This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers

Cross-Layer Software Dependability on Unreliable Hardware

Semeen Rehman[1], Kuan-Hsun Chen[2], Florian Kriebel[1], Anas Toma[1],
Muhammad Shafique[1], Jian-Jia Chen[2] and Jörg Henkel[1]
[1]Department of Informatics, Karlsruhe Institute of Technology (KIT), Germany
[2]Department of Informatics, TU Dortmund (TUD), Germany
Corresponding Author's Email: muhammad.shafique@kit.edu

***Abstract*—To enable reliable embedded systems, it is imperative to leverage the compiler and system software for joint optimization of functional correctness (i.e., vulnerability indexes) and timing correctness (i.e., deadline misses). This paper considers the optimization of the Reliability-Timing (RT) penalty, defined as a linear combination of the vulnerability and deadline misses. We propose a cross-layer approach to achieve reliable code generation and execution at compilation and system software layers for embedded systems. This is enabled by the concept of generating multiple versions for given application functions, with diverse performance and reliability tradeoffs, by exploiting different reliability-guided compilation options. As the execution time of a function is not fixed, the selection of the versions depends upon the execution behavior of the previous functions. Based on the reliability and execution time profiling of these versions, our reliability-driven system software decides the prioritization of the functions for determining their execution order and employs dynamic version selection to dynamically select a suitable version of a function. Specifically, our scheme builds a schedule table offline to optimize the RT penalty, and uses this table at run time to select suitable versions for the subsequent functions. A complex real-world application of "secure video and audio processing" composed of various functions is evaluated for reliable code generation and execution.**

I. Introduction

Aggressive technology scaling in the deep nanometer regime has led to serious system reliability threats like aging, soft errors, etc. [5, 10, 22, 30]. Soft errors are transient faults due to internal or external sources (e.g., high energy particle strikes) that manifest as spurious bit flips in the hardware and finally corrupt the correct application execution [5, 10]. Several reliability optimization techniques have been proposed at different system layers to mitigate soft error effects, most of which primarily rely on full-scale redundancy [10].

Hardware-based approaches mainly target spatial/temporal architectural redundancy using Dual/Triple Modular Redundancy (DMR, TMR) and design with reduced architectural vulnerability [22, 31]. However, these techniques introduce extra hardware circuitry and incur significant area/power overhead, which may violate the stringent design constraints of embedded systems. To alleviate this overhead, compiler-/software-based techniques have emerged as an attractive option.

Compiler-/Software-based approaches rely on instruction and/or data redundancy (DMR, TMR) [23, 27]. The approach in [12] duplicates the contents of narrow-width register values in 64-bit registers and performs error detection by checking the upper 32-bit and lower 32-bit values. The approaches in [23, 27] duplicate the instructions and insert check points during compilation for error detections. Other compiler-level approaches include reducing the register lifetime by rescheduling the assembly code [33], control flow checking [27, 32], etc. However, these approaches incur significant performance overhead (in most of the cases more than 2x-3x). Therefore, these techniques *do not respect the timing aspects* and thereby lead to an increased risk of deadline misses.

For real-time embedded systems, the *system software-based approaches* need to jointly account for: (1) **functional reliability**, i.e., for a given input, the correctness of the output values of a given application function considering faults in the underlying hardware; and (2) **timing reliability**, i.e., whether the correct application output is derived in time or after the deadline due to its prolonged execution exceeding estimations or expectations. An application that is the best from the perspective of functional reliability but incurring significant performance degradation might lead to deadline misses for delivering the correct output, i.e., degraded timing reliability. Consequently, it may jeopardize the overall system reliability. To ensure the reliability of an embedded system, it is imperative to make sure that the timing constraints are *mostly* obeyed, i.e., all the applications running in a system, most of the time, deliver their correct output in time. For *soft* and *firm* real-time systems, in which deadline misses are tolerable, but should be avoided or leveraged with functional correctness, compiler- and software-based approaches can be adopted.

System software-based approaches: To address soft errors in real-time systems, one might integrate fault tolerance techniques and task scheduling with on-time recovery, e.g., [9]. For distributed real-time systems, Izosimov et al. [16] propose a policy how to assign fault-tolerance to meet the timing constraints under specific reliability levels. The work in [19] analyzes the schedulability of real-time tasks for real-time systems based on processors with autonomic frequency scaling under aging. This analysis accounts for the aging-induced degradation for task mapping under life-time constraints. By considering the imprecise computation model, in which a task is composed of mandatory and optional parts of executions, Aydin et al. [3] explore how to maximize the total system reward under task recovery. Furthermore, Izosimov et al. [15] consider a more comprehensive imprecise computation model for maximizing the system reward in distributed systems by exploiting execution time properties (i.e., the best-, worst-, and

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers

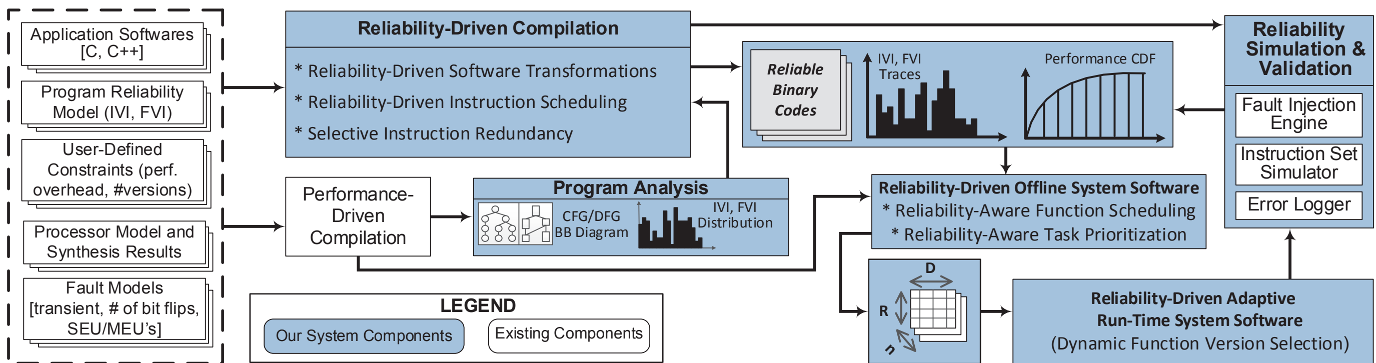


Fig. 1: Overview of our Cross-Layer Dependability illustrating the interactions between the reliability-driven compilation and reliability-driven system software layers for dependable code generation and execution.

average-case execution times).

Summarizing: state-of-the-art reliability methods mitigate reliability-related issues at a particular layer and do not fully leverage the cross-layer interactions and exploitation among different system layers. Therefore, these solutions provide optimizations for one design constraint, i.e., reliability by sacrificing other design constraints such as performance. State-of-the-art software-level techniques have, by far, not exploited their potential since the common belief, so far, was that reliability problems when occurring at the hardware level should also be addressed at the hardware level.

Challenge: *For highly dependable embedded systems, it is crucial to leverage/engage multiple system layers in an integrated fashion for joint optimization of functional and timing reliability in case the underlying hardware components are unreliable*.

The goal of this paper is to enable cross-layer software dependability on unreliable hardware by jointly addressing the issues related to the functional and timing reliability at multiple system layers (i.e., compiler, offline system software, and run-time system software), such that these system layers interact with each other and contribute towards the overall system reliability. In particular, this paper targets at reliable code generation and execution using integrated reliability-driven compilation and system software layers. Following the definitions in [6, 10, 11, 20] the term "cross-layer" is used referring to two or multiple adjacent layers which is not restricted to include both hardware *and* software layer.

To achieve high reliability, the flexibility in compilers to generate **multiple reliable versions** of a given function should be utilized such that, these function versions are identical in terms of their functionality and output, but differ in terms of their vulnerability and execution time properties. *The multiple versions of the same function provide the foundation for performance versus reliability tradeoffs and joint optimizations at both compiler and system software levels.*

A. Our Novel Contributions and Concept Overview

Our cross-layer software dependability scheme aims at optimizing the *Reliability-Timing* (RT) penalty, which is defined as the linear combination of vulnerability and deadline misses (see details in Section II-D). That is, given an application with n sequential functions, the studied *problem* is to generate and execute the application software such that the RT penalty is minimized. Fig. 1 shows an overview of our novel cross-layer scheme while illustrating the interaction between reliability-driven compiler, offline system software, and run-time system software layers. Our scheme employs the following **novel contributions**.

Concept of Multiple Reliable Function Versions (Section III, IV): Our reliability-driven compiler generates multiple versions (for a given function) with diverse performance and reliability properties under user-defined constraints of tolerable performance overhead and number of versions. Different reliability-aware transformations (like in [25, 26, 29, 33]) are employed to generate multiple reliable versions (with different assembly codes and hence different binary codes), which are then forwarded to the reliability-driven offline system software. In a cross-layer reliability stack, further reliability-performance tradeoff options can be obtained through traditional N-version programming. The distinction between multiple reliable compiled versions and N-version programming [7] is discussed in the Supplementary Material.

Reliability-Driven Offline System Software (Section V, VI): Given an application with n different functions each having multiple versions, our reliability-driven offline system software determines an appropriate executing sequence before the so-called "schedule table" (defining function execution schedules) is determined. Since exploring all possible ordering combinations is extremely time consuming, we provide a *function prioritization algorithm* to determine the execution order of the given functions. Since the actual execution time of a function is not always a constant, the probability distribution of the execution time of a function is taken into consideration by a *dynamic version selection scheme* to exploit the dynamic execution behavior by selecting suitable versions for the subsequent functions. When generating the *schedule table*, our scheme selects the versions of functions dynamically for minimizing the RT penalty.

Reliability-Driven Run-Time System Software (Section VI): It selects different function versions from the schedule table, prepared offline, depending upon the current execution behavior (the reliability penalty and the remaining time to the deadline) and *dynamically links* the corresponding binary codes of different functions[1].

[1]Note, such dynamic-linking techniques have also been used for the corrections of the software bugs, application reconfigurations, and dynamic applications in embedded systems, e.g., in wireless sensor networks [8].

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers

II. System Model

A. Fault and Architecture Models

The types of faults considered in this paper include transient faults/soft errors, single and multiple bit upsets in both the combinatorial and sequential logic. The processor under consideration is with RISC architecture, single core, and in-order execution. Let $\mathbf{C}$ be the set of hardware components in the processor, in which $\mathbf{C} = \{$pipeline, register file, address generation unit, $\ldots\}$. For each component c in $\mathbf{C}$, the area size A_c and the microarchitecture-dependent error probability ψ_c are both given a priori for evaluating the vulnerability w.r.t. component c.

To estimate the functional correctness at the instruction-level, we adopt the program reliability model "Instruction Vulnerability Index (IVI)" of [26] that captures the vulnerability of an instruction by considering both the spatial vulnerability (w.r.t. the area of different processor resources) and temporal vulnerability (w.r.t. instruction vulnerable periods in different processor components) of an instruction during execution. Based on the definition in [26], we can also define $\text{IVI}_{inst,c}$ as the vulnerability index of an instruction $inst$ in a component c. The total IVI_{inst} of an instruction $inst$ is then defined based on the weighted vulnerability in the components that the instruction is executed on:

$$\text{IVI}_{inst} = \frac{\sum_{c \in \mathbf{C}} \text{IVI}_{inst,c} \cdot A_c \cdot \psi_c}{\sum_{c \in \mathbf{C}} A_c}. \tag{1}$$

To characterize/estimate the vulnerability of a function version, this paper adopts the *Functional Vulnerability Index* (FVI) [25, 26] that is based on IVI and denotes the probability that a fault during the execution of this version leads to a visible error. For a binary version implementation V for a function, suppose that ψ_{inst} is the probability that instruction $inst$ is executed. The FVI of a version V is given as [25, 26]:

$$\text{FVI}_V = \sum_{\text{each instruction } inst \text{ in } V} \psi_{inst} \cdot \left(\sum_{c \in \mathbf{C}} \text{IVI}_{inst,c} \right). \tag{2}$$

We refer the readers to [25, 26] and the Supplementary Material for more details about the definitions and arguments on IVI and FVI. According to the definition, a version with higher FVI is more unreliable.

B. Application Model

We consider a soft real-time application that comprises of n functions $F = \{F_1, F_2, \ldots, F_n\}$. An instance of the application is required to finish the execution of all the given n functions. The execution order of these functions is obtained using our prioritization algorithm that minimizes the expected RT penalty. A simple example sequence would be: starting from function F_1, then F_2, $\ldots$, and finishing with function F_n. For the simplicity of presentation, we will implicitly assume that these n functions are independent (therefore they have the highest freedom of orderings). For completeness, we will discuss how our algorithm works if the functions have precedence constrains at the end of Section V.

We consider a function as the basic unit for making scheduling decisions. Each function may be implemented using different algorithms, e.g., different sorting algorithms, which may be implemented by different programmers. Each implemented function can be compiled using different transformations, e.g., reliability-aware loop unrolling [26], reliability-driven instruction scheduling [25, 33], and selective instruction redundancy [29].

We target soft real-time systems, in which deadline misses are possible. It is typically insufficient to examine the timing and reliability properties of an application by inspecting one execution instance. A common practice is to model the recurrent execution of the application by specifying its periodic execution behaviour [18, 21] with a period T and a relative deadline D with $D \leq T$. However, it is possible that an instance misses the deadline, which may cause a *domino effect* of deadline misses for the subsequent instances. For most control applications, in which the period T is fixed due to the specification and the required timing properties, such domino effects may make the system unstable, and, therefore deadline misses should be avoided. For most multimedia applications, in which the period T is a soft constraint imposed to increase the comfort of users' experience, deadline misses are possible, but the arrival time (release time) of the upcoming execution instances should be adjusted accordingly after a deadline miss. We focus on the latter applications, in which the arrival times of subsequent jobs are adjusted accordingly after a deadline miss. That is, if an execution instance of the application finishes before the periodicity, the periodicity will be enforced; otherwise, the next instance of execution will start after this instance finishes. That is, suppose that an instance is released at time t and finishes at time t'. When t' is less than $t+T$, the next instance starts at time $t+T$; otherwise, the next instance starts at time t'. Another option is to abort this deadline-missed instance before the next instance starts. For such a case, the corresponding reliability penalty has to be evaluated only till the relative deadline of the task.

Based on the above definition, it is clear that the probability of a deadline miss of one execution instance of the application is independent from the other execution instances. As a result, the deadline miss rate of one execution instance is also the deadline miss rate of the application for multiple executions.

C. Compilations

Among the possible versions of function F_i, we would like to characterize K_i versions that are *effective*. For each version of these K_i versions, the reliability-aware compiler [26] considers the tradeoff between the vulnerability index, represented by FVI, and the performance by considering the performance distributions. The details about the compilation tradeoffs will be presented in Section III. This paper assumes that K_i is given a priori as a user-defined parameter. These K_i versions should have the same functional guarantees, e.g., the bounded errors should be the same when operating on numerical data so that the error of the application is bounded. Moreover, we consider systems with a given (and fixed) fault rate η (in the unit of number of faults per time unit).

For each version $F_{i,j}$ of function F_i, the following information is obtained based on the *profiling* after compilation:

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers

- The cumulative density function (CDF) of the execution time, where $C_{i,j}(e)$ denotes the probability in which the execution time of such a version is less than or equal to e. When obtaining the density function, the influence of an error on the execution time is not considered.
- The probability density function (PDF) of the execution time, where $P_{i,j}(e)$ denotes the probability that the execution time of such a version is e.
- The reliability penalty $R_{i,j}$ of function version $F_{i,j}$ is defined by the system designers. Here, based on the functional vulnerability index and the given fault rate η, the expected number of faults in one execution of function $F_{i,j}$ can be defined by the product of η and the average-case execution time, i.e., $\eta \int_0^\infty P_{i,j}(x)xdx$. Since FVI denotes the probability of an error, the reliability penalty $R_{i,j}$ is defined as $\text{FVI}_{F_{i,j}} \cdot \eta \int_0^\infty P_{i,j}(x)xdx$.

Even though we define the metric of the reliability penalty specifically based on the FVI and the fault rate, the proposed approach for version selection in the system software also works for other vulnerability indexes, in which the reliability of the application is the summation of the vulnerability indexes of the selected versions.

We will present the design flow based on the assumption that the cumulative and probability density functions are continuous. The results can be easily extended to consider discrete cases (with probability massive functions). For the rest of this paper, we will assume that the profiling is precise. Based on this information, we will optimize the functional reliability and the timing reliability.

D. Optimization Objective

The design objective is to improve the reliability while meeting the timing constraints. Therefore, we need to exploit the deadline misses versus the functional reliability tradeoff. The Reliability-Timing (RT) penalty is defined as the linear combination of functional reliability (i.e., the reliability penalties in form of vulnerabilities) and timing reliability (i.e., the deadline misses). Specifically, for a user-defined parameter $0 \leq \alpha \leq 1$, the RT penalty is: $\alpha R + (1-\alpha) miss_rate$, where $miss_rate$ is the percentage of deadline misses for the application and R is the sum of the reliability penalties of the selected versions, defined in Section II-C. When α is closer to 0, the timing satisfaction is more important; when α is closer to 1, the functional reliability in the presence of faults in the underlying hardware is more important.

The **objective** of the studied problem in this paper can be described as follows: *Given an application with n functions, the objective is to leverage multiple system layers to generate and execute the application software such that the RT penalty is minimized.*

If function F_i already misses the deadline, it is clear that the application will miss its deadline no matter how the rest of the functions are executed. However, it is the users' choice to further improve the reliability or further improve the performance. Suppose that improving performance is preferred, the version with the minimum average execution time should be chosen. Suppose that the reliability improvement is preferred, the version with the minimum reliability penalty (hence, higher reliability) should be chosen.

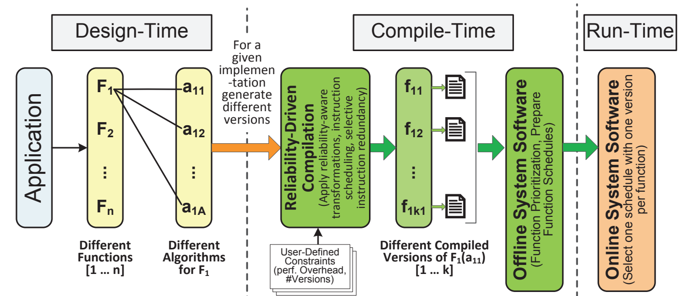


Fig. 2: Partitioning between design-, compile-, and run-time for the generation and utilization of multiple function versions

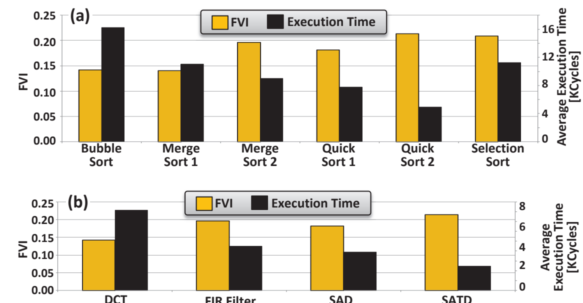


Fig. 3: Different algorithms have different FVI and average execution time: (a) comparing different sorting algorithms; (b) comparing different functions of the same application.

Note: although function versions with instruction-level redundancy for software-level error detection and recovery are provided, the system software layers aims at minimizing the probability of program errors as a means of improving the overall system reliability. The proposed approach is orthogonal to other means like thread replication, rollback recovery, etc. It is noteworthy that a reduced probability to program errors also corresponds to a reduced number of rollback recovery operations.

III. Reliability-Driven Compilation

Fig. 2 shows the high-level flow of design-, compile-, and run-time steps showing different versions generated by our reliability-driven compiler and their utilization.

Our experimental study in Fig. 3(b) illustrates that different functions of the same application exhibit unique reliability (i.e., FVI) and performance properties (i.e., execution time) because of their distinct algorithmic properties in terms of instruction profile and control flow. Moreover, same function may be implemented using different algorithms that exhibit distinct performance and reliability properties. Fig. 3(a) illustrates that different sorting algorithms and even different implementations by different programmers have different FVI and execution time values. Similarly, different compiler optimization options may also result in significant impacts on the FVI and execution time of the same function (an extensive study of this fact can be found in [25, 26, 29, 33]).

The proposed approach is to select representative function versions to cover a wide range of possible FVIs and average execution times for the resulting binary code of a function version.

Algorithm 1 shows the pseudo-code for generating up to

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers

Algorithm 1 Compilation for Function F_i

Input: Transformation Methods $\mathbf{T}$ [26], Instruction Rescheduling Methods $\mathbf{S}$ [25], Instruction Protection Method $\mathbf{IP}$, Function Implementations $\mathbf{I}_i$, maximum number of versions K_i, overhead Ω_i;
1: **for** each $\tau \in \mathbf{T}$ and each $I \in \mathbf{I}_i$ **do**
2: evaluate $\text{FVI}_{I,\tau}$ and the average execution time based on transformation τ for I under the given overhead constraint Ω_i (the details are in [26]);
3: **end for**
4: let $\mathbf{B}$ be the combinations of $\tau \in \mathbf{T}$ and $I \in \mathbf{I}_i$ in the RT penalty Pareto frontier;
5: **for** each $b_j \in \mathbf{B}$ and each $s \in \mathbf{S}$ **do**
6: evaluate the $\text{FVI}_{j,s}$ based on instruction rescheduling method s under the implementation b_j (the details are in [25]);
7: **end for**
8: replace $\mathbf{B}$ by taking the implementations in the RT penalty Pareto frontier based on the above loop;
9: **for** each $b_j \in \mathbf{B}$ **do**
10: perform selective instruction redundancy on the implementation b_j using $\mathbf{IP}$ (the details are in [29]);
11: **end for**
12: replace $\mathbf{B}$ by taking the implementations in the RT penalty Pareto frontier based on the above loop;
13: select K_i versions in $\mathbf{B}$ and profile the corresponding CDF/PDF and the FVI;

K_i versions of binary codes based on the function implementations $\mathbf{I}_i$ for function F_i, the compiler-based code transformations (called, transformation methods) $\mathbf{T}$ [26], instruction scheduling methods $\mathbf{S}$ [25], selective instruction redundancy using the given protection method [29] and a user-specified *tolerable performance overhead* Ω_i for reliability-driven compilations. Here, the tolerable performance overhead is defined as an upper bound of the increase in the average-case execution time, compared to the best performance version (under the average-case execution time). Unlike the RT-penalty based optimization by System Software, the specification of Ω_i is needed so that the reliability-driven compilation can limit the design space of transformations for generating the K_i versions.

As shown in [25, 26], it is usually better to first adopt the transformation methods to explore potential reliability improvement. Our approach here also first tries to adopt these transformation methods to obtain the corresponding FVI and average execution time (lines 1-3). Among the binary translations, the set of binary version implementations $\mathbf{B}$ for RT penalty Pareto frontier is taken (line 4). That is, none of any two binary version implementations in $\mathbf{B}$ will dominate each other in both FVI and the average execution time. Then, among all the binary version implementations in $\mathbf{B}$, we further consider the instruction rescheduling methods to further exploit some local improvement (lines 5-8). The set of binary version implementations $\mathbf{B}$ for RT penalty Pareto frontier is updated. These two reliability-driven compilation steps reduce the error probability. Afterwards, the selective instruction redundancy is applied for software-level error detection and recovery (lines 9-12) [29].

As the number of points in $\mathbf{B}$ may be more than K_i, the last step in Algorithm 1 is to select K_i (line 13). There can be many approaches to decide the final K_i versions by considering different strategies. For example, one possibility is to cluster $\mathbf{B}$ into K_i clusters by minimizing certain metrics. Another possibility is to divide the spectrum of the average execution time into K_i intervals, and find a representative in each interval. Here, we adopt the strategy by iteratively removing a Pareto point with the minimum slope, as this implies the improvement is less significant.

The compile-time prepared functions versions are then forwarded to the offline system software to generate execution schedules while optimizing for the RT penalty.

IV. Version Classifications and Properties

Before presenting the system software optimization for the minimization of RT penalty, this section presents how to classify the given K_i versions of function F_i based on their reliability and timing characteristics. Our classification is to identify a suitable high-performance version F_{i,h_i} so that we can redefine the properties of the other versions by referring to the high-performance version. For the rest of this paper, we define h_i to represent the index of the high-performance version for function F_i. In this section, we present how to identify h_i and use F_{i,h_i} as a reference version and evaluate two quantities for presenting the properties of each version with respect to this version h_i.

A. Identification of High-Performance Versions

To identify which version has higher performance, we need to analyze the difference of stochastic execution time (gap) between two versions x and y for function F_i.

Definition 1: Suppose that X and Y are the independent random variables that represent the execution times of versions $F_{i,x}$ and $F_{i,y}$, respectively. The cumulative density function $\text{C}_{i,x,y}(e)$, i.e., the probability, in which $X - Y$ is no more than a given gap e, is

$$\begin{aligned}\text{C}_{i,x,y}(e) &= P\{X - Y \leq e\} \\ &= P\{X \leq e + Y\} \\ &= \int_{-\infty}^{\infty} P_{i,Y}(y) \left(\int_{-\infty}^{e+y} P_{i,X}(x) dx \right) dy.\end{aligned}$$

Therefore, $\text{C}_{i,x,y}(e)$ provides a distribution function on the difference between two versions so that we can estimate the stochastic additional execution time by considering different versions $F_{i,x}$ and $F_{i,y}$. For two versions $F_{i,x}$ and $F_{i,y}$ of function F_i, we can say that $F_{i,y}$ has higher performance than $F_{i,x}$, where the probability of random variable Y larger than or equal to random variable X is at least 0.5 if

$$\text{C}_{i,x,y}(0) \leq 0.5.$$

However, the above definition of higher performance by comparing two versions does not have a transitive property. That is, when version $F_{i,x}$ outperforms version $F_{i,y}$ and version $F_{i,y}$ outperforms version $F_{i,z}$, we are not able to guarantee that $F_{i,x}$ outperforms version $F_{i,z}$. In such cases, the version with lowest-reliability penalty can be a reasonable reference.

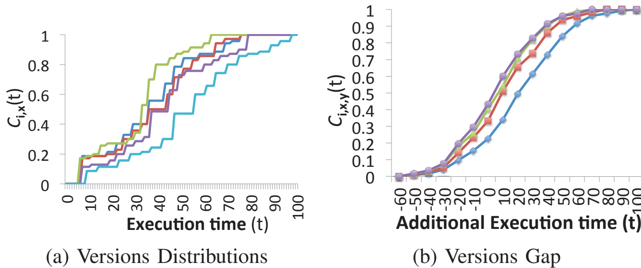
In Algorithm 2, we collect a set Γ of versions with better performance in the above definition and choose the one which has the lowest reliability penalty as the reference version h_i in set Γ. That is, if a version outperforms all the other versions in the above definition, it is put to Γ (lines 2-10). Please note, the version will be removed from Γ if there exists another version which outperforms it (lines 5-7). Due to the lack of the transitive property in the above definition, if Γ is an empty

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers

Algorithm 2 Classifications

Input: function F_i with K_i versions;
1: $\Gamma \leftarrow \emptyset$;
2: **for** each $j = 1, 2, \ldots, K_i$ **do**
3: $\quad \Gamma \leftarrow \Gamma \cup \{F_{i,j}\}$;
4: $\quad$ **for** each $k = 1, 2, \ldots, K_i, j \neq k$ **do**
5: $\quad\quad$ **if** $F_{i,k}$ outperforms $F_{i,j}$, i.e., $C_{i,j,k}(0) \leq 0.5$, **then**
6: $\quad\quad\quad \Gamma \leftarrow \Gamma \setminus \{F_{i,j}\}$;
7: $\quad\quad\quad$ break;
8: $\quad\quad$ **end if**
9: $\quad$ **end for**
10: **end for**
11: **if** Γ is an empty set **then**
12: $\quad$ put the version with the lowest-reliability penalty to Γ;
13: **end if**
14: return the version h_i which has the lowest-reliability penalty in set Γ;



(a) Versions Distributions (b) Versions Gap

Fig. 4: Multiple versions with stochastic execution time

set (lines 11-13), we will greedily place the version with the lowest-reliability penalty in Γ.

Fig. 4(a) shows the distribution function of execution time for multiple versions in one function. Please note, this example uses SHA as described in our experimental setup in Section VII-A. Fig. 4(b) presents the distribution function of additional execution time for each version which refers to high-performance version h_i. Note, the y-axis of charts ($C_{i,x}(t)$ and $C_{i,x,y}(t)$) are at most 100%, because they are the cumulative density functions representing the probabilistic properties of version $F_{i,x}$.

B. Properties of Each Version

We further define two quantitative properties to describe the benefit and additional execution time of each version by referring to the high-performance version h_i of function F_i:

- $w_{i,j}$ is the *benefit/effective profit* for version $F_{i,j}$ upgrading from version F_{i,h_i}. Suppose that $\Phi_{i,j}$ is the probability of deadline misses for a version $F_{i,j}$ by considering only itself for execution. That is,

$$\Phi_{i,j} = 1 - C_{i,j}(D), \tag{3}$$

where $C_{i,j}(D)$ is the probability in which the execution time is less than or equal to D. Therefore, $w_{i,j}$ is defined as follows:

$$w_{i,j} = \alpha \cdot (R_{i,h_i} - R_{i,j}) + (1 - \alpha) \cdot (\Phi_{i,h_i} - \Phi_{i,j}). \tag{4}$$

- $\mu_{i,j}$ is the *expected additional execution time* for version $F_{i,j}$ upgrading from version F_{i,h_i}. Suppose that $E[g_{i,j}]$ is the expected gap between version j and version h_i. That is,

$$\mu_{i,j} = E[g_{i,j}] = \int_{-\infty}^{\infty} g \cdot P_{i,j,h_i}(g) dg. \tag{5}$$

Here, the $P_{i,j,h_i}(g)$ is the probability density function (PDF) of the additional execution time (gap) g between version j and version h_i, i.e., the probability to have additional execution time equal to g.

V. Function Prioritization Algorithm

This section explores the function prioritization algorithm with multiple versions of each function F_i for Section VI to minimize the RT penalty. We first present a motivational example to explain why the execution ordering of the given functions matters. Then, we present a heuristic algorithm to decide the prioritization of the functions for determining the execution ordering.

A. Motivational Example

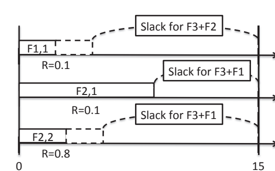


Fig. 5: Motivational example of prioritization.

We provide a motivational example to explain why the ordering of the execution of the functions matters for the optimization of the RT penalty. Suppose that we are given three functions, F_1, F_2 and F_3. Suppose that function F_1 has one version with variability in the execution time, in which $R_{1,1} = 0.1$, $P_{1,1}(2) = 0.5$, and $P_{1,1}(4) = 0.5$. Function F_2 has two versions, one version with variability in the execution time and the other with fixed execution time, in which $R_{2,1} = 0.1$ and $P_{2,1}(8) = 1$; $R_{2,2} = 0.8$, $P_{2,2}(3) = 0.9$ and $P_{2,2}(5) = 0.1$. Moreover, function F_3 has two versions with variability in the execution time, in which $R_{3,1} = 0.1$, $P_{3,1}(3) = 0.5$ and $P_{3,1}(9) = 0.5$; $R_{3,2} = 0.3$, $P_{3,2}(2) = 0.9$ and $P_{3,2}(8) = 0.1$. Suppose that the deadline D is 15.

With the above setting, there are 6 different orderings to execute the three functions. Due to the probability distribution of the execution times, the version chosen for the second function depends on how much time the first function takes. Fig. 5 illustrates the effect of different execution sequences on the remaining time.

For an executing sequence, we can try all possible execution scenarios to obtain the best execution plan under the execution sequence. Please note, the approach presented in Section VI can be used to decide how to optimally minimize the RT penalty with the slack, via building a dynamic programming table. The optimality proof of the proposed approach can be found in Theorem 1 in the Supplementary Material. In the above example, when $\alpha = 0.05$, the best result is with RT penalty 0.03325, whereas the RT penalty is 0.06475 for the worst ordering (see Table I). The gap between the best and the worst RT penalty is up to 95% as shown in Table I for different αs. Please also note that, the prioritization does not have any impact when considering only static version selections, since the reliability penalty and the miss rate can be both calculated statically. In other words, no matter which function is executed first, the RT penalty is not affected in that sense.

According to this motivational example, we know that an appropriate executing ordering also matters significantly for the minimization of the RT penalty. Enumerating all possible orderings exhaustively is of course a possible way to find a best executing ordering. However, with n functions, there are $n!$ different orderings, in which evaluating one given order

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers

	RT penalty when $\alpha = 0.01$	RT penalty when $\alpha = 0.05$
F_1, F_2, F_3	0.01605	0.06025
F_1, F_3, F_2	0.01065	0.03325
F_2, F_1, F_3	0.01605	0.06025
F_2, F_3, F_1	0.01695	0.06475
F_3, F_1, F_2	0.01065	0.03325
F_3, F_2, F_1	0.01065	0.03325

TABLE I: Individual Optimal RT penalty when $D = 15$, where the sequence is the corresponding ordering of the functions.

takes significant amount of time to do the minimization of RT penalty. It is clear that such an option is usually not possible when n is large.

Since each function has several versions, it is complicated to decide which function should be executed first and which of the available versions should be selected. In order to reduce the search space, we will perform a preprocessing procedure to estimate the potential profit with respect to RT penalty reduction by looking at only a subset of the given K_i versions of function F_i. After the preprocessing, we will put all the above subsets of the given versions of all the functions together and decide how to order the functions.

Please note that the following steps are just to decide the ordering of the functions. We may artificially reduce the available versions when deciding the orders, but all these available versions will be considered when we apply the dynamic programming approach in Section VI.

B. Preprocessing

The proposed algorithm starts with the initial step by considering that all the functions will use their high-performance versions. That is, we start from F_{i,h_i} for each function F_i. Then, the algorithm checks two properties of each version $F_{i,j}$: the effective profit $w_{i,j}$ and the expected additional execution time $\mu_{i,j}$.

For notational brevity, we define the expected execution time of function $F_{i,j}$ as $E[F_{i,j}] = \int_0^\infty P_{i,j}(x)xdx$. Throughout this section, we assume that the total expected execution time of the high-performance versions of the n functions, i.e., $\sum_{i=1}^n E[F_{i,h_i}]$ is no more than D. Otherwise, the probability of deadline misses is too high to be applied for real-time applications.

Moreover, we further define D' as the residual time by subtracting $\sum_{i=1}^n E[F_{i,h_i}]$ from the original relative deadline D, i.e.,

$$D' = D - \sum_{i=1}^{n} E[F_{i,h_i}], \tag{6}$$

in which $D' \geq 0$ as we assume that $\sum_{i=1}^n E[F_{i,h_i}]$ is no more than D.

The first step in the preprocessing is to find the Pareto curve with respect to $\mu_{i,j}$ and $w_{i,j}$. That is, a version $F_{i,k}$ is excluded if there exists a version $F_{i,j}$, in which $\mu_{i,j} \leq \mu_{i,k}$ and $w_{i,j} \geq w_{i,k}$. This can be done by applying the standard convex hull method (line 2) to identify the convex hull frontier with time complexity $O(K_i \log K_i)$ for a function F_i (lines 3-6). Fig. 2 in the Supplementary Material gives an example, in which the solid line shows the efficient convex hull frontier and the dotted line is the set of inefficient nodes for getting the better benefit. Therefore, the time complexity for this step

Algorithm 3 Preprocessing

Input: n functions, K_i versions, and μ_i and w_i of each version;
1: **for** each $i \in n$ **do**
2: adopt the standard convex hull method to construct the convex hull for F_i function by K_i versions;
3: **for** each $j \in K_i$ **do**
4: start from the node which has the smallest $\mu_{i,j}$ to identify the efficient frontier until the relative slope becomes negative with the previous node;
5: **end for**
6: return the nodes on the efficient frontier;
7: **end for**

is $O(\sum_{i=1}^n K_i \log K_i)$.

Now, suppose that there are $\kappa_i \geq 2$ points in the resulting Pareto curve of function F_i, and $\pi_i(\ell)$ is the index of the version for the ℓ-th point in the Pareto curve, in which $\mu_{i,\pi_i(\ell-1)} < \mu_{i,\pi_i(\ell)}$ for $\ell = 2, 3, \ldots, \kappa_i$. Moreover, we further define that

- $\Delta\mu_{i,\pi_i(\ell)} = \mu_{i,\pi_i(\ell)} - \mu_{i,\pi_i(\ell-1)}$ and
- $\Delta w_{i,\pi_i(\ell)} = w_{i,\pi_i(\ell)} - w_{i,\pi_i(\ell-1)}$,

for $\ell = 2, 3, \ldots, \kappa_i$.

Moreover, if the Pareto curve only has one version for function F_i, i.e., κ_i is 1, we will only consider the high-performance version h_i of function F_i when deciding the ordering of the given n functions. The pseudo code of the above preprocessing is presented in Algorithm 3.

C. Our Heuristic Algorithm for Function Prioritization

After the preprocessing, we sort the $\sum_{i=1}^n \kappa_i$ points for the n functions by a non-increasing order of the *benefit density* defined as $\frac{\Delta w_{i,\pi_i(\ell)}}{\Delta\mu_{i,\pi_i(\ell)}}$, in which ties are broken arbitrarily. The motivation for such a greedy ordering is to consider that all the possible elements in $\mathbf{S}$ can contribute benefit with execution time penalty (line 1 in Algorithm 4). For notational brevity, we denote the sorted list of these $\sum_{i=1}^n \kappa_i$ points as $\mathbf{S}$, and $|\mathbf{S}|$ as the number of elements in $\mathbf{S}$. For the j-th element in $\mathbf{S}$, let $\bar{w}_j$ be the corresponding value $\Delta w_{i,\pi_i(\ell)}$ and $\bar{\mu}_j$ be the corresponding value $\Delta\mu_{i,\pi_i(\ell)}$. As a result, we have

$$\frac{\bar{w}_1}{\bar{\mu}_1} \geq \frac{\bar{w}_2}{\bar{\mu}_2} \geq \ldots \geq \frac{\bar{w}_{|\mathbf{S}|}}{\bar{\mu}_{|\mathbf{S}|}}. \tag{7}$$

Moreover, for notational brevity, let $Sort(F_{i,\pi_i(\ell)})$ be the index of the implementation $F_{i,\pi_i(\ell)}$ in the sorted list $\mathbf{S}$ (line 1 in Algorithm 4). That is, $Sort(F_{i,\pi_i(\ell)})$ is j when $\frac{\Delta w_{i,\pi_i(\ell)}}{\Delta\mu_{i,\pi_i(\ell)}}$ has the j-th largest benefit density in the sorted list $\mathbf{S}$.

Let r be the minimum index such that $\sum_{i=1}^r \mu_i \geq D'$ (line 2). That is, by selecting the first r elements in $\mathbf{S}$, the summation of the expected additional execution time (with respect to the high-performance versions) of the first r elements in $\mathbf{S}$ is at most D'. Moreover, if $\sum_{i=1}^{|\mathbf{S}|} \mu_i < D'$, we greedily set r to $|\mathbf{S}|$.

For function F_i, we define the index θ_i in which $Sort(F_{i,\theta_i}) = r$ and $\frac{\Delta w_{i,\theta_i}}{\Delta\mu_{i,\theta_i}}$ is the minimum (lines 3-6). Note that θ_i is -1 if none of the points in the Pareto curve for function F_i is chosen before index r in $\mathbf{S}$. The index θ_i provides the reasonable reference point to evaluate the potential benefit . With this step, we have the specific index θ_i for each function F_i with respect to the residual time D'. That is, the index r defines the effective range of upgrading among these $\sum_{i=1}^n \kappa_i$ points for the n functions. This specific

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers

Algorithm 4 Function Prioritization

Input: n functions, K_i versions, the κ_i points in the Pareto curve for a function F_i, and the residual time D';
1: create the sorted list $\mathbf{S}$ based on (7) and the invert function $Sort()$;
2: let r be the minimum index such that $\sum_{i=1}^{r} \mu_i \geq D'$;
3: **for** each $i = 1, 2, \ldots, n$ **do**
4: find θ_i in which $Sort(F_{i,\theta_i}) = r$ and $\frac{\Delta w_{i,\theta_i}}{\Delta \mu_{i,\theta_i}}$ is the minimum;
5: let θ_i be -1 if there is no index θ_i with $Sort(F_{i,\theta_i}) = r$;
6: **end for**
7: let $\mathbf{N}_1 = \{F_i | \theta_i \geq 0, i = 1, 2, \ldots, n\}$;
8: let $\mathbf{N}_2 = \{F_i | \theta_i = -1, i = 1, 2, \ldots, n\}$;
9: order the functions in $\mathbf{N}_1$ ahead of the functions in $\mathbf{N}_2$;
10: order the functions F_is in $\mathbf{N}_1$ non-increasingly according to $\frac{w_{i,\theta_i}}{\mu_{i,\theta_i}}$;
11: order the functions F_is in $\mathbf{N}_2$ non-increasingly according to $\frac{\alpha \cdot R_{i,h_i} + (1-\alpha) \cdot \Phi_{i,h_i}}{E[F_{i,h_i}]}$;

index θ_i will be the possible efficient point in the κ_i points to use the residual time D' for upgrading from F_{i,h_i} to $F_{i,j}$.

Among the given n functions, we can now classify them into two sets $\mathbf{N}_1$ and $\mathbf{N}_2$ (lines 7-9), in which

- $\mathbf{N}_1 = \{F_i | \theta_i \geq 0, i = 1, 2, \ldots, n\}$, and
- $\mathbf{N}_2 = \{F_i | \theta_i = -1, i = 1, 2, \ldots, n\}$.

It is reasonable to then order the functions in $\mathbf{N}_1$ ahead of functions in $\mathbf{N}_2$. Among the functions in $\mathbf{N}_1$, ordering them according to the *benefit density*, i.e., from h_i version to θ_i version, would give the system better improvement in the RT reliability. Therefore, for the functions F_is in $\mathbf{N}_1$, we order these functions according to $\frac{w_{i,\theta_i}}{\mu_{i,\theta_i}}$ in a non-increasing order (line 10). For the functions in $\mathbf{N}_2$, in our definitions, they are selected with their high-performance versions. Therefore, for functions F_is in $\mathbf{N}_2$, we order them in a non-increasing order of $\frac{\alpha \cdot R_{i,h_i} + (1-\alpha) \cdot \Phi_{i,h_i}}{E[F_{i,h_i}]}$ (line 11), i.e., the RT penalty divided by the expected execution time.

Algorithm 4 presents the pseudo-code of the proposed algorithm for deciding the prioritization of the functions. The overall time complexity is $O(\sum_{i=1}^{n} K_i \log K_i)$. After the executing order is decided, we can further use the dynamic programming approach proposed in next section VI to obtain the optimal RT penalty with the given executing ordering.

Specifically, if the functions have *a partial order instead of an unknown order*, we can adopt the prioritization algorithm to obtain the partial executing order for the subset of functions which have no precedence constraints with each other in directed acyclic graphs (DAG). After the priorities of subset functions are determined, we can update their successors dependence by removing the predecessors in the selection pool of functions. As a consequence, we can derive a set of executing functions sequentially until the remaining dependences cannot be removed. Then we start the prioritization algorithm for the optimization of application reliability.

VI. Dynamic Version Selection Scheme

After deriving the execution ordering in Section V, we present the system software optimization based on the compiled versions for the given functions to minimize the RT penalty. Note that, throughout this section, we will consider that the execution ordering is given. For notational brevity, we execute the functions in the order of $F_1, F_2, \ldots, F_n$. We will start from the simplest case by considering applications with only one function. Then, we will move further to consider multiple functions.

A. One Function

When the application has only one function, based on the given information for the applications, we can evaluate the probability $\Phi_{1,j}$ of deadline misses for a version j of function F_1 by Eq. 3. Therefore, by selecting the version j, the RT penalty for the application with one function is defined as $\alpha R_{1,j} + (1-\alpha)\Phi_{1,j}$. It is clear that the version j^* that minimizes $\alpha R_{1,j^*} + (1-\alpha)\Phi_{1,j^*}$ should be selected for execution.

B. Multiple Functions

This subsection explores the minimization of the RT penalty when the application has multiple functions. The simplest way is to select the versions statically. However, finding the optimal selection with the minimum RT penalty with static version selections is in general $\mathcal{NP}$-hard. The proof of NP-hardness is presented in Theorem 2 in the Supplementary Material.

Unfortunately, the optimal static version selections with the minimum RT penalty may still be too pessimistic. Consider the following motivational example with two functions. Suppose that function F_1 is executed with one specific version with high variability in the execution time, in which $R_{1,1} = 0.1$, $P_{1,1}(1) = 0.5$, and $P_{1,1}(9) = 0.5$. Function F_2 has two versions with fixed execution times, in which $R_{2,1} = 0.1$, $P_{2,1}(9) = 1$, $R_{2,2} = 0.3$, and $P_{2,2}(1) = 1$. Suppose that D is 10. There are only two options for static version selections as shown in Fig. 6(a), i.e., with $\{F_{1,1}, F_{2,1}\}$ or $\{F_{1,1}, F_{2,2}\}$. For the case $\{F_{1,1}, F_{2,1}\}$, as the deadline miss rate is 50%, we know that the RT penalty is $0.2\alpha + 0.5 \cdot (1-\alpha)$. For the case $\{F_{1,1}, F_{2,2}\}$, as the deadline miss rate is 0%, we know that the RT penalty is 0.4α. Since $0.4\alpha > 0.2\alpha + 0.5 \cdot (1-\alpha)$ when $\alpha > \frac{5}{7}$, for the above example, we should choose

- the execution versions $\{F_{1,1}, F_{2,1}\}$ if $\alpha > \frac{5}{7}$, and
- the execution versions $\{F_{1,1}, F_{2,2}\}$ otherwise.

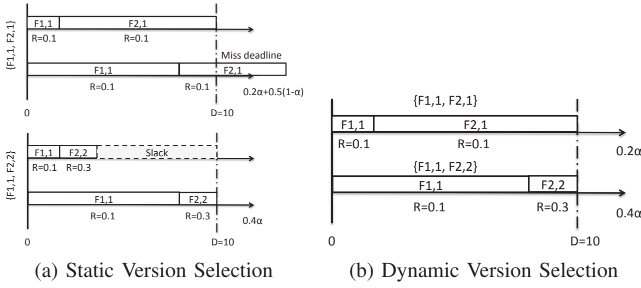
However, the above static assignment is pessimistic, as function F_2 can react according to different execution behaviours of function F_1. When function F_1 finishes very early, i.e., at time 1, function F_2 can adopt the high-reliability version $F_{2,1}$ with longer execution time, when the remaining time to the deadline is sufficiently large. Moreover, when function F_1 finishes very late, i.e., at time 9, function F_2 has to run the low-reliability version $F_{2,2}$ with shorter execution time, when the remaining time to the deadline is too small. The above dynamic version selection of function F_2 is with RT penalty equal to 0.3α, which is lower than the RT penalty 0.4α of the optimal static version selection $\{F_{1,1}, F_{2,2}\}$ when $\alpha \leq \frac{5}{7}$ as shown in Fig. 6(b).

Therefore, for the rest of this section, we will present the dynamic version selections, in which a schedule table is prepared offline and the scheduler adopts the suitable versions of the functions according to the run-time execution behavior in an online fashion. Also, we assume that θ_i is the preferred version of function F_i when F_i has already missed the deadline. For notational brevity, we define $\rho(i) = \sum_{\ell=i}^{n} R_{\ell,\theta_\ell}$ for the total reliability penalty from F_i to F_n when function F_i already misses the deadline.

The decision for function F_i depends on the execution behavior of functions $F_1, F_2, \ldots, F_{i-1}$. In the literature of

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers



(a) Static Version Selection (b) Dynamic Version Selection

Fig. 6: Motivational example of version selections. In figure (b), it is shown that the dynamic version selection reconciles the advantage of both selections and avoids the deadline miss and slack wastage.

real-time systems, when the actual execution time is less than the estimated worst-case execution time, the unused time is called *slack*. Slack reclamation and management have been studied to improve the quality of the system or reduce the energy consumption of the system dynamically. However, as this paper does not assume any worst-case behavior, we will analyze the timing behavior based on the estimated probability functions to minimize the RT penalty.

The selection of the execution version for function F_i depends on (1) the execution behavior up to now of functions $F_1, F_2, \ldots, F_{i-1}$ and (2) the reaction of functions F_{i+1}, $F_{i+2}, \ldots, F_n$ according to the execution behavior of function $F_1, F_2, \ldots, F_i$. To capture the properties in the first part, we need to know how much reliability penalty the functions $F_1, F_2, \ldots, F_{i-1}$ have incurred and how much execution time the functions $F_1, F_2, \ldots, F_{i-1}$ have elapsed. We will consider all possible scenarios for these properties by exploring possible values. The properties in the second part will be captured by referring to a table entry which stores the reactions of F_{i+1}, $F_{i+2}, \ldots, F_n$.

Our approach builds a 3-dimensional table $G()$ for execution behavior. Let $G(i, r, t)$ be an entry that stores the minimum RT penalty for the given n functions under the following conditions:

- $F_1, F_2, \ldots, F_{i-1}$ have finished at time $D - t$, and
- $F_1, F_2, \ldots, F_{i-1}$ have total reliability penalty r.

Furthermore, the decision of $G(i, r, t)$ also depends how the functions F_{i+1}, $F_{i+1}, \ldots, F_n$ will react according to the execution behavior of function F_i.

According to the above structure, we have to build $G(i + 1, r', t')$ for all possible r' and t' values first so that the entries can be used when we need to consider $G(i, r, t)$. Therefore, the procedure starts from the last function F_n. The entries for $F_{n-1}, F_{n-2}, \ldots, F_1$ are built later sequentially. To build $G(n, r, t)$, we first find $j^*(n, r, t)$ which is the version of function F_n with the minimum RT penalty (lines 1-5 in Algorithm 5), defined as follows:

$$j^*(n, r, t) = \begin{cases} \arg\min_j \alpha(r + R_{N,j}) + (1 - \alpha)(1 - C_{N,j}(t)), & t > 0 \\ \theta_n, & t \leq 0 \end{cases} \tag{8}$$

Therefore, we know that

$$G(n, r, t) = \alpha(r + R_{N,j^*}) + (1 - \alpha)(1 - C_{N,j^*}(t)), \tag{9}$$

where j^* is $j^*(n, r, t)$.

Now, let's consider the case to build an entry $G(i, r, t)$ where $i = n - 1, n - 2, \ldots, 1$ (lines 6-18). Clearly, when $t \leq 0$, we know that $j^*(i, r, t)$ is θ_i and $G(n, r, t)$ is $\alpha(r + \rho(i)) + (1 - \alpha)$. We now consider the other case when $t > 0$. If that function F_i selects version j, we know that

- the probability that F_i finishes with execution time x (when $x \leq t$) is $P_{i,j}(x)$,
- the minimum RT penalty for the n functions has been calculated and stored in $G(i + 1, r + R_{i,j}, t - x)$ when the execution time of F_i is x, where $x \leq t$, and
- the probability when $x > t$ is $(1 - C_{i,j}(t))$ by executing all the functions $F_{i+1}, F_{i+2}, \ldots, F_n$ with the default versions $\theta_{i+1}, \theta_{i+2}, \ldots, \theta_n$, respectively.

For notational brevity, we define $H_j(i, r, t)$ as the penalty by using the above properties, where

$$\begin{aligned} H_j(i, r, t) = & \int_{x=0}^{t} P_{i,j}(x) \cdot G(i + 1, r + R_{i,j}, t - x)dx \\ & + (1 - C_{i,j}(t)) \cdot (\alpha(r + R_{i,j} + \rho(i + 1)) + (1 - \alpha)). \end{aligned} \tag{10}$$

The first part in the right hand side in (10) for the integration considers the convolution when the execution time of $F_{i,j}$ is no more than t, while the second part considers the impact that $F_{i,j}$ already misses the deadline. Suppose that $j^*(i, r, t)$ is the index of j which minimizes $H_j(i, r, t)$ in Equation (10). Therefore, for $i = n - 1, n - 2, \ldots, 1$, we know that

$$G(i, r, t) = H_{j^*}(i, r, t) \tag{11}$$

where j^* is $j^*(i, r, t)$. Clearly, building $G(i, r, t)$ requires time complexity $O(K_i t) = O(K_i D)$.

Therefore, by building $G(i, r, t)$ from $i = n$ to $i = 2$, we can build $G(1, 0, D)$ and find $j^*(1, 0, D)$ (line 19), which gives the solution how the first function F_1 should be executed. Clearly, function F_1 uses only one version $j^*(1, 0, D)$. The other functions may require multiple versions to fully exploit the dynamic behavior of function executions.

Let $R_{\max}$ be the maximum (i.e., worst) reliability penalty that the system can achieve, i.e., $\sum_{i=1}^{n} \max_j R_{i,j}$. The above procedure requires time complexity $O(K_i D^2 R_{\max})$ for building $G(i, r, t)$ for r in the range of 0 and $R_{\max}$ and t in the range of 0 and D. As a result, the total time complexity is $O(\sum_{i=1}^{n} K_i D^2 R_{\max})$.

When all the possible values of execution time and reliability penalties are discretized, it is not difficult to see that the above procedure can optimally minimize the RT penalty. The optimality of the above procedure is proved by using the mathematical induction hypothesis in Theorem 1 included as supplementary material, where the base case starts from function F_n.

The above presentation requires to build the table for all possible values of t and r. However, it is not necessary to build some non-achievable entries in the table. For notational brevity, suppose that $R_{\max}(i)$ is $\sum_{\ell=1}^{i-1} \max_j R_{\ell,j}$ and $R_{\min}(i)$ is $\sum_{\ell=1}^{i-1} \min_j R_{\ell,j}$. For building $G(i, r, t)$, we only have to consider r in the range of $R_{\min}(i)$ and $R_{\max}(i)$. Moreover, we can change the units of the time and the reliability penalties. For example, we can build the table based on the timing unit of 0.1 msec or 0.01 msec. The larger the timing unit and the reliability unit adopted, the more the loss of accuracy and the less the complexity of table construction.

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers

Algorithm 5 Offline Table Construction

Input: n functions, CDF and PDF of the functions, units δ and σ, weighted parameter α, and the default versions $\theta()$ after observing the deadline misses;

1: **for** $r \leftarrow \left\lfloor \frac{R_{\min}(n)}{\sigma} \right\rfloor \sigma, \ldots, \left\lceil \frac{R_{\max}(n)}{\sigma} \right\rceil \sigma$ stepped by σ **do**
2: **for** $t \leftarrow 0, \ldots, \left\lceil \frac{D}{\delta} \right\rceil \delta$, stepped by δ **do**
3: calculate $j^*(n,r,t)$ and $G(n,r,t)$ by using Equations (8) and (9);
4: **end for**
5: **end for**
6: **for** $i \leftarrow n-1, n-2, \ldots, 2$ **do**
7: **for** $r \leftarrow \left\lfloor \frac{R_{\min}(i)}{\sigma} \right\rfloor \sigma, \ldots, \left\lceil \frac{R_{\max}(i)}{\sigma} \right\rceil \sigma$ stepped by σ **do**
8: **for** $t \leftarrow 0, \ldots, \left\lceil \frac{D}{\delta} \right\rceil \delta$, stepped by δ **do**
9: **if** $t = 0$ **then**
10: $j^*(i,r,t) \leftarrow \theta_i$; $G(i,r,t) \leftarrow \alpha(r + \rho(i)) + (1-\alpha)$;
11: **else**
12: for each $j = 1, 2, \ldots, K_i$, calculate $H_j \leftarrow \int_{x=0}^{t} P_{i,j}(x) \cdot G(i+1, \left\lfloor \frac{r+R_{i,j}}{\sigma} \right\rfloor \sigma, \left\lfloor \frac{t-x}{\delta} \right\rfloor \delta) dx + (1 - C_{i,j}(t)) \cdot (\alpha(r + R_{i,j} + \rho(i+1)) + (1-\alpha))$;
13: $j^*(i,r,t) \leftarrow \arg\min_{j=1,2,\ldots,K_i} H_j$;
14: $G(i,r,t) \leftarrow H_{j^*(i,r,t)}$;
15: **end if**
16: **end for**
17: **end for**
18: **end for**
19: calculate $j^*(1,0,D)$ and $G(1,0,D)$ with the same procedure in Steps 13 and 14;
20: return the table j^*;

Algorithm 5 adopts the above approximations by using δ as the timing unit and σ as the reliability penalty unit. The procedure is the same as the flow presented above. The time complexity of Algorithm 5 is $O(\sum_{i=1}^{n} K_i (\frac{D}{\delta})^2 \frac{R_{\max}}{\sigma})$. The space complexity is $O(n \frac{D}{\delta} \frac{R_{\max}}{\sigma})$.

C. Adaptive Run-Time System Software

The run-time system software performs version selection at run time by determining which function version F_i should be executed. F_1 is executed by the version $j^*(1,0,D)$. But, the other functions F_is may have different versions, depending upon the achieved reliability penalty of functions $F_1, F_2, \ldots, F_{i-1}$ and the remaining time to the deadline of this execution instance of the application.

According to Algorithm 5 when the remaining time to the relative deadline is t and the reliability penalty for the first $i-1$ functions is r, the run-time system software looks up the table entry $j^*(i,r,t)$. However, by considering the timing unit δ and the unit σ of the reliability penalties, the run-time system software instead looks up the entries with $j^*(i, \left\lfloor \frac{r}{\sigma} \right\rfloor \sigma, \left\lfloor \frac{t}{\delta} \right\rfloor \delta)$. Therefore, the binary version implementation F_{i,j^*} is selected, where j^* is $j^*(i, \left\lfloor \frac{r}{\sigma} \right\rfloor \sigma, \left\lfloor \frac{t}{\delta} \right\rfloor \delta)$.

Therefore, after the table j^* is built, the entries of j^* should be stored in the main memory as a look-up table. With such a mechanism, deciding a version to be executed requires only $O(1)$ time. However, the space complexity becomes a problem. Moreover, this may result in many redundant entries. It is not necessary to keep all the entries of j^*. For example, when $j^*(i,r,t)$ remain the same in the range of $[r_1, r_2]$ and in the range of $[t_1, t_2]$, the run-time system software only needs to keep one entry for the index. Therefore, only the representative entries are stored. Moreover, some entries can be further removed if the difference of the RT penalty is too small between two entries to reduce the memory overhead. In general, the system designers can decide the tolerable overhead for the resulting table.

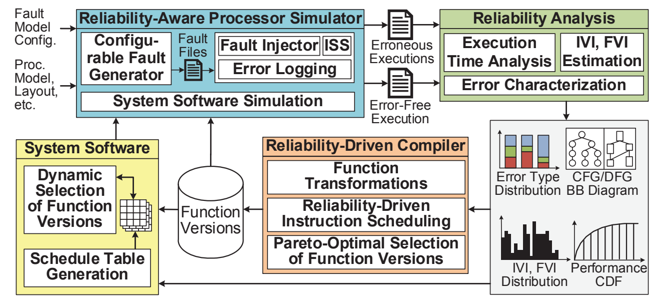


Fig. 7: Experimental setup with reliability-driven compiler, system software, and processor simulator.

Note that the table j^* should be protected so that the run-time system software can select the correct versions for minimizing the RT penalty. By removing the sparse entries, as discussed above, we can reduce the memory protection overhead. Following the prominent industrial and research trends of AMD [2] and IBM [14], in our experiments, we consider protected caches and memory. The performance overhead is at most $(n-1)$ multiplied with the overhead of fetching one table entry from the main memory. This is considered negligible, compared to the execution time of one application iteration.

VII. Results and Discussion

This section presents our experimental results based on simulations using a reliability-aware processor simulator and a reliability analysis program with profilers for obtaining the reliability and timing properties.

A. Experimental Setup and Compilations for Multiple Versions

The overview of the experimental setup is shown in Fig. 7. The reliability-driven compiler is based on the GCC framework and extended with several reliability-driven transformations [26] and an instruction scheduling algorithm [25] along with our Pareto optimal selection of function versions. The reliable binary codes of various application functions are forwarded to the system software which is implemented in C++. A reliability-aware Leon-II processor simulator is employed for executing all function versions with different inputs generating the information (e.g., instruction trace, register read/write accesses, etc.) that is required by the Reliability-Driven offline system software to analyze the performance and reliability properties of the function versions and to create the tables used by the Reliability-Driven online system software for dynamically selecting appropriate function versions. The simulator is a SPARC-v8 instruction set simulator (ISS) that is generated using the ArchC architecture description language and related tools [4]. The simulator is enhanced with an in-house developed configurable fault generator and injector, and error logging capabilities which is required for a detailed reliability analysis. For an accurate estimation, processor synthesis results (area, frequency, etc.) and fault model configurations (number of bit flips per fault, fault rate, fault distribution, etc.) are input to the reliability-aware processor simulator. Realistic fault rates are obtained using the neutron flux calculator [1] and coordinates of a given location. In our experiment, considering a processor frequency of 100MHz (representing embedded processors), we have used three different fault rates

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers

of $10^{-8}, 10^{-7}$, and 10^{-6} (in the unit of $\#faults/cycles$) to cover both terrestrial and aerial use cases. For representing processors with higher frequency, further accelerated fault rates could be used as typically adopted for fault injection experiments [13, 17]. After randomly generating several fault scenarios using the fault generator (i.e., determining for a large number of analysis runs, when and in which processor component faults should occur), the faults are injected by the fault injector during the function version execution on the ISS. The manifested effects on the application program layer are finally monitored using the error logger and categorized based on the severity from the user's perspective (e.g., application failure, incorrect output, correct output). More details on the fault injection and compilation infrastructure are presented in [24, 26] and the Supplementary Material. Faults are injected randomly (as it is done in [22, 28]) in different processor components and their effects are observed at the application program layer. The results of the fault injection experiments are finally used to accomplish two tasks (a) estimating the software-level masking properties of the applications used and (b) analyzing the main reason e.g. for application failures which could range from accessing prohibited memory regions to non-decodable instructions.

For experimental evaluation, we have employed various functions of different applications from MiBench like: (1) the "H.264" video encoder with three key functions "SAD", "DCT", and "SATD"; (2) "ADPCM"; (3) "CRC"; (4) "SusanS"; and (5) "SHA". In order to realize a complex real-world application scenario for reliable code generation and execution, we have integrated all these applications into one big application namely "secure video and audio processing". However, they can also be seen as parallel independent demands on a single core processor. Note, for each application at least 3 (up to 7) different function versions are generated and evaluated. For the defined order experiment, we use the following ordering of these 7 functions: "SAD", "SATD", "DCT", "SusanS", "CRC", "ADPCM", "SHA". In the experimental setup, we choose the first version returned from the compiler as the preferred version θ_i for each function F_i. For the unknown order experiment, we break the ties of functions arbitrarily and decide the execution order by the prioritization scheme in Section V. It is worthwhile to mention that, the functions of "H.264" video encoder have dependences which are not possible to remove similar to the case we already mentioned in Section V. Therefore, we keep their execution order, i.e., "SAD" $\rightarrow$ "SATD" $\rightarrow$ "DCT", and combine them as a wrapped function for the prioritization algorithm.

Fig. 8 shows the performance and reliability properties of different compiled versions in terms of average execution time and function vulnerability index (FVI), respectively. A user-provided maximum tolerable performance overhead Ω_i equal to 100% is provided to our reliability-driven compilation flow as illustrated in Algorithm 1. Different compiled versions of each function are profiled on the reliability-aware processor simulator using various different input data sets and distributions of errors, vulnerabilities, and execution time are obtained. The CDF of the execution time is generated and partitioned into 10 different steps. Note that a function can have different

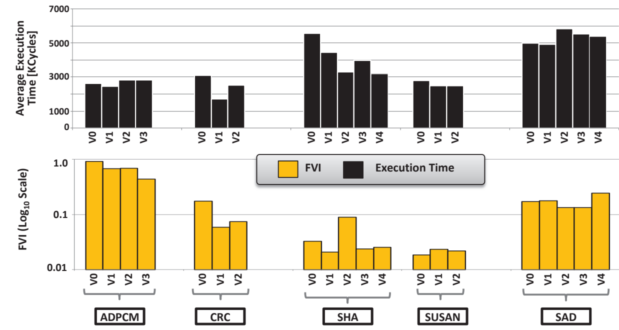


Fig. 8: Average execution time and FVI of different compiled versions of different functions.

reliability and execution time properties depending on its input which is one reason why a reliability-driven runtime system is required, i.e., different versions are selected at runtime. For simplifying the version selection problem the average vulnerability of a function version for different inputs is considered here as reliability variations for different inputs are also rather small in range.

For evaluating the system software, in addition to Algorithm 5 for dynamic version selections, we also evaluated three static version selections:

- *Min. R*: is to choose the version j with the minimum $R_{i,j}$ for each function F_i to improve the reliability;
- *Min. Avg*: is to choose the version j with the minimum average execution time for each function F_i to improve the performance;
- *Min. RTP*: is to choose the version j with the minimum RT penalty of αR + (1-α)$(1 - C_{i,j}(D))$ for each function F_i to jointly consider the reliability and performance, since it directly maps to the optimization objective.

B. *Simulation Results for a Defined Sequence*

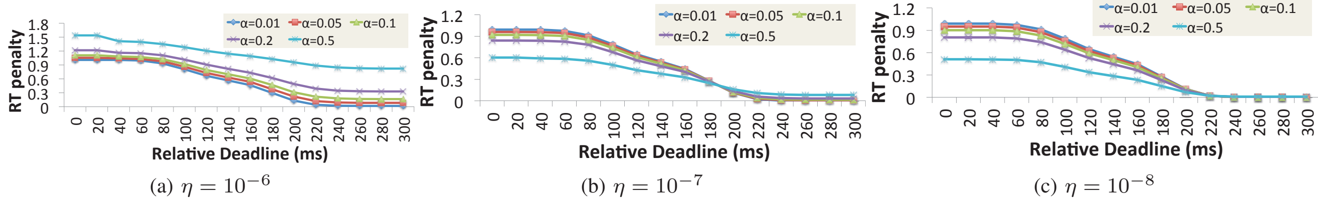
This subsection presents the results when the execution ordering is given. In such a case, the dynamic programming approach derives the minimum (optimal) RT penalty under the given execution ordering.

Fig. 9 presents the results for the "secure video and audio processing" application (as discussed above) simulated under a certain given ordering and three different settings of the fault rates, i.e., $\eta = 10^{-6}$ in Fig. 9(a), $\eta = 10^{-7}$ in Fig. 9(b), and $\eta = 10^{-8}$ in Fig. 9(c). According to the obtained FVIs of the functions and the fault rates, we derived the corresponding reliability penalties. Among these three configurations, we have $\sum_i \max_j R_{i,j} \approx 2.47$, $\sum_i \max_j R_{i,j} \approx 0.247$, and $\sum_i \max_j R_{i,j} \approx 0.0247$ when $\eta = 10^{-6}$, $\eta = 10^{-7}$, and $\eta = 10^{-8}$, respectively. Similarly, we have $\sum_i \min_j R_{i,j} \approx 1.72$, $\sum_i \min_j R_{i,j} \approx 0.172$, and $\sum_i \min_j R_{i,j} \approx 0.0172$ when $\eta = 10^{-6}$, $\eta = 10^{-7}$, and $\eta = 10^{-8}$, respectively. Note, the average total execution time of application is $178ms$, which is calculated by all functions executing on the highest performance versions. For the memory capacity, the dynamic programming with the simplification requires 921KBytes to build the table when the relative deadline is $D = 300$ms.

In our simulation results in Fig. 9, it is clear that when the relative deadline increases, the RT penalty decreases, since the timing constraint is less stringent. Therefore, when the relative deadline is too small, i.e., less than 40ms, the miss rate is

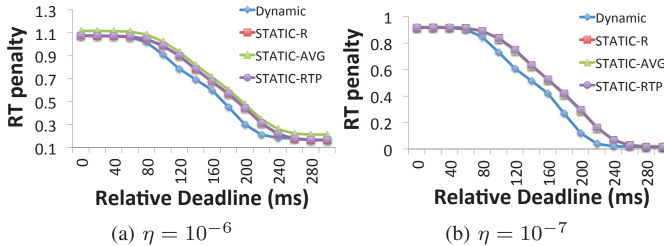
0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers



(a) $\eta = 10^{-6}$ (b) $\eta = 10^{-7}$ (c) $\eta = 10^{-8}$

Fig. 9: The RT penalty under different fault rates.



(a) $\eta = 10^{-6}$ (b) $\eta = 10^{-7}$

Fig. 10: The RT penalty under different fault rates by comparing to the static version selections when $\alpha = 0.1$.

almost 100% for all the version selections, and, hence, the most reliable versions will be selected (or the default versions will be enabled). On the other hand, when the relative deadline is large, i.e., more than 240ms, the miss rate becomes nearly 0%. A proper version selection will have almost no deadline misses, and will try to minimize the reliability penalties as well. In all the figures, the RT values change more in the range when the relative deadline is between 80ms and 240ms, and Algorithm 5 tries to balance the miss rate and the reliability penalties.

In Fig. 9, for one given α under a given deadline D, when the fault rate is high, we can also see that the RT penalty is also higher, due to the fact that $R_{i,j}$ is larger. When $\alpha = 0.01$, the reliability penalties $R_{i,j}$ do not play a very significant role in our settings, while the deadline miss rate matters. As a result, we can find very similar curves for $\alpha = 0.01$ in Fig. 9(a), Fig. 9(b), and Fig. 9(c). In other words, these curves for $\alpha = 0.01$ are basically very similar to the miss rate.

Interestingly, when the fault rate is 10^{-6}, the setting with $\alpha = 0.5$ is the one with the maximum RT penalty in the simulated cases, but it becomes the one with the minimum when the fault rate is 10^{-8}. The main reason comes from the settings of $R_{i,j}$. When $\eta = 10^{-6}$, the achievable reliability penalty is between 1.72 and 2.47. When $\eta = 10^{-8}$, the achievable reliability penalty is between 0.0172 and 0.0247. Therefore, compared to the deadline miss rates, the reliability penalties play a very significant role when $\eta = 10^{-6}$, play a comparable role when $\eta = 10^{-7}$, and play a very minor role when $\eta = 10^{-8}$. In Fig. 9(a), even though our scheme tries to minimize RT penalty, the values of $\alpha R_{i,j}$ are still too significant so that the RT penalty remains very high, especially when α is larger. In Fig. 9(b), we try to make a balance between the reliability penalties and the miss rate. However, since $\sum_i \min_j R_{i,j} \approx 0.172$ when $\eta = 10^{-7}$, the minimum RT penalty in Fig. 9(b) for $\alpha = 0.5$, is still about 0.1.

Fig. 10 presents our simulation results when the fault rate is 10^{-6} and 10^{-7} with $\alpha = 0.1$ by considering the static version selections (Min. R, Min. Avg, and Min. RTP) and the dynamic version selections based on Algorithm 5. As shown in Fig. 10, all the static version selections are worse than the dynamic version selection in Fig. 10.

As shown in Fig. 10(a), when the fault rate is higher (i.e., $\eta = 10^{-6}$), the static version selection with the minimum average execution time is the worst, since the reliability penalties play an important role for reducing the RT penalty. This also explains why the static version selection with the minimum $R_{i,j}$ for each function F_i is better. When the fault rate is lower (i.e., $\eta = 10^{-7}$), in Fig. 10(b), the differences between the above static version selections are very limited. Although the static version selection with the minimum RT penalty reconciles the advantage of the above static selections, the dynamic version selection can utilize the spare time (slack), which is changed from time to time, more cleverly than the others within the range of 80ms and 240ms. The advantage of the dynamic algorithm can achieve 33% improvement on average.

C. Simulation Results for Prioritization Ordering

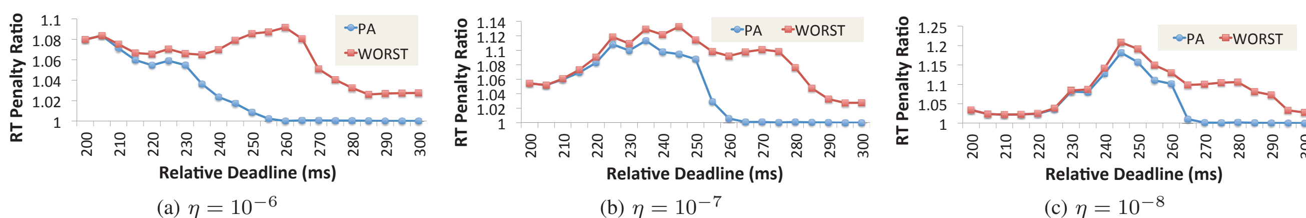
This subsection presents the results when the execution ordering can be determined by the system software. In such a case, we adopt our function prioritization algorithm in Section V and the dynamic programming approach in Section VI.

We illustrate the evaluation results by presenting the normalized *RT penalty ratio*, which is defined as the RT penalty of the resulting solution divided by *the optimal RT penalty*. For comparison, we also demonstrate a normalized result of the *worst* ordering. Please note, the optimal and worst orderings both are obtained by an exhaustive search with the factorial timing complexity. With this normalization, we can evaluate the effectiveness of our technique and show the potential of improvement between two extreme orderings, i.e., the best ordering and the worst ordering. As we are not aware of any other function prioritization algorithms, these two extreme cases also show the potential improvement space. Fig. 11 shows the RT penalty ratios of our proposed prioritization algorithm (with legends “PA”) and the worst ordering (with legends “Worst”) when considering the 5 functions as defined in the experimental setup in Section VII-A. Note that, since the application suffers from violating the performance constraint most of time from 0ms to 178ms, we only present our results by setting D in the range of 200ms and 300ms with $\alpha = 0.1$. In such cases, i.e., 0ms to 178ms, employing the prioritization would not obtain too much difference in terms of dependability, since the degradation of timing reliability as mentioned above will dominate the RT penalty.

Fig. 11 shows that our prioritization algorithm as a heuristic can reach the optimal RT penalty with the best execution order of function when the residual time is sufficient, i.e., $D \geq 260$ms. In such cases, we observe that most of the

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers



(a) $\eta = 10^{-6}$ (b) $\eta = 10^{-7}$ (c) $\eta = 10^{-8}$

Fig. 11: The RT penalty ratio under different fault rates

functions can upgrade the executing version to the lowest possible RT penalty state, in which our prioritization algorithm can evaluate the benefit density of all the best versions and prioritize the higher expected benefit version with higher priority. Though the proposed algorithm is not as good as the optimal result during 220ms to 260ms, it starts to deliver more efficient execution orders with lower RT penalty, where the efficiency of the proposed approach at the higher fault rate, i.e, $\eta = 10^{-6}$, is getting better than the results at the lower fault rates. When the fault rate is higher, i.e, $\eta = 10^{-6}$, the prioritization algorithm is getting better quickly than the others at lower, which starts from 220ms and the others start from 240ms and 250ms, respectively.

For the time consumption, we also report the required time to derive the RT penalty by adopting our approaches and by performing an exhaustive search of function prioritization, for the experiment reported in Fig. 11. For the case with fault rate 10^{-6} and $\alpha = 0.1$, our approach requires 2.35 sec on average whereas the exhaustive search takes 12 sec. Note, since the worst result for RT penalty is obtained along with the exhaustive search for the optimal ordering, we only do the exhaustive search once.

Naturally, the exhaustive search does not scale with the number of functions. Fig. 12 presents the results under different fault rates for a more complicated application scenario. This application is constructed using different functions selected from MiBench as mentioned previously, and composes various function versions with partial and full duplications to provide selective or full protection against functional errors. The motivation of this simulation result is to present that our approach still works while the application is more complicated. As mentioned before, this application is too complicated to provide the best ordering by an exhaustive search due to the high complexity. Our approach takes 3.3 sec on average for such a case.

VIII. Conclusions

This paper explored the issues related to the functional and timing reliability at multiple system layers (i.e., compiler, offline system software, and run-time system software) to improve the overall system reliability. We presented a scheme for reliable code generation and execution using reliability-driven compilation and system software.

We (1) develop a heuristic function prioritization algorithm to improve the optimization of the RT penalty efficiently, (2) enable the generations of multiple function versions with different performance versus reliability trade-off, (3) utilize these multiple versions in the system software and builds a schedule table offline to optimize the RT penalty, and (4) adapt the dynamic execution behavior to select suitable versions for the subsequent functions properly. We evaluated our scheme with a real-world application of "secure video and audio processing" composed of various functions by using a reliability-aware processor simulator.

References

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

[2] AMD. AMD PhenomTM II Processor Product Data Sheet 2010.

[3] H. Aydin, R. G. Melhem, and D. Mossé. Tolerating faults while maximizing reward. In *ECRTS*, pages 219–226, 2000.

[4] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. C. de Araujo, and E. Barros. The ArchC Architecture Description Language and Tools. *International Journal of Parallel Programming*, 33(5):453–484, 2005.

[5] R. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Trans. on Device and Materials Reliability*, 5(3):305 – 316, 2005.

[6] Y. Cao, J. Velamala, K. Sutaria, M. S.-W. Chen, J. Ahlbin, I. S. Esqueda, M. Bajura, and M. Fritze. Cross-layer modeling and simulation of circuit reliability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 33(1):8–23, 2014.

[7] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *International Symposium on Fault-Tolerant Computing*, 1995.

[8] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys*, pages 15–28, 2006.

[9] C.-C. Han, K. G. Shin, and J. Wu. A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults. *IEEE Trans. Comput.*, 52(3):362–372, 2003.

[10] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Design Automation Conference (DAC)*, 2013.

[11] J. Henkel, L. Bauer, H. Zhang, S. Rehman, and M. Shafique. Multi-layer dependability: From microarchitecture to application level. In *Design Automation Conference (DAC)*, pages 1–6, 2014.

[12] J. Hu, S. Wang, and S. Ziavras. In-register duplication: Exploiting narrow-width value for improving register file reliability. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 281 –290, june 2006.

[13] J. S. Hu, S. Wang, and S. G. Ziavras. In-Register Duplication: Exploiting Narrow-Width Value for Improving Register File Reliability. In *2006 International Conference on Dependable Systems and Networks (DSN 2006), 25-28 June 2006, Philadelphia, Pennsylvania, USA, Proceedings*, pages 281–290. IEEE Computer Society, 2006.

[14] IBM. IBM XIV Storage System cache. http://publib.boulder.ibm.com/infocenter/ibmxiv/r2/index.jsp.

[15] V. Izosimov, P. Eles, and Z. Peng. Value-based scheduling of distributed fault-tolerant real-time systems with soft and hard timing constraints. In *ESTImedia*, pages 31–40, 2010.

[16] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Synthesis of fault-tolerant embedded systems with checkpointing and replication. In *DELTA*, pages 440–447, 2006.

[17] L. Li, V. Degalahal, N. Vijaykrishnan, M. T. Kandemir, and M. J. Irwin. Soft error and energy consumption interactions: a data cache perspective. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design, 2004, Newport Beach, California, USA, August 9-11, 2004*, pages 132–137, 2004.

[18] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[19] A. Masrur, P. Kindt, M. Becker, S. Chakraborty, V. Kleeberger, M. Barke, and U. Schlichtmann. Schedulability analysis for processors with aging-aware autonomic frequency scaling. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2012.

[20] S. Mitra, K. Brelsford, and P. N. Sanda. Cross-layer resilience challenges: Metrics and optimization. In *DATE*, pages 1029–1034, 2010.

[21] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Cambridge, MA, USA, 1983.

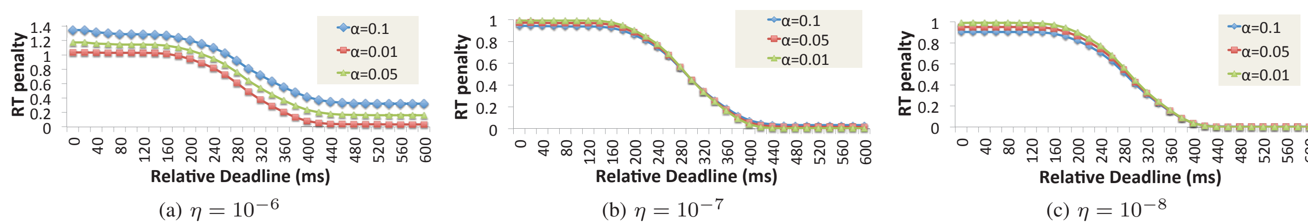
[22] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 29–, 2003.

[23] N. Oh, P. Shirvani, and E. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. on Reliability*, 51(1):63 –75, 2002.

[24] S. Rehman, F. Kriebel, M. Shafique, and J. Henkel. Reliability-driven software

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2417554, IEEE Transactions on Computers



(a) $\eta = 10^{-6}$ (b) $\eta = 10^{-7}$ (c) $\eta = 10^{-8}$

Fig. 12: The RT penalty under different fault rates for the complicated application

transformations for unreliable hardware. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 33(11):1597–1610, Nov 2014.

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

[26] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel. Reliable software for unreliable hardware: embedded code generation aiming at reliability. In *CODES+ISSS*, pages pp. 237–246, 2011.

[27] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. *ACM Trans. Archit. Code Optim.*, 2(4):366–396, Dec. 2005.

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

[29] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel. Exploiting program-level masking and error propagation for constrained reliability optimization. In *50th Design Automation Conference (DAC)*, 2013.

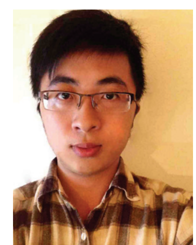
[30] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389 – 398, 2002.

[31] R. Vadlamani, J. Zhao, W. P. Burleson, and R. Tessier. Multicore soft error rate stabilization using adaptive dual modular redundancy. In *DATE*, pages 27–32, 2010.

[32] R. Venkatasubramanian, J. Hayes, and B. Murray. Low-cost on-line fault detection using control flow assertions. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, pages 137 – 143, july 2003.

[33] J. Yan and W. Zhang. Compiler-guided register reliability improvement against soft errors. In *Proceedings of the 5th ACM international conference on Embedded software (EMSOFT)*, pages 203–209, 2005.

Semeen Rehman (S'11) received the B.Sc. degree in computer science from the University of Peshawar, Pakistan, in 2004. She has been pursuing the Ph.D. degree from the Chair for Embedded Systems, Karlsruhe Institute of Technology, Germany, since 2008. From 2005 to 2007, she was an Information Systems Manager at Mardan Surgical Centre Pvt. (Ltd.), Pakistan. Her current research interests include dependable computing, cross-layer reliability, and reliability-driven compilation for embedded processors. Ms. Rehman was the recipient of the CODES+ISSS 2011 Best Paper Award and several HiPEAC Paper Awards.



Kuan-Hsun Chen received the M.Sc. degree in computer science from the 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. His current research interests include dependable computing, embedded systems and reliability-aware resource management.

Florian Kriebel received the M.Sc. degree in computer science from the Karlsruhe Institute of Technology (KIT), Germany, in 2013. He is currently pursuing the Ph.D. degree from the Chair for Embedded Systems, KIT, Germany. His current research interests include dependable computing, cross-layer reliability modeling, and optimization. Mr. Kriebel was the recipient of the CODES+ISSS 2011 Best Paper Award and two HiPEAC Paper Awards.

Anas Toma has been pursuing his Ph.D. at the Department of Informatics, Karlsruhe Institute of Technology, Germany, since October 2011. His main research interests are real-time systems, high-performance and energy-efficient scheduling, algorithm design and analysis, and computer vision. He received his B.Sc. in Computer Engineering from An-Najah National University, Palestine in 2005. In 2008, he received his M.Sc. in Computer Engineering from Jordan University of Science and Technology. From 2008 to 2011, Anas worked as lecturer at An-Najah National University. Anas was awarded two scholarships by DAAD for his master and doctoral studies.

Muhammad Shafique (M'11) received the Ph.D. degree in computer science from the Karlsruhe Institute of Technology (KIT), Germany, in 2011. He is currently a Research Group Leader at the Chair for Embedded Systems, KIT. He has over ten years of research and development experience in power-/performance-efficient embedded systems in leading industrial and research organizations. He holds one U.S. patent. His current research interests include design and architectures for embedded systems with focus on low power and reliability. Dr. Shafique was the recipient of six gold medals, CODES+ISSS'14 Best Paper Award, CODES+ISSS'11 Best Paper Award, AHS'11 Best Paper Award, DATE'08 Best Paper Award, DAC'14 Designer Track Poster Award, ICCAD'10 Best Paper Nomination, several HiPEAC Paper Awards, and the Best Masters Thesis Award. He has served as a TPC Member at ICCAD, CASES, DATE, and ASPDAC.

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.

Jörg Henkel (M'95-SM'01) is currently with the Karlsruhe Institute of Technology (KIT), Germany, where he is directing the Chair for Embedded Systems (CES). Dr. Henkel received the masters and the Ph.D. (Summa cum laude) degrees, both from the Technical University of Braunschweig, Germany. He then joined the NEC Laboratories, Princeton, NJ, USA. He holds ten U.S. patents. His current research interests include design and architectures for embedded systems with focus on low power and reliability. Prof. Henkel was the recipient of the 2008 DATE Best Paper Award, the 2009 IEEE/ACM William J. McCalla ICCAD Best Paper Award, the CODES+ISSS 2011 and 2014 Best Paper Awards. He was the Chairman of the IEEE Computer Society, Germany Section, and the Editor-in-Chief of the ACM Transactions on Embedded Computing Systems. He is also an Initiator and the Spokesperson of the national priority program called Dependable Embedded Systems of the German Science Foundation, and the General Chair of ICCAD 2013.

0018-9340 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.