SHARC: an Efficient Metric for Selective Protection of Software against Soft Errors

J. Isaza-González[a], F. Restrepo-Calle[b], A. Martínez-Álvarez[a], S. Cuenca-Asensi[a,*]

[a]*Dept. of Computer Technology, University of Alicante, Alicante, Spain*
[b]*Dept. of Systems and Industrial Engineering, Universidad Nacional de Colombia, Bogotá, Colombia*

Abstract

This paper presents a metric for the efficient application of selective hardening using software based techniques against *soft-errors*. It offers a method for selecting the resources to be protected obtaining maximum fault coverage with the minimum overhead. Common approaches are based on exhaustive exploration of the solution space or time-consuming fault injection campaigns. Contrarily, our Software based HARdening Criticality metric (SHARC) relies on early estimations of the impact that protection techniques will have on the global reliability of the application. SHARC estimations are based on features extracted from the dynamic analysis of source code, and produce a prioritization of the resources involved accordingly. For assessing our approach two case studies were carried out using low-cost embedded microprocessors. Results were compared to traditional approaches like brute-force exploration and the Architectural Vulnerability Factor (AVF) metric. Experiments show that SHARC improves the results between 5% and 21% at a fraction of the effort.

1. Introduction

The miniaturization of electronic components has significantly increased the susceptibility of processor-based systems to radiation effects, increasingly affecting reliability. For these systems, the effect of radiation could cause Single Event Effects (SEEs), which provoke execution faults known as *soft-errors*. These are transient faults that affect the system behavior by altering temporarily signal transfers (*Single Event Transients* - SETs) or stored values (*Single Event Upsets* - SEUs) [1].

The protection against soft-errors, also known as hardening, has traditionally relied on hardware redundancy. Although these techniques offer excellent results, they produce expensive and power costly solutions. Moreover, fine-grain hardware replication does not fit well on Commercial Off-The-Shelf (COTS) processors, which are becoming a key component in industry for producing low-cost reliable systems [2].

In recent years, a large number of techniques based on redundant software, also known as *Software Implemented Hardware Fault Tolerance* (SIHFT) techniques [3], have been proposed as a cost-effective alternative to traditional approaches based on hardware. Although SIHFT can be easily applied to COTS processors, they cause non-negligible overheads in terms of code size, execution time, and data storage [4]. These overheads present a challenge for designing fault tolerant systems and, in any case, they impose new limitations to their reliability. In fact, performance degradation caused by the execution of redundant instructions may prevent their application to systems with severe time constraints. Furthermore, the time span that vulnerable resources are exposed to radiation increases, making the system more prone to faults. In the same way, the augmented memory footprint increases the area susceptible to faults and, in some cases, may exceed the resources' capacity.

Recent approaches propose the partial or selective application of hardening techniques to overcome those drawbacks [5]. These are also known as selective-SIHFT in the case of the software-based ones [6, 7]. The general approach consists of protecting only a selected subset of elements (e.g., processor registers, basic blocks of instructions, among others) to achieve significant reductions in overheads with minimal impact on the system's reliability. Theoretically, it is possible to find an optimum trade-off between reliability, cost, and performance from the hardening design space. However, identifying the more critical resources and selecting the order that they should be protected is not a trivial task.

In first place, when a SIHFT technique is applied to a specific resource, and contrarily to the effect of the

*Corresponding author

equivalent hardware protection, the vulnerability of the rest of elements can be negatively impacted. Consequently, common hardware-based metrics of vulnerability, like Architectural Vulnerability Factor (AVF) [8] [9], are not always suitable for selective-SIHFT techniques. In second place, software offers designers a huge number of possibilities when exploring the hardening design space. In terms of flexibility this could be great, but in practice assessing the reliability of different solutions is a time-consuming process that makes the exploration both costly and cumbersome. In fact, just a few brute-force approaches have been proposed in that direction. For instance, in [6] reliability was estimated for every protected version of the code. Some refinements have been proposed to reduce the exploration area, like in [10] where an evolutionary algorithm was used for guiding the exploration of a dense solution space in the search of the optimal trade-offs. However, in spite of the reduction of the number of candidate solutions, it still supposes a considerable cost and effort, since agile fault injection tools and processor models are not always available for implementing these kinds of analyses.

In this context, the present work is based on [11] and [12], extending the initial efforts in two directions. Firstly, a lightweight metric named SHARC (Software based HARdening Criticality) is proposed to early assess the impact that the protection of each individual resource will produce on the overall system reliability. Secondly, a refined strategy is presented to improve the mitigation of soft errors keeping overheads as low as possible. It relies on an incremental and iterative process where successive software versions are re-evaluated and resources are hardened following a strict order given by SHARC. In this way, maximum fault coverage increase is ensured for each hardening step.

The rest of the paper is organized as follows. Section 2 defines the SHARC metric and proposes a method for the selective hardening of software based on it. Section 3 reports and discusses the experimental results obtained using the method. It also includes a brute-force hardening strategy and the comparative analysis between AVF and SHARC metrics for selective SIHFT techniques. Finally, Section 4 describes the conclusions of the work.

2. SHARC: metric for Selective Hardening based on Software

SHARC is aimed to obtain an assessment about the suitability of an individual resource to be protected using a specific SIHFT technique. As software hardening techniques can be applied to both high-level code (e.g., C/C++) and low-level code (e.g., assembly), in each case different kinds of resources might be considered: registers, variables, arrays, etc. Moreover, every technique produces a different effect to the overall reliability and introduces new potential vulnerabilities in the program. Therefore, to improve the accuracy of the estimations, SHARC is tightly coupled to the SIHFT under consideration. In this work, the SHARC metric is adapted to an assembly-level hardening technique and is applied for assessing the processor's register file, however, it could be employed for any kind of SIHFT and resources.

In this particular context, SHARC tries to identify the register that, once hardened, will produce the higher impact in the overall reliability of the application. For this purpose, two factors are taken into account: the criticality of the resources, and the vulnerability overhead. Both factors are formally expressed by the corresponding terms in the following definition:

$$SHARC(R_x) = \frac{\sum_{i \neq x} ABC_i}{\sum_i ABC_i} \cdot W_{ABC} + \frac{CI_{Rh_x}}{TI_{Rh_x}} \cdot W_{CI} \quad (1)$$

As defined in equation 1, the first term is weighted by coefficient W_{ABC} and corresponds to the contribution of the unprotected resources to the global criticality. Assuming that hardening of register x will have a negligible effort on the rest of resources, it is calculated as the summation of the individual criticalities and normalized by the original overall criticality (i.e., including register x). The individual criticality is proportional to the probability that a fault, affecting the resource, becomes an error during the code execution. The ABC metric was used for estimating this criticality in advance by means of dynamic analysis of the program (assembly code). It is calculated as a weighted sum of three parameters: the effective time span when useful data is present in a resource (i.e. effective lifetime), the number of times that a resource is involved in the evaluation of a branch condition (i.e. weight in conditional branches) and the number of times that a resource contains the result of an expression involving the value stored on another resource (i.e. functional dependencies) [11].

The second term, weighted by the coefficient W_{CI}, describes the vulnerability overhead. Unlike hardware replication, software redundancy does not guarantee a full protection of the data. Depending on the type of instructions employed by the protection technique and where they are inserted in the original code, new weaknesses may appear in the hardening code. Since SIHFT techniques are well defined by a set of code transformation rules, these vulnerabilities can also be estimated in advance during the analysis of the original code. They are expressed in the equation as the number of *Critical Instructions* induced by the hardening of register x

Table 1: SWIFT-R transformation example.

Original code	Reg. R4 protected	Additional instructions	Critical instructions
...	...		
MOV #8, R4	MOV #8, R4		
MOV #15, R5	MOV #8, R8	*	0
loop:	MOV #8, R9	*	0
CMP #0, R5	MOV #15, R5		
JZ loop_done	loop:		
ADD R4, R6	CMP #0, R5		
JMP loop	JZ loop_done		
...	CMP R4, R8	*	1
	JZ voter_0	*	1
	MOV R9, R6	*	0
	voter_0:	*	0
	ADD R4, R6		
	DEC R5		
	JMP loop		
	...		

(CI_{Rhx}). CI_{Rhx} takes into account that the application can pass through the same vulnerable state several times during the execution, and it is normalized by the total number of instructions executed (TI_{Rhx}).

For illustrative purposes, Table 1 shows an example of code protected with SWIFT-R technique (*SoftWare Implemented Fault Tolerance - Recovery*) [13]. The transformed code can be seen on the right, where injected instructions are marked by * in the second column. SWIFT-R triplicates the register *R*4, creating two identical copies (*R*8 and *R*9 registers), with one majority voter ensuring correctness provided at least two out of three copies of the register remain the same (*voter*_0).

However, the insertion of verification instructions adds weak points to the program, classified as critical instructions and marked with a 1 in the third column. This is due to the error propagation resulting from a fault in the register *R*4 after executing the first or second verification instruction inserted in the majority voter. The content of register *R*4 changes, so the addition between registers *R*4 and *R*6 will output an incorrect value.

2.1. Method for selective hardening based on SHARC

Usually, selective methods are guided by the order defined in the criticality rank, obviating the fact that protections have an effect on the reliability of the unprotected resources. Alternatively, we propose to recalculate the criticality rank at each successive code version. As shown in the next section, our method renders more accurate criticality estimations and guarantees the maximum fault coverage for each incremental protection step.

The procedure shown at Algorithm 1 has the original code (un-hardened code) as input, jointly with the SIHFT technique and the number of protection steps *N*. *SIHFT()* is defined by a set of code transformations that can be automatically applied to any version of the code. The *N* parameter specifies the number of protection steps to reach the final protected version of the program. It is defined as the maximum number of registers used by the application but also can be associated to the maximum level of overhead allowed by the application. This way, our method can be driven by the application reliability or by memory/time constraints.

Algorithm 1 Selective SIHFT based on SHARC

```
1: function SelectiveHard(SIHFT, code, N)
2:     version[0] ←code                                  ▷ Step 1
3:     for i ←1 to N do
4:         R_C ← SHARCrank(version[i − 1])               ▷ Step 2
5:         version[i] ← SIHFT(version[i − 1], R_C)
6:     end for
7: return version[i]
```

The registers to be hardened are successively selected using the *SHARCrank()* procedure for each version[*i*]. In the first step, version[0] corresponds to the *non-hardened* code and is used to select the most critical register, i.e., the best candidate to be hardened (R_C). This register is classified as the first in the criticality rank, and then, it is protected using the specific SIHFT technique to produce the next version of the code. In the second step, the version[*i*] is used to estimate the criticality of the remaining registers. As a result, we obtain the highest criticality register of the version[*i*], which is classified according to the order of criticality, and it is hardened in the following version[$i + 1$]. This loop is repeated until the total number of steps is completed (*N*). As a final result, we obtain the coverage percentages against faults of the different versions, when evaluating the selective hardening with 1, 2, 3, up to *N* hardened registers.

3. Experimental Results and Discussion

To validate the feasibility of SHARC, two target processors and one SIHFT technique were selected. The first, Xilinx PicoBlaze [14], is a widely used IP (Intellectual Property) core for embedded systems based on FPGAs. PicoBlaze processor includes 16 byte-wide general-purpose data registers (denoted as sX). The second, TI-MSP430, is a 16-bit RISC processor used in the Texas Instruments MSP430 low-power microcontroller family. It includes a register file with 16 registers (r0-r15). The first four registers are intended for special purposes and the remaining r4-r15 are general purpose registers.

The selective version of SWIFT-R was used to carry out the selective protection [6]. This detection and recovery technique was automatically applied by means of low-level instruction transformation rules to different subsets of processor register file. Code and performance overheads produced vary significantly with the selected resources, being greater than 2.5× when all registers are protected (fully protected version).

The benchmark software suite used in the experiments covers from basic to complex applications of different nature (data intensive vs control intensive), they are: proportional-integral-derivative controller (PID), matrix multiplication (MM), exponentiation (POW), bubble sort (BUB), matrix addition (MADD), fibonacci (FIB).

3.1. Brute-force strategy for software hardening

Following the brute-force strategy, the hardened versions of a test program were evaluated by means of fault injection campaigns. For every version, 10,000 faults per register were injected in the register file. Each fault consisted of a bit-flip, one per run, randomly selecting one clock cycle and one bit of the target resource. The different versions of the programs were automatically generated using the *Software Hardening Environment* (SHE) [15], and their reliability measured on processor ISA simulators. Naken open source simulator [16] was modified to support fault injection on MSP430 processor. Similarly SimPicoBlaze, included in the SHE toolchain, was used for PicoBlaze campaigns.

The fault effects were classified as *unACE - unnecessary for Architecturally Correct Execution* in case the system completes its execution, and obtains the expected output after a fault is injected. Otherwise they were classify as *ACE - Architecturally Correct Execution*, which comprises any undesirable effect categories such as uncorrected faults, abnormal program termination or infinite execution loop [8].

Tables 2 and 3 present the fault coverage for each version of the code in terms of unACE percentage. Hardened versions are grouped according to the number of protected registers. The horizontal divisions separate each group, so that the different versions evaluated can be observed using one, two, three, and up to four registers in the case of the TI-MSP430 processor, and five registers in PicoBlaze (except for the FIB case where only four registers were needed for coding the application). The registers used in the different hardened versions are labeled using a comma-separated list, for example, the version "1, 2" of an application means that the register 1 and 2 are hardened. It is worth to note that the protection technique applied in these experiments not only provides detection, but also correction of the faults. It involves the usage of two copies of data, limiting the number of registers that can be protected. Other SIHFT techniques, which only include detection, do not suffer of this limitation and would produce even more hardened versions of the code.

Table 2: Fault coverage on PicoBlaze (unACE%)

Hard regs	PID	MM	POW	BUB	MADD	FIB
none	86.0	78.1	81.7	78.3	87.6	83.3
0	87.5	78.7	85.7	78.8	92.8	84.8
1	87.1	82.7	87.3	78.3	89.9	85.3
2	89.7	80.2	84.6	74.5	89.4	85.6
3	89.7	83.3	84.0	84.4	86.3	88.1
4	90.5	84.5	85.1	83.9	86.4	
0, 1	87.5	84.3	92.0	82.0	95.3	89.0
0, 2	90.8	84.1	90.3	80.3	95.0	88.9
0, 3	86.6	83.2	91.4	85.9	92.8	90.6
0, 4	91.5	84.8	92.4	85.5	93.1	
1, 2	91.2	82.4	89.3	80.4	93.0	90.7
1, 3	87.7	87.6	87.3	85.2	87.2	91.8
1, 4	91.9	88.8	87.6	85.1	89.2	
2, 3	89.5	86.3	87.8	84.0	89.3	90.6
2, 4	94.6	86.3	88.5	83.5	87.9	
3, 4	90.1	89.0	88.3	90.1	86.1	
0, 1, 2	92.0	86.2	94.0	83.6	97.7	94.1
0, 1, 3	88.1	88.4	93.6	88.1	94.1	95.0
0, 1, 4	92.1	89.8	94.0	87.6	95.5	
0, 2, 3	90.5	88.7	91.8	86.1	95.7	94.0
0, 2, 4	95.2	89.2	93.4	86.3	93.9	
0, 3, 4	91.5	89.4	94.5	91.5	93.8	
1, 2, 3	91.4	87.9	90.0	86.5	91.4	95.4
1, 2, 4	96.2	88.4	90.6	85.9	90.7	
1, 3, 4	91.1	93.7	91.9	91.7	87.8	
2, 3, 4	94.6	91.6	88.5	90.8	87.6	
0, 1, 2, 3	92.7	90.3	96.4	88.7	97.0	99.3
0, 1, 2, 4	96.9	91.5	96.9	89.1	96.8	
0, 1, 3, 4	92.5	94.1	97.2	94.7	94.1	
0, 2, 3, 4	94.9	94.4	96.9	92.2	94.8	
1, 2, 3, 4	96.1	94.1	94.5	92.3	89.1	
0, 1, 2, 3, 4	97.4	96.5	99.9	95.3	96.0	

Table 3: Fault coverage on TI-MSP430 (unACE%)

Hard regs	PID	MM	POW	BUB	MADD	FIB
none	88.4	80.9	67.7	84.5	83.6	76.3
4	91.4	88.0	75.3	90.9	89.4	79.9
5	90.4	81.7	76.5	86.3	83.1	84.9
6	90.6	83.4	75.1	87.7	84.1	77.6
7	96.1	86.0	72.6	90.7	86.9	80.8
4, 5	91.4	89.1	83.8	90.5	88.5	88.1
4, 6	91.7	89.9	83.1	91.9	91.0	82.5
4, 7	97.0	92.8	83.4	91.9	93.6	84.4
5, 6	89.9	83.5	83.5	87.7	84.6	85.1
5, 7	96.3	86.5	81.2	91.8	88.2	88.5
6, 7	96.0	88.6	81.5	93.7	89.1	84.1
4, 5, 6	91.8	90.6	91.4	91.9	90.3	90.1
4, 5, 7	97.2	94.0	91.8	96.1	95.9	92.6
4, 6, 7	97.0	95.5	91.4	97.9	96.0	88.1
5, 6, 7	96.1	89.2	89.9	94.0	90.6	91.6
4, 5, 6, 7	97.2	95.6	99.7	98.2	98.2	95.96

In general, fault coverage increases with the protec-

tion level (i.e. the number of registers protected). For instance, in the case of PID (Table 2) the baseline is 86.00% of unACE when no protection is applied, and the maximum values obtained are 90.53%, 94.61%, 96.29%, 96.94% and 97.40%, corresponding to one register protected (version "4"), two registers protected (version"2, 4"), three registers protected (version "1, 2, 4"), four register protected (version "0, 1, 2, 4") and full protection (version "0, 1, 2, 3, 4") respectively.

However, an increment in the number of protected registers does not always produce an improvement of fault coverage. As it can be observed, version "4" offers better result than the majority of two register versions. In the same way, version "2, 4" exceeds the value of all three registers versions except just one, the version "1, 2, 4". This apparent contradiction arises from the fact that software redundancy enhances the fault tolerance of protected resources, but also modifies the criticality of everything else in the code introducing new vulnerable states. This effect becomes particularly important for the MADD application. In this case, the maximum unACE is obtained by protecting only three registers (97.70% in the version "0, 1, 2"), which is higher than the best result offered by any version of four and five protected registers (i.e., 96.00% in the version "0, 1, 2, 3, 4").

The results for TI-MSP430 (Table 3) show a similar behavior. Although the extreme case of MADD is not repeated here, increasing the number of protected registers does not guarantee the fault coverage enhancement. As an example, the MM application offers a maximum unACE percentage of 88.08% for one register (version "4") which surpasses versions "5, 6" and "5, 7" with two registers protected. It corroborates the trend observed and clearly shows the different nature of software hardening techniques in contrast to the hardware-based ones.

3.2. Comparative analysis of SHARC metric

Previous results helped us to compare the new method with the state-of-the-art AVF metric. For this purpose, fault coverage percentages were used to elaborate a reference ranking (golden rank) with the best register candidates to be hardened. The golden rank was obtained as follows: for each one of the test programs, the maximum fault coverage was selected among the hardened versions with only one protected register, i.e, the versions "0", "1", "2", "3", (and "4"). The register protected in the selected version was labeled at the top in the golden criticality rank. Then, the maximum fault coverage was chosen among the hardened versions with two protected registers, and at the same time, containing the first register in the golden rank. The same procedure was followed for the versions with protection in three, four, and five registers to complete the rank.

To obtain a criticality rank based on AVF, each register in the non-hardened version of the programs was attacked separately by means of fault injection campaigns. In this way, the rank was built according to the percentage of unACE faults exposed by each register in increasing order (from the most critical to the least critical register).

In contrast to the AVF metric which need, on average, $10,000$ runs per register, SHARC metric can be estimated with just one run. The statistics of each resource involved are collected, during the execution of the original code, by the ISA simulators and the estimations are calculated on the fly. It represents an important saving in development effort of, at least, four orders of magnitude in comparison to any other fault-injection based metric. In addition to the accuracy of the criticality estimations, this is one of the main advantages of SHARC. The weight coefficients used in (1) were adjusted to 0.5 each. This configures an equal weight for each criterion. However, these coefficients can be modified according to the vulnerability overhead introduced by the SIHFT technique.

Pearson Product-Moment Correlation Coefficient (PCC) was used to evaluate the quality of the criticality ranks with respect to the golden rank for every program in the benchmark. These correlations show the strength of the linear association between the variables.

Tables 4 and 5 show the results of correlation coefficients obtained by SHARC and AVF metrics. Comparing these results we can see that the coefficients obtained using the SHARC metric on PicoBlaze are equal to or higher than those obtained using the AVF metric. In average, the correlation is 88% in the case of the SHARC metric, whereas it is only 76% in case of the AVF metric. As for the results obtained in TI-MSP430, the correlation coefficients show a minimum difference of 1.66% between the two metrics. On average, the AVF metric obtains the highest percentage with 88.33%, whereas the SHARC metric obtains a lower percentage with 86.67%.

However, regarding the number of matches of the two most critical registers (marked with circles), SHARC offers better results in all cases. As it can be seen, in the case of the PicoBlaze processor, the maximum number of matches of the most critical register according to the golden rank was obtained by the SHARC metric with a total of 6. Meanwhile, the AVF metric obtained only 3 matches. For the second most critical register, the SHARC metric scored 4 matches against 2 of AVF. The results on TI-MSP430 processor show the same trend. SHARC metric scored a total of 5 matches on the most critical register with respect to 4 of the AVF metric. For the second most critical register, the SHARC metric

Table 4: AVF vs. SHARC in PicoBlaze

Test	Rx	Golden	AVF metric			SHARC metric		
		Rank	% ACE	Rank	PCC	Criticality	Rank	PCC
PID	s0	4	64.6	3	60%	0.50	4-5	90%
	s1	3	31.2	5		0.48	3	
	s2	②	79.4	1		0.47	②	
	s3	5	39.6	4		0.50	4-5	
	s4	①	69.9	2		0.46	①	
MM	s0	5	40.4	5	58%	0.53	5	90%
	s1	3	74.3	2		0.40	2	
	s2	4	74.3	2		0.52	4	
	s3	②	69.5	4		0.48	3	
	s4	①	96.6	①		0.40	①	
POW	**s0**	②	92.1	1	80%	0.46	②	100%
	s1	①	49.4	2		0.43	①	
	s2	5	36.5	4		0.56	5	
	s3	4	36.1	5		0.53	3-4	
	s4	3	48.0	3		0.53	3-4	
BUB	s0	4	65.9	4	90%	0.55	3-4	90%
	s1	3	70.1	3		0.55	3-4	
	s2	5	20.9	5		0.56	5	
	s3	①	99.0	②		0.38	①	
	s4	②	99.2	①		0.42	②	
MADD	**s0**	①	72.8	①	90%	0.41	①	90%
	s1	②	52.3	②		0.45	3	
	s2	3	49.7	3		0.43	2	
	s3	4	9.20	5		0.51	4-5	
	s4	5	9.50	4		0.51	5-5	
FIB	s0	4	57.2	3	80%	0.52	4	70%
	s1	②	79.2	②		0.53	②	
	s2	3	54.9	4		0.53	3	
	s3	①	81.0	①		0.42	①	
Average					76%			88%

Table 5: AVF vs. SHARC in TI-MSP430

Test	Rx	Golden	AVF metric			SHARC metric		
		Rank	% ACE	Rank	PCC	Criticality	Rank	PCC
PID	r4	②	30.6	②	90%	0.45	3	70%
	r5	3	5.4	4		0.52	4	
	r6	4	12.8	3		0.43	2	
	r7	①	89.6	①		0.42	①	
MM	r4	①	84.5	2	90%	0.41	①	90%
	r5	4	10.9	4		0.53	4	
	r6	3	39.0	3		0.46	2	
	r7	②	94.7	1		0.51	3	
POW	r4	②	95.2	3	90%	0.43	②	100%
	r5	①	99.2	①		0.38	①	
	r6	4	94.1	4		0.58	4	
	r7	3	98.8	2		0.55	3	
BUB	r4	①	62.8	2	80%	0.37	①	90%
	r5	4	19.2	3		0.49	3	
	r6	3	16.0	4		0.51	4	
	r7	②	87.2	1		0.47	②	
MADD	r4	①	80.9	①	90%	0.45	2	80%
	r5	4	34.2	3		0.52	3	
	r6	3	29.0	4		0.53	4	
	r7	②	52.6	②		0.44	1	
FIB	r4	②	63.3	3	90%	0.46	②	90%
	r5	①	94.0	①		0.41	①	
	r6	4	38.5	4		0.49	3	
	r7	3	87.7	2		0.56	4	
Average					88.33%			86.67%

scored 3 matches and the AVF 2 matches. These results support the method for selective hardening depicted in Algorithm 1.

Tables 6 and 7 show the comparative analysis after implementing the proposed method (Section 2.1). Note that in this case, the Criticality column represents the SHARC value of the best candidate for each hardened version. Therefore, those values do not follow a rank order.

As expected, the coefficients obtained by the method based on the SHARC metric have improved considerably in both processors. On average, the benchmarks executed in PicoBlaze show the highest correlation percentages with an overall increase up to 97%, whereas the AVF metric is 76%. Regarding the benchmarks executed in TI-MSP430, we can see a notable increase in the correlation percentages obtained. On average, the SHARC-based method gets 93.33%, whereas the AVF metric is 88.33%.

Table 6: Selective Hardening with SHARC vs AVF in PicoBlaze

Test	Rx	Golden	AVF			Method based on SHARC		
		Rank	% ACE	Rank	PCC	Criticality	Rank	PCC
PID	s0	4	64.6	3	60%	0.08	5	90%
	s1	3	31.2	5		0.37	3	
	s2	2	79.4	1		0.42	2	
	s3	5	39.6	4		0.19	4	
	s4	1	69.9	2		0.46	1	
MM	s0	5	40.4	5	58%	0.08	5	90%
	s1	3	74.3	2		0.37	2	
	s2	4	74.3	2		0.20	4	
	s3	2	69.5	4		0.37	3	
	s4	1	96.6	1		0.40	1	
POW	s0	2	92.1	1	80%	0.16	2	100%
	s1	1	49.4	2		0.43	1	
	s2	5	36.5	4		0.10	5	
	s3	4	36.1	5		0.25	4	
	s4	3	48.0	3		0.40	3	
BUB	s0	4	65.9	4	90%	0.27	4	100%
	s1	3	70.1	3		0.37	3	
	s2	5	20.9	5		0.11	5	
	s3	1	99.0	2		0.38	1	
	s4	2	99.2	1		0.36	2	
MADD	s0	1	72.8	1	90%	0.41	1	100%
	s1	2	52.3	2		0.41	2	
	s2	3	49.7	3		0.21	3	
	s3	4	9.20	5		0.26	4	
	s4	5	9.50	4		0.07	5	
FIB	s0	4	57.2	3	80%	0.13	4	100%
	s1	2	79.2	2		0.41	2	
	s2	3	54.9	4		0.36	3	
	s3	1	81.0	1		0.42	1	
Average					76%			97%

Table 8 summarizes the results offered by the different selective methods studied. The brute-force strategy is the more effective method with a 100% match in every case. It also implies the higher effort, in terms of development time expressed as the number of executions needed to get the best hardened version (#*runs*) . The effort level depends on three factors: the number of code versions (N_v), the number of registers involved (R_{rgs}) and the minimum number of injected faults needed for an accurate estimation of the fault coverage (F_f). AVF based methods rely also on fault injection campaign, but just the criticality of the registers in the original code is evaluated. It takes a time proportional to the parameters R_{rgs} and F_f. The average match levels reach to 76% and 83.3% for PicoBlaze and TI-MSP430 respectively. Finally, the SHARC method is able to make estimations with just one run of each code version (R_{rgs} versions in total). In spite of this limited effort, the match level is

Table 7: Selective Hardening with SHARC vs AVF in TI-MSP430

Test	Rx	Golden	Method based on AVF			Method based on SHARC		
		Rank	%ACE	Rank	PCC	Criticality	Rank	PCC
PID	r4	2	30.67	2	90%	0.33	2	90%
	r5	3	5.47	4		0.08	4	
	r6	4	12.84	3		0.16	3	
	r7	1	89.69	1		0.42	1	
MM	r4	1	84.56	2	90%	0.41	1	100%
	r5	4	10.94	4		0.12	4	
	r6	3	39.01	3		0.25	3	
	r7	2	94.73	1		0.35	2	
POW	r4	2	95.27	3	90%	0.40	2	100%
	r5	1	99.2	1		0.38	1	
	r6	4	94.1	4		0.13	4	
	r7	3	98.84	2		0.30	3	
BUB	r4	1	62.85	2	80%	0.37	1	90%
	r5	4	19.2	3		0.30	3	
	r6	3	16.05	4		0.10	4	
	r7	2	87.21	1		0.37	2	
MADD	r4	1	80.91	1	90%	0.36	2	80%
	r5	4	34.22	3		0.30	3	
	r6	3	29.04	4		0.11	4	
	r7	2	52.64	2		0.44	1	
FIB	r4	2	63.39	3	90%	0.39	2	100%
	r5	1	94.07	1		0.41	1	
	r6	4	38.57	4		0.11	4	
	r7	3	87.71	2		0.31	3	
Average					88.33%			93.33%

Table 8: Summary of results

Method	Brute-force	AVF	SHARC
Effort Level (#runs)	$N_v \cdot R_{rgs} \cdot F_f$	$R_{rgs} \cdot F_f$	R_{rgs}
Match Level PicoBlaze	100%	76.0%	97.0%
Match Level MSP430	100%	88.3%	93.4%

above 90% on average for both processors.

4. Conclusion

In this paper we presented a metric for the selective hardening of software based on the dynamic measure of the registers' criticality (SHARC metric). The estimation of the registers' criticality is calculated during the execution of the original code, avoiding time-consuming fault injection campaigns. In addition, based on the SHARC metric, an incremental method is proposed for protecting software, taking into account design restrictions and ensuring maximum reliability in terms of fault coverage.

Experiments carried out with programs of different nature (control oriented, data processing, etc.) show that criticality estimates based on SHARC metric not only significantly improve the accuracy of results compared to the AVF metric, but that higher performance is also achieved for prioritizing the candidates to be protected by implementing the proposed method. As a result, the rank obtained better matches the ideal hardening configuration. Experiments show that SHARC improves the results of state of the art methods between 5% and 21% on average, at a fraction of the effort.

Acknowledgment

This work was funded by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund with the project *Evaluación temprana de los efectos de radiación mediante simulación y virtualización. Estrategias de mitigación en arquitecturas de microprocesadores avanzados* (Ref: ESP2015-68245-C4-3-P MINECO/FEDER, UE).

References

[1] M. Nicolaidis (Ed.), Soft Error in Modern Electronic System, volume 41 of *Frontiers in Electronic Testing*, Springer US, 2011.

[2] M. O'Halloran, J. G. Hall, L. Rapanotti, Safety engineering with cots components, Reliability Engineering & System Safety 160 (2017) 54–66.

[3] S. Goloubeva, Rebaudengo, Violante (Eds.), Software-Implemented Hardware Fault Tolerance, Springer, 2006.

[4] J. R. Azambuja, S. Pagliarini, L. Rosa, F. L. Kastensmidt, Exploring the Limitations of Software-based Techniques in SEE Fault Coverage, Journal of Electronic Testing 27 (2011) 541–550.

[5] J. H. I. Polian, Selective Hardening: Toward Cost-Effective Error Tolerance, IEEE Design and Test of Computers (2011) 45–63.

[6] F. Restrepo-Calle, A. Martínez-Álvarez, S. Cuenca-Asensi, A. Jimeno-Morenilla, Selective SWIFT-R: A flexible software-based technique for soft error mitigation in low-cost embedded systems, Journal of Electronic Testing: Theory and Applications (JETTA) 29 (2013) 825–838.

[7] E. Chielle, G. S. Rodrigues, F. L. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, H. Quinn, S-seta: Selective software-only error-detection technique using assertions, IEEE Transactions on Nuclear Science 62 (2015) 3088–3095.

[8] 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, in: 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, 2003, pp. 29–40.

[9] S. Raasch, A. Biswas, J. Stephan, P. Racunas, J. Emer, A fast and accurate analytical technique to compute the avf of sequential bits in a processor, in: Proc. 48th International Symposium on Microarchitecture, MICRO-48, ACM, New York, NY, USA, 2015, pp. 738–749. doi:10.1145/2830772.2830829.

[10] A. Martínez-Álvarez, F. Restrepo-Calle, L. A. Vivas Tejuelo, S. Cuenca-Asensi, Fault tolerant embedded systems design by multi-objective optimization, Expert Systems with Applications 40 (2013) 6813–6822.

[11] F. Restrepo-Calle, S. Cuenca-Asensi, A. Martínez-Álvarez, E. Chielle, F. L. Kastensmidt, Application-Based Analysis of Register File Criticality for Reliability Assessment in Embedded Microprocessors, Journal of Electronic Testing 31 (2015) 139–150.

[12] F. Restrepo-Calle, S. Cuenca-Asensi, A. Martinez-Alvarez, An effective strategy for selective hardening of software, in: 2017 18th IEEE Latin American Test Symposium (LATS), 2017, pp. 1–6. doi:10.1109/LATW.2017.7906744.

[13] G. A. Reis, J. Chang, D. I. August, Automatic instruction-level software-only recovery, IEEE Micro 27 (2007) 36–47.

[14] K. Chapman, PicoBlaze KCPSM3. 8-bit Micro Controller for Spartan-3, Virtex-II and Virtex-II Pro, Xilinx Ltd., 2003.

[15] A. Martinez-Alvarez, S. Cuenca-Asensi, F. Restrepo-Calle, F. R. P. Pinto, H. Guzman-Miranda, M. A. Aguirre, Compiler-Directed Soft Error Mitigation for Embedded Systems, IEEE Transactions on Dependable and Secure Computing 9 (2012) 159–172.

[16] M. Kohn, Naken, 2018. URL: `http://www.mikekohn.net/micro/naken_asm.php`.