RTailor: Parameterizing Soft Error Resilience for Mixed-Criticality Real-Time Systems

Shao-Yu Huang*, Jianping Zeng*, Xuanliang Deng†, Sen Wang†, Ashrarul Sifat†, Burhanuddin Bharmal†, Jia-Bin Huang‡, Ryan Williams†, Haibo Zeng†, Changhee Jung*
*Purdue University, USA †Virginia Tech, USA ‡University of Maryland, USA

***Abstract*—Equipping real-time systems with soft error resilience can be challenging due to the tradeoff of the timing and failure requirements for mixed-criticality tasks. Violation of these requirements yields failed task scheduling in one way or another. However, not every task requires the same *degree* of soft error resilience. For example, low-criticality tasks can run with low or even no soft error resilience, whereas mid- or high-criticality tasks may require relatively high resilience depending on their inherent failure requirement. Unfortunately, existing soft error resilience schemes do not have the ability to control the degree of their resilience in a fine-grained way, *i.e.,* they can only be turned on or off as a whole during task execution. To this end, this paper presents *RTailor (Resilience Tailor)*, a compiler-directed parameterized soft error resilience scheme that achieves the desired level of soft error protection according to the demand of each task. The key idea is that for a given protection ratio, compilers can transform a hot loop such that the number of its iterations protected over the total iterations matches the ratio. Compared to full resilience protecting every iteration, RTailor's parameterized soft error resilience significantly reduces the performance overhead of tasks, thereby improving their real-time schedulability. The experimental results highlight that for four representative fault rates, RTailor achieves 15%∼21% average schedulability improvements over the state-of-the-art work that lacks parameterized soft error resilience.**

I. Introduction

Soft errors have been a major cause of failures in mission-critical embedded systems. Taking an unmanned aerial vehicle (UAV) as an example, an undetected soft error in the flight control kernel may result in sending erroneous signals to actuators, which can crash the vehicle. One typical source of soft errors is an energetic particle strike, *e.g.,* cosmic rays, that can change the data held in registers and memory cells without causing physical hardware damage [1]–[4]. Such unexpected changes can lead to failures such as crashes, hangs, and silent data corruption (SDC). Even worse, transistor size shrinking and aggressive voltage scaling make electronic circuits more vulnerable to soft errors [5]. Thus, providing soft error resilience is a must in mission-critical systems.

Further complications arise when mission-critical systems run real-time tasks with a timing requirement for meeting deadlines [6]. In particular, equipping such a real-time system with soft error resilience can be challenging as the performance overhead—paid for satisfying failure rate requirements of the tasks—inevitably increases their execution time. Such a phenomenon then places significant pressure on task schedulability in that violation of either the failure rate or the timing requirements for a real-time system leads to failed scheduling.

Nevertheless, not every task requires the same reliability in a mixed-criticality real-time system that runs both critical and non-critical tasks. The actual failure requirement depends on the criticality of each task, *e.g.,* the aviation domain uses DO-178C [7] as a standard for defining failure requirements of software components. Specifically, it assigns the components different levels based on the failure effect: DAL A (Catastrophic), DAL B (Hazardous), DAL C (Major), DAL D (Minor), and DAL E (No Safety Effect). Each level specifies the corresponding allowable failure rate, taking into account the severity of the failure impact. For example, DAL A has the smallest allowable failure rate 10^{-9}/hr while the other four requirements have higher allowable failure rates.

Likewise, in a mixed-criticality system, low-criticality tasks can run with low or even no soft error resilience, whereas mid- or high-criticality tasks may demand relatively high resilience depending on their inherent failure rate requirement. Thus, it is desirable to have the ability to configure the reliability according to the task's demand, reducing the performance overhead while maintaining the failure rate requirement. Unfortunately, existing resilience schemes do not have such an ability to control the degree of resilience in a fine-grained way, *i.e.,* they can only be turned on or off as a whole during task execution.

To this end, this paper presents RTailor (Resilience Tailor), the first of its kind that achieves the desired level of soft error protection according to the resilience demand of mixed-criticality tasks. The protection leverages (1) *instruction duplication* [8], [9]—comparing the outputs of the original and the replica—and (2) *idempotent processing* [10], [11]—dividing the program into side-effect-free regions, each of which can be re-executed to correct any error detected therein—for error detection and recovery, respectively. Unlike prior instruction duplication work that blindly replicates all instructions thus being called full soft error resilience, RTailor replicates only a portion of instructions with a given target protection ratio[1] in mind, thereby achieving parameterized soft error resilience. In other words, RTailor's compiler transforms each hot loop such that the number of its iterations protected over the total iterations matches the ratio.

Compared to the full resilience protecting every iteration, RTailor's parameterized soft error resilience significantly re-

[1]It is defined for a given hot loop as the number of protected iterations divided by the total iteration count. To protect the rest of the instructions residing outside of such a loop, RTailor can fall back to the full resilience, *i.e.,* duplicating all instructions in each idempotent region.

duces the performance overhead of mixed-criticality tasks, which in turn improves their real-time schedulability. The experimental results highlight that across four representative fault rates, RTailor achieves 15%∼21% and 17%∼20% average schedulability improvements for NPB (NASA Parallel Benchmarks) [12] and MiBench [13] applications, respectively, over the state-of-the-art tree-based scheduler [14] that lacks parameterized soft error resilience. RTailor's principal contributions can be summarized as follows:

- RTailor, for the first time, parameterizes soft error resilience in an application-tailored way. It can offer mixed-criticality real-time tasks the right level of resilience without overprotecting them against soft errors.
- Due to the lack of the overprotection, RTailor reduces the execution time of tasks without compromising their failure rate requirement. It turns out that the saved time helps a real-time system to schedule more tasks.
- RTailor's parameterized soft error resilience is effective. It allows fine-grained control over the resilience necessary for a given task, and the performance overhead is proportional to the target protection ratio in general.

II. Background and Motivation

A. Soft Error Resilience: Detection and Recovery

Soft error resilience has been an active research area due to the correctness/safety demand of mission-critical embedded systems. Among prior resilience schemes, one popular way to detect soft errors is conducting instruction-level dual modular redundancy (DMR) in a software manner [8], [15]–[20]. At compile time, it duplicates instructions of program and injects checking code at synchronization points, *e.g.,* store instructions, so that the outputs of the original and the replica are compared at run time. This is so-called instruction duplication based error detection in which any mismatch between the outputs is treated as a soft error.

To lower the run-time overhead of the software-based DMR, researchers have proposed hardware-based soft error resilience approaches, *e.g.,* core-level DMR [21], [22], triple modular redundancy (TMR) [23]–[25], and acoustic-sensor-based resilience schemes [26]–[32]. However, hardware-based schemes come with the costs of high power consumption and complex hardware logic; they also lack the ability to adapt the level of soft error resilience to the varying demand of each task. Meanwhile, software-based schemes are attractive since they are affordable and flexible for RTailor to achieve parameterized soft error resilience. Thus, this section focuses on them leaving the hardware-based schemes out of scope.

As always, the detection alone cannot offer soft error resilience, *i.e.,* instruction duplication needs to be accompanied by either backward or forward recovery to correct detected errors. Idempotent processing is one of the most popular backward recovery schemes [10], [33]–[39]. It partitions program into a series of *idempotent* regions—each of which does not have any Write-After-Read (WAR) dependence—so that they can be re-executed yet still generate the same output; such side-effect-free re-execution of idempotent regions is the basis for them to recover from any soft error detected therein. On the other hand, forward recovery schemes such as SWIFT-R [40] and InCheck [41] rely on TMR without retreating the faulty execution when a soft error is detected. Swift-R runs three copies of program and adds recovery code—that can correct an error by majority voting—at synchronization points. To mitigate the silent data corruption (SDC) problem of SWIFT-R, InCheck utilizes additional register/memory checkpointing. It leverages a special recovery procedure that consults the checkpointed values to determine if a detected soft error is recoverable. Otherwise, InCheck informs the program of the irrecoverability, which forces it to restart, rather than blindly trying the recovery procedure which leads to SDC.

In addition to the above fine-grained instruction-level soft error resilience schemes, there are coarse-grained schemes, the recovery procedure of which is realized in the process level [42]. They run multiple replica processes of the same program and perform the majority voting among all the processes for error detection and correction purposes. To the best of our knowledge, no prior work is capable of adjusting soft error resilience in a fine-grained and application-tailored way. In contrast, RTailor can offer real-time tasks the necessary level of soft error resilience according to the failure rate requirement, which reduces their over-protection and the resulting run-time overhead thereby providing better schedulability.

B. Re-execution of Tasks to Meet Failure Rate Requirements

Mission-critical embedded systems should be equipped with soft error resilience which may otherwise lead to catastrophic failure. Among them, it is a particular interest to protect a mixed-criticality real-time system against soft errors due to the importance and popularity. In the mixed-criticality system, each task features both an actual failure rate and a required failure rate; the actual failure rate represents how likely a task can fail during the execution, which is affected by multiple factors including program characteristics, the underlying hardware, and the surrounding environment. One common way to estimate the actual failure rate is conducting fault injection campaigns. On the other hand, the required failure rate determines the maximum allowable failures within a given time period, *e.g.,* a regulation like DO-178C specifies the required failure rate of each task. Indeed, it is critical to ensure the actual failure rate of each task is lower than its required failure rate, which we call a *failure rate constraint*. In general, the scheduler leverages both actual and required failure rates of all tasks—along with their period and worst-case execution time—to figure out if the tasks are schedulable.

Especially for those tasks that do not satisfy the constraint, *i.e.,* the actual failure rate is over the required rate, the task scheduler can leverage a re-execution mechanism to lower the actual failure rate. The rationale is that if any task is executed multiple times, then its actual failure rate becomes smaller across re-executions. That is because the original rate is defined for a time period of a single task run, and thus the task's rate over multiple N runs is obtained by multiplying the

original rate (< 1) by itself N times; additional details of the rate over re-executions are deferred to §III-B. The following equation from Reghenzani *et al.*'s work [14] shows how to find the minimum value of N that satisfies the failure rate constraint. In other words, for a given task, we compute such a minimum required number of re-executions ($N_A^{re\text{-}exec}$) with the actual and required failure rates in mind like below:

$$N_i^{re\text{-}exec} = \max(0, \lceil \log_{p_i^F} p_i^{req} \rceil - 1) \quad (1)$$

where p_i^F is the actual failure rate of Task i while p_i^{req} stands for the required failure rate.

Fig. 1: Example of task re-executions to satisfy the failure rate constraint of Task A, which requires executing the task at least 3 times; the failure rates are transformed with a negated logarithm function so that the failure rate after re-execution can be visually represented as a summation; the relationship between the bars is equivalent to $p_A^{req} \geq {p_A^F}^3$.

Figure 1 depicts an example where Task A in a mixed-criticality system has an original failure rate of $10^{-3}/hr$ and a required failure rate of $10^{-7}/hr$. Thus, Task A should be executed at least 3 times (*i.e.,* $N_A^{re\text{-}exec}$ is 2) to satisfy the failure rate constraint—according to Equation (1). As such, this leads to the 3x higher occupation of processor resources.

Note that as shown in Figure 1, Task A is overprotected; the actual failure rate is excessively lower than the required failure rate, *i.e.,* the concatenated yellow bar is too much longer than the red bar in the figure. Here, Task A does not have to be run 3 times with full resilience. It is rather possible to relax the soft error resilience of the task with the goal of minimizing overprotection. For example, we can protect only a portion of task instructions, while satisfying the failure rate constraint.

C. Our Solution: Parameterized Resilience

With the above insight in mind, this paper proposes RTailor, a compiler-directed flexible yet efficient approach to parameterizing soft error resilience for mixed-criticality real-time systems. By providing each task with its own custom soft error resilience, RTailor can prevent the task from being overprotected unnecessarily, thereby achieving significant schedulability improvement.

Figure 2 shows a mixed-criticality system with parameterized soft error resilience; it provides 3 versions of the execution of Task A with different levels of soft error resilience: 40% protection, 80% protection, and full (100%) protection used as a standard non-parameterized resilience scheme. The execution version sequence of the task with parameterized soft error resilience ("full", "80%", "40%") has a higher actual failure rate than that of the non-parameterized execution sequence ("full", "full", "full"), *i.e.,* $p_A^{req} \geq p_{A_{full}}^F p_{A_{80\%}}^F p_{A_{40\%}}^F \geq (p_{A_{full}}^F)^3$, yet still complies with the failure rate constraint[2]. The upshot is that by minimizing the overprotection of a task, its execution time is significantly reduced, which gives the scheduler more room to accommodate other tasks.

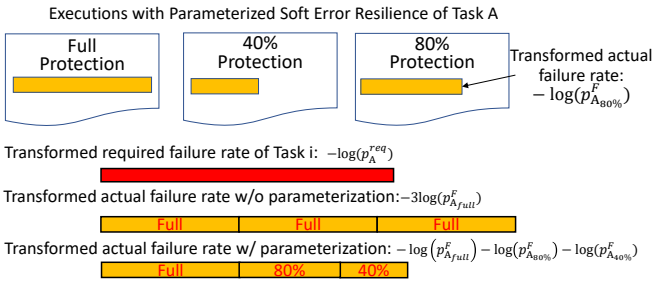


Fig. 2: The impact of parameterized soft error resilience on lowering overprotection and processor utilization; less reliability indicates less execution time (more idle processor time).

III. Task Model and Fault Model

A. Task Model

RTailor's task model is built upon prior work [14]. The task set is defined with $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task is modeled with $\tau_i = (C_i, D_i, T_i, p_i^{req})$; C_i is a vector for representing the worst-case execution times (WCETs) of τ_i's different versions τ_{i_j}, *i.e.,* $C_i = \{c_{i_j} | j \in V\}$ where V is a set of the task versions, and c_{i_j} is the WCET of task τ_{i_j}. Here, D_i, T_i, and p_i^{req} are the deadline, period, and required failure rate of τ_i, respectively. Unlike the prior work, RTailor compiles a given task τ_i to multiple versions with different protection ratios, *i.e.,* τ_{i_j}, whose actual failure rate is $p_{i_j}^F$.

Also, the fault tolerance mechanism of RTailor is derived from the same prior work [14]. In essence, if a task τ_i has a failure during the execution, the task scheduler spawns a new job containing the same computation, which is repeated unless it succeeds. The spawned jobs of task τ_i are defined as the sequence: $\tau_{i(0)}, \tau_{i(1)}, \ldots, \tau_{i(N_i^{re-exec})}$ where $\tau_{i(k)}$ is the k$^{\text{th}}$ re-execution of τ_i ($\tau_{i(0)}$ is the original execution), and $N_i^{re-exec}$ is the total number of re-executions of τ_i. Considering $\tau_{i(k)}$'s WCET $c_{i_{v_k}}$ where v_k is the task version found in the re-execution version sequence $VSEQ_i = (v_0, \ldots, v_{N_i^{re-exec}})$, RTailor generates the optimal sequence that satisfies the task τ_i's failure rate constraint while keeping the execution time ($\sum_{k=0}^{N_i^{re-exec}} c_{i_{v_k}}$) minimal as will be shown in §VI.

B. Fault model

RTailor only considers transient faults, also known as soft errors, in the processor. As in prior resilience work [31], [40], control flow and address generation units, caches, and main memory are supposed to be hardened against soft errors. RTailor assumes that the task scheduler in the mixed-criticality system is also hardened with some form of hardware resilience support. Therefore, a failure happening in the task scheduler is beyond the scope of this paper.

[2] §VI details how RTailor obtains an optimal re-execution version sequence with the failure rate constraint satisfied.

It is important to note that the failure rate of a task remains independent over its re-executions. That is because RTailer only considers transient faults, not permanent ones. Due to the transient nature, the fault rate of the following re-execution is independent of faults that have happened in prior execution(s).

Even if transient faults somehow affect task outputs, RTailor guarantees the integrity of the task inputs, *i.e.,* a task always reads correct inputs during its execution(s). The reason is two-fold: (1) The inputs of each task are stored in the main memory backed with error correction code (ECC); (2) The input memory locations are disjoint with the output memory locations, which is the case for all modern operating systems that start program with clean memory status. This technically prevents any incorrect output of prior task execution(s) from being propagated to inputs to the current execution. As a result, the re-execution of each task always reads its correct inputs from the ECC memory.

IV. RTailor Overview

The goal of RTailor is to achieve a fine-grained and flexible soft error resilience scheme for mixed-criticality real-time systems. With RTailor's parameterized soft error resilience, they can not only improve the task schedulability but also meet the failure rate constraint of all tasks. Overall, RTailor consists of four major functionality blocks to accomplish the goal: (1) *loop selection* to find those loops suited for the resilience parameterization, (2) *loop restructuring* to transform them into the form for which the resilience can be easily parameterized, (3) *soft error protection code generation* to augment the loop body with instructions that detect and recover from soft errors, and (4) *optimal re-execution version sequence searching* to figure out $VSEQ_i$ (§III-A) of a given task τ_i so that its actual failure rate through re-executions can satisfy the failure rate constraint with the total execution time minimized. Note that the first three functionality blocks are implemented as RTailor's compiler passes, while the fourth is done as a standalone algorithm.

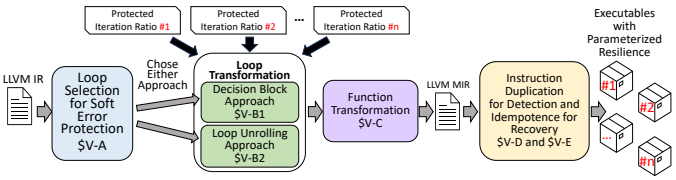


Fig. 3: The workflow of RTailor's compiler passes

Figure 3 shows RTailor's compilation workflow. It begins with taking input program in the form of LLVM intermediate representation (IR) [43] produced by the language frontend such as Clang [44] for C/C++. RTailor's compiler then selects the candidate loops for parameterized soft error resilience[3](§V-A) and transforms their control flow so that the protected iteration ratio—input atop of the figure—matches the ratio of the number of iterations being protected in each loop to its total iteration count (§V-B). Then, the compiler also transforms those functions called inside the parameterized loop for their protection to be parameterized as well (§V-C). Finally, the compiler performs code generation for instruction duplication [8] and idempotent processing [33]—over the transformed loops and functions—to detect (§V-D) and correct (§V-E) soft errors, respectively. Note that RTailor compiles a task τ_i into multiple versions τ_{i_j} with each different protected iteration ratio. They are fed into the *optimal re-execution version sequence searching* (§VI) along with the WCET c_{i_j}, actual failure rate $p_{i_j}^F$ of each version τ_{i_j}, and the required failure rate p_i^{req} to find $VSEQ_i$, *i.e.,* the optimal sequence of re-executing τ_i's versions.

[3]It is interchangeably referred to as parameterized soft error protection.

V. RTailor Compiler Implementation

A. Loop Selection for Parameterized Soft Error Protection

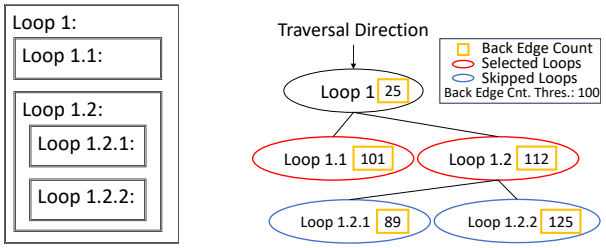
To parameterize soft error resilience, RTailor only enables soft error protection during a part of program (task) execution. That is, RTailor selects some parts of code in input program and decides if they should be protected or not; it determines the right level of the resilience for them according to how much protection is enough to achieve the failure rate requirement without overprotection. When selecting the candidate code to which the resilience parameterization is applied, RTailor seeks two important properties of their execution. First, the program must spend most of its execution time in the code (*execution time dominance*), which would otherwise make the impact of the parameterization on the overall resilience and performance negligible. Second, the code must be executed a sufficient number of times (*high execution count*) during program execution so that RTailor achieves the parameterization in a fine-grained manner. The execution count translates to the opportunities that RTailor can make a decision to switch on/off the soft error protection of the code.

TABLE I: The ratio of loop execution time to the total execution time of program in the NPB benchmark suite

	BT	CG	EP	FT	IS	LU	MG	SP
Ratio	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.98

Taking into account the two properties of candidate code, we believe that loops are the right one to be protected with parameterized soft error resilience. It is well known that lots of applications in real-time systems are loop-intensive [45], and our experimental results also justify the decision of selecting loop candidates to be protected. As shown in Table I, for all NPB benchmark applications, 99% of the total execution time is spent on their loops. Note that for some program that is not loop-intensive, RTailor can still protect the code outside of the loop candidates with conventional non-parameterized soft error resilience.

In particular, care must be taken for nested loops. Although they are common in program, not all levels of loops in the nested structure have the same degree of the two execution properties mentioned above. That is, parameterized soft error resilience performs better on some level of the loop than other levels. With that in mind, RTailor proposes three strategies to



(a) Nested loops (b) Loop structure tree

Fig. 4: Examples of nested loops and their loop structure tree

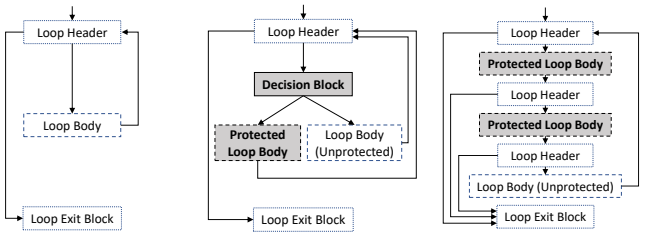
find the ideal loop level in nested loops as the target of parameterized soft error protection: (1) outermost, (2) innermost, and (3) profiling-based selection strategies. First, the outermost strategy prioritizes those with *execution time dominance*, so it selects the outermost loop for the parameterized protection. Second, the innermost strategy prioritizes those with *high execution count*, i.e., many loop iterations, and therefore it selects the innermost loops as the target.

Finally, the profile-based strategy tries to strike a balance between the goals of the innermost and outermost strategies. To illustrate, for each loop shown in Figure 4a, the profiler records the loop back edge count, *i.e.,* how many times the back edge—from a loop latch block [46] to the loop header—is taken. Using the profile results, RTailor builds a loop structure tree [47] and annotates each node with its loop back edge count. For example, Figure 4b shows such a tree corresponding to the nested loops in Figure 4a. RTailor then traverses the tree starting from the root node (i.e., outermost loop) to find the loops whose back edge count is greater than a threshold, in which case those nodes underneath are skipped as shown in Figure 4b; if there is no such loop found at the end of the traversal, RTailor falls back to the innermost strategy. According to empirical results, a back edge count threshold of 100 strikes a good balance between the execution time dominance and the protection granularity.

B. Preparing the Loops for Parameterized Protection

This section discusses two loop transformation approaches to varying the frequency of protecting loop iterations: (1) decision block formation and (2) speculative loop unrolling [37]. For the loops selected for parameterized soft error protection, RTailor first prepares two types of loop bodies, i.e., unprotected (the original in Figure 5a) and protected (the replica for accommodating protection code later). Then, using one of the two approaches, RTailor transforms the control flow of the loop so that the protected loop iteration will be executed at the frequency of the protected iteration ratio.

1) Naive Approach - Decision Block Formation: To execute either the protected or the unprotected code of a loop according to a user-defined protected iteration ratio, RTailor's compiler inserts the decision logic block after the header of each selected loop. As shown in Figure 5b, the block decides whether the upcoming loop iteration should be protected or not. Algorithm 1 describes how the decision logic enables switching between protected and unprotected code in a controlled way;



(a) Original loop (b) Decision block formation (c) Speculative loop unrolling [37]

Fig. 5: Loop structure transformation

RTailor sets $Threshold$ of the algorithm correspondingly in order to reflect the protected iteration ratio. By the law of large numbers [48], as the decision block is executed a huge number of times in a frequently executed loop, the ratio of protected iterations to the total number of iterations becomes close to the user-defined protected iteration ratio.

Algorithm 1: Decision Logic

```
1 Sample uniform random number R ∈ [0 : 100) ⊆ ℕ;
2 Protected Iteration Ratio N ∈ [0 : 1] ⊆ ℝ ;
3 Initialize Threshold ← N × 100;
4 if R < Threshold then
5 |   branch to protected code;
6 else
7 |   branch to unprotected code;
```

Note that the protected iteration ratio is not necessarily equivalent to the program's actual failure rate. Although there is a positive correlation between these two variables, the actual failure rate depends on multiple factors such as program characteristics, the implementation of soft error resilience, the underlying hardware, and so on. This implies that a fault injection campaign is essential to measure the program's actual failure rate. §VII-B describes RTailor's methodology for the fault injection campaign while §VII-G shows the resulting actual failure rates of the benchmarks tested.

2) RTailor's Approach - Speculative Loop Unrolling [37]: Although the decision block approach is simple, it incurs a high run-time overhead for a couple of reasons: (1) the random number generator (line 1 in Algorithm 1) used by the block significantly degrades the effectiveness of the branch predictor [49] in the processor core—due to the randomness hindering the prediction—and thus results in frequent mispredictions leading to expensive pipeline flushing [49]; (2) the random number generation itself takes a while, making the decision block dominate the execution time of a small loop.

To address the above two issues of the decision block approach, RTailor leverages loop unrolling [46] which essentially eliminates the need for decision logic. To be specific, RTailor unrolls each selected loop and protects a portion of the unrolled loop bodies according to the protected iteration ratio. For example, as shown in Figure 5c, suppose an unrolling factor[4] is two (three loop bodies in total after unrolling), and the user-defined protected iteration ratio is $\frac{2}{3}$; to match the

[4]The unrolling factor is the number of times the loop body is copied.

ratio, the first two loop bodies are protected while the last one is not. In general, for the unrolling factor of X and the protected iteration ratio of R, RTailor protects $\lceil R \times (X+1) \rceil$ copies of the unrolled loop bodies. In particular, RTailor can unroll even the loops whose iteration count is unknown at compile time by explicitly checking the exit condition before each unrolled loop body as shown in Figure 5c. This is so-called speculative loop unrolling [37]; the rest of paper refers to it simply by loop unrolling.

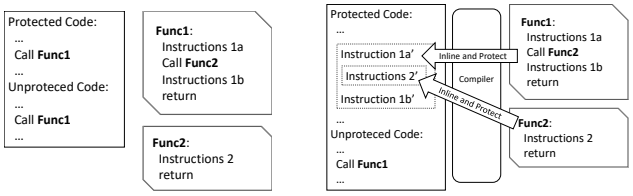
TABLE II: Average slowdown of the decision block approach compared to the loop unrolling approach for NPB bechmarks

Protected Iteration Ratio	20%	40%	60%	80%
Slowdown of the decision block formation approach	1.35x	1.08x	1.05x	1.01x

Unlike the decision block approach, the loop unrolling does not introduce additional dynamic instructions during program execution—though it slightly increases the code size. As a result, the loop unrolling approach incurs a much less performance overhead than the decision block approach. Table II highlights that for NPB benchmarks protected with parameterized soft error resilience, the decision block approach[5] incurs significant slowdowns compared to RTailor's loop unrolling approach across different protected iteration ratios.

C. Preparing Functions for Parameterized Protection

To achieve parameterized soft error resilience correctly, RTailor must be careful with the function whose callsites are not always protected. The crux of the problem is that the (un)protection of the function is subject to whether or not its callsites are protected, which leads to a conflicting situation as it can be called by both unprotected and protected callsites. To address this problem, RTailor should be able to protect the function if and only if it is called from protected code—leaving it unprotected otherwise.



(a) Original code (b) Nested function inlining

(c) Function cloning and name mangling

Fig. 6: Function protection transformation
Instruction X' is the protected equivalence of instruction X

[5]In the decision block approach, the decision logic utilizes a 32-bit LFSR (linear-feedback shift register) based pseudorandom number generator.

To this end, RTailor compiler generates both protected and unprotected versions of such a function and picks which version should be called depending on the location of the callsite (i.e., whether it is protected or not). For this purpose, RTailor proposes two techniques: (1) nested function inlining and (2) function cloning and name mangling. The former technique recursively inlines all functions in the call chain rooted at the protected callsite. For example, Figure 6b shows that RTailor replaces the protected callsite, i.e., Call Fun1, with the body of Func1 which in turn does the same for Call Func2 by inlining its body.

When the recursive function inlining is not feasible, *e.g.*, the resulting code size increase is prohibitively large due to a long call chain, RTailor takes advantage of the function cloning and name mangling instead. As shown in Figure 6c, RTailor's compiler first finds and copies those functions called in the protected callsites. It then renames the replica functions so that they can only be called by protected callsites with different names (Func1.P and Func2.P). Finally, the compiler changes the target of each call instruction in the protected code to the corresponding renamed replica function, *e.g.*, resulting in Call Fun1.P and Call Fun2.P as shown in the figure.

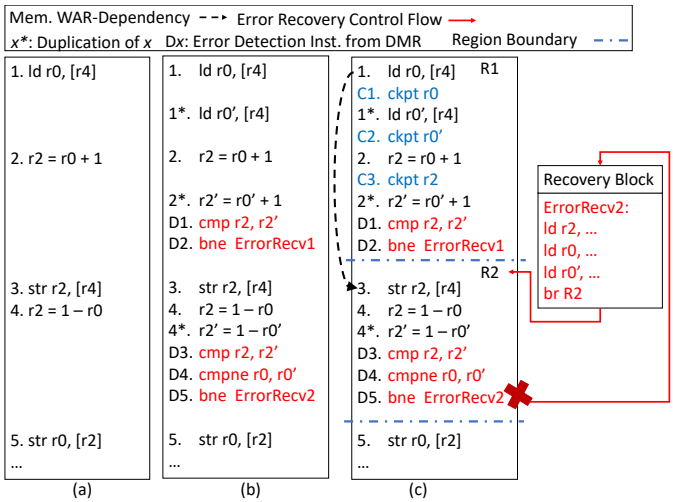


Fig. 7: Instruction duplication and idempotent region formation; (a) original assembly code; (b) code after DMR ; (c) code after idempotent processing with a per-region recovery block

D. Instruction Duplication for Error Detection

Once the seleted code candidates are prepared for soft error resilience parameterization using the code restructuring techniques (§V-B and §V-C), RTailor should enable the resulting code to detect and correct errors. This and next sections describe RTailor's DMR-based soft error detection and idempotence-based error recovery, respectively.

RTailor leverages instruction duplication [8] to detect the occurrence of soft errors in the protected code. Specifically, RTailor's compiler replicates each instruction and inserts error checking instructions at each synchronization point [8][6]. They

[6]Synchronization points are instructions that may affect the output of the program, *e.g.*, store instruction, and they are not replicated.

detect a soft error by comparing the output values of the original and the replica instructions, i.e., any mismatch between the values indicates the occurrence of a soft error. Figure 7(b) shows how the original code in Figure 7(a) changes with the instruction duplication, *e.g.,* load `ld r0',[r4]` is followed by the replica `ld r0,[r4]` numbered with $*$ in Figure 7(b).

In particular, it uses a different target register (`r0'`), as with other replica instructions of the figure, in case the register (`r0`) of the original instruction can be corrupted by a soft error. Before a synchronization point, which is instruction 3 (`store`) in Figure 7(b), RTailor's compiler inserts a comparison instruction $D1$ to check the occurrence of a potential soft error on `r2`; although every value read by instruction 3 is supposed to be compared against the original register, our example here omits the error checking code for the other store operand `r4`. Finally, a conditional branch (instruction $D2$) follows the error checking code to direct program control to the recovery code in case of an error detected, i.e., provided the comparison results in mismatch.

E. Idempotence-Based Recovery for Error Correction

For the recovery of program interrupted by the detection of a soft error, RTailor exploits idempotent processing [50] that can correct errors by its side-effect-free re-execution. To achieve this, RTailor's compiler partitions program to a series of idempotent regions so that each region lacks memory write-after-read (WAR) dependence [33]; the compiler places a region boundary to break every memory WAR dependence, *e.g.,* the first region boundary in Figure 7(c) cuts the dependence between instruction 1 (`ld`) and instruction 3 (`str`).

Note that the above region partitioning only takes care of memory WAR dependences with register WAR dependences unaddressed, and thus the re-execution of such a region in its current form leads to incorrect error recovery—because the values of input registers of the region get changed upon re-execution. To resolve this issue, RTailor leverages the live-out register checkpointing [50]. That is, for each region, RTailor's compiler inserts a checkpoint store after the last updating point of a register that is used by some following regions. The implication is two-fold: (1) RTailor only checkpoints the live-out register once even if it is defined multiple times in a region, saving checkpointing stores; (2) all input registers of each region is already sure to have been checkpointed to memory (i.e., checkpoint storage) before starting the region[7]. Thus, upon an error detected in a region, RTailor can use the checkpointed values to restore (WAR dependent) the input registers of the region before re-executing it for recovery.

Figure 7 (c) shows an example where three live-out registers $r0$, $r0'$, $r2$ of region R2 have been checkpointed (instruction `C1-3`) before; though they are all checkpointed in region R1 in the figure, they could be done in other prior regions. To illustrate the recovery protocol, when a soft error is detected in R2, the program is first interrupted. Then, RTailor's recovery runtime redirects the program control to the beginning of a compiler-generated per-region recovery block shown in the rightmost box of Figure 7(c). Finally, the recovery block reloads the checkpointed values to the input registers of R2 from memory and then jumps back to the beginning of the interrupted region R2 to re-execute it for error recovery.

[7]We assume that the entire memory is protected with ECC

VI. Searching for Optimal Re-execution Version Sequence

This section presents how RTailor finds the optimal sequence of re-executing task versions which minimizes the total execution time—including the original task's—with the failure rate constraint satisfied. Compared to the prior work [14] that only determines how many times the fully protected (non-parameterized) task should be re-executed, RTailor's parameterized soft error protection significantly reduces the total execution time and improves the overall task schedulability—without violating any failure rate constraint. RTailor builds the optimal re-execution version sequence for a given task τ_i with taking into account the actual failure rates $p^F_{i_j}$, required failure rate p^{req}_i, and WCET c_{i_j} of each task version τ_{i_j}. In practice, the actual failure rate $p^F_{i_j}$ is obtained from fault injection campaigns (§VII-B), and the WCET c_{i_j} is estimated by using static analysis or measuring task execution times on a real machine [51]–[54]. The following defines the problem of computing a task's optimal re-execution version sequence:

Problem 1: Given a task τ_i and the set of its task versions V, RTailor should find the optimal re-execution version sequence including the original task execution $\tau_{i(0)}$, *i.e.,* $VSEQ_i = (v_0, \ldots, v_{N_i^{re-exec}}), v_i \in V$ such that $\prod_{k=0}^{N_i^{re-exec}} p^F_{i_{v_k}} \leq p^{req}_i$ and $\sum_{k=0}^{N_i^{re-exec}} c_{i_{v_k}}$ is the minimum.

In particular, RTailor reduces the problem of finding the optimal version sequence to an unbounded 0-1 knapsack problem as follows: First, the WCET of every single task version ($\forall_{j \in V}\ c_{i_j}$) is quantized by rounding to two decimal places; according to our empirical result, they strike a balance between the resolution and the speed of running the knapsack algorithm. Second, the required failure rate of a given task τ_i and the actual failure rates of the task versions τ_{i_j} are all negated-log-transformed to turn the product of the actual failure rates into the sum of the negated-log-transformed actual failure rates; this allows RTailor to fit Problem 1 into the knapsack problem. With this transformation, $\prod_{k=0}^{N_i^{re-exec}} p^F_{i_{v_k}} \leq p^{req}_i$ in Problem 1 can be rewritten like below:

$$-\log_{10} p^{req}_i \leq -\log_{10} \prod_{k=0}^{N_i^{re-exec}} p^F_{i_{v_k}} = -\sum_{k=0}^{N_i^{re-exec}} \log_{10} p^F_{i_{v_k}} \tag{2}$$

Table III shows the mapping of the variables between Problem 1 and the unbounded 0-1 knapsack problem. Once the log-transformed actual failure rate is negated, a high total value (the last entry of the table) in the unbounded 0-1 knapsack problem implies a low actual failure rate of the corresponding task, which facilitates the knapsack formulation of Problem 1.

TABLE III: Variable mapping between RTailor's optimal re-execution version sequence problem and the knapsack problem

0-1 Knapsack Problem	Optimal Re-execution Version Sequence Problem
Collection of items	$VSEQ_i$ Re-execution version sequence of τ_i
Item j	τ_{i_j} (The version j of a given task i)
Weight of item j	c_{i_j} (The WCET of τ_{i_j})
Total weight	$\sum_{k=0}^{N_i^{re-exec}} c_{i_{v_k}}$ where v_k is from $VSEQ_i = (v_0, \ldots, v_{N_i^{re-exec}})$
Value of item j	-$\log_{10} p_{i_j}^F$ (The negated-log-transformed actual failure rate of τ_{i_j})
Total value	$-\sum_{k=0}^{N_i^{re-exec}} \log_{10} p_{i_{v_k}}^F$

For a given task τ_i, Algorithm 2 shows how RTailor finds the optimal re-execution version sequence $VSEQ_i$ using dynamic programming [55] for solving the knapsack problem. Algorithm 2 initializes the knapsack's maximum weight (W)—which is the total WCET of the $VSEQ_i$—to 0 (line 1).

Note that the termination condition of Algorithm 2 (line 6) is different from that of the original unbounded 0-1 knapsack problem. That is, Algorithm 2 terminates when the collection's total value (Table III's last entry) is not less than τ_i's negated-log-transformed required failure rate (input $M = -\log_{10} p_i^{req}$) while keeping the total weight (Table III's fourth entry) under the knapsack's maximum weight (W). In other words, the algorithm found a $VSEQ_i$ which satisfies the failure rate constraint of τ_i with the least total WCET.

If the termination condition is not satisfied, each iteration begins with increasing the knapsack's maximum weight W (line 7); to ensure that the final $VSEQ_i$ solution has the minimal total WCET, the increasing step (W_s at line 7) must be set to a value equal to the quantization step size of WCET, which is 0.01 in this work. Then, it forms a collection of items, *i.e.,* $VSEQ_i$, such that their total weight is not greater than W and they have the highest total value (lines 6-17).

Overall, the complexity of Algorithm 2 is like below:

$$O(|V|\lceil \frac{\max_{j \in V}(M(\textit{weight of item j})/(\textit{value of item j}))}{W_s} \rceil) \quad (3)$$

which is technically the number of task versions that need to be searched in the loop (line 10) times the while loop's maximum number of iterations (line 6). Although the complexity is pseudo-polynomial, the algorithm takes only a few seconds to compute the optimal re-execution version sequence for each task in our schedulability simulations (§VII-H). In particular, RTailor can pre-compute the sequence of each task using Algorithm 2 so that at run time, the task scheduler spawns the next re-execution of the task by consulting the pre-computed sequence rather than executing the algorithm each time.

Algorithm 2: Optimal re-execution sequence finding with the unbounded 0-1 knapsack problem

```
Input  : Set of task versions V
Input  : Item weights w_j for all j in V
Input  : Item values u_j for all j in V
Input  : Required minimum value of the item collection M = -log10 p_i^req
Input  : Increasing step of knapsack's maximum weight W_s
Output: Optimal re-execution version sequence VSEQ_i
1  Initialize knapsack's maximum weight: W ← 0;
2  Initialize 0-1 knapsack value array: A_v;
3  Initialize 0-1 knapsack traceback array: A_h;
   /* In the implementation of this algorithm, weights
      are scaled to integers, e.g., W, W_s, and w_j are
      multiplied by 100, so that A_v and A_h can be
      accessed with the conventional integer indices.  */
4  A_v[0] ← 0 ;
5  A_h[0] ← (Null, Null) ;
6  while A_v[W] < M do
7      W ← W + W_s;
8      A_v[W] ← A_v[W − W_s] ;
9      A_h[W] ← (Null, W − W_s);
10     foreach j ∈ V do
11         x ← W − w_j;
12         if x < 0 then
13             continue;
14         y ← A_v[x] + u_j;
15         if y > A_v[W] then
16             A_v[W] ← y;
17             A_h[W] ← (j, x);
   /* Now trace back A_h to find the optimal re-execution
      version sequence VSEQ_i                          */
18 Intialize VSEQ_i to empty sequence;
19 (J, W) ← A_h[W];
20 while W ≠ Null do
21     VSEQ_i.append(J);
22     (J, W) ← A_h[W];
23 Return VSEQ_i;
```

VII. Evaluation

A. RTailor Compiler Implementation

We implemented RTailor atop Clang/LLVM 13 [43] for ARMv7-a ISA with NEON and VFPv3. The compiler passes for soft error detection and protection deal with both NEON registers including floating point registers in ARMv7 and general purpose registers excluding the stack pointer (SP) and the program counter (PC). Although RTailor does not cover SP and PC, there are existing soft error resilience schemes that can be combined with RTailor to protect those registers [18].

B. Fault Injection Campaign Design

RTailor's ability to parameterize soft error resilience is evaluated with statistical fault injection campaigns [56]. They run program a number of times, each of which flips a random bit position of an arbitrary register selected from the register file covering both general purpose registers (except for SP and PC) and NEON registers. We implemented our fault injector as a DynamoRIO client [57]. Basically, it pauses program at a randomly selected dynamic instruction, then flips the selected bit in the chosen registers, and finally resumes the program. Note that the instruction at which the fault injector pauses the program can belong to either the protected or unprotected code. Depending on the termination status of the program and its output, there are 5 kinds of results: Masked, Corrected, Crash, Hang, and SDC. Masked means the output is correct while Corrected shows the fault is captured and corrected.

C. Benchmark Setup

To evaluate RTailor's parameterized soft error resilience, we utilized two benchmark suites, *i.e.,* MiBench[8] and NPB (serial version) [12]. For run-time overhead evaluation of NPB

[8]Some benchmarks are omitted because of redundancy issues [58].

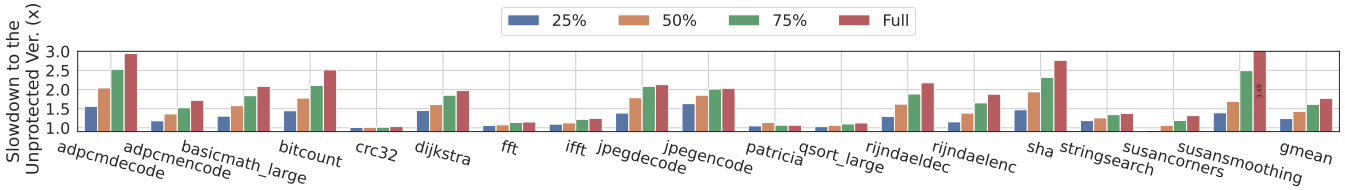


Fig. 8: Slowdown of RTailor for MiBench applications compared to the unprotected version; lower is better

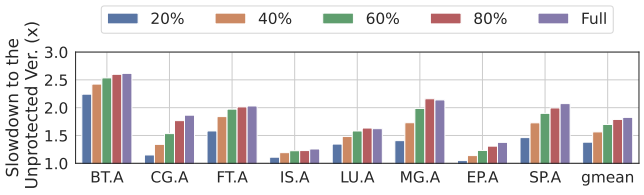


Fig. 9: Slowdown of RTailor for NPB applications compared to the unprotected version; lower is better

applications (`bt`, `cg`, `ep`, `ft`, `is`, `lu`, `mg`, and `sp`), we ran them with Class A inputs [12] (§VII-E), though Class W inputs [12] were used for their statistical fault injection campaigns; this is rather essential to finish the fault injection campaigns in a reasonable amount of time because we have to run each benchmark and its multiple versions a huge number of times to get statistically meaningful results. On the other hand, Mibench applications use the default inputs for both run-time overhead evaluation and fault injection campaigns. To enable the parameterized soft error resilience, all applications from both benchmark suites were compiled with RTailor's compiler passes (§V) using the profile-based strategy with the loop selection threshold set to 100 (§V-A). For better performance, RTailor leveraged the loop unrolling approach (§V-B2) that prepares the selected loops for parameterized protection.

We compiled NPB applications with four different protected iteration ratios: 20%, 40%, 60%, and 80% with the unrolling factor of 4 (5 loop bodies in total) (§V-B2). Since NPB applications spend the majority of the execution time in the loops (Table I), we left the instructions outside the loops unprotected, which has only zero or minimal effect on the overall soft error resilience and the execution time.

In the meantime, MiBench applications were compiled with 3 different protected iteration ratios: 25%, 50%, and 75%, with the unrolling factor of 3. The fewer versions than those used in compiling NPB applications result from the resilience characteristics of MiBench applications; it turns out that compiling them into a larger number of resilient versions, *i.e.,* more protected iteration ratios, does not provide more fine-grained soft error resilience parameterization. For example, multiple MiBench applications show nearly identical fault injection results at 60% and 80% protected iteration ratios. Moreover, while having more resilient versions requires RTailor to use a higher unrolling factor (§V-B2), *i.e.,* loop bodies are copied more times, this increases the code size and thus can cause performance degradation due to more cache misses. Therefore, we decided to use a smaller number of resilient versions when compiling MiBench applications. Instead, we protected the instructions outside the selected loops with the conventional non-parameterized soft error resilience scheme [8], [33].

Finally, all benchmarks were compiled with non-parameterized soft error resilience (shown as "full" in following figures and called fully protected version [8], [33]) and also without any soft error protection to serve as the baseline (called unprotected version) for comparison.

D. Evaluation Enviroment

TABLE IV: Experimental Board Configuration

Board	Raspberry Pi 4B
CPU	Broadcom BCM2711, Quad core Cortex-A72 @ 1.5GHz
Memory	8GB LPDDR4-3200 SDRAM
OS	Ubuntu 22.04 LTS (64-bit ARM)
Target ISA	ARMv7-a with NEON and VFPv3

We measured the execution times of NPB applications on a Raspberry Pi 4B board whose configuration details are shown in Table IV. For MiBench applications, we measured their execution time using Gem5 simulator with SE mode [59] since most of them have very short execution times on real hardware. For example, MiBench applications' execution times measured atop Raspberry Pi 4B board are dominated by factors that are not related to RTailor's parameterized soft error resilience, *e.g.,* OS's system calls and IO operations. In contrast, the Gem5 simulator can measure applications' execution times precisely. To model ARM's typical embedded core, we configured the Gem5 simulator [59] using single core 8-issue out-of-order pipeline with 32KB/32KB 4-way set-associate L1 instruction/data caches (2 cycles hit) and a 1MB 16-way set-associative L2 cache (20 cycles hit).

E. Run-time Overhead with Varying Protection Ratios

Figure 8 and Figure 9 show the slowdown of RTailor for MiBench and NPB applications, respectively, compared to the unprotected version, when the protected iteration ratio varies from 20% to 100% (Full)[9]. On average, the geometric mean of RTailor's overhead is 27%∼80% across the protected iteration ratios. Except for `Patricia` from MiBench, the execution time of each application is proportional to the protected iteration ratio used. That is because as the protected iteration ratio increases, the program executes more protected iterations—each of which is of course slower than the unprotected iteration. Regarding `Patricia`, its performance anomaly results from the pointer-chasing loop where the majority of

[9]Prior work [8] has shown that the overhead from soft error detect scheme (DMR) can exceed 100%, let alone the overhead from idempotent recovery

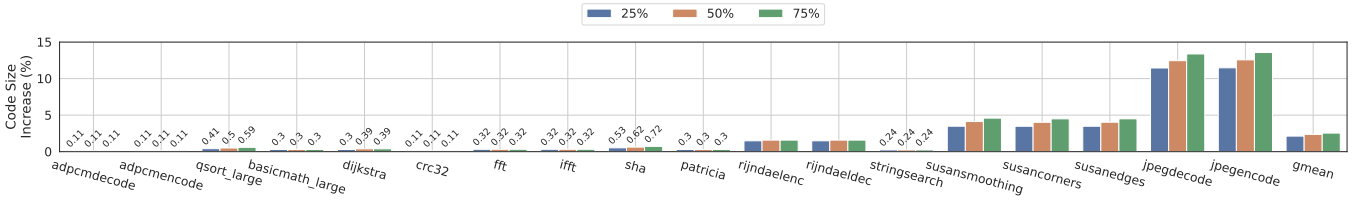


Fig. 10: Mibench application code size increase in percentage; baseline is the fully protected binary; lower is better

the execution time is spent waiting for load cache misses to be resolved. It turns out that the underlying out-of-order processor is able to schedule and execute those instructions inserted for soft error detection and correction while waiting for the load instructions to retire. This can effectively hide the execution time overhead of the protection code in `Patricia` regardless of the protected iteration ratio.

F. Code Size Increase

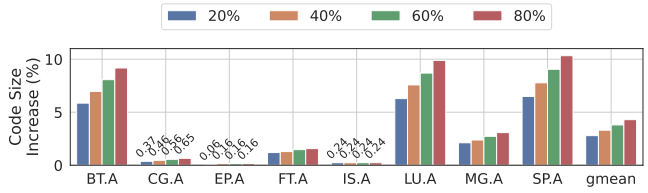


Fig. 11: NPB application code size increase in percentage; baseline is the fully protected binary; lower is better

Figure 10 and Figure 11 show the code size increase of RTailor for MiBench and NPB applications, respectively, across varying protected iteration ratios. In both figures, the code size of the fully protected (non-parameterized protected) version is set as the baseline and normalized to 1. The main difference between the baseline and RTailor's executables is that they conduct loop unrolling (§V-B2) and function inlining/cloning (§V-C) to parameterize soft error resilience. While these compiler transformations generally increase the code size and thus can lead to performance degradation in case of more instructions cache misses, it turns out that the maximum resulting code size increase is only 14%. Consequently, this rarely affects the performance due to the negligible instruction cache miss behavior in our experiments.

G. Statistical Fault Injection Campaign

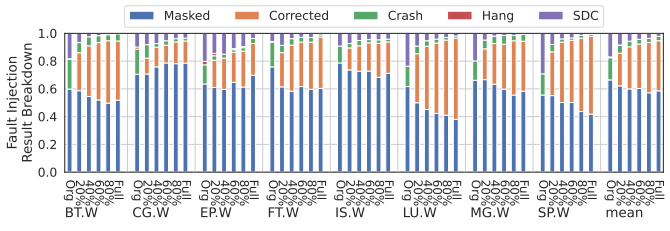


Fig. 12: Fault injection results of NPB applications

To evaluate RTailor's ability to parameterize soft error resilience, we leveraged statistical fault injection campaigns (§VII-B). For this purpose, we ran each benchmark application and its resilient versions, *i.e.,* including full and parameterized protection binaries varying the protected iteration ratio, 1200 times per each to reach 95% confidence level [56]. Figure 12 and Figure 13 show the fault injection results of NPB and MiBench applications, respectively. The ratio of each fault injection outcome (masked, corrected, crash, hand, or SDC) in the figures is the number of the runs that correspond to the outcome divided by 1200. As shown in the two figures, the ratios of correct runs (masked/corrected) increase in proportion to the protected iteration ratio used. Thus, these results highlight RTailor's ability to parameterize soft error resilience.

H. Schedulability Simulation of Real-Time Systems

To evaluate the schedulability improvement of RTailor's parameterized soft error protection over the fully protected scheme, we performed mixed-criticality system simulations with the testing framework of prior work [14].

1) Setup: The mixed-criticality system is based on a single-core processor. Each simulation scenario is defined by a tuple (n, λ, NU), where $n \in \{5, 10, 25, 50\}$ is the number of tasks, *i.e.,* the cardinality of the task set in the system, $NU \in [0:1]$ is the total processor utilization of tasks excluding re-executions, and $\lambda \in \{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}/hr$ is the system's fault rate per hour, picked from other prior works [60]–[62].

For a given (n, λ, NU) tuple, *i.e.,* simulation scenario, we randomly generate a number of mixed-criticality systems (MCS). To illustrate, under the same system fault rate (λ), MCS_1 and MCS_2 both have 5 (n) tasks but differ in terms of their utilization assignment to each task with the NU satisfied. For the task set of each MCS, a simulation run determines whether it is schedulable or not; MCS and its task set are interchangeably used because of their 1-1 correspondence.

We evaluated RTailor's schedulability improvements for the aforementioned mixed-critical system using both the EDF scheduler and the state-of-the-art scheduler proposed by the prior work [14]. Unlike the EDF scheduler, the novel scheduler of the prior work [14] has an ability to drop tasks if necessary. When it is impossible to schedule all tasks in the task set, the scheduler tries to drop some (re-)executions of the low-criticality tasks to reduce processor utilization with the failure rate constraints of all tasks still satisfied, which gives high-criticality tasks more time for execution. Since the prior work uses a tree-based algorithm [14] to find appropriate tasks (or their re-executions) to drop, we called it the tree scheduler hereafter for the sake of simplicity.

On the other hand, for the EDF scheduler, all tasks and their re-executions are admitted, *i.e.,* no task is dropped though

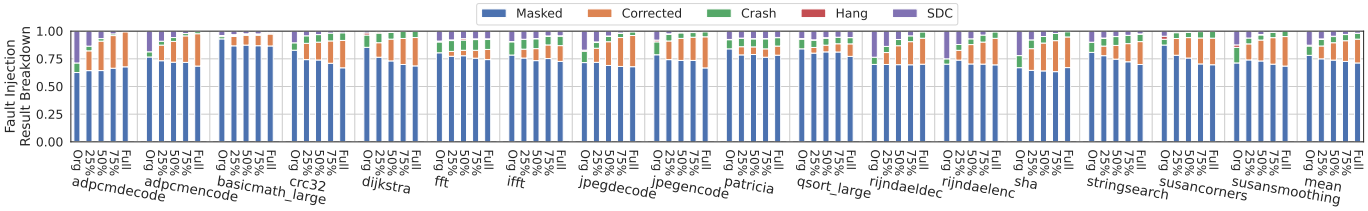


Fig. 13: Fault injection results of MiBench applications

they are unschedulable. This implies that if the task set is schedulable, then all tasks satisfy their failure rates.

TABLE V: Variables used in the schedulability simulation

Variable	Definition
n	Number of tasks in the task set
γ_i	The benchmark application; it serves as an index to some factors for generating c_{i_j} and $p^F_{i_j}$ of τ_{i_j}
c_{i_j}	WCET of τ_{i_j}
T_i	Period of τ_i
D_i	Deadline of τ_i
NU_i	Processor utilization of τ_i without re-execution
NU	Total processor utilization of the task set without re-execution
U_i	Processor utilization of τ_i with re-executions
U	Total processor utilization of the task set with re-executions
$p^F_{i_j}$	Actual failure rate of τ_{i_j}
p^{req}_i	Required failure rate of τ_i
λ	System's fault rate per hour

2) Task Set Generation: Again, each (n, λ, NU) tuple yields many randomly generated task sets. For each tuple, RTailor's parameterized soft error resilience and the conventional non-parameterized resilience schemes both have 1000 and 100 task sets for the EDF scheduler and the tree scheduler[10], respectively. For each task set generation, we take a two-step approach: (step-1) generating a task set with random tasks that are fully protected and (step-2) extending it with their re-execution versions in two different ways; the resulting two task sets share the same random tasks but differ in how to construct the re-execution version sequences, *i.e.,* one (RTailor's) consists of task versions created by RTailor's resilience parameterization—including full protection—while the other (the prior work's) of the replicas of the same fully protected tasks. To satisfy the failure rate constraint (§III-A), RTailor's task set with parameterized protection utilizes Algorithm 2 to construct the re-execution version sequence of each task[11], while the task set of the prior work [14] uses Equation (1) to determine how many times its task (fully protected) should be re-executed.

Table V shows the definitions of the variables used in the above task set generation, and they are detailed below:

- γ_i is a variable that we added to derive the WCET and actual failure rate of each task version complied with RTailor's resilience parameterization.
- Each task is assigned a random benchmark γ_i that serves as an index to the factors for generating c_{i_j} and $p^F_{i_j}$ below.
- The utilization NU_i of each task τ_i is determined using the UUnifast algorithm [63] such that $NU = \sum_{i=1}^{n} NU_i$.
- The period T_i of each task τ_i is uniformly sampled from $[50 : 1000] \subseteq \mathbb{N}$, and τ_i's deadline D_i is set to T_i.
- The WCET c_{i_j} of τ_{i_j} is defined as $c_{i_j} = T_i \times NU_i \times \frac{E_{\gamma_{i_j}}}{E_{\gamma_{i_{full}}}}$, where $E_{\gamma_{i_j}}$ is the measured execution time of the benchmark application γ_i's version j from §VII-E.
- The actual failure rate $p^F_{i_j}$ of τ_{i_j} is defined as $p^F_{i_j} = \lambda \times \epsilon_{\gamma_{i_j}}$[12], where $\epsilon_{\gamma_{i_j}}$ is the failure ratio of the benchmark application γ_i's version j; the failure means crash, hang, or SDC in the fault injection campaigns (§VII-G).
- The required failure rate p^{req}_i of τ_i is uniformly sampled from four values: $\{10^{-3}, 10^{-5}, 10^{-7}, 10^{-9}\}/hr$ that are obtrained from DO-178C standard [7].
- The processor utilization with re-executions U_i of τ_i is defined as $U_i = \sum_{k=0}^{N^{re-exec}_i} c_{i_{v_k}}/T_i$, where $VSEQ_i = (v_0, \ldots, v_{N^{re-exec}_i})$ is the re-execution version sequence satisfying the failure rate constraint.
- The total processor utilization U of the task set with re-executions is defined as $U = \sum_{i=1}^{n} U_i$.

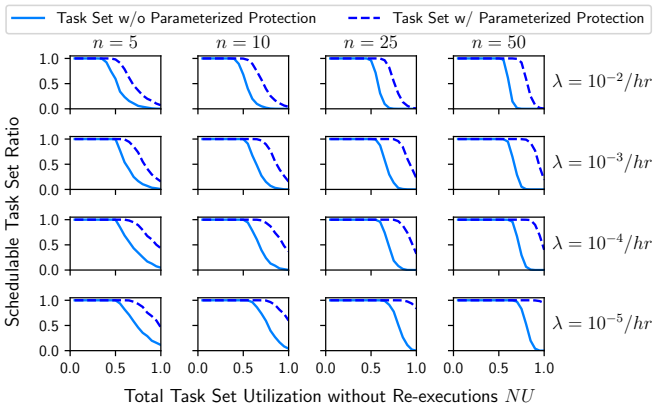


Fig. 14: Schedulability simulation of the EDF scheduler with γ_i from NPB. n : number of tasks in a task set, λ : fault rate/hour.

3) Schedulability Simulation Results: Figure 14 and Figure 16 show the schedulability results of the EDF scheduler and the tree scheduler, respectively, when γ_i is selected from NPB benchmark suite. Similarly, Figure 15 and Figure 17 provide the schedulability results from the EDF scheduler and the

[10]The purpose is to get the simulation runs down to a reasonable amount of time in that the tree scheduler algorithm [14] takes a while to find appropriate task re-executions to drop. Note that the prior work [14] generated the same number of task sets for the simulation of the tree scheduler.

[11]The original execution $\tau_i(0)$ of a given task τ_i can be a task version other than the fully protected version as long as the re-execution version sequence (including the original execution) satisfies τ_i's failure rate constraint.

[12]When computing $p^F_{i_j}$, λ are first scaled to same time period of c_{i_j} for correct actual failure rate calculation. We omit this for the sake of simplicity.

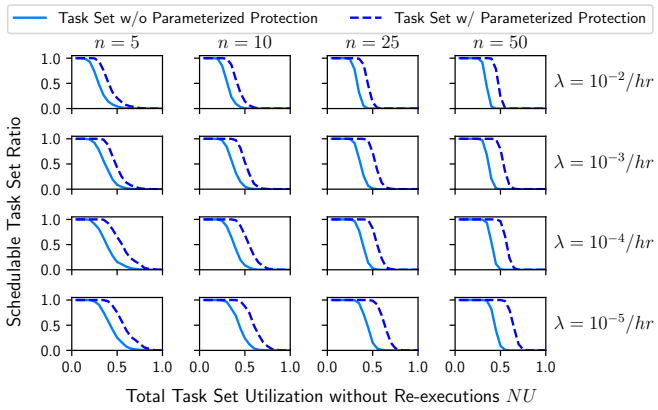


Fig. 15: Schedulability simulation of the EDF scheduler with γ_i from MiBench. n : number of tasks in a task set, λ : fault rate/hour.

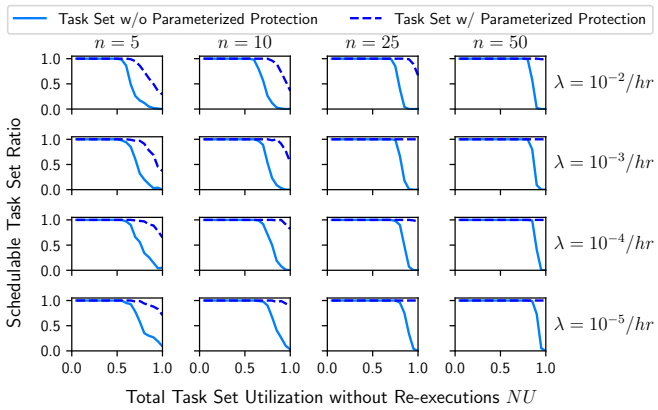


Fig. 16: Schedulability simulation of the tree scheduler with γ_i from NPB. n : number of tasks in a task set, λ : fault rate/hour.

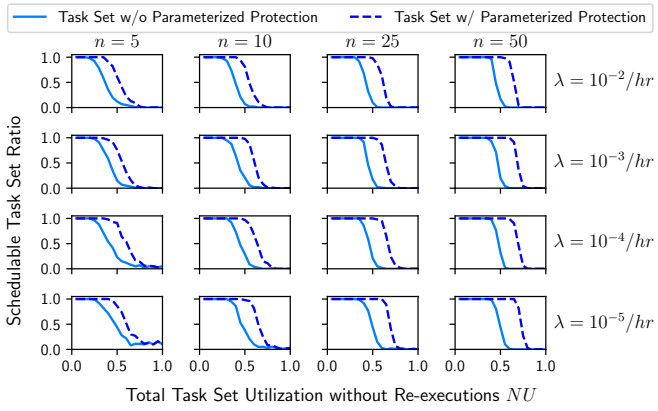


Fig. 17: Schedulability simulation of the tree scheduler with γ_i from MiBench. n : number of tasks in a task set, λ : fault rate/hour.

tree scheduler, respectively, when γ_i is selected from MiBench benchmark suite. In the figures, the x-axis corresponds to NU the total utilization of the task set without re-execution, *i.e.,* the one generated in the step one (§VII-H2), while the y-axis to the ratio of schedulable task sets to the total number of task sets. Since RTailor can minimize over-protection, which in turn reduces the processor utilization of each task, the resulting schedulability of the mixed-criticality system gets significantly improved as shown in the figures. Table VI summarizes the schedulability improvement of RTailor over the conventional non-parameterized resilience for both the EDF and the tree schedulers across different fault rates. Overall, with the help of RTailor's soft error resilience parameterization, the schedulability of the EDF scheduler improves up to 24%, while that of the tree scheduler improves up to 21%.

TABLE VI: The summary of the schedulability improvements

λ	10^{-2}	10^{-3}	10^{-4}	10^{-5}
EDF, $\gamma_i \in$ NPB	19%	23%	24%	20%
Tree, $\gamma_i \in$ NPB	21%	20%	18%	15%
EDF, $\gamma_i \in$ MiBench	12%	15%	16%	18%
Tree, $\gamma_i \in$ MiBench	17%	18%	20%	19%

VIII. Other Related Work

Prior schemes have studied energy-aware reliability management for real-time systems [64], [65]. As with RTailor, they provide real-time tasks with application-tailored reliability. To achieve this, the schemes leverage task replication, *i.e.,*, for a task violating the failure rate constraint (§II-B), they keep spawning the replica of the task until the probability of failing all task executions becomes lower than the required failure rate. However, the prior schemes often end up overprotecting tasks due to the lack of parameterized soft error resilience (Figure 1). In particular, the schemes take advantage of the overprotection as a safety net to enable aggressive dynamic voltage and frequency scaling (DVFS) that trades offs the reliability for energy efficiency [66], [67].

In contrast to the prior schemes that inherently cause task overprotection and mitigate it with the DVFS, RTailor can address the overprotection in the first place using soft error resilience parameterization. For a given task that requires re-execution(s) to bring its failure rate down to the required rate, RTailor composes the optimal sequence of the task versions whose resilience is parameterized in a fine-grained manner with a different protection ratio. Consequently, the resulting overprotection of RTailor is much smaller than that of the prior schemes. On the other hand, RTailor can work with their DVFS technique in synergy to further reduce overprotection and improve energy efficiency without violating the failure rate constraint of real-time tasks. This would be helpful especially when RTailor is extended for energy-harvesting systems [68]–[75], which we leave as future work.

IX. Conclusion

This paper presents RTailor that can parameterize soft error resilience for mixed-criticality real-time systems in a flexible and fine-grained manner. To offer custom soft error resilience for a given reliability demand, RTailor's compiler transforms the loops of each task so that their iterations are protected as frequent as the demand. The simulation results demonstrate that RTailor's parameterized soft error resilience can significantly improve the schedulability of the real-time tasks up to 21% compared to the state-of-the-art work.

X. Acknowledgements

We thank the anonymous reviewers and our shepherd for their valuable and insightful comments as well as the members of the Purdue CompArch research group for early discussions on the project. This work was supported by NSF grants 2001124 (CAREER), 2314681, and 1932074.

References

[1] W. Jang, "Soft-error tolerant quasi delay-insensitive circuits," Ph.D. dissertation, California Institute of Technology, USA, 2011.

[2] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *11th International Symposium on High-Performance Computer Architecture*, 2005.

[3] N. DeBardeleben, S. Blanchard, V. Sridharan, S. Gurumurthi, J. Stearley, K. Ferreira, and J. Shalf, "Extra Bits on SRAM and DRAM Errors - More Data From the Field," *Silicon Errors in Logic - System Effects*, 2014.

[4] F. Reghenzani, Z. Guo, and W. Fornaciari, "Software fault tolerance in real-time systems: Identifying the future research questions," *ACM Comput. Surv.*, mar 2023, just Accepted.

[5] L. Wang and K. Skadron, "Implications of the power wall: Dim cores and reconfigurable logic," *IEEE Micro*, vol. 33, no. 5, pp. 40–48, 2013.

[6] A. Kritikakou, P. Nikolaou, I. Rodriguez-Ferrandez, J. Paturel, L. Kosmidis, M. K. Michael, O. Sentieys, and D. Steenari, "Functional and timing implications of transient faults in critical systems," in *2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2022, pp. 1–10.

[7] B. Brosgol and C. Comar, "Do-178c: A new standard for software safety certification," ADA CORE TECHNOLOGIES NEW YORK NY, Tech. Rep., 2010.

[8] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *International symposium on Code generation and optimization*, 2005.

[9] M. Didehban and A. Shrivastava, "A compiler technique for processor-wide protection from soft errors in multithreaded environments," *IEEE Transactions on Reliability*, vol. 67, no. 1, pp. 249–263, 2018.

[10] M. A. De Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," in *ACM SIGPLAN Notices*, vol. 47. ACM, 2012, pp. 475–486.

[11] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed soft error detection and recovery to avoid due and sdc via tail-dmr," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 2, p. 32, 2017.

[12] S. Seo, G. Jo, and J. Lee, "Performance characterization of the nas parallel benchmarks in opencl," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, 2011, pp. 137–148.

[13] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.

[14] F. Reghenzani, Z. Guo, L. Santinelli, and W. Fornaciari, "A mixed-criticality approach to fault tolerance: Integrating schedulability and failure requirements," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 27–39.

[15] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[16] K. Mitropoulou, V. Porpodas, and M. Cintra, "Drift: Decoupled compiler-based instruction-level fault-tolerance," in *Workshop on Languages and Compilers for Parallel Computing*, 2013.

[17] I. Laguna, M. Schulz, D. F. Richards, J. Calhoun, and L. Olson, "Ipas: Intelligent protection against silent output corruption in scientific applications," in *IEEE/ACM Symposium on Code Generation and Optimization*, 2016.

[18] M. Didehban and A. Shrivastava, "nzdc: A compiler technique for near zero silent data corruption," in *Design Automation Conference*, 2016.

[19] J. Ma and Y. Wang, "Identification of critical variables for soft error detection," in *International Conference on Human Centered Computing*. Springer, 2016, pp. 310–321.

[20] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," in *ACM Sigplan Notices*, vol. 50. ACM, 2015, p. 2.

[21] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* IEEE, 2003, pp. 98–109.

[22] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe, "Reunion: Complexity-effective multicore redundancy," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 223–234.

[23] J. F. Wakerly, "Microcomputer reliability improvement using triple-modular redundancy," *Proceedings of the IEEE*, vol. 64, no. 6, pp. 889–895, 1976.

[24] S. Hudson, R. Shyama Sundar, and S. Koppu, "Fault control using triple modular redundancy (tmr)," in *Progress in Computing, Analytics and Networking: Proceedings of ICCAN 2017*. Springer, 2018, pp. 471–480.

[25] P. Ulbrich, M. Hoffmann, R. Kapitza, D. Lohmann, W. Schroder-Preikschat, and R. Schmid, "Eliminating single points of failure in software-based redundancy," in *2012 Ninth European Dependable Computing Conference*, 2012, pp. 49–60.

[26] G. Upasani, X. Vera, and A. González, "Setting an error detection infrastructure with low cost acoustic wave detectors," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012, pp. 333–343.

[27] G. Upasani, X. Vera, and A. González, "Reducing due-fit of caches by exploiting acoustic wave detectors for error recovery," in *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. IEEE, 2013, pp. 85–91.

[28] G. Upasani, X. Vera, and A. González, "Avoiding core's due & sdc via acoustic wave detectors and tailored error containment and recovery," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 37–48.

[29] G. Upasani, X. Vera, and A. Gonzalez, "A case for acoustic wave detectors for soft-errors," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 5–18, 2016.

[30] Q. Liu, C. Jung, D. Lee, and D. Tiwarit, "Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[31] J. Zeng, H. Kim, J. Lee, and C. Jung, "Turnpike: Lightweight soft error resilience for in-order cores," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 654–666.

[32] Y. Zhang and C. Jung, "Featherweight soft error resilience for gpus," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 245–262.

[33] M. De Kruijf and K. Sankaralingam, "Idempotent code generation: Implementation, analysis, and evaluation," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2013, pp. 1–12.

[34] H. Kim, J. Zeng, Q. Liu, M. Abdel-Majeed, J. Lee, and C. Jung, "Compiler-directed soft error resilience for lightweight gpu register file protection," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 989–1004.

[35] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "ido: Compiler-directed failure atomicity for nonvolatile memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 258–270.

[36] J. Jeong and C. Jung, "Pmem-spec: Persistent memory speculation (strict persistency can trump relaxed persistency)," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 517–529.

[37] J. Jeong, J. Zeng, and C. Jung, "Capri: Compiler and architecture support for whole-system persistence," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 71–83.

[38] J. Jeong, J. Hong, S. Maeng, C. Jung, and Y. Kwon, "Unbounded hardware transactional memory for a hybrid dram/nvm memory system," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 525–538.

[39] J. Zeng, J. Jeong, and C. Jung, "Persistent processor architecture," in *MICRO-56: 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023.

[40] J. Chang, G. Reis, and D. August, "Automatic instruction-level software-only recovery," in *International Conference on Dependable Systems and Networks (DSN'06)*, 2006, pp. 83–92.

[41] M. Didehban, S. R. D. Lokam, and A. Shrivastava, "Incheck: An in-application recovery scheme for soft errors," in *ACM/IEEE Design Automation Conference*, 2017.

[42] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "Plr: A software approach to transient fault tolerance for multicore architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 135–148, 2009.

[43] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization*, 2004.

[44] "Clang: a C language family frontend for LLVM." [Online]. Available: http://clang.llvm.org/

[45] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, may 2008.

[46] S. Muchnick *et al.*, *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[47] V. Sarkar, "Automatic selection of high-order transformations in the ibm xl fortran compilers," *IBM Journal of Research and Development*, vol. 41, no. 3, pp. 233–264, 1997.

[48] F. Dekking, *A Modern Introduction to Probability and Statistics: Understanding Why and How*, ser. Springer Texts in Statistics. Springer, 2005.

[49] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[50] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[51] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, may 2008.

[52] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "A comprehensive survey of industry practice in real-time systems," *Real Time Syst.*, vol. 58, no. 3, pp. 358–398, 2022.

[53] K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar, "Wcet measurement-based and extreme value theory characterisation of cuda kernels," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 279–288.

[54] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis, "An empirical survey-based study into industry practice in real-time systems," in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020, pp. 3–11.

[55] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms, third edition*, ser. The MIT Press. MIT Press, 2009.

[56] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. Leuven, BEL: European Design and Automation Association, 2009, p. 502–506.

[57] D. L. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, USA, 2004.

[58] Q. Liu, X. Wu, L. Kittinger, M. Levy, and C. Jung, "Benchprime: Effective building of a hybrid benchmark suite," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 1–22, 2017.

[59] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[60] X. Li, M. C. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. USA: USENIX Association, 2010.

[61] A. Jacobs, G. Cieslewski, A. D. George, A. Gordon-Ross, and H. Lam, "Reconfigurable fault tolerance: A comprehensive framework for reliable and adaptive fpga-based space computing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 5, no. 4, dec 2012.

[62] L. A. Tambara, P. Rech, E. Chielle, J. Tonfat, and F. L. Kastensmidt, "Analyzing the impact of radiation-induced failures in programmable socs," *IEEE Transactions on Nuclear Science*, vol. 63, 2016.

[63] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Syst.*, vol. 30, no. 1–2, p. 129–154, may 2005.

[64] M. A. Haque, H. Aydin, and D. Zhu, "On reliability management of energy-aware real-time systems through task replication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 813–825, 2017.

[65] M. Salehi, M. K. Tavana, S. Rehman, F. Kriebel, M. Shafique, A. Ejlali, and J. Henkel, "Drvs: Power-efficient reliability management through dynamic redundancy and voltage scaling under variations," in *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2015, pp. 225–230.

[66] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez, "Power-aware scheduling for periodic real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 5, pp. 584–600, 2004.

[67] M. Weiser, B. Welch, A. Demers, and S. Shenker, *Scheduling for Reduced CPU Energy*. Boston, MA: Springer US, 1996, pp. 449–471.

[68] Q. Liu and C. Jung, "Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2016, pp. 1–6.

[69] J. Choi, Q. Liu, and C. Jung, "Cospec: Compiler directed speculative intermittent computation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 399–412.

[70] J. Choi, H. Joe, Y. Kim, and C. Jung, "Achieving stagnation-free intermittent computation with boundary-free adaptive execution," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 331–344.

[71] Y. Zhou, J. Zeng, J. Jeong, J. Choi, and C. Jung, "Sweepcache: Intermittence-aware cache on the cheap," in *MICRO-56: 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023.

[72] J. Choi, H. Joe, and C. Jung, "Capos: Capacitor error resilience for energy harvesting systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, 2022.

[73] J. Choi, J. Zeng, D. Lee, C. Min, and C. Jung, "Write-light cache for energy harvesting systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[74] J. Choi, L. Kittinger, Q. Liu, and C. Jung, "Compiler-directed high-performance intermittent computation with power failure immunity," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2022, pp. 40–54.

[75] J. Zeng, J. Choi, X. Fu, A. P. Shreepathi, D. Lee, C. Min, and C. Jung, "Replaycache: Enabling volatile cachesfor energy harvesting systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 170–182.