

Exploiting Program-Level Masking and Error Propagation for Constrained Reliability Optimization

Muhammad Shafique, Semeen Rehman, Pau Vilimelis Aceituno, Jörg Henkel
Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Germany
{muhammad.shafique, henkel}@kit.edu; seemeen.rehman@student.kit.edu

Abstract—Since embedded systems design involves stringent design constraints, designing a system for reliability requires optimization under tolerable overhead constraints. This paper presents a novel reliability-driven compilation scheme for software program reliability optimization under tolerable overhead constraints. Our scheme exploits program-level error masking and propagation properties to perform reliability-driven prioritization of instructions and selective protection during compilation. To enable this, we develop statistical models for estimating error masking and propagation probabilities. Our scheme provides significant improvement in reliability efficiency (avg. 30%-60%) compared to state-of-the-art program-level protection schemes.

I. INTRODUCTION AND RELATED WORK

Reliability has become a major design objective for on-chip systems due to advanced technology scaling [1]. Several hardware level (TMR, pipeline protection, etc. [3]) and software level (SWIFT-R [6], EDDI [7], in-register duplication [8], etc.) schemes have emerged. These schemes principally rely on full-scale duplication either at logic or instruction level, thus they incur significant overhead. Hence, trends have been set for adaptive and selective protection, i.e. applying reliability to more susceptible logic blocks [4][5] or program code [12] to obtain cost-efficient reliable designs. Hardware-level adaptive techniques target at reducing the power overhead by activating and deactivating the redundant hardware while monitoring the architectural vulnerability [2][4][5] or instruction vulnerability [19][30]. However, these schemes are still prohibitively expensive in terms of area and power for embedded computing. The area overhead may be reduced by using, for instance, reliable ultra-reduced instruction set co-processors to detect permanent faults [20][28].

In order to complement/alleviate hardware level schemes or to provide reliability in cases where hardware redundancy is prohibitive, *reliability-driven compilation with selective instruction protection* has emerged as a promising layer to improve the overall system reliability. Principally, these selective instruction protection schemes [11]–[17] identify so-called *critical instructions* that are more sensitive with respect to correct program execution *either* for a user-defined output data range *or* in order to avoid program crashes. The scheme of [11] performs fault injection to identify an instruction vulnerability factor (IVF) and protect instructions for IVF greater than a user-defined threshold. The scheme in [12] first partitions the program output in user-defined tolerable and non-tolerable sets for multimedia applications. Afterwards, it identifies critical instructions through static analysis of instruction dependencies in the program data flow graph. The scheme of [13] also targets multimedia applications and performs a static analysis on the program to identify instructions that affect the control flow, loop counters, etc. The schemes of [14][15] exploit inherent resilience of multimedia applications to reduce power consumption. The schemes of [16][17][29] perform transformations during the compilation or dynamic code compilation to reduce the probability of errors for improved soft error resilience. In summary, the above-mentioned state-

of-the-art schemes suffer from three main limitations:

- 1) They mainly target multimedia applications (which have a high degree of intrinsic error tolerance) but they do not efficiently protect non-multimedia applications.
- 2) They account only for instruction dependencies and ignore the probability of error masking at instruction level.
- 3) They protect all instructions that ultimately affect the program output but they do not provide measure for the relative importance of instructions w.r.t. reliability. Therefore, these schemes are hardly applicable in cases where constraints are applied to guide reliability.

A. Motivational Case Study on Program Reliability Analysis

Our reliability analysis experiments in Fig. 1 show that different instructions have varying vulnerabilities¹ to errors due to hardware-level faults. Even different programs have varying vulnerability distributions. This is due to their varying instruction profiles and vulnerable time they dwell in different pipeline components. Fig. 1 shows that the instruction vulnerabilities in “SusanC” are relatively higher compared to that in “ADPCM”. “SusanC” has a relatively sparse distribution. Variations in instruction vulnerabilities and data/control flow properties of “SusanC” and “ADPCM” also hint towards their varying degree of instruction-level error masking and error propagation. This leads to a noticeable difference in program output errors of both applications: i.e. 5% output errors in “SusanC” vs. 45% in “ADPCM” at 5faults/MCcycle (using Monte Carlo simulations; see details of experimental setup in Supplementary Section S3). This shows that despite low instruction vulnerabilities in “ADPCM”, there is a significantly high propagation of unmasked errors to different execution paths and program outputs that leads to a higher susceptibility to program errors.

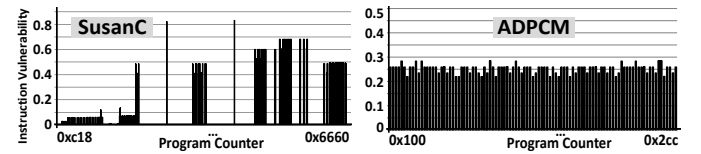


Fig. 1 Distribution of Instruction Vulnerability Index [10] for Different Instructions in “SusanC” and “ADPCM”

Summary and conclusion from observations: The above analysis illustrates that ignoring inherent error masking properties and only accounting for instruction vulnerability or instruction dependencies may lead to *inefficient protection*. Exploiting error masking and propagation properties provide a strong potential for efficient program reliability optimization under performance constraints. Furthermore, *not all* parts of a given program may require the same level of protection due to their diverse data/control flow properties. The error propagation and masking properties may be leveraged to limit the growing overhead of reliability optimization, while utilizing the tolerable overhead for protecting the sensitive parts of a program.

B. Problem and Research Challenges

The problem of constrained program reliability optimization poses the following **key research challenges**:

- 1) Choosing a set of instructions for program level protection while accounting for program level error masking and propagation properties under a given performance overhead budget.

¹ Computed using the model of [10], see Section S2 for further details.

- 2) To enable constrained program optimization, it is important to accurately model and estimate the error masking and propagation probabilities at instruction level, such that instructions can be prioritized w.r.t. their impact on reliability.

C. Our Novel Contributions and Concept Overview

To address the above-mentioned challenges, a novel scheme for constrained program reliability optimization is proposed that employs:

- 1) **A Selective Instruction Protection Algorithm (Section IV)** that chooses a set of instructions for program-level protection considering their reliability-wise higher/lower impact on the program output, (i.e. jointly accounting for instruction vulnerability, masking probabilities, and error propagation probabilities) under a given performance overhead budget. To enable this, our scheme requires:
- 2) **Modeling and Estimation of Instruction-Level Error Masking Index (Section III.A)** that jointly accounts for masking effects due to program data/control flow properties and microarchitectural effects.
- 3) **Modeling and Estimation of Instruction-Level Error Propagation Index (Section III.B)** as a joint function of non-masking probability of all the successor instructions and all the possible execution paths along with their path execution probabilities.

Fig. 2 illustrates an overview of our novel scheme that is integrated in a reliability-aware compiler.

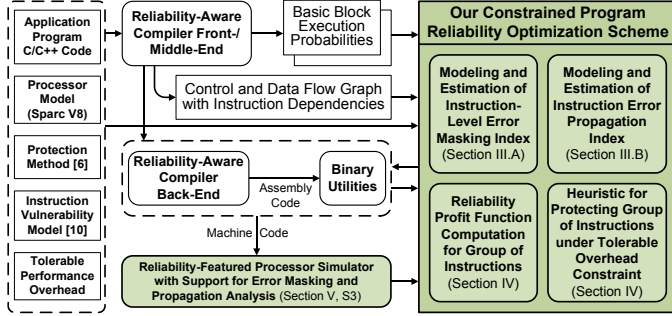


Fig. 2 Our Constrained Program Reliability Optimization Scheme (in highlighted boxes) Integrated into a Reliability-Aware Compiler

II. SYSTEM MODELS

Application Model: A program function F is a set of basic blocks $B = \{B_1, \dots, B_m\}$ each having a control flow probability P_{CF} . $B_i = \{I_i, f_i, l_i\}$. f_i and l_i denote the basic block number of executions and execution time, respectively. I_i denotes the set of instructions in the basic block, such that the complete function can be represented as a graph $G = (V, E)$. V is the set of instruction nodes, s.t. $V = \{I_1, \dots, I_n\}$. L_G defines a set of leaf nodes of G . Each instruction node is given as a tuple $I = (l, o, S, P, \xi, IMI, \phi)$. l and o denote the instruction latency and set of operands, respectively. S and P denote the set of successor and predecessor instructions, respectively. ξ , IMI , and ϕ denote the instruction error propagation index, instruction masking index, and instruction vulnerability index, respectively. For estimation of the instruction vulnerability index ϕ , we employ state-of-the-art technique of [10] (see details in supplementary Section S2). E is the set of edges that denote dependencies between instructions, s.t. $E = \{e_{xy} | (I_x, I_y) \in V\}$. The weight of the edge represents the latency of moving from instruction I_x to I_y . We define the term *instruction path* p in the graph G as a sub-graph with a sequential set of instructions, such that each instruction in p has exactly one predecessor and successor, i.e. $\forall i \in p | i \notin \text{Leafs} | i.S | = I \wedge | i.P | = I$. Note that an instruction may appear in different paths, as it has several successors. We define the term *execution path* ep in the control flow as a set of instructions that execute with a conditional probability of $P_{CF}(ep | I)$, given an instruction I executes.

Processor and Fault Model: in-order RISC single core processor with multiple *pipeline* stages subjected to “transient faults” with single or multiple bit upsets.

III. MODELING AND ESTIMATION OF PROGRAM-LEVEL ERROR MASKING AND ERROR PROPAGATION PROPERTIES

In the following, we model the error masking and error propagation properties at instruction level and identify different parameters that affect error masking/propagation in software programs. These models jointly provide a measure of severity of an error at a given instruction, thus can be used to prioritize instructions for constrained reliability optimization. Fig. 3 provides an overview and flow of steps to estimate the error masking probabilities and the error propagation index (for the ease of reader, we have provided the complete set of parameters in Table I, Supplementary Section S1.D).

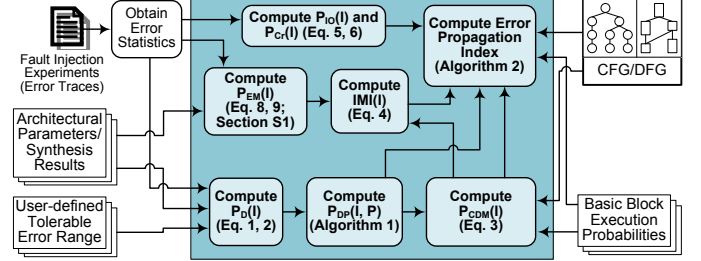


Fig. 3 Flow of Steps to Compute the Instruction-Level Masking Probabilities and Error Propagation Index

A. Program-Level Error Masking

Definition: we quantify program-level error masking as the *Instruction Masking Index* ($IMI(I)$) of an instruction I , which is defined as the total probability of an error at instruction I being masked until the last instruction of all of its instruction paths p (i.e. leaf nodes), such that the output of p is correct.

Parameter Identification: The masking of an error at an instruction I can occur due to the following factors that constitute the parameters of the $IMI(I)$.

- 1) *Error masking due to data flow properties* ($P_{DF}(I, p)$): The error at instruction I may be masked due to the successor instruction in path p depending upon two factors: (a) instruction type, and (b) value of the operand variables.

- 2) *Error masking due to the changing control flow properties* ($P_{CDM}(I)$), where a highly probable execution path may exhibit masking instructions that block the propagation of the error to the relevant program output. It may also happen that a highly probable execution path does not even use the erroneous value from the preceding basic block.

Example: Fig. 4(a) illustrates an example of control flow graph showing different basic blocks and the control flow probabilities. The error in basic block B_1 may be blocked in B_3 due to the “&” and/or “or” instructions (see Fig. 4(b)). However, if the control flow follows B_2 , the error will propagate to B_4 and ultimately to the visible program output. Note that B_3 has a higher probability of execution compared to B_2 . Fig. 4(b) shows that the error masking occurs only in case of the “&” and “or” instructions due to the value of the other operand, while “+” does not mask the error.

- 3) *Masking in pipeline components during the execution of instruction I* ($P_{EM}(I)$) due to microarchitecture-level logical masking effects, i.e. the error within a pipeline component (combinatorial logic) is not visible at the output latch as the error propagation through different logic elements/gates is blocked due to subsequent logic element(s) and the output of the pipeline component remains correct (see an example in Section S1.A).

From software program’s perspective the most important challenge is to estimate the masking probabilities $P_{DF}(I, p)$ and $P_{CDM}(I)$ that represent the novel contributions of this paper and are discussed in the following. $P_{EM}(I)$ is estimated using fault injection in different pipeline components (see details in Section S1.B and S3).

Estimation of $P_{DP}(I, p)$: For each instruction I , the masking probability depending upon the data flow can be modeled through Eq. 1.

$$P_D(I) = \sum_{x \in I.O.Bits} P_D(x, I) \times P_e(x) \quad (1)$$

O is the set of operands of instruction I with a set of $Bits$. $P_D(x, I)$ is the probability of masking of each bit depending upon the instruction type (as discussed in Fig. 4(b)). $P_e(x)$ is the error probability of each bit, which can be simplified to $1/N_{Bits}$ by assuming the same error probability for all bits, where N_{Bits} is the bit-width of operand registers. In case, the user specifies a tolerable range th for the error in the output value ($\log_2(th)$ provides the number of bits), Eq. 1 can be modified as Eq. 2.

$$P_D(I) = \sum_{x \in I.O.Bits \setminus \log_2(th)} P_D(x, I) \times P_e(x) + \log_2(th) / N_{Bits} \quad (2)$$

In a given instruction path p , the error masking depends upon the individual instructions as well as the combination of consecutive instructions. Note, the scheme of [21] does not account for combined masking effects of consecutive instructions, thus may provide inaccurate estimates for the masking probability. Depending upon their masking behavior, instructions can be classified into two categories:

Type A: instructions like “&” and “or” with a variable value assuming a random bit masking; the theoretical masking probability for such instructions is given as $1/2$;

Type B: instructions like “shift” left or right by a constant value; where the masking bits can be inferred from the source code; these instructions affect the computation of masking probabilities of the predecessor instruction. In case the joint masking effects of consecutive instructions are ignored, the total masking probabilities are under-estimated (see an example in Section S1.C).

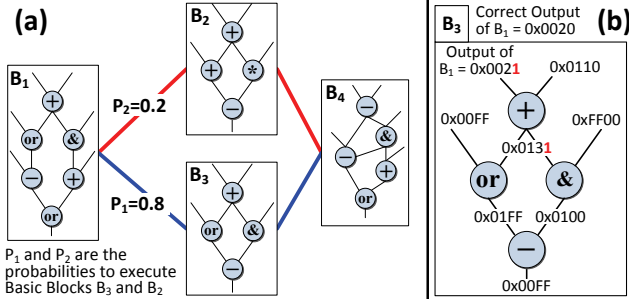


Fig. 4 An Example Control & Data Flow Graph showing the Error Masking due to Successor Instructions and Changing Control Flow

Algorithm 1 illustrates the pseudo-code for computing the error masking probability $P_{DP}(I, p)$ for instruction I of path p in an iterative manner starting from the leaf nodes. First, for all instructions in G , $P_D(I)$ is computed using Eqs. 1-2 (line 1). Afterwards, $P_{DP}(I, p)$ is initialized for all the leaf nodes with their corresponding $P_D(I)$, as they represent the last nodes of the graph (line 2). The list of ready nodes (i.e. instructions for which $P_{DP}(I, p)$ can be computed) L is initialized with the predecessors of leaf nodes (lines 3). The while loop (lines 4-19) to compute the $P_{DP}(I, p)$ iterates till the list is empty. For each iteration of the loop, first, the set of all possible instruction paths $Paths$ is generated from every instruction I in L until the leaf nodes L_G (line 6). Afterwards, for all possible paths of every instruction, i.e. all possible combinations of (I, p) , the number of consecutive instructions of *type B* N_B is computed (lines 9-12) and the masking probabilities $P_{DP}(I, p)$ are computed (lines 13-16). If there are 2 or more consecutive instructions of *type B*, their cumulative masking probabilities are considered to compute $P_{DP}(I, p)$ (lines 14-16); otherwise, the independent masking probabilities are considered (line 13). After computing $P_{DP}(I, p)$, the instruction I is removed from the list and its predecessors are added to the list (line 18). The loop of instructions is iterated until the list is empty (lines 18, 19). Fig. 9 in Section S1.C illustrates step-by-step Algorithm 1 with the help of an example instruction graph.

Algorithm 1: Computing the Error Masking Probability $P_{DP}(I, p)$

INPUT: Instruction flow graph $G = (V, E)$, Set of leaf nodes L_G , Set of predecessors and successors for each instruction (P, S) .
OUTPUT : $P_{DP}(I, p)$ – Masking probabilities due to data flow for each instruction I for path p .
BEGIN
1. $\forall I \in G \quad P_D(I) \leftarrow \text{compute}P_D(I); \quad // \text{Eq. 1; Eq. 2}$
2. $\forall I \in L_G(G) \quad P_{DP}(I, p) \leftarrow P_D(I);$
3. $List \ L(); \quad \forall x \in L_G.P \quad L.add(x); \quad // \text{List of ready nodes}$
4. **while** ($L.isEmpty() == \text{false}$) **{**
5. $\forall I \in L \{$
6. $\quad I.Paths \leftarrow \text{generatePaths}(I); \quad // \text{generate all instruction paths}$
7. $\quad \forall p \in I.Paths \{$
8. $\quad \quad N_B \leftarrow 0; \quad p' \leftarrow p;$
9. $\quad \quad \forall x \in p' \{$
10. $\quad \quad \quad \text{if}(x.typeB) \{ N_B \leftarrow N_B + 1; \quad p'.remove(x); \}$
11. $\quad \quad \quad \text{else} \{ N_B \leftarrow 0; \quad p'.remove(x); \};$
12. $\quad \quad \}$
13. $\quad \quad \text{if}(N_B \leq 1) \{ P_{DP}(I, p) \leftarrow P_D(I) + (1 - P_D(I)) \times P_D(I.S);$
14. $\quad \quad \text{else} \{ // \text{add masking probabilities of } N_B \text{ consecutive successor instrs.}$
15. $\quad \quad \quad P_D'(I) \leftarrow \sum_{x=I}^{I + N_B} P_D(x);$
16. $\quad \quad \quad P_{DP}(I, p) \leftarrow P_D'(I) + (1 - P_D'(I)) \times P_D(I.S); \}$
17. $\quad \quad \}$ **// end loop of paths**
18. $\quad L.remove(I); \quad \forall ip \in I.P \quad L.add(ip); \}$
19. **}** **// end While loop when the List of nodes is empty.**
20. **END**

Estimation of $P_{CDM}(I)$: For each instruction I , the error masking probability due to both data and control flow $P_{CDM}(I)$ can be modeled as the weighted masking probability due to data flow $P_{DP}(s, p)$ for all the *successor* instructions s of instruction I and in all the corresponding instruction paths $s.Paths$ while considering the execution path probabilities $P_{CF}(ep | I)$; see Eq. 3.

$$P_{CDM}(I) = \sum_{ep | I \in ep} (P_{CF}(ep | I) \times \prod_{s \in I.S; \forall p \in s.Paths} P_{DP}(s, p)) \quad (3)$$

The execution path probabilities $P_{CF}(ep | I)$ are estimated using the GCC framework with option “-fguess-branch-prob”.

Estimation of Instruction Masking Index ($IMI(I)$): we model the *Instruction Masking Index* ($IMI(I)$) using Eq. 4. $P_{CDM}(I)$ and $IMI(I)$ are computed using the *breadth-first search* starting from the leaf nodes and explores G in a bottom-up fashion.

$$IMI(I) = P_{EM}(I) + (1 - P_{EM}(I)) \times P_{CDM}(I) \quad (4)$$

Non-masked errors may propagate to subsequent instructions in different execution paths, thus corrupting the correct program output. Error propagation to multiple execution paths denote a high susceptibility of program errors. Therefore, we *leverage the error masking probabilities to model the program-level error propagation to quantify the severity of the unmasked errors*.

B. Program-Level Error-Propagation

Definition: we quantify the program-level error propagation effects as the *Instruction Error Propagation Index* $\xi(I)$, which is defined as the product of the non-masking probability (i.e. the probability that an error is *visible* at the program output) of all the *successor* instructions of a given instruction I for all possible instruction paths.

Modeling and Estimation of Instruction Error Propagation Index ($\xi(I)$):

Algorithm 2 shows the procedure for estimating ξ for each instruction in graph G . First the error propagation index for all the leaf nodes (outputs) is initialized with 1, as the errors in the leaf nodes are considered as propagated to the next stages of the program execution (line 1). Afterwards, a list C is created that contains the traversed instructions for which ξ is computed (line 2). The algorithm uses a FIFO based queue Q to store the instructions for which ξ can be computed considering all successors are completed (line 2). Initially the predecessors of the leaf nodes are inserted in Q (lines 3). $\xi(I)$ is computed for all the instructions whose successors are in the C list (lines 4-14). Otherwise, the instruction I is inserted back into the queue Q

Algorithm 2: Computing the Instruction Error Propagation Index

INPUT: Instruction flow graph $G=(V, E)$, Set of leaf nodes L_G , masking probabilities due to data $P_{DP}(I, p)$, control flow probabilities $P_{CF}(p | I)$.

OUTPUT: $\xi(I)$ – Set of error propagation index for each instruction

BEGIN

```

1.  $\forall I \in L_G(G) \ \xi(I) \leftarrow 1;$ 
2.  $List \ C(L_G); \ \text{Queue } Q();$ 
3.  $\forall I \in L_G \ \{ \forall i \in I.P \ Q.Enqueue(i); \}$ 
4. while ( $Q.isEmpty() \neq true$ ) do
5.    $I \leftarrow Q.Dequeue();$ 
6.   if ( $\forall s \in I.S \ s \in C$ ) then
7.      $\xi \leftarrow 0;$ 
8.      $\forall s \in I.S \{$ 
9.       if ( $IMI(I) = 0$ )  $\xi \leftarrow \xi + \xi(s) \times P_{Execution}(s | I);$ 
10.      else  $\xi \leftarrow \xi + \sum_{p \in s.Paths} ((1 - P_{DP}(s, p)) \times P_{CF}(p | I) \times \xi(LG(p)));$ 
11.    end loop of successors
12.     $\xi(I) \leftarrow \xi \times (P_{IO}(I) / (P_{IO}(I) + P_{Cr}(I))); \ C.add(I); \ \forall i \in I.P \ Q.Enqueue(i);$ 
13.  else  $Q.Enqueue(I);$ 
14. end while loop when the Queue of instructions is empty
15. END
```

(line 13). For $\xi(I)$ computation for an instruction I , all of its successors instructions $I.S$ are considered (line 6-11). In case, a successor instruction s is a non-masking instruction, its $\xi(s)$ is directly used in the ξ computation along with its corresponding execution probability $P_{Execution}(s | I)$ (line 9). Otherwise, ξ is computed as the weighted sum of non-masking probability $(1 - P_{DP}(s, p))$ multiplied with ξ of the leaf nodes of the path p , for all the successors $I.S$ and their instruction paths $s.Paths$ (line 10). Note that the weight is determined using the path probability $P_{CF}(p | I)$. An error in an instruction may lead to an “incorrect output” or “program failure” (like crash, halt, abort). Therefore, $\xi(I)$ accounts for the probability of crash $P_{Cr}(I)$ (see Eq. 5) and probability of incorrect output $P_{IO}(I)$ (see Eq. 6); line 12. After computing the $\xi(I)$ for instruction I , all of its predecessors $I.P$ are added into the queue Q (line 12).

$$P_{Cr}(I) = \sum_{b \in COB(I)} P_{eOP}(b, I) + (1 - IMI(I)) \times \sum_{b \in CAB(I)} P_{eAd}(b, I); \quad (5)$$

$$P_{IO}(I) = 1 - IMI(I) - P_{Cr}(I) + IMI(I) \times P_{Cr}(I); \quad (6)$$

$COB(I)$ is the number of critical opcode bits that lead to a “non-decodable instruction” error. $P_{eOP}(b, I)$ and $P_{eAd}(b, I)$ are the error probabilities in the opcode and address bits. $CAB(I)$ is the number of critical address bits that lead to a “memory segmentation” error due to an access to the invalid or restricted memory region.

The above-discussed error propagation index ($\xi(I)$) captures the error masking and propagation properties of the software program, which are crucial for prioritizing the instructions in basic blocks with respect to reliability. Therefore, our instruction-level error masking and propagation models ($IMI(I)$ and $\xi(I)$) enable constrained software reliability optimization on unreliable hardware that facilitates tradeoff between performance loss and reliability improvement.

IV. OUR CONSTRAINED INSTRUCTION PROTECTION SCHEME

We propose a heuristic that selectively protects instructions or group of consecutive instructions in different execution paths in a given function. For this, our scheme leverages the error propagation index ξ , and instruction vulnerabilities ϕ to compute the instruction reliability profit.

Reliability Profit Function (RPF) for Choosing Instructions for Protection: Our instruction protection heuristic employs a *reliability profit function* (RPF, Eq. 7) which is defined as the accumulated reliability efficiency of a group of instructions g , such that the reliability efficiency is given as the product of error propagation index ξ and instruction vulnerability ϕ divided by the protection overhead ω . The overhead of the instruction group g is computed using Eq. 7; csi is the set of consecutive instructions and ci is the checking instruction inserted only at the end of g or at the point of multiple outputs.

$$RPF = (\xi(g) \times \phi(g)) / \omega_g; \quad s.t., \ \omega_g = \sum_{csi \in g} t_{csi} + \sum_{ci \in g} t_{ci} \quad (7)$$

Example: Fig. 5 shows an excerpt from an example instruction graph with 13 instructions and the effect of different parameters in the optimization goal on the efficiency of selective instruction protection under a tolerable overhead budget of 20 cycles. The table in Fig. 5 provides ϕ , ξ , and ω of each instruction I .

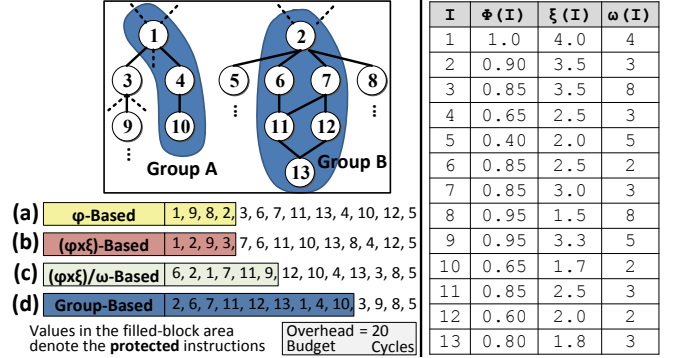


Fig. 5 An Example Showing the Effect of Different Parameters on the Reliability Efficiency of the Selective Instruction Protection

a) ϕ -based Selection: Fig. 5(a) illustrates that when considering only instruction vulnerability ϕ , only four instructions are chosen for protection. There may be cases, where an instruction’s vulnerability to error is quite high, but the probability that this error will be masked until the visible output is also high.

b) $(\phi \times \xi)$ -based Selection: Fig. 5(b) illustrates that when jointly accounting for ϕ and ξ , instruction 3 is chosen instead of instruction 8. However, still a total of 4 instructions are protected due to the high protection overhead of instruction 3 and 9. In such scenarios, it might be beneficial to choose several instructions with a slightly lower $(\phi \times \xi)$ profit and low protection overhead, rather than protecting only few instructions with high protection overhead. Note, depending upon the instruction types and protection scheme, the protection overhead may vary significantly for different instructions.

c) *Instruction Reliability Efficiency based Selection:* Fig. 5(c) shows that 6 instructions are protected, while the total reliability efficiency is 0.854 (computed using Eq. 7 and values of table in Fig. 5), which is 46% and 29% better compared to the case (a) having a reliability efficiency of 0.585 and case (b) having a reliability efficiency of 0.663, respectively. Note, the protection overhead depends upon the protection scheme. For instance, in case of simple software level error recovery like SWIFT-R [6], the voting instructions are inserted at the store instructions or leaf nodes of the groups. Therefore, protection overhead may be curtailed by computing the group reliability efficiency, i.e. cumulative reliability efficiency for a group of consecutive instructions.

d) *Group Reliability Efficiency based Selection:* Fig. 5(d) shows the marked regions in the graph as groups of protected instructions. Note, using group reliability efficiency, 9 instructions are protected, while the overall reliability efficiency is 0.966, which is 13% better compared to that of the case (c).

Constrained Protection Algorithm: Algorithm 3 shows the pseudo-code of our selective instruction protection heuristic. The *inputs* are: unprotected function F , user-provided tolerable performance overhead budget β_F in cycles, set of instruction vulnerabilities and error propagation indexes (ϕ , ξ), and a user-defined reliability method R . The *output* is the protected function F' . The *optimization goal* is to choose individual or groups of instructions while maximizing the total reliability profit function. Our scheme operates on the instruction graph G . First, the reliability profit function for each instruction is individually computed and inserted into a list that is then sorted in descending order (lines 1-3). The *while* loop iterates until the list is

empty or the budget β_F is exhausted. Since the generation of all groups of all instructions leads to a significant complexity, the heuristic starts with protecting individual instructions (lines 5-7) and incrementally builds groups of instructions for protection considering the predecessor and successors of the protected instructions (lines 8-17). For each predecessor and successor instruction of the protected instruction I , all possible instruction groups GI are generated (line 9) and their combined protection overhead is computed (lines 10-16). If an instruction i in the group g has more successors and a successor s do not belong to the group g , a *check* instruction is inserted at instruction i (lines 12-15). Afterwards, the reliability profit function is computed for all the groups (line 16). The list of instruction is re-sorted such that the instructions of group g_{Best} with the highest reliability profit RPF_{Best} (Eq. 7) appears first in the list (line 17), which is later evaluated for protection in the subsequent iteration of the loop (lines 6-7). For each protected instruction or group of instructions, the overhead budget β_F is updated accordingly (line 7).

Note, in this work, we assume that the control flow is protected using standard techniques like basic block signature [9]. This work employs SWIFT-R [6] as a protection scheme. However, our proposed models for error propagation and masking, and our scheme for constrained program reliability optimization is equally beneficial for selective applicability of other protection schemes and orthogonal to improvements in such program-level recovery schemes.

V. RESULTS AND EVALUATION

A. Experimental Setup and Tool Flow

The proposed error masking and propagation models along with our constrained program reliability optimization scheme are integrated in a GCC-based compiler. For evaluation, we have employed a reliability-featured ISS with an integrated fault injection engine. Numerous fault injection campaigns have been carried out at three different fault rates (1, 5, 10 faults/MCycles). Parameters to the fault injection module are particle strike rate (obtained from the flux calculator [23]), coordinates of the location, processor layout and frequency (in this work, a Leon-II processor [24] @ 100MHz is used), etc. An extensive analysis of errors, vulnerabilities, and reliability efficiency has been performed at the software program layer. Results of fault injection campaigns, vulnerability analysis, and program data/control flow are used to build and verify the models for error masking and propagation in MATLAB. Section S3 provides details on the tool flow, experimental setup, and fault injection configurations.

For evaluation, we use different applications: “ADPCM”, “SUSAN” from MiBench [25], and kernels from a complex “H.264 video encoder” [18], namely “SAD” for motion estimation and coding mode decision; and “DCT” for transformation. The selected applications vary significantly in terms of their instruction vulnerabilities, error masking index, and error propagation properties due to distinct control and data flow. All the applications are compiled with -O3 option.

B. Comparison to State-of-the-Art

We compare our constrained program reliability optimization scheme with state-of-the-art program-level protection schemes [12][11][6] (Fig. 7 b, c, d) and the baseline unprotected case (Fig. 7d) for varying number of tolerable performance overhead cases for different applications. Fig. 7 illustrates the reliability efficiency improvement of our scheme over state-of-the-art.

Fairness of Comparison: for fairness, we have provided the same fault scenarios, same compiler options, thus same application binaries, control and data flow graph, same input data, and same instruction protection method (i.e. instruction level TMR). The results solely represent the effect due to difference in the protection cost function and selection scheme.

Overall Comparison with all State-of-the-Art Schemes: First, we discuss experimental observations that are common in all comparisons. For “Susan” application, our scheme obtains a very high reliability

Algorithm 3: Pseudo-Code of Our Selective Instruction Protection

INPUT: Unprotected Function F from the software program as $G=(V, E)$, User-provided tolerable performance overhead in cycles β_F ; Set of instruction vulnerabilities ϕ , Set of error propagation indexes for all instructions ξ , User-provided program reliability method R (like SWIFT-R [6]), etc.

OUTPUT: F' – Function with selective instruction protection

BEGIN

```

1. List  $L$ ;
2.  $\forall I \in G$   $RPF(I) \leftarrow (\xi(I) \times \phi(I)) / \omega(I)$ ;
3. Sort( $L, RPF, DescendingOrder$ );
4. while( $L.Empty() || (\beta_F > 0)$ ) {
5.    $I \leftarrow L.Pull()$ ;
6.   if( $\omega(I) \leq \beta_F$ ) {
7.     Protect( $I$ );  $\beta_F \leftarrow \beta_F - \omega(I)$ ;
8.      $\forall i \in (I.S \cup I.P)$  {
9.        $GI = generateGroups(I, i)$ ; // groups of consecutive instructions
10.       $\forall g \in GI$  // compute overhead of instruction groups
11.         $g' \leftarrow g$ ;
12.         $\forall i \in g$  {
13.          if( $(|i.S| > 1) \& (\exists s \in i.SinGroup(s, g') = False))$  setCheckInstrPt( $i, g'$ );
14.          setCheckInstructionPt(Leaf( $g'$ ),  $g'$ ); }
15.         $\omega(g') \leftarrow getOverhead(g', R)$ ; // see Eq. 7
16.      }  $RPF(i) \leftarrow (\xi(i) \times \phi(i)) / \omega(i, g')$ ;
17.    } Sort( $L, RPF, DescendingOrder$ );
18.  }
19. } // end While loop of Budget finished or all instructions protected

```

END

improvement (avg. 30%-60%) over all comparison partners because “Susan” exhibits instructions with high error propagation index having varying distribution (see Fig. 6) and at the same time high IMI and high vulnerability. Additional results and discussion on instruction masking and error propagation is given in Supplementary Section S4. Due to the joint consideration of error propagation/masking probabilities and vulnerability in the cost function (Eq. 7), our scheme stays superior compared to all schemes and achieves a reliability efficiency improvement of up to 60%-99% (avg. 30%-60%). Fig. 7 shows that for 50% tolerable overhead, the reliability of “Susan” reaches close to 100%, because all the important non-masking instructions are protected within this budget, while masking instructions are left un-protected as errors during these instructions do not affect the correct program output. This illustrates the benefit of our scheme since it accounts for error masking and propagation probabilities in the protection cost function.

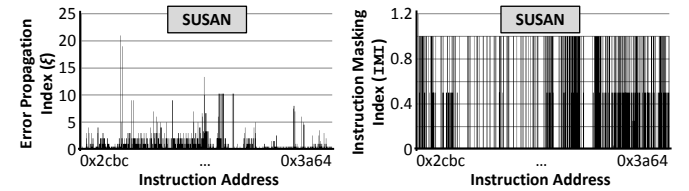


Fig. 6 Distribution of Error Masking and Propagation Indexes

The improvements are also high in case of “ADPCM” for overhead cases of 30% and higher (avg. 30%-40% reliability efficiency improvement). Below 20% the efficiency is low, as important instructions require more overhead than the tolerable overhead due to their high execution frequency. The improvements in “SAD” are also noticeable (up to 45% and avg. 10%-30% reliability efficiency improvement). However, the improvements for “DCT” are relatively low, due to limited masking probability of instructions and dependency on the earlier instructions of the algorithm (avg. 8%-10% reliability efficiency improvement). For low overhead cases, in several cases (like in “DCT”, “SAD”, and rarely in “ADPCM”), the savings of our scheme are below 10%. This is because of the fact that important instructions (that typically occur in loops) have many executions and their required protection overhead cannot be fulfilled under a cap of 5% or 10% tolerable overhead.

Now, we discuss specific observations for different comparison cases.

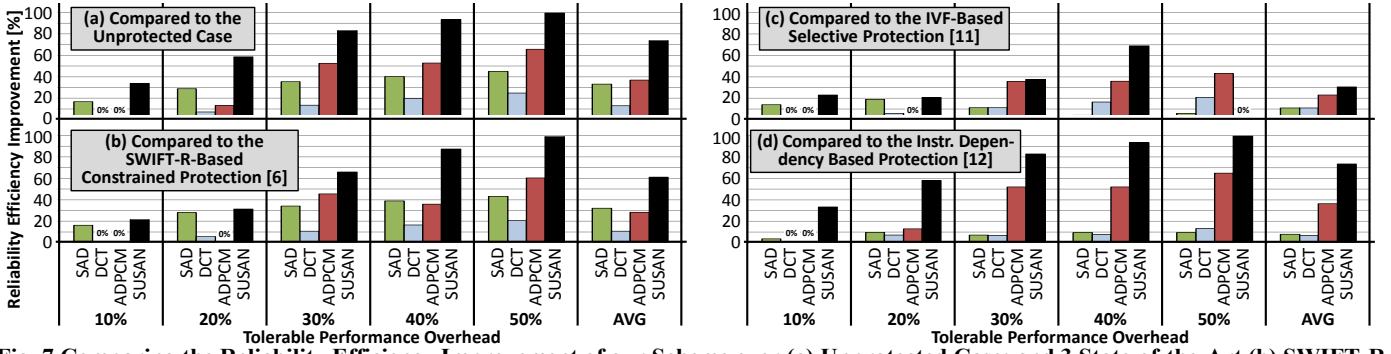


Fig. 7 Comparing the Reliability Efficiency Improvement of our Scheme over (a) Unprotected Case; and 3 State-of-the-Art (b) SWIFT-R [6] under Constraint; (c) IVF-Based Selective Protection [11]; and (d) Instruction Dependency based Selective Protection [12]

Comparing to the Unprotected Case: Comparison with the unprotected case shows the best possible reliability efficiency improvement of our scheme. Fig. 7a shows that our scheme provides up to 25%-99% and average 30%-70% improvement in the reliability efficiency compared to the unprotected case for different applications.

Comparing to the Scheme of SWIFT-R [6]: SWIFT-R [6] is the most prominent program-level instruction protection scheme that employs TMR with majority voting for protecting all instructions. Compared to original SWIFT-R (which is unconstrained to performance overhead), our scheme achieves $>3\times$ better reliability efficiency, since the overhead of SWIFT-R is $>5-6\times$. For fairness of comparison, we have modified the SWIFT-R implementation towards constrained optimization, such that the overhead constraint is used to determine the number of instructions that can be protected. Afterwards, the instructions are selected for protection in a sequential manner, i.e. first execute, first protect. Fig. 7b shows that our scheme provides up to 20%-97% and average 10%-60% improvement in the reliability efficiency compared to the constrained SWIFT-R variant.

Comparing to the IVF-based Selective Protection Scheme [11]: this scheme computes the instruction vulnerability and protects the instruction with the highest vulnerability factor first. However, this scheme does not account for the error propagation properties and instruction dependencies. Therefore, this scheme works well only for the cases where vulnerability is dominant and error propagation is very low with smooth distribution. In contrast, our scheme accounts for both vulnerability and error propagation. As a result, our scheme provides up to 20%-70% (avg. 10%-30%) improvement in reliability efficiency, compared to the scheme of [11].

Comparing to the Instruction Dependency based Selective Protection Scheme [12]: this scheme prioritizes and protects the instructions with the highest number of dependent instructions. However, this scheme ignores the error masking probabilities (which is an important parameter to be considered in the error propagation) and instruction vulnerabilities. As a result, in some cases (like "Susan") the scheme of [12] provides significantly less protection. However, in cases where error propagation is crucial and dominant over vulnerability (i.e. "DCT"), this scheme provides good reliability. In contrast, our scheme provides high reliability efficiency in all cases, as it jointly accounts for error propagation and masking probabilities, vulnerabilities, and overhead of different instructions individually or jointly in a group. Our scheme thereby achieves up to 12%-99% (avg. 7.5%-80%) improved reliability efficiency compared to the scheme of [12], see Fig. 7d.

VI. CONCLUSIONS

We presented a constrained program reliability optimization scheme that exploits the program level error masking and propagation along with instruction vulnerabilities to selectively protect critical parts of the program for a given tolerable performance overhead. Our scheme is integrated in a reliability-driven compiler and exploits the data and control flow properties to estimate the error masking and propagation indexes. Our scheme provides significant improvement in reliability

efficiency (avg. 30%-60%) compared to state-of-the-art protection schemes. Due to the novel enhancements in Section III and IV, state-of-the-art selective program protection techniques cannot reach the level of program reliability efficiency of our scheme. Our proposed models enable a whole new range of program-level or program-guided hardware-level constrained reliability optimizations.

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] J. Henkel et al., "Reliable On-Chip Systems in the Nano-Era: Lessons Learnt and Future Trends", IEEE DAC, 2013.
- [2] 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.
- [3] 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.
- [4] R. Vadlamani et al., "Multicore soft error rate stabilization using adaptive dual modular redundancy", IEEE DATE, pp. 27-32, 2010.
- [5] M. D. Powell, A. Biswas, S. Gupta, S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance", International Symposium on Computer architecture (ISCA), pp. 93-104, 2009.
- [6] G. A. Reis, J. Chang, D. I. August, "Automatic instruction-level software only recovery", IEEE MICRO, pp. 36-47, 2007.
- [7] N. Oh et al., "Error detection by duplicated instructions in super-scalar processors", IEEE Transaction on Reliability, 51-1, pp. 63-75, 2002.
- [8] J. Hu et al., "In-Register Duplication: Exploiting Narrow-Width Value for Improving Register File Reliability," DSN, pp. 281-290, 2006.
- [9] E. Borin, C. Wang, Y. Wu, G. Araujo, "Software-Based Transparent and Comprehensive Control-Flow Error Detection", CGO, pp. 333-345, 2006.
- [10] S. Rehman et al., "Reliable software for unreliable hardware: Embedded code generation aiming at reliability", Codess+ISSS, pp. 237-246, 2011.
- [11] D. Borodin et al., "Protected Redundancy Overhead Reduction Using Instruction Vulnerability Factor," IEEE CF, pp. 319-326, 2010.
- [12] J. Cong, K. Gururaj, "Assuring Application-Level Correctness Against Soft Errors", ICCAD, pp. 150-157, 2011.
- [13] A. Sundaram et al., "Efficient fault tolerance in multi-media applications through selective instruction replication", WREFT, pp. 339-346, 2008.
- [14] M. Shafique et al., "Power-Efficient Error-Resiliency for H.264/AVC Context-Adaptive Variable Length Coding", DATE, pp. 697-702, 2012.
- [15] M. A. Makhzan et al., "A low power JPEG2000 encoder with iterative and fault tolerant error concealment", IEEE TVLSI, vol. 17, no. 6, pp. 827-837, 2009.
- [16] S. Rehman, M. Shafique, J. Henkel, "Instruction Scheduling for Reliability-Aware Compilation", IEEE DAC, pp. 1288-1296, 2012.
- [17] J. Lee et al., "Dynamic Code Duplication with Vulnerability Awareness for Soft Error Detection on VLIW Architectures", ACM TACO, vol. 9, no. 4, article 48, 2013.
- [18] M. Shafique, L. Bauer, J. Henkel, "Optimizing the H.264/AVC Video Encoder Application Structure for Reconfigurable and Application-Specific Platforms", JSPS, vol. 60, no. 2, pp. 183-210, 2010.
- [19] T. Li et al., "CSER: HW/SW Configurable Soft-Error Resiliency for Application Specific Instruction-Set Processors", IEEE DATE, 2013.
- [20] A. Rajendiran et al., "Reliable computing with ultra-reduced instruction set co-processors", IEEE DAC, pp. 697-702, 2012.
- [21] S. Rehman et al., "Leveraging Variable Function Resilience for Selective Software Reliability on Unreliable Hardware", IEEE DATE, 2013.

ADDITIONAL REFERENCES

- [22] S. Z. Shazli, M. B. Tahoori, "Obtaining Microprocessor Vulnerability Factor Using Formal Methods", DFTVS, 2008.
- [23] Flux calculator: www.seutest.com/cgi-bin/FluxCalculator.cgi.
- [24] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC v8 architecture", DSN, pp. 409-415, 2002.
- [25] MiBench (<http://www.eecs.umich.edu/mibench/>).
- [26] IBM® XIV®: <http://publib.boulder.ibm.com/infocenter/ibmxiv/t2/index.jsp>.
- [27] AMD Phenom™ II Processor Product Data Sheet 2010.
- [28] S. Ananthanarayan, S. Garg, H. D. Patel, "Low Cost Permanent Fault Detection Using Ultra-Reduced Instruction Set Co-Processors", IEEE DATE, 2013.
- [29] S. Rehman et al., "RAISE: Reliability Aware Instruction Scheduling for Unreliable Hardware", IEEE ASP-DAC, pp.671-676, 2012.
- [30] T. Li et al., "RASTER: Runtime Adaptive Spatial/Temporal Error Resiliency for Embedded Processors", IEEE DAC, 2013.

SUPPLEMENTARY MATERIAL

S1. Details on Modeling Program-Level Error Masking

A. Example –Error Masking in Pipeline Components

Fig. 8 illustrates an example of an instruction executing through different pipeline stages/components (PC) and corresponding masking probabilities $P_{EM}(I, PC)$, s.t., $PC = \{F, D, E, M, W\}$. An example of microarchitecture-level logical masking for an adder circuit (which is a part of the ALU) is shown in Fig. 8, where different error cases are denoted by red dots. It is shown in Fig. 8 (case-2) that an error at the "AND" gate is blocked by the subsequent "OR" gate. However, the error at the "XOR" gate is propagated to the output (Fig. 8, case-1). Note that only "OR" and "AND" gates may mask the error with a theoretical probability of 0.5, while any "XOR" gate does not mask an error. Moreover, the logical masking properties of a pipeline component depend upon its microarchitecture. Therefore, the logical masking properties of a carry-lookahead adder are different from a ripple-carry adder.

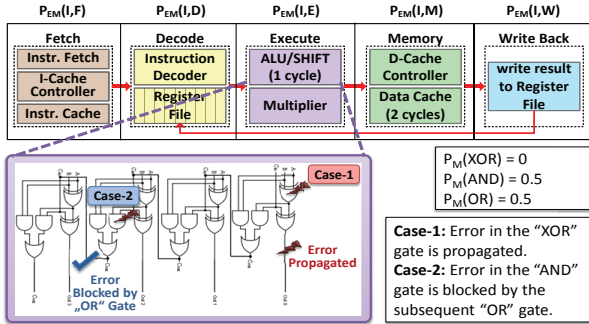


Fig. 8 Different Pipeline Stages Exhibit Distinct Masking During the Instruction Execution due to Combinatorial Logic

B. Parameter Estimation of Masking Probability due to Microarchitectural Effects ($P_{EM}(I)$)

The masking probability $P_{EM}(I)$ during the execution of an instruction I can be estimated as the cumulative probability of masking in different pipeline components PC it uses; see Eq. 8. Each pipeline component c can be seen as a set of connected logic elements LE (logic gates, latches, etc.) each having a certain masking probability $P_{EM}(c, l, t)$ and error probability $P_e(c, l, t)$. T is the total time spent in the pipeline in terms of cycles.

$$P_{EM}(I) = \sum_{c \in PC, \forall l \in c, \forall t \in T} (P_{EM}(c, l, t) \times P_e(c, l, t)) \quad (8)$$

We can simplify Eq. 8 by assuming similar error probability for each logic element of a given pipeline stage; see Eq. 9.

$$P_{EM}(I) = \sum_{c \in PC, \forall t \in T} (P_{EM}(c, t) / (N_{LE}(c) \times t(c))) \quad (9)$$

$N_{LE}(c)$ is the number of logic elements in each pipeline component c obtained from the hardware synthesis; $t(c)$ is the time spent in each pipeline component, and $P_{EM}(c, t)$ is the logical masking of each component. Several methods have been researched by the hardware-level community to estimate $P_{EM}(c, t)$, ranging from analytical/statistical approaches (like EPP [22]) to fault injection. In this paper, we employ a fault injection based technique to estimate $P_{EM}(c, t)$. See Section S3 for the details of experimental and fault injection setup.

C. Example –Effect of Joint Masking Probabilities of Consecutive Instructions

Fig. 9 illustrates an example showing the computation of error masking probabilities of different instruction in a given instruction graph, while showing the effect of consecutive instructions of type B on the total masking probability. It is shown in Fig. 9 that for instruction "1", the total masking probability is equal to "0.803" when considering the masking effects of consecutive instructions of type B (i.e. instructions "1" and "2"). If the masking effects of consecutive instructions are ignored, the total masking probability of instruction "1" is equal to "0.770", i.e. a difference of "0.033" compared to the earlier case. This shows that ignoring the effects of consecutive instructions may lead to an under-estimation of masking probabilities. Therefore, accurate estimation of instruction-level error masking needs to account for the joint error masking effects due to the combination of consecutive successor instructions of type B in the path p .

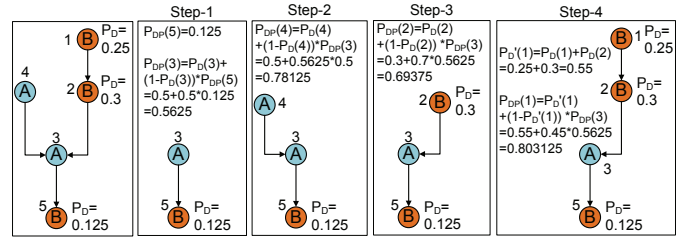


Fig. 9 An Example showing the Computation of Error Masking Probabilities illustrating the Effect of Consecutive Instructions of type B in the Path on the Total Masking Probability

D. Table of Model Parameters, Variables, and Notations

Table I provides the description of various model parameters, variables, and notations used in this paper. The table provides a summary, which is helpful in cross-referring from different algorithms and equations, in order to increase the readability and ease-of-understanding of the reader.

Table I: Description of Different Model Parameters and Variables

Parameter	Description
F	Program Function
I	Instruction
IMI (I)	Instruction masking index: The probability that an error in instruction I will not be seen at the output (any output)
p	Path of dependent instructions, such that each instruction has exactly one successor and one predecessor. Note that an instruction may appear in different paths, as it has several successors.
$P_{DP}(I, p)$	Error masking probability due to data flow properties for an instruction I in path p .
$P_{CDM}(I)$	Error masking probability for instruction I due to control and data flow properties. For instance, if an instruction is executed only in half of the executions, then its errors will be masked with a probability of 0.5.
$P_{EM}(I)$	Probability of masking in pipeline due to microarchitectural logical masking. An error appearing in an instruction may be masked due to the logic gates during the instruction execution.
$P_D(I)$	Probability of instruction I masking an error on the input due to data flow (operation type and operands).

O	Operands of an instruction
Bits	Bits of the operands of an instruction
$P_D(x, I)$	Probability that instruction I will mask an error in bit x
$P_e(x)$	Probability of an error in bit x
N_{Bits}	Number of operand bits
th	Threshold for tolerable output error, i.e. acceptable output value in presence of faults. Even if it is not exactly the correct output result, in some applications we may still have an acceptable result. The threshold is the accepted deviation from the correct result.
G	Graph of a program function given as a set of vertices and edges, where vertex denotes an instruction and edge denotes the dependency
L	List of nodes for which $P_{DP}(I, p)$ is computed
L_G	Set of dependencies from a node to the leaf (outputs)
N_B	Consecutive masking instructions of type B
$P_{DP}(s, p)$	Look $P_{DP}(I, p)$
$P_{CF}(ep I)$	Probability of execution path ep if the instruction I is executed.
$\xi(I)$	Error propagation index of instruction I . It is a measure of how many outputs are likely to be affected by an error at I .
C	List of nodes for which $\xi(I)$ is computed
$I.S$	Successors of instruction I
s	A particular successor instruction
$I.P$	Predecessors of instruction I
$P_{Execution}(s I)$	Probability of executing s if we know that I is executed
$s.Paths$	Dependency paths from s to leaf nodes
$P_{CF}(p I)$	Probability of executing the a set of instructions that contains path p knowing that instruction I is executed
$P_{Cr}(I)$	Probability that an error in instruction I will generate a crash
$P_{IO}(I)$	Probability that an error in instruction I will generate an incorrect output
COB(I)	Number of critical opcode bits in I . Critical opcode bits means those bits that if wrong will generate a crash
$P_{eOP}(b, I)$	Probability of error in bit b of opcode of instruction I
CAB(I)	Number of critical address bits. An error in one of those will generate a crash when the address is accessed
$P_{eAd}(b, I)$	Probability of error in bit b of address used in instruction I
$\phi(I)$	Instruction Vulnerability Index for instruction I
$\omega(I)$	Overhead needed to protect instruction I
ω	Overhead to protect the whole function
P_{Best}	Group of instructions with the highest reliability profit function value
RPF_{Best}	The reliability profit function value corresponding to the best selected group of instructions
csi	Consecutive successor instructions
ci	Checking instructions added at the end of a set of protected instructions to verify the results
β_F	Tolerable performance overhead for protecting a function F
F'	Protected Function
R	Protection Method

S2. Overview of Instruction Vulnerability Estimation [10]

The instruction vulnerability index (ϕ) is a program-level reliability model that quantifies the program reliability at instruction [10]. The instruction vulnerability index is bridges the gap between hardware and software by quantifying the effects of hardware-level faults at the software program level. It defines the vulnerability of an instruction as the accumulated sum of vulnerabilities when an instruction executes in different pipeline components $PC = \{F, D, E, M, W\}$ each having a logic area A_c and probability of fault $P_E(C)$. The instruction vulnerability in a pipeline stage can be quantified as the product of *spatial*

vulnerability (i.e. vulnerable bits/logic area of each pipeline stage during the instruction execution) and *temporal vulnerability* (i.e. vulnerable residency period in different pipeline stages); see Eq. 10.

$$\phi(I) = \sum_{C \in PC} (\phi(C, I) \times A_C \times P_E(C)) / \sum_{C \in PC} A_C \quad (10)$$

The instruction vulnerability index is estimated by executing the application binary on a given processor architecture for a given input data and analyzing the vulnerable periods and bits in different pipeline stages. Note, that the vulnerable bits and period vary significantly depending upon the data and control flow properties. This fact is evident from the instruction vulnerability distributed shown in Fig. 1(a) in Section I.A.

S3. Details on the Experimental Setup and Fault Injection

Fig. 10 shows the tool flow and experimental setup. The masking and error propagation models are developed in MATLAB and the input for parameter evaluation is provided from the program reliability analysis which is performed on a reliability-featured processor simulator. The input to the modeling is the program error distribution, data and control flow graph, and application program execution traces. The developed models are then implemented in our reliability-driven compiler for estimation of error masking and propagation probabilities. These models are also integrated the reliability analysis module of the processor simulator for evaluation of error masking and propagation properties of application programs.

The program is compiled on a GCC-based compiler extended with program reliability features. In this paper, we have proposed error masking and propagation models along with a constrained program reliability protection scheme. These models and protection scheme are integrated in the compiler. The error masking and propagation operate on the data and control flow graph. The instructions are protected using TMR [6]. We have also implemented and integrated various state-of-the-art protection techniques for comparison purposes.

The compiled program is simulated and evaluated for reliability on a reliability-featured processor simulator. Our simulator features a basic processor instruction set simulator (a Leon-II based on SPARC-V8 architecture is simulated), an integrated fault generation and inject module, and an enhanced program reliability analysis module.

Fault Generation and Injection Module: the fault generation module is used to generate various fault scenarios, such that same scenarios can be provided to all the comparison partners and the experiments can be repeatable. Numerous fault scenarios are generated considering the following fault parameters.

- 1) *Particle Flux Rate:* The particle flux rate is determined by the neutron flux calculator [23], which takes city coordinates and altitude (at which the device is operating) as an input and provides the flux rate in particle strikes per mm^2 per sec. An example input is 59° 55' N, 10° 45' O, altitude ranging from 1- 20km (covering various use cases from terrestrial to the aerial).
- 2) *Processor Layout/Area:* This parameter specifies the size of the complete target device. The value of this parameter is typically in mm^2 size or logic gates and technology parameters from where the absolute area in mm^2 can be obtained, which is important to determine the surface area exposed to particle strikes.
- 3) *Vulnerable/Unprotected Fault Location:* A list of processor components is specified, where the faults are injected. This list has option to specify protected or unprotected components, which is important to accelerate the fault injection simulations by simply skipping the fault injection in the protected parts. An example list of processor resources is: register file, program counter, instruction word, instruction memory, data memory, etc.
- 4) *Fault Probability:* This parameter specifies the probability that a neutron strike on a processor component results in a architectural-visible error.

- 5) *Fault Distribution*: The fault distribution model describes how faults in different processor components will be injected, for instance randomly or following a certain correlation model.
- 6) *Fault Model*: This parameter specifies the distribution of faults in terms of number of even upsets, i.e. single even upsets or multiple even upsets. This parameter may also partly cover fault sources other than soft errors. In this
- 7) *Processor Frequency*: This parameter specifies the operating frequency of the processor and it is used to convert the fault rate from per-sec unit to the per-MCycles unit.

A fault rate (in faults/MCycles) is computed using the neutron flux, city coordinates, fault probability, processor area/layout, and the processor frequency. In our experiments, we used various a Leon-II embedded processor from Gaisler [24] and obtained three different faults rates (1, 5, 10 faults/MCycles) for different altitude levels. In this paper, we employ single bit flip faults. In this work, we assume that caches are protected following the common practice in the prominent industrial and research projects by AMD [27] and IBM [26].

After generating various fault scenarios, the integrated configurable fault injection module injects the faults during the application execution on the processor instruction set simulator. For instance, in case of the instruction decoder or instruction word, corrupted operation or wrong source/destination addresses of processor is modeled by making one/multiple fields of the instruction word corrupted, consequently leading to a wrong opcode or wrong operand. The program output is categorized in “Correct Output”, “Incorrect Output”, and “Program Failure”, where the program failures are further characterized depending upon the type of fault, like memory segmentation fault, non-decodable instruction, etc. For each simulation run, the reliability analysis is performed, where instruction vulnerabilities, masked and propagated errors are monitored and analyzed. Note that the fault injection does not contribute in an undesirable consequence such as changing performance counters. The fault injection operates in parallel to the application execution while preserving the processor state in backup data structures.

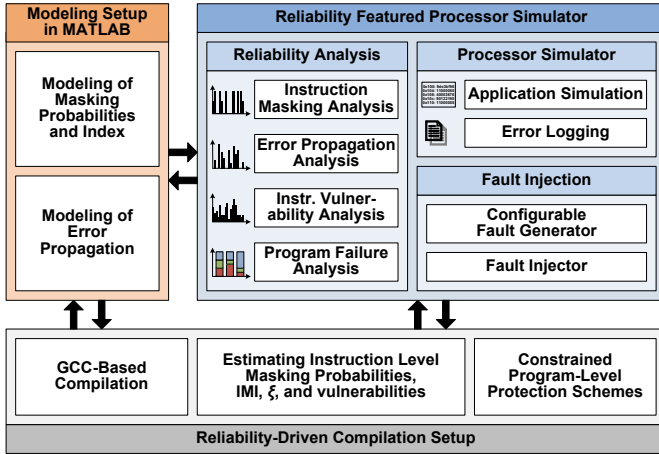


Fig. 10 Our Experimental Setup with Reliability-Featured Processor Simulator, Fault Injection Engine, Reliability-Aware Compiler, and Reliability Analysis Flow

S4. Additional Results on Program-Level Masking and Error Propagation

Fig. 11 shows the distribution of error propagation index (ξ) and instruction masking index (IMI) for 3 additional applications “ADPCM”, “DCT”, and “SAD” (see similar plots for “Susan” in Section V.B). the distribution illustrates summary from a selected execution run out of many. Horizontal axis shows the instruction address in the execution sequence.

Fig. 11 shows that in “ADPCM” and “DCT”, several instructions have zero value for ξ . This denotes that these instructions do not have any dependent instructions or the dependent instructions mask the errors completely. Such masking dependent instructions can be identified by comparing the plots of ξ and IMI corresponding to the same instruction address. These instructions are relatively less important for protection compared to those instructions which exhibit high error propagation index ξ .

In “ADPCM” and “SAD”, the ξ value is much lower compared to the ξ plot of “DCT”. Therefore, in cases of “ADPCM” and “SAD”, the protection is dominated by the instruction vulnerability index. In case of “DCT”, the ξ value for many instructions is low, and the number decreases exponentially as the value grows the same way. That is due to the butterfly form of the instruction dependencies. This illustrates that it is important to protect earlier executing instructions compared to the later ones. In case of “SAD”, the ξ value is low and the distribution is homogeneous. This is due to the fact that there are many parallel instruction paths with similar dependency structure in the data flow graph. Note that in case of “SAD” the loops are completely unrolled, therefore, the ξ plot is very dense. In case of “SAD”, the protection decision is primarily dominated by the value of the instruction vulnerability index.

Fig. 11 also shows the plots for IMI. It is noteworthy that most of the IMI values are close to 0, 0.5, and 1. This primarily reflects three important cases:

- 1) **Case-1**: an instruction is either not important (IMI=1, EPI=0) and therefore its output will not matter;
- 2) **Case-2**: IMI value close to 0.5 indicates the cases of comparison or logical instructions, where the error probability to become visible is controlled by the 50% probability of the operation type.
- 3) **Case-3**: IMI value close to 1 indicates the cases where a dependency path directly leads to an error without any intermediate masking, e.g., a sequence of arithmetic instructions leading to an output value as in case of “SAD”.

Note that the case-2 is dominant in “SAD” due to comparison operations of the absolute operation. However, case-1 is dominant in “DCT” due to dependent arithmetic instructions.

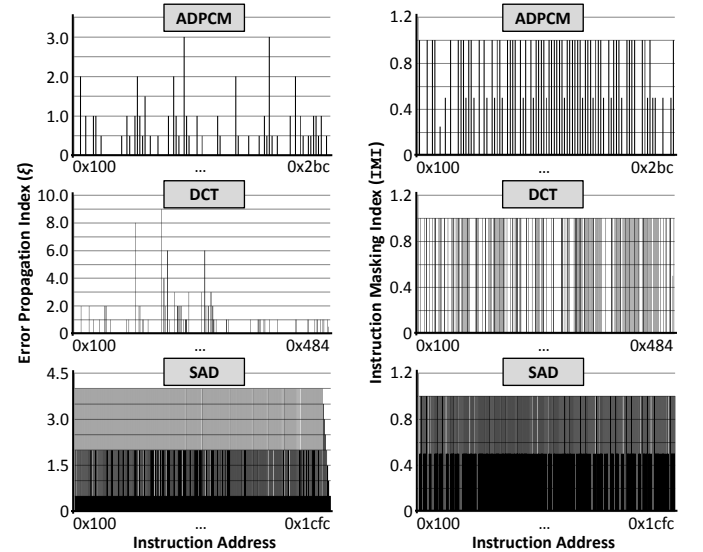


Fig. 11 Distribution of Error Masking and Propagation Indexes