Leveraging Variable Function Resilience for Selective Software Reliability on Unreliable Hardware

Semeen Rehman, Muhammad Shafique, Pau Vilimelis Aceituno, Florian Kriebel, Jian-Jia Chen, Jörg Henkel
Karlsruhe Institute of Technology (KIT), Germany
{semeen.rehman, florian.kriebel}@student.kit.edu; {muhammad.shafique, j.chen, henkel}@kit.edu

***Abstract*—State-of-the-art reliability optimizing schemes deploy spatial or temporal redundancy for the complete functionality. This introduces significant performance/area overhead which is often prohibitive within the stringent design constraints of embedded systems. This paper presents a novel scheme for selective software reliability optimization constraint under user-provided tolerable performance overhead constraint. To enable this scheme, statistical models for quantifying software resilience and error masking properties at function and instruction level are proposed. These models leverage a whole new range of reliability optimization. Given a tolerable performance overhead, our scheme selectively protects the reliability-wise most important instructions based on their masking probability, vulnerability, and redundancy overhead. Compared to state-of-the-art [7], our scheme provides a 4.84X improved reliability at 50% tolerable performance overhead constraint.**

I. Introduction and Related Work

Reliability has become an imperative design criterion for advanced computing systems. Aggressive shrinking of transistor dimensions, a reduced gap between threshold and nominal voltages, etc. result in unreliable hardware susceptible to different sources of faults like soft errors, aging, etc. [1]-[4]. Transient faults (like radiation-induced soft errors) manifest as spurious bit flips in the hardware that may propagate to the application software layer and jeopardize the correct application execution. To mitigate these issues, diverse reliability-enhancing schemes have been developed at hardware and software levels.

Hardware-based reliability schemes (like TMR, pipeline protection with shadow latches [6], design with reduced architectural vulnerability factor [4][5], adding ultra-reduced instruction set co-processors to execute faulty instructions [25], etc.) typically incur significant overhead in terms of area, power, and validation cost. To alleviate this overhead and in order to complement existing hardware schemes, several software-based reliability schemes have been proposed, such as redundant code (instruction & register duplication) and control flow checking using EDDI [8], [9][10], SWIFT/CRAFT [7], etc. These software-based schemes, however, duplicate *all* the instructions and incur a significant performance and/or memory overhead (> 2X) [7]-[10].

Significant overhead in typical design metrics like performance, area, or power is often prohibitive within the stringent design constraints of embedded systems. On the one hand, insufficient reliability may result in an ineffective product due to its functional degradation. On the other hand, an excessive protection may lead to a resulting product that is uncompetitive. Therefore, reliable embedded system design needs to optimize reliability under *tolerable* overheads while accounting for the error masking behavior of target applications and avoiding *'design for over-protection'*. Previous work on constrained reliability optimization leverage instruction scheduling and transformations in a reliability-driven compiler [18][26][27], but does not account for error masking effects and selective protection.

Here is an interesting observation: often, different application programs and even different functions in an application are *not* equally susceptible to transient faults [18][19] due to different data & control flow properties, internal error masking effects, etc. (see Fig. 1). Therefore, these functions exhibit distinct resilience to hardware-level faults. The observed difference in function-level resilience may be leveraged to limit the growing overheads of reliability optimizations under constraints like performance or power.

Motivational Example: Fig. 1(a) shows the error distribution in two applications "AES" and "SusanC" at a fault rate of 5 faults/MCycles (see Section VI for the experimental setup). When comparing "AES" to "SusanC", the percentage of "Incorrect Output" errors is lower in "SusanC", which is mainly due to the relatively high *instruction-level masking*, i.e. an error at an instruction will be masked until the visible output due to control flow properties and/or logical masking in the hardware. From the above observation, it is implicit that even for the same hardware and same fault scenario different applications exhibit distinct resilience properties due to their varying control flow, type/number and sequence of instructions. Moreover, the vulnerabilities of different instructions in a given function vary due to spatial effects (i.e. using different pipeline components of different hardware area) and temporal effects (i.e. different amount of time spent in the pipeline components). Fig. 1(b) illustrates the distribution of instruction vulnerabilities (estimated using the model of [18]) in "SusanC" for a selected range of program counter (PC, x-axis). It is noteworthy that a few instructions (like 'multiply', 'load', 'branches') have significantly high vulnerability compared to other instructions (like 'add' and 'sub') due to relatively large spatial and temporal effects.

The above-discussion and observation in Fig. 1 illustrate that different applications/functions differ in terms of their resilience and different instructions differ in terms of vulnerabilities. Therefore, not all functions and instructions require same level of protection. Pessimistically applying redundancy to all instructions of all functions may incur significant overhead that can be *curtailed by leveraging the variable resilience, masking, and vulnerability properties of application software at different granularities* (i.e. functions, basic blocks, and instructions).

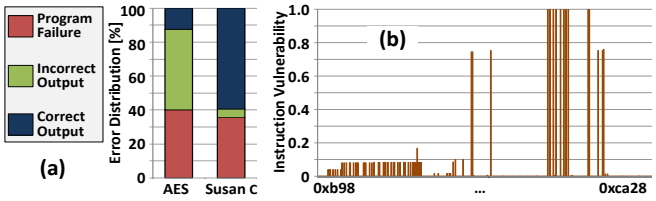

Fig. 1 (a) Error Distribution for two Applications; (b) Instruction Vulnerability distribution for different instructions in "SusanC"

State-of-the-art schemes in error-resilience exploit inherent resilience of image/video processing applications to tolerate errors while accepting incorrect output values with degraded output quality [11]-[15]. These schemes primarily aim at reducing the power consumption using aggressive voltage scaling, while tolerating errors introduced by voltage scaling. These state-of-the-art schemes *do not quantify and model application resilience in terms of functional correctness*. Therefore, these schemes cannot efficiently exploit resilience to guide reliability-optimizing schemes at the software and/or hardware layers. Moreover, state-of-the-art like [28][29] has not yet exploited application resilience and error masking properties *quantitatively* for selective reliability-optimization under tolerable performance overhead constraints. Note, AVF based techniques like [4][5] cannot be applied at the software/compiler layer as they do not provide a quantification of vulnerabilities and masking probabilities at the instruction level.

Problem: there is a need to *selectively optimize the software reliability* under tolerable performance overhead constraints. This requires *modeling and quantitative estimation* of (1) resilience of application software at the function level; (2) error masking effects at the instruc-

978-3-9815370-0-0/DATE13/©2013 EDAA

tion level. Selective reliability-optimizing schemes need to be guided with these models to curtail their protection overhead and to realize *constrained reliability-optimization*.

A. Our Novel Contributions and Idea: Overview

1) **A Resilience-Driven Selective Software Reliability Scheme (Section V)** that selects a set of reliability-wise most important instructions in different functions for protection depending upon function resilience, instruction-level error masking, and instruction vulnerability under user-provided tolerable performance overhead constraint. The key is to give more protection to the less-resilient part of the application, while less protection to more-resilient part.
2) **Resilience Modeling and Estimation (Section III):** Modeling and estimating the resilience of application at function-level w.r.t. the functional correctness based on statistics and information theory concepts;
3) **Software Masking Analysis and Modeling (Section IV):** A probabilistic analysis, modeling, and estimation of instruction masking effects inside a function, i.e. probability that an error during the execution of this instruction will be masked until the visible function output.

Fig. 2 shows an overview of our novel scheme integrated in a reliability-driven compiler. To the best of authors' knowledge, this is the first work that targets selective software reliability under constraints leveraged by modeling and quantitative estimation of resilience and masking properties of application software.

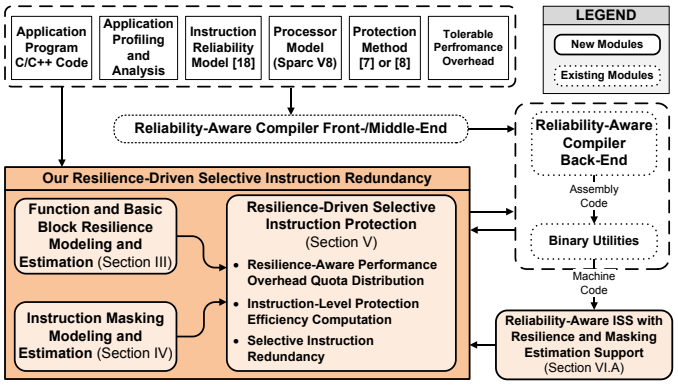

Fig. 2 Overview our Selective Software Reliability Scheme integrated in a Reliability-Driven Compiler

Before proceeding to the details of our novel contribution, we present formal model & system models for clarity/consistency of discussion.

II. System Models and Preliminaries

Application Model: An application software ***A = (F, E)*** is composed of different *functions* $\boldsymbol{F}=\{f_1, f_2, \ldots, f_l\}$.

Function: Each function may compose of multiple *basic blocks*; $\boldsymbol{f_i}=\{B_i, BDi, L_i, eF_i, R_i\}$ such that $B_i=\{B_1, B_2, \ldots, B_m\}$. BD_i is the se of basic block dependencies, L_i is the average execution time, eF_i is the expected execution frequency, and R_i is the resilience of the i^{th} function.

Basic Block: A basic block is a sequence of instructions without any jump/branch; $B_{ij}=\{I_{ij}, E_{ij}, L_{ij}, eF_{ij}, R_{ij}\}$, such that $I_{ij}=\{I_1, I_2, \ldots, I_n\}$, L_{ij} is the average execution time, eF_{ij} is the expected execution frequency, and R_{ij} is the resilience of the j^{th} basic block of i^{th} function. E_{ij} is the set edges that denote the instruction dependencies; $E_{ij}=\{e_{Ixi} \rightarrow e_{Iyi} \mid \forall I_{xi}, I_{yi} \in I_{ij}\}$ and is given as the edge weight that represents the latency from instruction I_{xi} to I_{yi}.

Instruction: Each instruction is represented as a tuple $I_{ijk}=\{P_{TM}, P_{im}, \upsilon, P, S, L, d, o\}_{ijk}$. P_{TM} is the total masking probability from the k^{th} instruction to the visible output. P_{im} is the internal masking during the execution of an instruction (i.e. masking through the pipeline components). υ denotes the vulnerability of an instruction in the pipeline. $P_{ijk}=\{p_1, \ldots, p_a\}_{ijk}$ and $S=\{s_1, \ldots, s_b\}_{ijk}$ are the sets of predecessor and successor instructions, respectively. L is the execution time of the instruction, while d and o are the sets of destination and source operands.

Processor Model: single-core RISC architecture, in-order, multiple *Pipeline* stages (like SPARC V8 with 5 stages).

Fault Model: Transient faults – single or multiple bit upsets.

Error Categories: We classify application outputs in 3 categories:
1) *Correct Output*: Output data values are correct and useful
2) *Incorrect Output*: Output data values are incorrect; may/may not be useful; and
3) *Application Failure*: like crash, halt, abort, etc.

Instruction Vulnerability Model: In order to estimate the vulnerability υ of an instruction I_{ijk}, we employ an existing model Instruction Vulnerability Index [18] (see Eq. 1),which quantifies the effects of hardware-level faults at the software level. The Instruction Vulnerability Index υ is defined as the weighted sum of instruction vulnerabilities in different *Pipeline* components p with area A_p and micro-architectural error probability $P_E(p)$.

$$\upsilon = \left(\sum_{\forall p \in Pipeline} \upsilon(p) \times A_p \times P_E(p)\right) \Big/ \sum_{\forall p \in Pipeline} A_p \quad (1)$$

III. Modeling the Resilience of Application Functions

Definition: The *resilience* of an application function is defined as the probabilistic measure of functional correctness (output quality) in presence of faults.

Modeling: Modeling resilience requires error probabilities for basic blocks outputs. There are two possible error types: "incorrect output" and "application failure". Therefore, output of each instruction in a given basic block can be modeled as a Markov Chain with three states: S_C, S_{IC}, and S_F denoting "correct", "incorrect", and "application failure" states, respectively (see Fig. 3). Considering that the execution of a program is a stochastic process, we adopt Markov Chain technique for output modeling as it provides a good tradeoff between the model complexity and accuracy when compared to exhaustive Monte-Carlo Simulations, Fault-Tree Analysis, and Principal Component Analysis based reliability models [21].

Assuming that each state depends upon the previous instructions' output and the error state can only be observed at the end or at the time of "application failure", the execution path can be modeled as a Hidden Markov Chain, with the above-discussed three states as hidden states and the observation state as "application failed" or "not-failed". The parameters of this model are the state transition probabilities (given in the matrix T, see Fig. 3) that depend upon the executed instructions. Note, the Markov Chain is non-homogeneous as the transition probabilities change depending upon an instruction I_{ijk}.

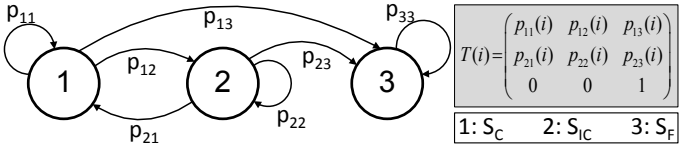

Fig. 3 Markov Chain for Instruction Output with State Transition Probabilities

Once these probabilities are estimated (see parameter estimation later in this section), we can compute the final state probability for a given basic block B_{ij} using Eq. 2, where ξ is the final state probability vector containing the probability of three states: p_C, p_{IC}, and p_F.

$$\xi(B_{ij}) = \begin{bmatrix} p_C & p_{IC} & p_F \end{bmatrix}_{B_{ij}} = \xi(B_{ij-1}) \times \prod_{x \in I_{ij}} T(x) \quad (2)$$

Following the information theory concepts, we model the *resilience of a function as the normalized mutual information between the required correct result (from a golden execution run X) and the result at the end of a function execution (from a potentially faulty execution under a given fault rate)*, i.e. amount of useful function output (see Eq. 3). A large value of mutual information illustrates that more information about the correct output can be inferred, i.e. high resilience. The

resilience of a basic block B_{ij} is given as $R(B_{ij})=1–H(X|Y)/H(X)$, where H(X) is the information about the correct execution, i.e. $H(X)=b_{Live}$, such that b_{Live} denotes the bits of live output registers of B_{ij}. The conditional entropy $H(X|Y)$ is the information lost in B_{ij} and given as Eq. 3, where $p_C(x)$ denotes the probability of correct value being x; and $p_{[IC,F]}(x, y)$ is the conditional probability of faulty output being "incorrect" or "application failure".

$$R(B_{ij}) = 1 - H(X|Y)/H(X); \quad H(X) = b_{Live}$$
$$H(X|Y) = \sum_{x\in X, y\in Y}\left(p_{[IC,F]}(x,y)\times log_2\left(p_C(x)/p_{[IC,F]}(x,y)\right)\right) \quad (3)$$

Assuming, resilience of a basic block $R(B_{ij})$ can be characterized as resilience to "incorrect output" and resilience to "application failures", we can compute the conditional entropy separately for both cases. $H(X|Y)_F$ is given as $p_F(B_{ij})$ using Eq. 2, while $H(X|Y)_{IC}$ is given by Eq. 4.

$$H(X|Y)_{IC} = -\left[p_{IC}\times log_2(p_{IC}/(2^n-1)) + (1-p_{IC})\times log_2(1-p_{IC})\right]_{B_{ij}} \quad (4)$$

By replacing the terms of Eq. 3 with Eq. 4, we can compute the resilience of a basic block against failures and incorrect outputs where the second term in Eq. 5 denotes the combined information loss.

$$R(B_{ij}) = 1-\left[H(X|Y)_{IC} + H(X|Y)_F - \left(H(X|Y)_{IC}\times H(X|Y)_F\right)\right]/H(X) \quad (5)$$

Given the resilience values of all basic blocks $\boldsymbol{B_i}$ of a function $\boldsymbol{f_i}$, resilience $R(f_i)$ can be computed using Eq. 6.

$$R(f_i) = \sum_{\forall b\in Bi}\left(R(b)/eF_i(b)\right)\times\sum_{\forall b\in Bi,\forall fi\in F}\left(eF_i(b)\right) \quad (6)$$

Parameter Estimation: For estimating the model parameters, i.e. transition probabilities given in Eq. 2, we make few assumptions:

- observation of faulty output is made at the end of function
- no recovery mechanism and no error protection is available; i.e. starting from a base case of unreliable hardware => $p_{33}=1$; $p_{21}=0$.
- Initial state and input is error-free; $[p_C\ p_{IC}\ p_F]_{(t=0)fI}=[1\ 0\ 0]$.

Note, $p_{11}+p_{12}+p_{13}=1$. To expedite the parameter estimation process, we have grouped instructions in N_T primitive instruction categories (like arithmetic, multiply, divide, logical, load/store, calls/jumps, floating point, etc.) such that all instructions in a given category share the same transition probabilities. The parameter can be estimated using fault injection campaigns. Consider there are N_S different fault-injection experiments at a given fault rate, N_C and N_{IC} are the number of cases with correct and incorrect output, respectively. For a particular fault injection experiment s, for a certain instruction category t_k, the transition probability p_{11} can be estimated using the maximum likelihood, thus deriving[1] Eq. 7. $NI(t,s)$ denotes the number of instruction type t in that simulation.

$$log\left(p_{11}(t_k)\right) = -NI(t_k,s)\times\left(\frac{\sum_{\forall s\in S} log\left(N_S/N_C(s)\right) + \sum_{t=0,t\neq t_k}^{N_T} NI(t,s)\times log\left(p_{11}(t)\right)}{\left(\sum_{\forall s\in S} NI(t_k,s)\right)^2}\right) \quad (7)$$

Assuming $p_{23}(t) = p_{13}(t)$, we can re-use Eq. 7 to obtain the probability $p_{22}(t_k)$. In this way we can compute all the remaining transition probabilities, such that, $p_{23}(t_k)=p_{13}(t_k)=1–p_{22}(t_k)$; and $p_{12}(t_k)=p_{22}(t_k)–p_{11}(t_k)$.

Fig. 4 shows a simplified flow of different steps of our scheme towards modeling and estimation of function resilience along with parameter estimation and computation of conditional entropy.

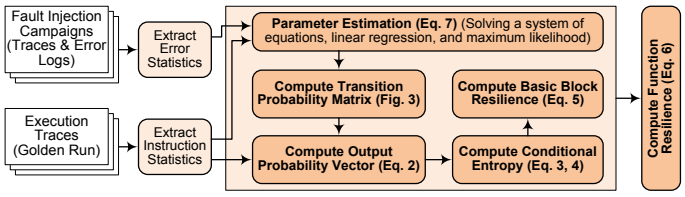

Fig. 4 Flow of Steps to Compute Basic Block & Function Resilience

[1] Complete derivation is omitted due to space limitations.

Complexity: The complexity of resilience estimation is $O(|B_i|\times N_T \times \log(|I_{ij}|)$, which is much smaller than the complexity of fault tree based methods (i.e. $O(|B_i|\times|I_{ij}|^3)$ and Monte-Carlo simulations (i.e. $O(|B_i|\times|I_{ij}|^2)$) for each basic block.

The resilience model quantifies the reliability properties at a coarse-grained level, i.e. function and basic block that facilitates prioritizing functions and basic blocks for selective protection. However, to enable the selective protection within a basic block and to prioritize different instructions, there is a need to model and estimate the instruction masking probabilities.

IV. Modeling Software Masking Effects

Definition: *Software masking probability $p_{TM}(I_{ijk})$* at a certain instruction I_{ijk} is defined as the probability that an error at I_{ijk} does not become visible at the application output and therefore is denoted as 'masked'.

Software-level masking impacts the application software reliability by blocking the error propagation such that: (1) the output value remains correct; or (2) a degraded value does not propagate to the subsequent execution iterations. In this paper, we assume that the program control flow is protected using, e.g., basic block signatures [30].

Modeling: $p_{TM}(I_{ijk})$ depends upon two key parameters:

1) $p_M(I_{ijk})$ which is defined as the total masking during the execution of an instruction I_{ijk} due to the microarchitecture-level masking effects, i.e. a transient fault during the instruction execution is blocked due to a subsequent gate in the pipeline components, thus the error is not latched by a memory element, thus does not affect the correct output of the instruction I_{ijk};
2) $p_{postM}(I_{ijk})$ which is defined as the total masking probability after the execution of an instruction I_{ijk} such that, an erroneous output value is masked in the path from I_{ijk} until the visible output point due to, for instance, dependent instructions, operation masking, control flow, etc.

$$p_{TM}\left(I_{ijk}\right) = p_M\left(I_{ijk}\right) + \left(1 - p_M\left(I_{ijk}\right)\right)\times p_{postM}\left(I_{ijk}\right) \quad (8)$$

Once $p_M(I_{ijk})$ and $p_{postM}(I_{ijk})$ are estimated, $p_{TM}(I_{ijk})$ is computed using dynamic programming starting from the leaf node (i.e. last instruction on an execution path before the value is written to the main memory or returned from the application function), which has $p_{postM}(I_{ijk})=0$.

Estimating $p_M(I_{ijk})$: An instruction I_{ijk} uses different pipeline components p, each having logical masking $LM(p|e_p)$ as the conditional probability of error masking given an error occurs in a pipeline component p, i.e. e_p. Considering that the probability of error in each cycle is same, the $p_M(I_{ijk})$ can be computed using Eq. 9.

$$p_M\left(I_{ijk}\right) = \sum_{p\in Pipeline} LM\left(p|e(p)\right) \quad (9)$$

$LM(p|e_p)$ can be computed using fault injection experiments or using statistical techniques like EPP [20]. In this paper, we determine $LM(p|e_p)$ through extensive fault injection campaigns considering area of different pipeline components.

Estimating $p_{postM}(I_{ijk})$: For a given path in the instruction flow graph, $p_{postM}(I_{ijk})$ can be computed using the total bit masking probability of the actual operation $[p_V(I_{ijk})]$ and p_{postM} of the dependent/successor instructions (S_{ijk}); see Eq. 10.

$$p_{postM}\left(I_{ijk}\right) = \prod_{\forall s\in S_{ijk}}\left[p_V(s) + \left(1 - p_V(s)\right)\times p_{postM}(s)\right] \quad (10)$$

S_{ijk} denotes the set of successors. Assuming single bit faults with same fault probability in all of the operand bits, we can compute the total bit masking probability using Eq. 11.

$$p_V\left(I_{ijk}\right) = \left(1/N_{Bits(o_{ijk})}\right)\times\left(\sum_{bv\in o_{ijk}} p_m\left(b_v, I_{ijk}\right)\right) \quad (11)$$

b_v denotes the bits of operands assuming that all bits having similar probability to get faults, while o_{ijk} is the set of operands.

Example: Let us assume an instruction $c=a\&b$, where operation is "*bit-wise and (&)*", $a=0x0000FFFF$, and $b=0xFFFFFFFF$ [bit sequence

31…0]. In this case, for *b*, error masking probability w.r.t. the operation type $p_m(b_v, I_{ijk})$ is given as: $p_m(b_{0...15}, \&)=0$ and $p_m(b_{16...31}, \&)=1$.

Our scheme computes $p_{postM}(I_{ijk})$ **recursively** using p_{postM} of the successor instructions (S_{ijk}), starting with the leaf node (i.e. last instruction of the execution path) that has $p_{postM}(I_{ijk})=0$ (i.e. all instructions occurring at the leaf node will propagate to memory). A **breadth-first bottom-up search** is employed that starts from the leaf nodes and explores the instruction flow graph.

Complexity: The time complexity of this search is $O(\Sigma_{\forall b \in Bi}(|I_{ij}|+|E_{ij}|))$ and space complexity is $O(\Sigma_{\forall b \in Bi}|I_{ij}|)$.

The above-presented software resilience and masking models *enable selective reliability-optimizing schemes under constrained scenarios* that provide a tradeoff between reliability and performance. Function and basic block resilience leverages selective redundancy schemes to prioritize functions and basic blocks w.r.t. their reliability importance, while the instruction-level masking model leverages prioritizing the instructions within basic blocks and functions.

V. Leveraging Resilience and Software Masking for Selective Software Reliability

In this paper, we propose a selective instruction redundancy scheme that leverages both function and basic block resilience along with instruction masking probability and vulnerability to selectively protect reliability-wise most important instructions in a given application software under user-provided tolerable performance overhead.

Our selective instruction redundancy scheme operates in two steps:

Step-1: First, distribute the tolerable performance overhead quota among different functions of an application and their constituting basic blocks based on their resilience value (i.e. R_i and R_{ij}).

Step-2: Afterwards, select a set of reliability-wise most important instructions within a basic block for protection using selective instruction redundancy depending upon their *masking probabilities* $p_{TM}(I_{ijk})$ and vulnerability index υ_{Iijk}.

The key is to provide more protection to the less-resilient functions, while less protection to more-resilient functions. Our *selective instruction redundancy scheme* provides means to reduce the redundancy overhead, while still ensuring a high probability of correct output.

Algorithm 1 shows the pseudo-code of our resilience-driven selective instruction redundancy scheme.

Input: Original unprotected application software **A=(F,E)**; see application model in Section II.
Output: Reliability-optimized application **A'** with selective instruction protection.
Constraint: A user-provided tolerable performance overhead constraint **Γ** in terms of cycles or percentage of performance-optimized execution that can be converted to a cycle quota accordingly.

Optimization Goal: The *protection efficiency* (λ) of an instruction is given as the instruction reliability benefit if protected (using a user-specified protection scheme **P**), such that the instruction with the highest protection efficiency (λ_{Best}) is selected for protection first given its protection overhead is under the cap of tolerable performance overhead **Γ**. The *protection efficiency* (λ) of an instruction is a joint function of total masking probability (p_{TM}, see Section IV), instruction vulnerability (υ, see Eq. 1), and protection overhead "γ" that depends upon the execution latency of the candidate instruction and protection scheme. The *protection efficiency* (λ) is defined as $((1-p_{TM})*\upsilon/\gamma$. The product of the term "$1-p_{TM}$" and υ provides the *effective vulnerability* that an error occurring in an instruction will ultimately propagate to the output. The algorithm should protect the instruction with the highest *effective vulnerability*. However, it might happen that an instruction with the highest effective vulnerability incurs significant protection overhead. It might be better to protect more instructions

Algorithm 1: Resilience-Driven Selective Instruction Redundancy

INPUT:
A : Original unprotected application software *// see formal model in Sec. II*
Γ : User-provided tolerable performance overhead in cycles
P : User-provided protection method like EDDI, SWIFT, CRAFT, etc.
OUTPUT: **A'** : Application software with redundant instruction
BEGIN

```
1.  ∀f_i ∈ F{          // compute resilience of basic blocks & functions
2.    ∀B_ij ∈ f_i {
3.      R_ij ← computeBBResilience(B_ij); }          // see Eq. 5
4.    R_i ← computeNormalizedFunctionResilience(f_i);   // see Eq. 6
5.  }
6.  ψ_F ← Σ_{∀fi∈F}((1 − R_i) × L_i × eF_i);
7.  ∀f_i ∈ F{
8.    Γ_i ← (((1 − R_i) × L_i × eF_i)/ψ_F) × Γ; // function's overhead quota
9.    ψ_B ← Σ_{∀Bij∈Bi}((1 − R_ij) × L_ij × eF_ij);
10.   ∀B_ij ∈ B_i {
11.     Γ_ij ← (((1 − R_ij) × L_ij × eF_ij)/ψ_B) × Γ_i; // BB's overhead quota
12.     ∀ins ∈ I_ij { // compute p_TM and υ for all instructions
13.       ins.P_TM ← computeP_TM(ins);          // see Eq. 8
14.       ins.υ ← computeInstrVul(ins);         // see Eq. 1
15.     }
16.     while((Γ_ij > 0) || (Σ_{∀ins∈Iij} isNotProtected(ins) > 0)){
17.       λ_Best ← 0; γ_Best ← 0; I_Best ← NULL;
18.       ∀ins ∈ I_ij {
19.         γ ← getProtectionOverhead(ins, L_ins, P);
20.         λ_ins ← (((1 - ins.p_TM) × ins.υ) / γ);
21.         if((λ_ins > λ_Best)&&((Γij − γ) ≥ 0)) {
22.           λ_Best ← λ_ins; γ_Best ← γ; I_Best ← ins; }
23.       }
24.       Protect(I_Best, P); // perform instruction redundancy
25.       Γij ← Γij − γ_Best;
26.     }
27.   }
28. }
```

END

with a slightly reduced effective vulnerability, rather than protecting one instruction with a high overhead. Fig. 5 illustrates an abstract example comparing two selection schemes (1) selecting instructions based on the effective vulnerability; and (2) selecting instructions based on the protection efficiency. The value in the box shows the effective vulnerability. Fig. 5 shows that the second scheme provides a higher protection reduction in the effective vulnerability for a given tolerable performance overhead of 7 cycles. Overall, the total protection efficiency of the second scheme is 0.71 compared to the 0.4 efficiency of the first scheme, i.e. an improvement of 0.31 => 77% better reliability compared to the first scheme.

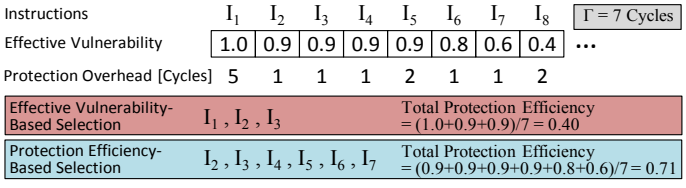

Fig. 5 Abstract Example Comparing Two Selection Schemes

Note: in this work, we employ standard protection methods (like instruction redundancy at the compilation level) to evaluate our models and scheme. However, our approach is equally beneficial for adapting/guiding other hardware-/software-based reliability enhancing mechanisms in order to reduce their overhead depending upon a given user constraint.

Algorithm Flow: The *goal* of our algorithm is to select a set of instructions for protection using user-specified protection scheme **P**)

such that, the application software reliability is maximized under given tolerable performance overhead constraint (i.e. maximizing the total *protection efficiency*), while accounting for the resilience and masking properties (see Sections III and IV).

First, the resilience values of all basic blocks and functions are computed using Eq. 5 and 6 (lines 1-5). Afterwards, the tolerable performance overhead quota for each function is computed using its resilience value (R_i, see Section III) and the user-provided tolerable performance overhead (lines 6-8). The idea is to allocate more overhead quota to less-resilient function, while providing less overhead quota to more-resilient function. The function's performance overhead quota is distributed among different basic blocks depending upon their resilience values (R_{ij}, see Section III) and execution frequencies (eF_{ij}, see Section II), such that more frequently executing and less-resilient block receives more quota (lines 9-11). Afterwards, the basic blocks overhead quota is distributed among different instructions by selecting reliability-wise more important instruction for redundancy-based protection. An instruction for redundancy is selected depending upon its protection efficiency "λ" (lines 17-25). The protection overhead is subtracted from the basic block's overhead quota (line 25). The loop is iterated until the tolerable overhead of the basic block is exhausted or all instructions are protected (line 16).

VI. Results and Discussion

A. Experimental Setup

Fig. 6 shows our experimental and modeling setup. A reliability-aware ISS is employed which exhibits an integrated fault generation and injection module which takes different fault models and parameters as input. Important parameters are: (1) fault rate obtained using the neutron flux calculator [16] and city coordinates, (2) processor layout and frequency (a Leon-II processor @100 MHz is deployed in this work [24]), (3) single bit flip transient faults, randomly distributed. We consider three different fault rates in our experiments (1, 5, 10 faults/MCycles) to cover a wide range of cases (terrestrial to aerial). Like in prominent industrial and research projects by AMD [23] and IBM [22], the caches are assumed to be protected. The errors are observed at the application software layer and classified in different categories (see Section II). Error distributions, application analysis, IVI traces, etc. are generated from the reliability-aware ISS and forwarded to the resilience modeling and parameter estimation. Further details of fault injection process and reliability analysis can be found in [18][26].

Modeling and parameter estimation is done in MATLAB. Application control and data flow graphs are used for computing the masking probabilities. The models and application reliability analysis is forwarded to the reliability-driven compiler (based on GCC framework), that performs the selective instruction redundancy.

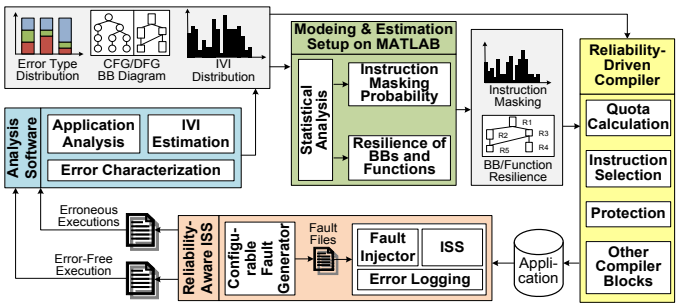

Fig. 6 Experimental and Modeling Setup

For evaluation, we employ various applications "H.264", "ADPCM", "SusanS", "CRC", "SHA" from MiBench and integrate them into a complex real-world application scenario of "secure audio-video coding and filtering" for quota distribution. The "H.264" application exhibits various compute intensive functions like "SAD" and "DCT".

B. Comparison to State-of-the-Art for Instruction Redundancy

For reliability comparison, we have selected the most prominent state-of-the-art in software based protection schemes, i.e. SWIFT [7], which performs instruction redundancy and recovery for all instructions in the program. The reliability comparison is performed for the vulnerability reduction using the model of [18] and protection efficiency (see Fig. 5, Section V). Since SWIFT [7] incurs >3X performance overhead for a RISC processor, its protection efficiency (i.e. reliability improvement per overhead) is ≤ 0.33. In contrast to this, the protection efficiency of our approach ranges from 0.95 to 1.21, see Fig. 7b. This corresponds to an improvement of 3.6X in the protection efficiency.

To have a more fair comparison, we adapted SWIFT towards selective instruction redundancy scheme by providing it our resilience based overhead quota distribution. However, instead of applying selective redundancy, SWIFT-variant selects instructions in a sequential manner. Fig. 7a shows such a comparison, where our experiments illustrates that, compared to SWIFT, our scheme provides a vulnerability reduction of 6% to 48% at 5% and 50% tolerable performance overheads, respectively.

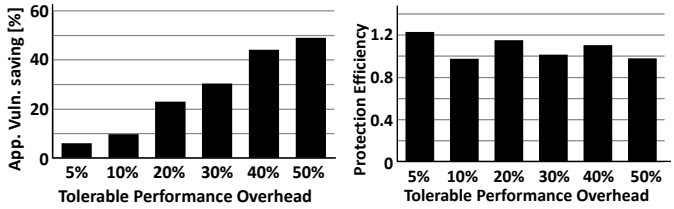

Fig. 7 (a) Overall reduction in the application vulnerability compared to the SWIFT [7] at different tolerable performance overhead constraints; (b) corresponding protection efficiency of our scheme; SWIFT has 0.33 protection efficiency

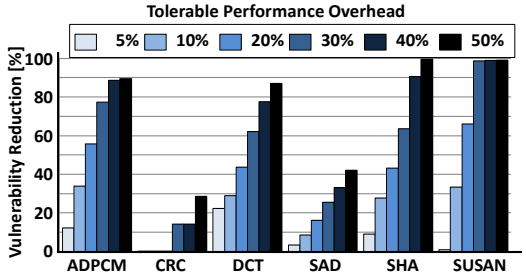

Fig. 8 Vulnerability reduction of our scheme for various application functions compared to the unprotected case at different tolerable performance constraints

We have additionally compared our scheme to the unprotected case. Fig. 8 shows the overall vulnerability reduction of various application functions at different tolerable performance overhead constraints. It is noticeable that our scheme reaches >80% vulnerability reduction at 50% overhead, since in this quota it already protects the most vulnerable instructions. It denotes a protection efficiency of 1.6 for "ADPCM", "SHA", and "SUSAN", which is 4.84X better compared to the 0.33 protection efficiency of full SWIFT [7]. Fig. 8 shows that "Susan" and "SHA" almost reach 95% vulnerability reduction with only a 50% overhead, while the vulnerability reduction is already more than 40% for only a 10% overhead. This also illustrates the benefit of using resilience for quota distribution, as in this case more quota is allocated to "ADPCM" and "SAD" and less quota is given to "DCT". High reliability is required for both "SHA" and "CRC" as they are critical applications in terms of data protection.

C. Results for Function Resilience and Software Masking

The resilience is used to distribute the tolerable performance overhead quota among different functions of the application. A more resilient function would get a less quota for protection compared to a less-resilient function that may not tolerate more errors. Fig. 9 illustrates

the resilience (in log scale) and the performance overhead quota for different application functions. The resilience and quota are provided separately for the "incorrect output" and "application failure" cases along with the combined case. Note, here "incorrect output" and "application failure" are both treated as information loss. In cases, where "incorrect output" is tolerable, resilience to "application failure" is important to be considered. In our experiments of selective instruction redundancy, we have employed the quotas for the combined case as we consider all types of errors. Due to its high resilience, "DCT" gets lesser quota compared to the "ADPCM", "SHA", and "SAD". The resilience of "DCT" is high because it is an unrolled version, with a relatively less number of branches compared to other applications that leads to less control flow errors in "DCT".

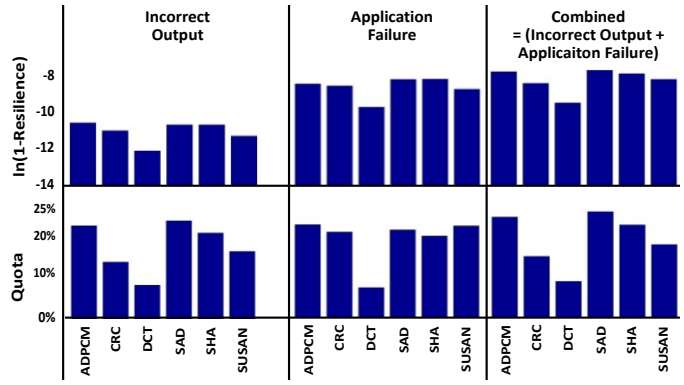

Fig. 9 Resilience of various application functions (inverse values in log scale): resilience is shown separately for "incorrect output", "application failures", and "combined"

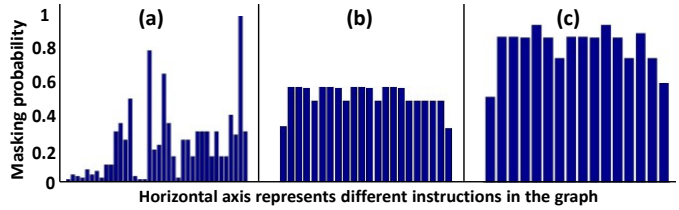

Fig. 10 Distribution of masking probabilities for different instructions in (a) DCT, (b) low-precision filter, (c) vertical edge-detector

Fig. 10 illustrates the distribution of instruction masking probability in three different functions "DCT", "Filter", and "Edge Detector". Fig. 10 (a) shows that the distribution of masking probabilities in "DCT" exhibits significant variations from instruction to instruction, depending upon the instruction dependencies. Zero masking probabilities denote the instruction on the sequential path, where all errors will propagate to the successor instructions. In the "Edge Detector", there is a high masking probability because the possible output values are "0" or "1", i.e. if there is an edge is or not. Therefore, all errors that are smaller than the *boundary value* (specified in the application program) of the difference between pixels from an edge will not be seen. A similar masking behavior is observed "Filter" function, where all errors that are smaller than the precision bits are dismissed. The small variations correspond to shift instructions, while similar group of bars in Fig. 10(b, c) show the symmetric execution paths in "Edge Detector".

VII. Conclusions

We illustrate that our selective software reliability scheme provides means to reduce the redundancy overhead, while still ensuring a high probability of correct output. This is leveraged by employing the variable function/basic block resilience and instruction masking probabilities to selectively protect instructions of application software under a user-provided tolerable performance overhead. Our novel resilience and masking models enable selective reliability optimization under constrained scenarios at both hardware and software levels. Due to the conceptual enhancements in Section III, IV, and V, state-of-the-art software reliability schemes by principal cannot reach the level of constrained reliability optimizations that our scheme provides.

Acknowledgment

This work is supported in parts by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 - spp1500.itec.kit.edu).

References

[1] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," IEEE TDMR, vol. 5, no. 3, pp. 305-316, 2005.

[2] S.Borkar et al., "Design and Reliability Challenges in Nanometer Technologies", IEEE DAC, pp. 75-75, 2004.

[3] P.Shivakumar, M.Kistler, "Modeling the effect of technology trends on the soft error rate of combinational logic". IEEE DSN, 2002.

[4] S. S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, T.Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor", MICRO, pp. 29-40, 2003

[5] R. Vadlamani et al.,"Multicore soft error rate stabilization using adaptive dual modular redundancy", IEEE DATE, pp. 27-32, 2010.

[6] D. Ernst et al., "Razor: circuit-level correction of timing errors for low-power operation," IEEE MICRO, vol. 24, no. 3, pp. 10-20, 2004.

[7] G. A. Reis, J. Chang, D. I. August, "Automatic instruction-level software only recovery", IEEE MICRO, pp. 36–47, 2007.

[8] N. Oh et al., "Error detection by duplicated instructions in super-scalar processors", IEEE Transaction on Reliability, 51-1, pp. 63-75, 2002.

[9] J. Hu et al., "In-Register Duplication: Exploiting Narrow-Width Value for Improving Register File Reliability," DSN, pp. 281-290, 2006.

[10] J. S. Hu et al., "Compiler-Directed Instruction Duplication for Soft Error Detection," DATE, vol.2, pp. 1056-1057, 2005

[11] G. V. Varatkar, N. R. Shanbhag, "Energy-efficient motion estimation using error-tolerance", IEEE ISLPED, pp. 113-118, 2006.

[12] V. Chippa, A. Raghunathan, K. Roy, S. Chakradhar, "Dynamic Effort Scaling: Managing the Quality-Efficiency Tradeoff", DAC, 2011.

[13] M. Shafique et al., "Power-Efficient Error-Resiliency for H.264/AVC Context-Adaptive Variable Length Coding", DATE, pp. 697-702, 2012.

[14] M. A. Makhzan, A. Khajeh, A. Eltawil, F. J. Kurdahi, "A low power JPEG2000 encoder with iterative and fault tolerant error concealment", IEEE TVLSI, vol. 17, no. 6, pp. 827-837, 2009.

[15] A. Heinig, M. Engel, F. Schmoll, P. Marwedel, "Improving transient memory fault resilience of an H.264 decoder", ESTIMedia, 2010.

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

[17] H.264 Codec JM 13.2: http://iphome.hhi.de/suehring/tml/index.htm.

[18] S. Rehman et al., "Reliable software for unreliable hardware: Embedded code generation aiming at reliability", Codess+ISSS, pp. 237-246, 2011.

[19] P. Giacinto et al., "An experimental Study of Soft Error in Microprocessors", MICRO, pp. 30-39, 2005.

[20] S. Z. Shazli, M. B. Tahoori, "Obtaining Microprocessor Vulnerability Factor Using Formal Methods", DFTVS, 2008.

[21] M. Xie, K.-L. Poh, Y.-S. Dai, "Computing Systems Reliability: Models and Analysis", Springer, ISBN 978-0-306-48496-4, 2004.

[22] IBM® XIV®: http://publib.boulder.ibm.com/infocenter/ibmxiv/r2/index.jsp.

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

[24] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC v8 architecture", DSN, pp. 409-415, 2002.

[25] A. Rajendiran et al. "Reliable computing with ultra-reduced instruction set co-processors", IEEE DAC, pp. 697-702, 2012.

[26] S. Rehman, M. Shafique, J. Henkel, "Instruction Scheduling for Reliability-Aware Compilation", IEEE DAC, pp. 1288-1296, 2012.

[27] S. Rehman et al., "RAISE: Reliability Aware Instruction SchEduling for Unreliable Hardware", IEEE ASP-DAC, pp.671-676, 2012.

[28] J. Cong, K. Gururaj, "Assuring Application-Level Correctness Against Soft Errors", ICCAD, pp. 150-157, 2011.

[29] A. Sundaram et al., "Efficient fault tolerance in multi-media applications through selective instruction replication", WREFT, pp. 339-346, 2008.

[30] E. Borin et al., "Software-Based Transparent and Comprehensive Control-Flow Error Detection", CGO, pp. 333-345, 2006.