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 vulnerability 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).

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.

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 $F = \{f_1, f_2, ..., f_l\}$.

Function: Each function may comprise of multiple basic blocks, $F_i = \{B_{ii}, D_{ii}, I_{ii}, E_{ii}, P_{ii}, R_{ii}\}$ such that $B_{ii} = \{B_{ii}, B_{i2}, ..., B_{il}\}$, $D_{ii}$ is the set of basic block dependencies, $I_{ii}$ is the average execution time, $E_{ii}$ is the expected execution frequency, and $R_{ii}$ is the resilience of the $i^{th}$ function.

Basic Block: A basic block is a sequence of instructions without any jump/branch; $B_{ii} = \{I_{i1}, E_{i1}, L_{i1}, F_{i1}, P_{i1}, R_{i1}\}$ such that $I_{i1} = \{I_{11}, I_{21}, ..., I_{il}\}$, $E_{ii}$ is the average execution time, $F_{i1}$ is the expected execution frequency, and $R_{ii}$ is the resilience of the $j^{th}$ basic block of $i^{th}$ function. $E_{ii}$ is the set edges that denote the instruction dependencies; $E_{ii} = \{E_{i1}, E_{i2}, ..., E_{il}\}$ and is given as the edge weight that represents the latency from instruction $I_{i1}$ to $I_{i2}$.

Instruction: Each instruction is represented as a tuple $I_{ijkl} = (P_{ijkl}, P_{i0}, \alpha, P, S, L, d, \sigma, \phi)$. $P_{ijkl}$ is the total masking probability of the $k^{th}$ instruction to the visible output. $P_{i0}$ is the internal masking during the execution of an instruction (i.e. masking through the pipeline). $P_{ijkl} = \{p_1, ..., p_d, \sigma\}$ and $S = \{s_1, ..., s_d\}$ are the sets of predecessor and successor instructions, respectively. $L$ is the execution time of the instruction, while $d$ and $\alpha$ 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.

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: $S_C$, $S_{CI}$, 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 $I_{ijkl}$.

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 $\xi$ is the final state probability vector containing the probability of three states: $p_{C}$, $p_{CI}$, and $p_{F}$.

$$\xi(B_{ij}) = [p_{C} \quad p_{CI} \quad p_{F}] = \xi(B_{ij-1}) \times \prod_{x=1}^{l} \xi(x)$$

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 $B_i$ is given as $R(B_i)=1−H(X|Y)/H(X)$, where $H(X)$ is the information about the correct execution, i.e. $H(X)=p_{Live}$, such that $b_{Live}$ denotes the bits of live output registers of $B_i$. The conditional entropy $H(Y|X)$ is the information lost in $B_i$ and given as Eq. 3, where $p_{IC}(x)$ 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”. 

$$R(B_i)=1−H(X|Y)/H(X); H(X)=b_{Live}$$  

Assuming, resilience of a basic block $R(B_i)$ can be characterized as resilience to “incorrect output” and resilience to “application failures”; we can compute the conditional entropy specifically for both cases. $H(X|Y)_F$ is given as $p_{F}(B_i)$ using Eq. 2, while $H(X|Y)_{IC}$ is given by Eq. 4.

$$H(X|Y)_{IC}=-[p_{IC}(x) \log_2 p_{IC}(x)/(2^{\pi}−1)+(1−p_{IC}(x)) \log_2(1−p_{IC}(x))]$$

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.

$$R(B_i)=1−[H(X|Y)_{IC}+H(X|Y)_F−(H(X|Y)_{IC} \times H(X|Y)_F)]/H(X)$$

Given the resilience values of all basic blocks $B_i$ of a function $f$, resilience $R(f)$ can be computed using Eq. 6.

$$R(f)=\sum_{b_{IC} \in B_i} R(b_i) = \sum_{b_{IC} \in B_i} \sum_{b_{F} \in F} e_F(b_i)$$

**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 $=\Rightarrow p_{11}=1$, $p_{21}=0$
- Initial state and input is error-free; $[p_{IC}, p_{F}]_{t=0}= [1, 0, 0, 0]$.

Note, $p_{11}+p_{12}+p_{21}=1$. To expedite the parameter estimation process, we have grouped instructions in $N^j$ 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 $N_e$ different fault-injection experiments at a given fault rate, $N_s$ and $N_e$ are the number of cases with correct and incorrect output, respectively. For a particular fault injection experiment $s$, for a certain instruction category $t$, the transition probability $p_{ij}$ can be estimated using the maximum likelihood, thus deriving Eq. 7. $N(t,s)$ denotes the number of instruction type $t$ in that simulation.

$$p_{ij}(t)=\frac{N(t,s)}{N(t,s)}$$

Assuming $p_{jj}(t)=p_{j0}(t)$, we can re-use Eq. 7 to obtain the probability $p_{j0}(t)$. In this way we can compute all the remaining transition probabilities such that $p_{jj}(t)=p_{0j}(t)=1−p_{j0}(t)$; and $p_{j0}(t)=p_{j2}(t)$. 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.

**Complexity:** The complexity of resilience estimation is $O(|B| \times N_t \times \log|B|)$, which is much smaller than the complexity of fault tree based methods (i.e. $O(|B| \times |B|^3)$) and Monte-Carlo simulations (i.e. $O(|B| \times |B|^3)$) 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 $p_{TM}(I_{jk})$ at a certain instruction $I_{jk}$ is defined as the probability that an error at $I_{jk}$ 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:** $p_{TM}(I_{jk})$ depends upon two key parameters:

1. $p_p(I_{jk})$ which is defined as the total masking during the execution of an instruction $I_{jk}$ 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 $I_{jk}$.
2. $p_{postM}(I_{jk})$ which is defined as the total masking probability after the execution of an instruction $I_{jk}$ such that, an erroneous output value is masked in the path from $I_{jk}$ until the visible output point due to, for instance, dependent instructions, operation masking, control flow, etc.

$$P_{TM}(I_{jk})=P_p(I_{jk})+P_{postM}(I_{jk}) \times P_{postM}(I_{jk})$$

Once $p_p(I_{jk})$ and $P_{postM}(I_{jk})$ are estimated, $p_{TM}(I_{jk})$ 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 $P_{postM}(I_{jk})=0$.

**Estimating $p_p(I_{jk})$:** An instruction $I_{jk}$ 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 $p_p(I_{jk})$ can be computed using Eq. 9.

$$PM(I_{jk})=\sum_{p \in Pipeline} LM(p|e_p)$$

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

$$P_{postM}(I_{jk})=\prod_{v \in S_{gk}} [P_v(s) \times (1−P_v(s))] \times P_{postM}(s)$$

$S_{gk}$ 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.

$$P_v(I_{jk})=\left(\frac{1}{N_{Bits \cdot o_{jk}}} \times \sum_{b \in o_{jk}} P_m(b, I_{jk})\right)$$

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

Example: Let us assume an instruction $c=adb$, where operation is “bitwise and” ($\&$”), $a=0x00000000$, and $b=0xFFFFFFFF$ [bit sequence

$$c=adb$$

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Masking Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{TM}$</td>
<td>$p_p(I_{jk})$</td>
</tr>
<tr>
<td>$p_{postM}$</td>
<td>$P_{postM}(I_{jk})$</td>
</tr>
</tbody>
</table>
```

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

1. Complete derivation is omitted due to space limitations.
31...0]. In this case, for b, error masking probability w.r.t. the operation type \( p_{bd}(b_i, l_i) \) is given as: \( p_{bd}(b_0,15, &)=0 \) and \( p_{bd}(b_{16}, &)=1 \).

Our scheme computes \( p_{\text{post}}(l_i) \) recursively using \( p_{\text{post}}(l_i) \) of the successor instructions \( (S_i) \), starting with the leaf node (i.e. last instruction of the execution path) that has \( p_{\text{post}}(l_i)=0 \; \text{(i.e. all instructions occurring at the leaf node will propagate to memory)} \). 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 \( O(\Sigma_{v \in b_i} (|l_i|+|E_j|)) \) and space complexity is \( O(2 \Sigma_{v \in b_i} |l_i|) \).

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 \( R_p \)).

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

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:** Resilience-Driven Selective Instruction Redundancy

**INPUT:**

- Original unprotected application software // see formal model in Sec. II
- User-provided tolerable performance overhead in cycles

**OUTPUT:**

- Application software with redundant instruction

**BEGIN**

1. \( \forall f_i \in F \}
2. \( \forall B_j \in f_i \}
3. \( R_q \leftarrow \text{computeBBResilience}(B_q); \}

**// see Eq. 5**

4. \( R_i \leftarrow \text{computeNormalizedFunctionResilience}(f_i); \}

**// see Eq. 6**

5. \( \forall f_i \in F \}
6. \( \forall f_i \in F \}
7. \( \forall f_i \in F \}
8. \( \forall f_i \in F \}
9. \( \forall f_i \in F \}
10. \( \forall f_i \in F \}
11. \( \forall f_i \in F \}
12. \( \forall f_i \in F \}
13. \( \forall f_i \in F \}
14. \( \forall f_i \in F \}
15. \( \forall f_i \in F \}
16. \( \forall f_i \in F \}
17. \( \forall f_i \in F \}
18. \( \forall f_i \in F \}
19. \( \forall f_i \in F \}
20. \( \forall f_i \in F \}
21. \( \forall f_i \in F \}
22. \( \forall f_i \in F \}
23. \( \forall f_i \in F \}
24. \( \forall f_i \in F \}
25. \( \forall f_i \in F \}
26. \( \forall f_i \in F \}
27. \( \forall f_i \in F \}
28. **END**

**Algorithm 1:** Resilience-Driven Selective Instruction Redundancy

**INPUT:**

- \( A \) : Original unprotected application software // see formal model in Sec. II
- \( P \) : User-provided protection method like EDDI, SWIFT, CRAFT, etc.

**OUTPUT:**

- \( A' \) : Application software with redundant instruction

**BEGIN**

1. \( \forall f_i \in F \}
2. \( \forall B_j \in f_i \}
3. \( R_q \leftarrow \text{computeBBResilience}(B_q); \}

**// see Eq. 5**

4. \( R_i \leftarrow \text{computeNormalizedFunctionResilience}(f_i); \}

**// see Eq. 6**

5. \( \forall f_i \in F \}
6. \( \forall f_i \in F \}
7. \( \forall f_i \in F \}
8. \( \forall f_i \in F \}
9. \( \forall f_i \in F \}
10. \( \forall f_i \in F \}
11. \( \forall f_i \in F \}
12. \( \forall f_i \in F \}
13. \( \forall f_i \in F \}
14. \( \forall f_i \in F \}
15. \( \forall f_i \in F \}
16. \( \forall f_i \in F \}
17. \( \forall f_i \in F \}
18. \( \forall f_i \in F \}
19. \( \forall f_i \in F \}
20. \( \forall f_i \in F \}
21. \( \forall f_i \in F \}
22. \( \forall f_i \in F \}
23. \( \forall f_i \in F \}
24. \( \forall f_i \in F \}
25. \( \forall f_i \in F \}
26. \( \forall f_i \in F \}
27. \( \forall f_i \in F \}
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 \( \times 77% \) better reliability compared to the first scheme.

**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 ($R_{bb}$, see Section III) and execution frequencies ($EF_{bb}$, 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 “$\eta$” (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.

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.

C. Results for Function Resilience and Software Masking

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.

We want to conclude that resilience is an important aspect of modern application development. We have shown that resilience can be effectively integrated into compilers to provide a reliable execution environment. Our results demonstrate that resilience can be achieved with minimal performance impact, making it a viable solution for software developers.
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”.

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