $u_{i,j} \geq 1$, or
- $G(i-1, m, k - u_{i,j})$ when $u_{i,j} < 1$.

Let $P_j(i, m, k)$ be the resulting reliability penalty for the selection of level j for task τ_i, defined as:

$$P_j(i,m,k)\begin{cases} R_{i,j}+G(i-1,m-H_{i,j},k) & \text{if } u_{i,j}\geq 1 \text{ and } m\geq H_{i,j}\\ R_{i,j}+G(i-1,m,k-u_{i,j}) & \text{if } u_{i,j}<1 \text{ and } k\geq u_{i,j}\\ \infty & otherwise\end{cases} \tag{6}$$

Suppose that j_i^* is the j which minimizes $P_j(i,m,k)$ for given values of m and k. This means we know that

$$G(i,m,k)=P_{j_i^*}(i,m,k) \tag{7}$$

and $j^*(i,m,k)$ is j_i^*. This calculations have to be done for all $i=2,3,\dots,n$, all integer m with $0\leq m\leq M$, and all utilization values k in the range of $[0,0.5\cdot M]$.

The pseudo-code of the presented level selection can be found in Algorithm 1, using a scaling factor ω for the utilization values, i.e., the third dimension of the table. This is necessary to upper bound the number of entries in the 3-dimensional tables and thus bounding the time needed to construct those tables. Obviously, the number of values we have to consider for the first dimension is the number of tasks N while for the second dimension we only have to consider up to M integer values. Note, that the maximum number of dedicated cores is bounded by 3 for each task.[3] However, if the utilization values are not discretized, we would have to consider an infinity number of values for the third dimension. Therefore, we discretize all utilization values based on a scale unit ω, i.e., $0<\omega\leq 1$ and all values of $u_{i,j}$ are replaced with $u_{i,j}/\omega$. Under the assumption that all scaled utilization values u_i/ω are integers, our dynamic programming approaches in Section VI and Section VII find the solution with the minimized reliability penalty that is possible when Federated Scheduling [18] is used and the schedulability of the task set under a selection of execution levels is tested based on the sufficient schedulability test in Lemma 1.

Theorem 1: Let ω with $0<\omega\leq 1$ be given and let $\frac{u_{i,j}}{\omega}$ be an integer $\forall u_{i,j}$. For all $i\in\{1,2,\dots,N\}$, $m\in\{1,2,\dots,M\}$, and $k\in\left\{1,2,\dots,\frac{0.5M}{\omega}\right\}$, Eq. (4), Eq. (5), Eq. (6), and Eq. (7) compute the optimal task redundancy level selection $j^*(i,m,k)$ and the optimal solution overall reliability penalty $G(i,m,k)$ achievable under Federated Scheduling when the sufficient schedulability test in Lemma 1 is used.

Proof: This can be proved using mathematical induction:

Base case ($i=1$): Eq. (4) calculates the optimal level for each m and k and Eq. (5) calculates the resulting minimal reliability penalty, stored in $j^*(1,m,k)$ and $G(1,m,k)$, respectively. Thus $G(1,m,k)$ and $j^*(1,m,k)$ are optimal.

Inductive step ($i\geq 2$): Assume that $G(i-1,m,k)$ and $j^*(i-1,m,k)$ are optimal for the sub-problem considering the first $i-1$ tasks for all values of $m\leq M$ and k, i.e., $G(i-1,m,k)$ stores the minimal reliability penalty value and the selected version j^* of τ_{i-1} is stored in $j^*(i-1,m,k)$ for each m and k. Suppose for contradiction that $G(i,m,k)$ is not optimal. This means, that at least one of the level selections for $\tau_1,\ \dots,\ \tau_i$ is not optimal. For each version of τ_i the penalty $P_j(i,m,k)$ for a given m and k can be calculated by adding the reliability penalty $R_{i,j}$ to either $G(i-1,m-H_{i,j},k)$ for a version where $u_{i,j}\geq 1$ or $G(i-1,m,k-u_{i,j})$ if $k\geq u_{i,j}$. If the version is not applicable, i.e., $m<H_{i,j}$ for versions with $u_{i,j}\geq 1$ or $k<u_{i,j}/\omega$ for versions with $u_{i,j}<1$, $P_j(i,m,k)$ is set to ∞. As we take the version j^* that minimizes $P_j(i,m,k)$ we know that $G(i,m,k)$ and $j^*(i,m,k)$ are calculated correctly based on $G(i-1,m,k)$ and $j^*(i-1,m,k)$. Therefore, if $G(i,m,k)$ or $j^*(i,m,k)$ is wrong for any combination of m and k, at least one of the previously selected $i-1$ task levels is not optimal, which contradicts the induction hypothesis. ■

Algorithm 1 Offline Table Construction

```
Input: $N$ tasks, $M$ cores, $\omega$ scale unit
Output: $j^*$ level selection table
 1: for $m \leftarrow 0,\dots,M$ do
 2:     for $k \leftarrow 0,\dots,\left\lceil\frac{0.5M}{\omega}\right\rceil$ do
 3:         if $m+2k\cdot\omega > M$ then
 4:             $j^*(1,m,k) \leftarrow \infty$
 5:             $G(1,m,k) \leftarrow \infty$
 6:         else
 7:             calculate $j^*(1,m,k)$ and $G(1,m,k)$ by using Equations (4) and (5)
 8:         end if
 9:     end for
10: end for
11: for $i \leftarrow 2,3,\dots,N$ do
12:     for $m \leftarrow 0,\dots,M$ do
13:         for $k \leftarrow 0,\dots,\left\lceil\frac{0.5M}{\omega}\right\rceil$ do
14:             if $m+2k\cdot\omega > M$ then
15:                 $j^*(i,m,k) \leftarrow \infty$
16:                 $G(i,m,k) \leftarrow \infty$
17:             else
18:                 for each $j \in$ possible redundancy levels do
19:                     calculate $P_j$ by using Equation (6)
20:                 end for
21:                 $j^*(i,m,k) \leftarrow \arg\min_{j=1,2,\dots,K_i} P_j$
22:                 $G(i,m,k) \leftarrow P_{j^*(i,m,k)}$
23:             end if
24:         end for
25:     end for
26: end for
```

After the tables G and j^* are calculated, the minimum value stored in $G(N,m,k)$ is the minimum penalty value. We denote this position by m_N^* and k_N^*. The redundancy level of θ_N can be found in the related entry of table j^*, i.e., at $j(N,m_N^*,k_N^*)$. From this value, we can easily trace back the redundancy levels selected for $\tau_{N-1},\tau_{N-2},\dots,\tau_1$ iteratively, i.e., if the utilization of the selected level $u_{N,\theta_N}<1$, for τ_{N-1} the selected version is stored at $j^*(N-1,m^*,k^*-\frac{u_{N,\theta_N}}{\omega})$, otherwise it is the version at $j^*(N-1,m^*-H_{N,\theta_N},k^*)$, and so on. As k is defined as $M/(2*\omega)$, the time complexity of Algorithm 1 is $O((\sum_{i=1}^{N}|K_i|)\cdot M^2/\omega)$ and the space complexity is $O(NM^2/\omega)$.

If the scaling factor that is necessary to ensure that all

[3]In the general case (explained in Section V) the number of cores needed for the execution of a single task can be arbitrary large, depending on the relation of $C_{i,\theta_i}-L_{i,\theta_i}$ to T_i-L_{i,θ_i}. However, the number of processors needed to activate CRT-TMR is bounded by 3 due to the assumption that $L_{i,\theta_i}\leq T_i$ and at most 3 instances are executed in parallel. If the fine-grained selection in Section VII is used, still none of the sequential stages will be executed more then 3 times in parallel and thus the bound of 3 still holds.

utilization values are integers is too small, the number of entries that have to be considered in the table would be too large. To avoid this, a ceiling function can be used when calculating the utilization values for a given scale unit ω, i.e, all values of $u_{i,j}$ are replaced with $\lceil u_{i,j}/\omega \rceil$. This leads to a trade-off between the accuracy of our dynamic programming approach on one hand and the space and the time complexity on the other hand.

That the dynamic programming finds the optimal solution under Federated Scheduling using the schedulability test in Lemma 1 implicates that better reliability penalties can be achieved if other scheduling approaches or tighter schedulability tests are used. This also means that in some cases other scheduling strategies can perform better as shown in Section VIII. However, our approach in general is not limited to Federated Scheduling and the schedulability test in Lemma 1. It can be applied for other strategies and tests by reformulating Eq.(4), Eq.(5), Eq.(6), and Eq.(7) accordingly if the suboptimality to construct an optimal solution to schedule the first i tasks on m cores can be achieved by referring to the optimal schedules of the first $i-1$ tasks on m' processors (with $m' \leq m$) in a similar manner.

One specific example is to adopt semi-partitioned scheduling instead of partitioned scheduling for both the tasks in τ_{BIG} and τ_{LITTLE}. Rate-Monotonic Scheduling with Task Splitting (RM-TS) as proposed in [13] can be applied for the tasks in τ_{LITTLE} under Federated Scheduling. In this case, Eq.(2) should be reformulated as

$$\sum_{\tau_j \in \tau_{\text{LITTLE}}} u_{j,\theta_j} \leq H_{\text{LITTLE}} \cdot \frac{2\Theta(\Gamma)}{1+\Theta(\Gamma)} \tag{8}$$

where $\Theta(\Gamma) = N(2^{1/N}-1)$. Eq. (8) directly leads to the following sufficient test. A prove is therefore omitted.

Lemma 2: Federated Scheduling can schedule the tasks $\tau_{\text{BIG}} \cup \tau_{\text{LITTLE}}$ in Γ on M cores, if the following condition holds:

$$\sum_{\tau_i \in \tau_{\text{BIG}}} H_{i,\theta_i} + \sum_{\tau_j \in \tau_{\text{LITTLE}}} u_{j,\theta_j} \cdot \frac{1+\Theta(\Gamma)}{2\Theta(\Gamma)} \leq M. \tag{9}$$

Please note that Eq.(4), Eq.(5), Eq.(6), and Eq.(7) should be reformulated accordingly. The corresponding results derived by using Federated Scheduling with Eq.(8) and Eq.(9) are also presented in Section VIII for completeness.

For the tasks in τ_{BIG} it is possible that nearly 50% of the utilization of 3 cores is wasted when CRT is used, i.e., when one activation of a task has a utilization slightly bigger than 0.5.[4] In this case using MRT-TMR directly is not possible as two complete activations of the task cannot be placed on the same core. However, using MRT-TMR under semi-partitioned scheduling is possible as long as the total utilization of 3 activations is below 2, i.e, by starting the original on core 1 and one replica on core 2, preempting the replica on core 2 after 50% of the replica is executed, starting the second replica on core 2 and finishing the previously preempted first replica on core 1 after the original task is finished.

[4]We neglect the workload due to the synchronization in this example as it only adds additional complexity in the description without adding any insight.

In addition, using MRT-TMR for tasks in τ_{BIG} may also result in unbalanced utilization of CPUs and the waste of nearly 50% of the utilization of two CPUs in a similar scenario. This utilization could be used by other tasks. However, balancing the utilization of CPUs is not the main focus of this work. Our goal is to explain the approach in general and to show its effectiveness. Therefore we use well known techniques, i.e., Federated Scheduling and list scheduling.

VII. Fine-Grained Selection

In this section we extend our approach from selecting among given redundancy levels to a more fine-grained approach where different stages of the task execution can be hardened individually. For each given task τ_i, we assume that it has S_i sequential stages, e.g., a function or a basic block, that can be hardened by redundancy individually as in the fork-join task model adopted by Axer et al. [3]. This means, we can decide whether we run a stage of a task with TMR, DMR, or without any redundancy. If a stage is executed with redundancy, the task execution is forked at the beginning of this stage and joined at the end of this stage.

If task τ_i's stage s is executed in the redundancy level $\theta_{i,s} \in \{\phi, \text{DMR}, \text{TMR}\}$ we assume its critical-path length, WCET, and reliability penalty are all given with corresponding mapping functions, i.e., $L_i(s,\theta)$, $C_i(s,\theta_{i,s})$, and $\mathbb{R}_i(s,\theta_{i,s})$, respectively, thus its utilization $\mathbb{U}_i(s,\theta_{i,s})$ is $C_i(s,\theta_{i,s})/T_i$. Our objective is to select the redundancy level $\theta_{i,s}$ for each task's stage that minimize the overall reliability penalty while satisfying the given timing constraints.

Definition 2 (Overall Reliability Penalty Ψ'_Γ): The overall reliability penalty of task set Γ, denoted by Ψ'_Γ, is the sum of the tasks' reliability penalties given by $\Psi'_\Gamma = \sum_{\tau_i \in \Gamma} R_{i,\theta_i}$, where R_{i,θ_i} is the reliability penalty of task τ_i for a given selection $\theta_i = \{\theta_{i,1}, \theta_{i,2}, \ldots, \theta_{i,s}\}$ of stage redundancy levels.

A. Preprocessing

To make the final scheduling/task partition decision, the number of cores for τ_{BIG} and utilization for τ_{LITTLE} are both required. We prepare two reference tables $\hat{U}$ and $\hat{C}$ to record the optimal reliability penalty under given resource constraints and refer to both tables to obtain the best redundancy for each task stage. When for a stage s of τ_i a redundancy level $\theta_{i,s}$ is selected, we have to consider two possibilities: 1) $\tau_i \in \tau_{\text{LITTLE}}$ and the constraint is the utilization of the task, or 2) $\tau_i \in \tau_{\text{BIG}}$ and the constraint is the number of dedicated cores.

Utilization Demand Table: For the first case, we prepare a table $\hat{U}(i,k)$ to record the optimal reliability penalty of τ_i with given utilization k and a corresponding cost table $\mathbb{K}(i,k)$ for recording the exact required utilization of $\hat{U}(i,k)$. Again we use ω to scale the utilization values and assume that all the scaled utilization values u_i/ω are integers. To find the optimal reliability penalty in $\hat{U}(i,k)$, we use a stage-wise dynamic programming algorithm. Therefore, we construct two 2-dimensional tables $s_i^u(s,k)$ and $q_i^u(s,k)$. The first dimension saves the considered stages and therefore is in the range $[1, S_i]$ while the second dimension depends on the corresponding utilization values k in the range of $[0, 1/\omega]$. In

Algorithm 2 Preprocessing-utilization

```
Input: $N$ tasks
Output: Utilization-grained table $\hat{U}$
1: for $i \leftarrow 1, 2, \dots, N$ do
2:     for $k \leftarrow 0, 1, \dots, 1/\omega$ do
3:         $s_i^u(1,k) \leftarrow$ by using Eq.(10)
4:         $q_i^u(1,k) \leftarrow \mathbb{R}_i(1, s_i^u(1,k))$
5:         for $s \leftarrow 2, 3, \dots, S_i$ do
6:             calculate $s_i^u(s,k)$ by using Eq.(11)
7:             $q_i^u(s,k) \leftarrow \mathbb{R}_i(s,j) + q_i^u(s-1, k-\mathbb{U}_i(s,j))$
8:         end for
9:         $\hat{U}(i,k) \leftarrow q_i^u(S_i,k)$
10:        $\mathbb{K}(i,k) \leftarrow$ backtrack with table $s_i^u$ and $\mathbb{U}_i$
11:    end for
12: end for
```

Algorithm 3 Preprocessing-cores

```
Input: $N$ tasks, $M$ cores
Output: Cores-grained table $\hat{C}$
1: for $i \leftarrow 1, 2, \dots, N$ do
2:     calculate $\xi_i^{\max}$ and $l_i^{\max}$ with all stages in TMR
3:     for $s \leftarrow 1, 2, \dots, S_i$ do
4:         for $\xi \leftarrow 0, 1, \dots, \xi_i^{\max}$ do
5:             for $l \leftarrow 0, 1, \dots, l_i^{\max}$ do
6:                 calculate $s_i^c(s,\xi,l)$ by using Eq.(12) and Eq.(13)
7:                 $q_i^c(s,\xi,l) \leftarrow \mathbb{R}_i(s, s_i^c(s,\xi,l))$
8:             end for
9:         end for
10:    end for
11:    $\hat{C}(i,0) \leftarrow \infty$
12:    $\mathbb{H}(i,0) \leftarrow \infty$
13:    for $m \leftarrow 1, \dots, M$ do
14:        $\hat{C}(i,m) \leftarrow$ by using Eq.(14)
15:        $\mathbb{H}(i,m) \leftarrow$ by $\left\lceil \frac{\xi'-l'}{T_i-l'} \right\rceil$ with corresponding $\xi'$ and $l'$
16:    end for
17: end for
```

each stage s, all levels $j \in \{\phi, \text{DMR}, \text{TMR}\}$ are considered with their corresponding utilization $\mathbb{U}_i(s,j)$ and reliability penalty $\mathbb{R}_i(s,j)$. The pseudo code of the preprocessing is provided in Algorithm 2. For the first stage, we find the level $j_{i,1}^*$ with the minimal reliability penalty for each k in $[0, 1/\omega]$ and record $j_{i,1}^*$ in the $s_i^u(1,k)$ entry:

$$s_i^u(1,k) = \arg \min_{j \in \{\phi, \text{DMR}, \text{TMR}\}} \begin{cases} \mathbb{R}_i(1,j) & \text{if } k \geq \mathbb{U}_i(1,j) \\ \infty & otherwise \end{cases} \tag{10}$$

The corresponding reliability penalty $\mathbb{R}_i(1, s_i^u(1,k))$ is recorded in $q_i^u(1,k)$. For the following stage $s = 2, 3, \dots, S_i$

$$s_i^u(s,k) = \arg \min_{j \in \{\phi, \text{DMR}, \text{TMR}\}}$$
$$\begin{cases} \mathbb{R}_i(s,j) + q_i^u(s-1, k-\mathbb{U}_i(s,j)) & \text{if } k \geq \mathbb{U}_i(s,j) \\ \infty & otherwise \end{cases} \tag{11}$$

After all the entries in table q_i^u are calculated, we can find the optimal selection for each task with utilization k and record the exact utilization demand (scaled up by ω) in $\mathbb{K}(i,k)$. Both the time and the space complexity of Algorithm 2 are $O((\sum_{i=1}^{N} S_i) \cdot N/\omega)$.

Core Demand Table: For the second case, we prepare a core-level table $\hat{C}(\tau_i, m)$ to record the optimal reliability penalty and record the number of required cores in table $\mathbb{H}(i,m)$. Similarly, we prepare a stage-wise table $s_i^c(s,\xi,l)$ to find the stage redundancy $j \in \{\phi, \text{DMR}, \text{TMR}\}$ with the minimal reliability penalty under the critical length l and the worst-case execution time ξ constraints, where $\xi \geq l \geq L_i(s, s_i^c(s,\xi,l))$. As each task τ_i in τ_{BIG} is assigned to $\left\lceil \frac{C_i-L_i}{T_i-L_i} \right\rceil$ cores in Federated Scheduling, we intend to find the optimal redundancy selection for each stage s by which the sum of the critical length l and the total execution time ξ among all the stages does not exceed L_i and C_i, respectively. The pseudo code is shown in Algorithm 3. Here we calculate the maximal critical length $l_i^{\max}$ and the maximal total execution time $\xi_i^{\max}$ by profiling each task τ_i's critical length $\sum_{s=1}^{S_i} L_i(s, \theta_{i,s})$ and total execution time $\sum_{s=1}^{S_i} C_i(s, \theta_{i,s})$ with TMR, respectively, on all its stages. We start from the first stage with the minimal reliability penalty and record the redundancy level in $s_i^c(1,\xi,l)$:

$$s_i^c(1,\xi,l) = \arg \min_{j \in \{\phi, \text{DMR}, \text{TMR}\}}$$
$$\begin{cases} \mathbb{R}_i(1,j) & \text{if } \xi \geq C_i(1,j) \text{ and } l \geq L_i(1,j) \\ \infty & otherwise \end{cases} \tag{12}$$

Its corresponding $R_i(1, s_i^c(1,\xi,l))$ reliability penalty is recorded in $q_i^c(1,\xi,l)$. For the following stage $s = 2, 3, \dots, S_i$

$$s_i^c(s,\xi,l) = \arg \min_{j \in \{\phi, \text{DMR}, \text{TMR}\}}$$
$$\begin{cases} \mathbb{R}_i(s,j) + \mathbb{R}_i^t & \text{if } \xi \geq C_i(s,j) \text{ and } l \geq L_i(s,j) \\ \infty & otherwise \end{cases} \tag{13}$$

where $\mathbb{R}_i^t = q_i^c(s-1, \xi - C_i(s,j), l - L_i(s,j))$ and the reliability penalty is recorded in $q_i^c(s,\xi,l)$. After all the entries in q_i^c are calculated, we can find a certain combination of ξ' and l' under the condition that $\xi \geq l \geq L_{i,\phi}$ to obtain the minimal reliability penalty $\hat{C}(i,m)$, defined as:

$$\hat{C}(i,m) = \min \begin{cases} q_i^c(S_i,\xi,l) & \text{if } m \geq \left\lceil \frac{\xi-l}{T_i-l} \right\rceil \text{ and } \xi \geq l \geq L_{i,\phi} \\ \infty & otherwise \end{cases} \tag{14}$$

where $L_{i,\phi}$ is the critical length of task τ_i without any redundancy. For each m and i combination, those ξ' and l' are recorded to $\xi_i^m = \xi'$ and $l_i^m = l'$. The time complexity of Algorithm 3 is $O((\sum_{i=1}^{N} S_i) \cdot \xi l)$ while the space complexity is $O(NM\xi l)$.

B. Selecting and Scheduling

Using the reference tables $\hat{U}$ and $\hat{C}$, our fine-grained approach builds two 3-dimensional tables $j^*(i,m,k)$ and $G(i,m,k)$ to record the sub-optimal selections of task τ_i and the resulting penalty values, respectively, for m dedicated cores and utilization, presented as pseudo-code in Algorithm 4. We use the tables $\hat{U}$ and $\hat{C}$ to find the selection with the minimal reliability penalty between the fine grained versions that are in τ_{BIG} or τ_{LITTLE}, i.e., with utilization > 1 and utilization ≤ 1, respectively. Again, all the possible combinations of utilization value k and number of available cores m have to be checked for all tasks, i.e., $0 \leq k \leq 1/\omega$ and $1 \leq m \leq M$.

Algorithm 4 Fine-Grained Table Construction

```
Input: $N$ tasks, $M$ cores, $\hat{U}$ and $\hat{C}$ fine-grained tables;
 1: for $m \leftarrow 0, \dots, M$ do
 2:     for $k \leftarrow 0, \dots, \left\lceil \frac{0.5M}{\omega} \right\rceil$ do
 3:         $G(1,m,k) \leftarrow \min \left\{ \hat{U}(1,k), \hat{C}(1,m) \right\}$
 4:         $j^*(1,m,k) \leftarrow \theta_{1,m,k}$
 5:     end for
 6: end for
 7: for $i \leftarrow 2, 3, 4, \dots, N$ do
 8:     for $m \leftarrow 0, \dots, M$ do
 9:         for $k \leftarrow 0, \dots, \left\lceil \frac{0.5M}{\omega} \right\rceil$ do
10:             if $m + 2k \cdot \omega > M$ then
11:                 $G(i,m,k) \leftarrow \infty$;
12:             else
13:                 $P^*_{\text{BIG}}(i,m,k) \leftarrow \infty$
14:                 for $m' \leftarrow 0, \dots, m$ do
15:                     $P(i,m',k) = \hat{C}(i,m') + G(i-1, m-\mathbb{H}(i,m'), k)$
16:                     $P^*_{\text{BIG}}(i,m,k) = \min \left\{ P^*_{\text{BIG}}(i,m,k), P(i,m',k) \right\}$
17:                 end for
18:                 $P^*_{\text{LITTLE}}(i,m,k) \leftarrow \infty$
19:                 for $k' \leftarrow 0, \dots, k$ do
20:                     $P(i,m,k') = \hat{U}(i,k') + G(i-1, m, k-\mathbb{K}(i,k'))$
21:                     $P^*_{\text{LITTLE}}(i,m,k) =$
                        $\min \left\{ P^*_{\text{LITTLE}}(i,m,k), P(i,m,k') \right\}$
22:                 end for
23:                 $G(i,m,k) = \min \left\{ P^*_{\text{BIG}}(i,m,k), P^*_{\text{LITTLE}}(i,m,k) \right\}$
24:                 $j^*(i,m,k) \leftarrow \theta_{i,m,k}$
25:             end if
26:         end for
27:     end for
28: end for
```

For the first task τ_1 (lines 1-5 in Algorithm 4) the minimum reliability penalty for each m and each k can be calculated as:

$$G(1,m,k) = \min \left\{ \hat{U}(1,k), \hat{C}(1,m) \right\} \tag{15}$$

For the other tasks, i.e., τ_i with $i > 1$ (lines 6-26 in Algo. 4), for each combination of m and k we need to consider all possible k' with $0 \leq k' \leq k$ to select the best achievable penalty when a selection θ_i with $\mathbb{U}_i(s, \theta_{i,s}) \leq 1$ is chosen and all possible m' with $1 \leq m' \leq m$ to select the best achievable penalty when a selection θ_i with $\mathbb{U}_i(s, \theta_{i,s}) > 1$ is chosen.

As a result, the best possible selection $P^*_{\text{LITTLE}}(i,m,k)$ for a solution with $\mathbb{U}_i(s, \theta_{i,s}) \leq 1$ can be found as:

$$P^*_{\text{LITTLE}}(i,m,k) = \min_{0 \leq k' \leq k} \left\{ \hat{U}(i,k') + G(i-1, m, k - \mathbb{K}(i,k')) \right\} \tag{16}$$

The best possible selection $P^*_{\text{BIG}}(i,m,k)$ for a solution with $\mathbb{U}_i(s, \theta_{i,s}) > 1$ can be found as:

$$P^*_{\text{BIG}}(i,m,k) = \min_{1 \leq m' \leq m} \left\{ \hat{C}(i,m') + G(i-1, m - \mathbb{H}(i,m'), k) \right\} \tag{17}$$

Therefore, for τ_i with $i > 1$ the best possible selection for m and k is:

$$G(i,m,k) = \min \left\{ P^*_{\text{BIG}}(i,m,k), P^*_{\text{LITTLE}}(i,m,k) \right\} \tag{18}$$

Please note, that $P^*_{\text{BIG}}(i,m,k)$ and $P^*_{\text{LITTLE}}(i,m,k)$ and therefore $G(i,m,k)$ may be ∞ for some values of m and k. The number of dedicated cores or the utilization of the chosen reliability selection θ^*_i is stored in $j^*(i,m,k)$.

Afterwards, the table $G(i,m,k)$ contains the optimal reliability values for each combination of i, m, and k. It contains entries with value ∞ if the condition of the schedulability test in Lemma 1 does not hold, i.e., $m + 2k \cdot \omega > M$. Note, that some other values may be ∞ as well, if the combination of the number of processors m and the utilization value k is too small to schedule any selection redundancy stages, i.e, if $m = 0$ and the sum of the utilizations of the tasks versions with no redundancy at all is larger than k. We search for the entry $G(N, m^*, k^*)$ with the minimal reliability penalty. Based on the related entry in $j^*(N, m^*, k^*)$ we know the chosen selection θ^*_N and can backtrack to the entry in $j^*(N-1, m, k)$ etc. The time and space complexity of Algorithm 4 both are $O(NM^2/\omega)$. To schedule tasks in Γ with their redundancy selection the tasks are classified in τ_{BIG} and τ_{LITTLE} and scheduled using Federated Scheduling.

VIII. Result and Discussion

We analyzed the performance of both the coarse-grained and the fine-grained approaches compared to the greedy approach **dTune** [25] with respect to the number of feasible configurations and by comparing the best resulting reliability penalties among those feasible configurations, using real tasks from an embedded benchmark.

A. Experimental Setup

In this subsection we describe the setup we use for the embedded benchmark when evaluating our approaches and the greedy approach **dTune** [25] we compare to.

For the task set we analyzed, seven tasks were chosen from the embedded benchmark MiBench [14]: (1) SAD, (2) ADPCM, (3) CRC, (4) SusanS, (5) SHA, (6) SATD, and (7) DCT. All tasks were compiled with a reliability-driven GCC-based compiler and extended with several reliability-driven transformations [24, 28] and instruction scheduling algorithm [27], by which redundancy levels of each task were generated based on its DAG abstraction. For each compiled level, we determined the WCET by a large number of measurements and estimated the reliability penalty values under two different fault rates η, i.e., $\eta = 10^{-6}$ and 10^{-7} $fault/cycles$, as adopted in [15, 19]. Similarly to [15, 19], the reliability penalty for each function/task is estimated using the same metrics as in [23, 28]. The time overhead of the synchronization for DMR and TMR redundancy levels are integrated into the total execution time $C_{i,j}$ and the critical-path length $L_{i,j}$ for level $\tau_{i,j}$. From those values we generated the tasks period using two different approaches:

- *Random Periods:* We randomly drew T_i in the range of $[1.65 \cdot C_{i,\phi}, (1.65+\rho) \cdot C_{i,\phi}]$ using a uniform distribution, i.e., analyzing task sets with smaller / larger periods compared to the execution time and therefore larger / smaller utilization values. The ρ values we used in the experiments were 0.6, 1.2, and 1.8, leading to upper bounds of 2.25, 2.85, and 3.45, respectively, for the periods compared to the execution time. We generated 500 task sets for each ρ to get a sufficient sample size.

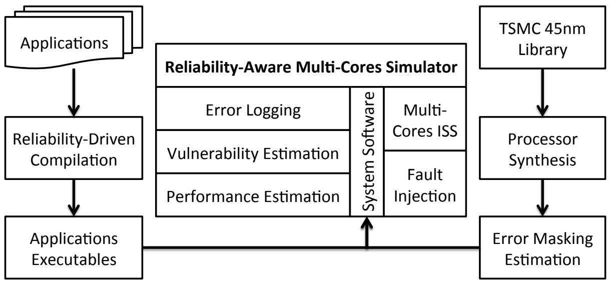


Fig. 4: Experimental setup with reliability-driven compilation, system software, and multi-cores simulator

- *Given Total Utilization:* We applied the UUnifast-Discard method proposed by Davis et al. [11] to generate task sets of size N with a given total utilization U^* for the execution without any redundancy, i.e., $U^* = \sum_{\tau_i \in \Gamma} u_{i,\phi}$. Each task τ_i is assigned with a utilization value $u_{i,\phi}$ and the period is $T_i = C_{i,\phi}/u_{i,\phi}$, i.e., the the execution time of the non-RMT level divided by the assigned utilization from the UUnifast-Discard method. We evaluated using three different values of U^*, i.e., 250%, 300%, and 350% creating 500 task sets for each utilization value.

As **dTune** only can decide to activate CRT-TMR or not, it needs at least 9 cores to activate CRT-TMR for one task and 21 cores to activate CRT-TMR for all tasks. Therefore, we evaluated the range of 9 to 21 cores.

For the setting of U^*, we choose the range of 250% to 350% due to the following reasons: (1) When U^* is larger than 350%, the average utilization of a task is more than 50%. Therefore most tasks cannot activate SRT-TMR or MRT-TMR due to the additional computations for replicas and synchronization. (2) If the total utilization is less than 250%, the average task utilization is below 33% and therefore most tasks can activate SRT-TMR.

We adopted a reliability-aware many-core simulator, integrated with fault generation and injection modules, i.e., exactly the same simulator as in [7]. For each core, the *SPARC-v8 ISA* is implemented for synthesizing the *LEON3* processor. The *Synopsys Design Compiler* with the *TSMC* 45 *nm* library was used to obtain area, frequency, and logical masking probabilities to get an accurate estimation. As detailed in [23, 28], these probabilities are used to obtain the instruction vulnerabilities that can estimate the reliability penalty for each task. The fault model (e.g., fault rate, fault distribution, etc.) and the processor synthesis information were used as the input to the many core simulator in the fault scenario generations. Those realistic fault rates were obtained by using the *neutron flux calculator [1]* and coordinates of a given location, fault distribution, etc. During the execution of a given function version, transient faults are randomly injected in different processors as done in [22, 31]. The impacts of faults on the application output are monitored, and soft errors are classified by the severity from the user's perspective. Overall, the results of the fault injection experiments are analyzed to accomplish two objectives: (1) estimating the software-level vulnerability and masking properties; (2) analyzing the main reasons for system failures. An overview of the experimental setup is provided in Figure 4.

We implement our dynamic programming approaches with C++, i.e., Algorithm 1 and Algorithm 4, using an *Intel Core i7-4770 CPU* with 16GB DDR3 RAM for the evaluations. The derived reliability penalties of Algorithm 1 and Algorithm 4 with Lemma 1 (partitioned scheduling) and Lemma 2 (semi-partitioned scheduling) are compared with **dTune** [25][5] by evaluating the number of feasible configurations and the delivered overall reliability penalties $\Psi_\Gamma(\theta)$. The greedy approach **dTune** [25] works as follows:

- The tasks are sorted by their reliability penalty $R_{i,\phi}$.
- The $\left\lfloor \frac{M-N}{2} \right\rfloor$ tasks with the highest reliability penalty are selected to activate CRT-TMR.
- The remaining tasks are executed in NON-RMT.

B. Evaluation of the Coarse-Grained Approach

To show that Mixed Redundant Threading (MRT) provides additional possibilities for reliability optimization we adopted a similar comparison as in [7]. We restrict the coarse-grained approach to only use Triple Modular Redundancy (TMR) with partition scheduling and semi-partition scheduling, as **dTune** can only choose between CRT-TMR and no redundancy as well. For each of the seven tasks, TMR can be activated individually, leading to $2^7 = 128$ configurations. We report the number of feasible configurations for both the coarse-grained approach (Algorithm 1) and **dTune** [25]. The average analysis duration for one run of the coarse-grained approach considering 7 tasks was 0.22 seconds. As results under different fault rates η are similar, we only present the results for $\eta = 10^{-6}$.

Figure 5 shows the number of feasible configurations with respect to ρ and the number of cores M. A log scale with base-10 is used for the Y-axis to improve the readability. The number of feasible configurations using the coarse-grained approach is generally not less than for the greedy approach. Some exceptions are the cases using partitioned-scheduling under tight constraints, e.g., 9 cores and $\rho = 0.6$. This is due to the limitation of the sufficient bound of the partitioned-scheduling. As all tasks without redundancy are by construction in τ_{LITTLE} and the sufficient bound only accepts tasks with a total utilization of $\sum_{\tau_i \in \tau_{\text{LITTLE}}} u_{i,\phi} \leq M/2$ some task sets are barely schedulable without any redundancy. When the parameter ρ is increased, i.e., the utilization of tasks are relatively lower, the coarse-grained approach adopts SRT/MRT-TMR to exploit the spare utilization of cores, while the greedy approach can only decide if CRT-TMR should be activated or not without any adaptation.

As shown in Figure 6, for different U^* settings, the number of feasible configurations using the the coarse-grained approach are generally superior to the greedy approach when the number of available cores is less than 18. However, we can see that our approach does not always outperform the greedy approach if the number of available cores is sufficient to activate CRT-TMR for many tasks. For example, when the

[5]Although the task mapping approach in [7] outperforms the greedy approach in [25] under variation-performance multi-cores, the approaches from both papers will perform the same in our studied problem as all the cores in the considered system are homogeneous.

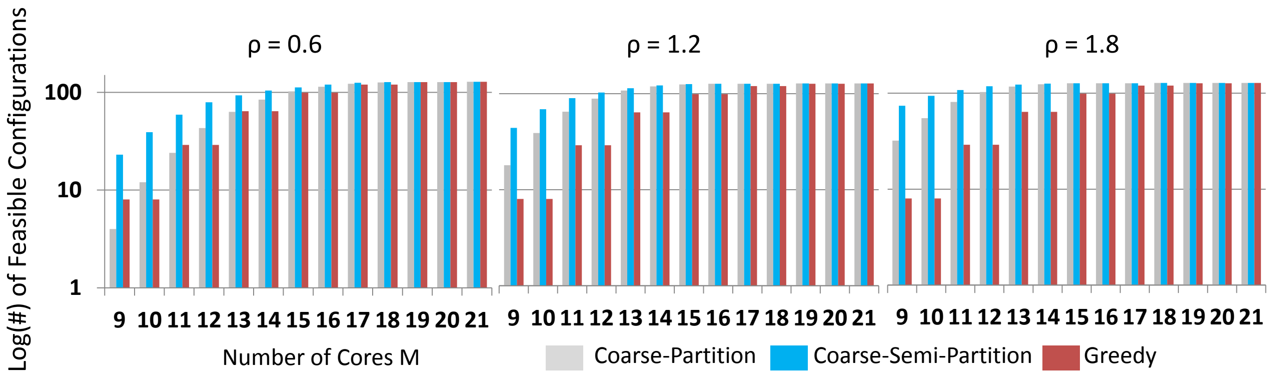


Fig. 5: Number of feasible configurations for different ρ.

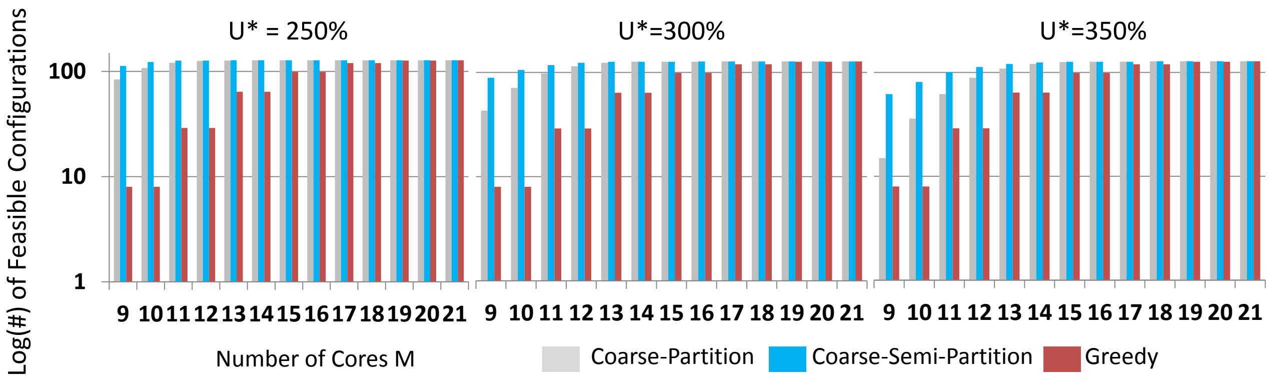


Fig. 6: Number of feasible configurations for different U^*.

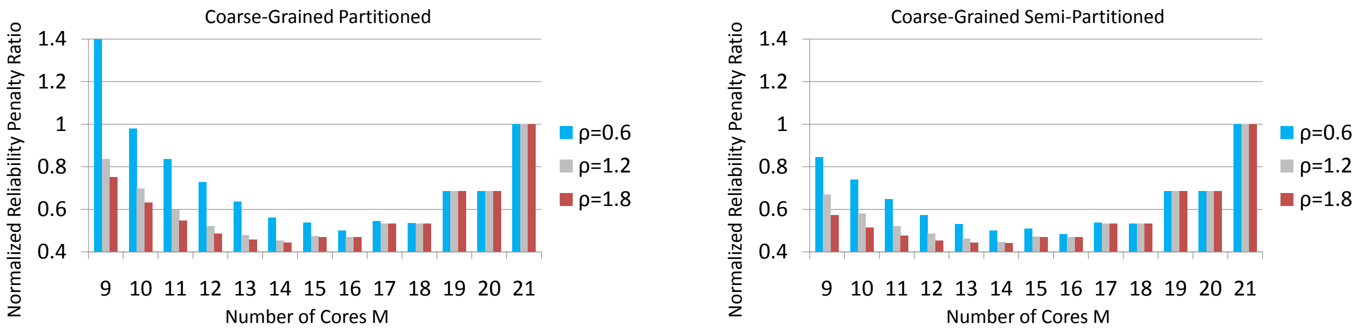


Fig. 7: Comparison of the coarse-grained approach and the greedy approach under different ρ.

number of available cores is 21 and U^* is close to 350%, the coarse-grained approach does not outperform the greedy approach. However, these results strongly support our claim that using Mixed-Redundant Threading is able to increase the schedulability for given selections of TMR.

Figure 7 shows the normalized ratio of overall system reliability, which is calculated as Ψ_Γ of the resulting solution divided by Ψ_Γ of the greedy approach. To generate the reliability penalties, $\eta = 10^{-6}$ was used. Instead of using the given redundancy configurations in the previous evaluation, the greedy approach decides which task can be executed with CRT-TMR. Our coarse-grained approach determines the executed redundancy level of each tasks. For the sake of fairness, we only compare those task sets where both approaches are able to provide feasible mappings. By the definition of the penalty value, lower values are better. Generally, most of the results delivered by the coarse-grained approach are better than the greedy approach results. However, when M is 9, we can see that our approach cannot outperform the greedy approach when partitioned scheduling is used in the coarse-grained approach. From the previous evaluation we know that our approach can find some feasible mappings, but the best one it can find is limited by the sufficient condition of Federated Scheduling and in this case the coarse-grained approach can barely execute the tasks without redundancy. For 12 up to 20 cores the coarse-grained approach achieves a significantly better reliability penalty than the greedy approach. For the normalized reliability penalty ratio displayed in Figure 8, we observe that the gain by using the coarse-grained approach is larger when the overall utilization is smaller. If semi-partitioned scheduling is used, the gain is higher when the number of available cores is in the range between 9 to 11

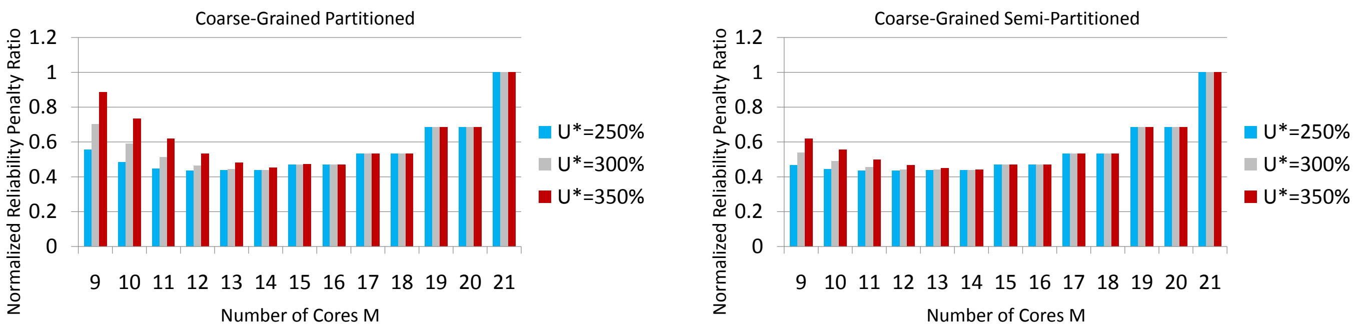


Fig. 8: Comparison of the coarse-grained approach and the greedy approach under different U^*.

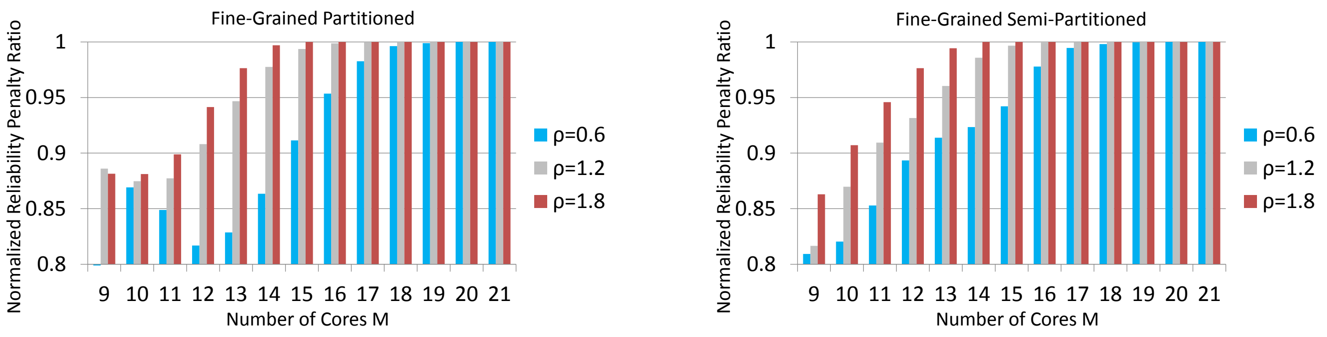


Fig. 9: Comparison of the fine-grained approach and the coarse-grained approach under different ρ.

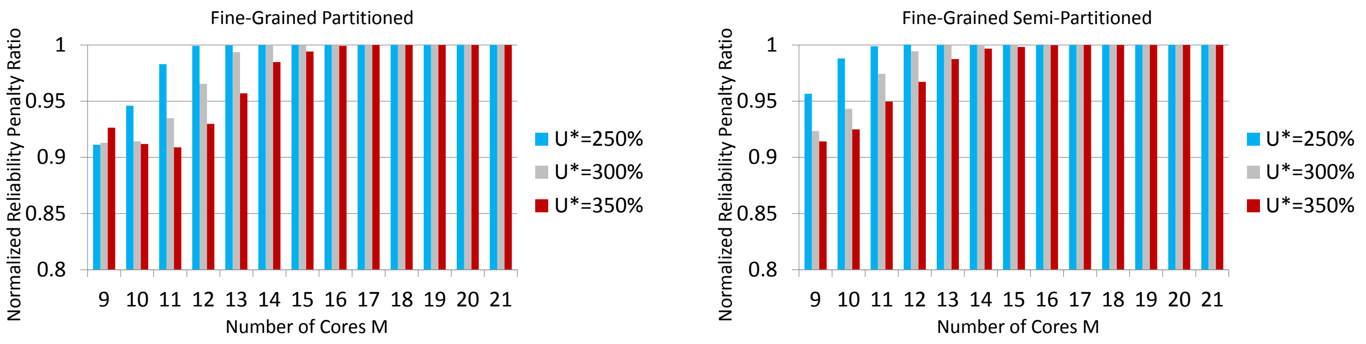


Fig. 10: Comparison of the fine-grained approach and the coarse-grained approach under different U^*.

especially for $U^* = 350$. Overall, as our approach can fully exploit the available cores with SRT or MRT rather than solely using CRT, it specifically performs well when the number of available cores is in the most interesting region, i.e., TMR can be activated for some tasks but not for most/all.

C. *Evaluation of the Fine-Grained Approach*

We compare the results of the fine-grained approach (Algorithm 4) with the coarse-grained approach to show the possible benefit if the stage redundancy can be determined arbitrarily. We show the comparison to the coarse-grained approach here as the fine-grained approach will always perform at least as good as the coarse-grained approach when compared to the greedy approach and the coarse-grained approach already performs at least as good as the greedy approach in all settings beside $M = 9$ and $\rho = 0.6$. That the fine-grained approach performs as least as good as the coarse-grained approach is due to the fact that the option to harden the complete task, which is the only option of the coarse-grained approach, is always considered by the fine-grained approach as well.

Figure 9 and Figure 10 show the average system reliability values achieved by the fine-grained approach over 500 runs, normalized by the values achieved by the coarse-grained approach. The number of task stages S_i was drawn uniformly distributed in the range of 2 to 5. Generally, the more stages a task has, the more flexibility for using redundancy it has. Please note, that when S_i is 1 for each task τ_i, the fine-grained approach is the same as the coarse-grained approach. The average analysis duration for one run of the fine-grained approach considering 7 tasks was 3.8 seconds.

In Figure 9, we see that the fine-grained approach improves the results of the coarse-grained approach more when ρ is smaller. While the coarse-grained approach with partitioned scheduling suffers from the small number of available cores which does not allow to harden many complete tasks if the utilization is high, i.e., $M = 9$ and $\rho = 0.6$, the fine-grained approach can harden some stages of the tasks with redundancy. In this case, the normalized ratio between the fine-grained and the coarse-grained approaches is 0.6292, but the bar is not shown due to the scale of Y-axis. However, we choose to use the scale from 0.8 to 1.0 as it improves the readability for the other settings. Under semi-partitioned scheduling the gain of the fine-grained approach is not as large as under the coarse-grained approach, since the coarse-grained approach can activate some redundancy already for some tasks due to a larger sufficient bound in Lemma 2.

In Figure 10, we observe that the benefit of using the fine-grained approach is not as large as in Figure 9, especially under semi-partitioned scheduling. The reason is that if the coarse-grained approach can already activate TMR for most complete tasks, the fine-grained approach can only perform as good as the coarse-grained approach for those tasks.

Overall, the results of the two proposed approaches are generally better than the greedy approach in terms of reliability when the number of available cores is too limited to activate CRT-TMR for all tasks. Furthermore, since the fine-grained approach has more flexibility to harden tasks in stage-level, the decrease of the system reliability penalty is at least as good as for the coarse-grained approach. When the resources are more limited, the benefit of adopting the fine-grained approach is more significant. While our approach already performs better than the greedy approach in most cases if partitioned scheduling is used for the tasks in τ_{LITTLE}, using semi-partitioned scheduling can increase the gain even further.

IX. Conclusion and Future Work

As multi-core systems have become the mainstream processors, exploiting redundant cores to mitigate soft-error effects by using RMT is a reasonable solution. This work provides software synthesis methodologies for real-time embedded system designers to efficiently exploit mixed redundancy techniques to decrease the system reliability penalty while satisfying the timing constraints in multi-core systems. We provide a combination of CRT and SRT called Mixed-Redundant Threading to achieve these goals. In addition, we provide two reliability optimization approaches to decrease the system reliability penalty with different granularity while scheduling the hard real-time tasks on multi-cores.

To the best of our knowledge, this paper provides the first solid foundation for using mixed redundancy techniques in multi-core systems. The methodologies in this paper are not limited to the mixture of two redundancy levels but also applicable to multiple redundancy levels up to the designer's choice. However, our study is limited to implicit-deadline real-time tasks under federated scheduling in homogeneous multi-core systems. It has been recently shown by Chen [6] that Federated Scheduling does not perform well for constrained- and arbitrary-deadline real-time task systems. Nevertheless, the proposed Mixed-Redundant Threading can directly be used for other scheduling strategies while the applicability of the proposed optimization techniques depends on the scheduling strategy that is used.

Acknowledgments

This research is supported in parts by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 - spp1500.itec.kit.edu). The authors thank our partner, Prof. Dr.-Ing. Muhammad Shafique, for providing the simulator and the data. Furthermore, we thank the anonymous reviewers for their suggestions on improving this paper.

References

[1] Flux calculator. http://www.seutest.com/cgi-bin/FluxCalculator.cgi.

[2] Andersson, B. and Jonsson, J. (2003). The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 33–40.

[3] Axer, P., Quinton, S., Neukirchner, M., Ernst, R., Dbel, B., and Hrtig, H. (2013). Response-time analysis of parallel fork-join workloads with real-time constraints. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 215–224.

[4] Baumann, R. (2005). Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*.

[5] Bolchini, C., Carminati, M., Miele, A., Das, A., Kumar, A., and Veeravalli, B. (2013). Run-time mapping for reliable many-cores based on energy/performance trade-offs. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pages 58–64.

[6] Chen, J.-J. (2016). Federated scheduling admits no constant speedup factors for constrained-deadline dag task systems. *Real-Time Systems*, 52(6):833–838.

[7] Chen, K. H., Chen, J. J., Kriebel, F., Rehman, S., Shafique, M., and Henkel, J. (2016). Task mapping for redundant multithreading in multi-cores with reliability and performance heterogeneity. *IEEE Transactions on Computers*, 65(11):3441–3455.

[8] Cui, X., Mills, B., Znati, T., and Melhem, R. (2014). Shadow replication: An energy-aware, fault-tolerant computational model for green cloud computing. *Energies*, 7(8):5151–5176.

[9] Das, A., Kumar, A., and Veeravalli, B. (2013a). Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 689–694.

[10] Das, A., Kumar, A., and Veeravalli, B. (2013b). Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 689–694.

[11] Davis, R. I. and Burns, A. (2011). Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Syst.*, 47(1):1–40.

[12] Graham, R. L. (1966). Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581.

[13] Guan, N., Stigge, M., Yi, W., and Yu, G. (2012). Parametric utilization bounds for fixed-priority multiprocessor scheduling. In *IEEE 26th International Parallel and Distributed Processing Symposium*, pages 261–272.

[14] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001). Mibench: A free, commercially representative embedded benchmark suite. In *WWC-4. IEEE International Workshop*.

[15] Hu, J., Wang, S., and Ziavras, S. (2006). In-register duplication: Exploiting narrow-width value for improving register file reliability. In *DSN*.

[16] Izosimov, V., Pop, P., Eles, P., and Peng, Z. (2012). Scheduling and optimization of fault-tolerant embedded systems with transparency/performance trade-offs. pages 261–272.

[17] Kwon, J., Kim, K. W., Paik, S., Lee, J., and Lee, C. G. (2015). Multicore scheduling of parallel real-time tasks with multiple parallelization options. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 232–244.

[18] Li, J., Chen, J. J., Agrawal, K., Lu, C., Gill, C., and Saifullah, A. (2014). Analysis of federated and global scheduling for parallel real-time tasks. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 85–96.

[19] Li, L., Degalahal, V., Vijaykrishnan, N., Kandemir, M., and Irwin, M. (2004). Soft error and energy consumption interactions: A data cache perspective. In *ISLPED*.

[20] Lyons, R. E. and Vanderkulk, W. (1962). The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, 6(2):200–209.

[21] Mukherjee, S. S., Kontz, M., and Reinhardt, S. K. (2002). Detailed design and evaluation of redundant multithreading alternatives. In *ISCA*, pages 99–110.

[22] Mukherjee, S. S., Weaver, C., Emer, J., Reinhardt, S. K., and Austin, T. (2003). A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *IEEE/ACM MICRO*.

[23] Rehman, S., Chen, K., Kriebel, F., Toma, A., Shafique, M., Chen, J., and Henkel, J. (2016). Cross-layer software dependability on unreliable hardware. *Computers, IEEE Transactions on*.

[24] Rehman, S., Kriebel, F., Shafique, M., and Henkel, J. (2014a). Reliability-driven software transformations for unreliable hardware. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*.

[25] Rehman, S., Kriebel, F., Sun, D., Shafique, M., and Henkel, J. (2014b). dtune: Leveraging reliable code generation for adaptive dependability tuning under process variation and aging-induced effects. In *DAC*, pages 1–6.

[26] Rehman, S., Shafique, M., Aceituno, P. V., Kriebel, F., Chen, J. J., and Henkel, J. (2013a). Leveraging variable function resilience for selective software reliability on unreliable hardware. In *DATE*.

[27] Rehman, S., Shafique, M., and Henkel, J. (2012). Instruction scheduling for reliability-aware compilation. In *DAC*.

[28] Rehman, S., Shafique, M., Kriebel, F., and Henkel, J. (2011). Reliable software for unreliable hardware: embedded code generation aiming at reliability. In *CODES+ISSS*.

[29] Rehman, S., Toma, A., Kriebel, F., Shafique, M., Chen, J. J., and Henkel, J. (2013b). Reliable code generation and execution on unreliable hardware under joint functional and timing reliability considerations. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 273–282.

[30] Reinhardt, S. K. and Mukherjee, S. S. (2000). Transient fault detection via simultaneous multithreading. In *ISCA*, pages 25–36.

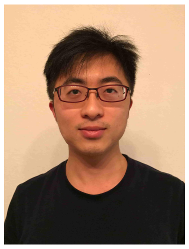
[31] Saggese, G. P., Wang, N. J., Kalbarczyk, Z., Patel, S. J., and Iyer, R. K. (2005). An experimental study of soft errors in microprocessors. *IEEE Micro*, 25(6):30–39.

[32] Shivakumar, P., Kistler, M., Keckler, S., Burger, D., and Alvisi, L. (2002). Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN*.

[33] Srinivasan, A. and Anderson, J. H. (2005). Fair scheduling of dynamic task systems on multiprocessors. *Journal of Systems and Software*, 77(1):67 – 80. Parallel and distributed real-time systems.

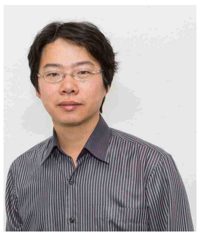
[34] Vadlamani, R., Zhao, J., Burleson, W., and Tessier, R. (2010). Multicore soft error rate stabilization using adaptive dual modular redundancy. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 27–32, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

[35] Vijaykumar, T. N., Pomeranz, I., and Cheng, K. (2002). Transient-fault recovery using simultaneous multithreading. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 87–98.



Kuan-Hsun Chen received his M.Sc. degree from Department of Computer Science from National Tsing Hua University, Hsinchu, Taiwan, in 2013, and is pursuing the Ph.D. degree from the Chair for Design Automation of Embedded Systems, TU-Dortmund, Germany, supervised by Prof. Dr. Jian-Jia Chen. His research interests include dependable computing, embedded systems, reliability-aware resource management, and real-time operating systems.

Georg von der Brüggen received his Diploma degree in computer science from TU Dortmund University, Germany, in 2013 and now is a PhD student at the Chair for Design Automation of Embedded Systems at TU Dortmund University, supervised by Prof. Dr. Jian-Jia Chen. His research interests are in the area of embedded and real-time systems with a focus on real-time scheduling. His main research topics are non-preemptive scheduling, speedup-factors, self-suspension, and mixed-criticality systems.



Jian-Jia Chen is Professor at Department of Informatics in TU Dortmund University, Germany. He was Juniorprofessor at Department of Informatics in Karlsruhe Institute of Technology, Germany from May 2010 to March 2014. He received his Ph.D. degree from Department of Computer Science and Information Engineering, National Taiwan University, Taiwan in 2006. Between Jan. 2008 and April 2010, he was a postdoc researcher at ETH Zurich, Switzerland. His research interests include real-time systems, embedded systems, energy-efficient scheduling, power-aware designs, temperature-aware scheduling, and distributed computing. He received Best Paper Awards at CODES+ISSS 2014, RTCSA 2005 and 2013, and SAC 2009. He has involved in Technical Committees in many international conferences.