This is a previous version of the article published in Journal of Electronic Testing. 2015, 31(2): 139-150. doi:10.1007/s10836-015-5513-9

Application-based Analysis of Register File Criticality for Reliability Assessment in Embedded Microprocessors

Felipe Restrepo-Calle, Sergio Cuenca-Asensi, Antonio Martínez-Álvarez, Eduardo Chielle, Fernanda Lima Kastensmidt

Abstract—There is an increasing concern to reduce the cost and overheads during the development of reliable systems. Selective protection of most critical parts of the systems represents a viable solution to obtain a high level of reliability at a fraction of the cost. In particular to design a selective fault mitigation strategy for processor-based systems, it is mandatory to identify and prioritize the most vulnerable registers in the register file as best candidates to be protected (hardened).This paper presents an application-based metric to estimate the criticality of each register from the microprocessor register file in microprocessor-based systems. The proposed metric relies on the combination of three different criteria based on common features of executed applications. The applicability and accuracy of our proposal have been evaluated in a set of applications running in different microprocessors. Results show a significant improvement in accuracy comparing to previous approaches and regardless the underlying architecture.

Index Terms—Embedded systems, Metrics, Microprocessors, Reliability

1 INTRODUCTION

Technological scaling is posing major challenges on the development of reliable systems, e.g.: voltage and temperature variability, sensitivity to soft errors and electromagnetic interferences, accelerated degradation as aging [1]. These challenges may cause timing faults and Single Event Effects (SEEs), which provoke permanent or temporary effects over the electronic components operation, increasingly affecttingreliability.Therefore, fault tolerant design has become a mandatory issue for an increasing number of application domains, including: space, avionics, automotive, defense, medicine, and communications [2].

A wide spectrum of design techniques has emerged to overcome these problems. Traditionally, fault tolerant design has relied mainly on expensive and power costly approaches based on hardware redundancy [3]. More recently, thanks to the proliferation of processor-based systems and the need for reliable low-cost solutions, a large number of techniques based on redundant software have been proposed [4, 25]. However, they cause non-negligible overheads in terms of code size, execution time, and data that designers have to cope with [5].

To reduce the costs and overheads inherent to the protection, whether hardware or software, recent approaches propose to use a selective protection strategy (selective hardening). It consists of protecting only the most critical parts of the designs. This partial protection can be achieved by means of selective redundancy applied to: hardware [6-7], software [8-9], or by means of hybrid hardware/software approaches [10-11].

In particular for processor-based systems, a well-known strategy consists of protecting only the most critical registers in the microprocessor register file [8-11].Thus, it is evident the necessity to identify the most critical registers properly during an early design stage in order to facilitate the selection of the hardened set of registers. A proper selection of registers determines to achieve good efficiency/cost tradeoffs for the designed solution and, at the same time, permits to explore the design space effectively, avoiding costly explorations guided by brute force strategies [24].

Furthermore, early estimation of the register file criticality is crucial in order to quickly obtain an evaluation before the system is fully implemented and fault injections can be performed. These estimations are strongly needed during the design phase in order to properly develop a hardening strategy that better fulfills the reliability requirements and system constraints.

In this context, our work presents a metric to estimate the

- *F. Restrepo-Calle is with the Department of Systems and Industrial Engineering, Universidad Nacional de Colombia, Bogotá, Colombia. E-mail: ferestrepoca@unal.edu.co.*
- *S. Cuenca-Asensi and A. Martínez-Álvarez are with the Computer Technology Department, University of Alicante, Carretera San Vicente del Raspeig s/n, 03690 Alicante, Spain. E-mail: {sergio, amartinez}@dtic.ua.es.*
- *E. Chielle and F. Lima Kastensmidt are with the Instituto de Informática, PPGC and PGMICRO, Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil. E-mail: {echielle, fglima}@inf.ufrgs.br.*

criticality of each register from the microprocessor register file, which allows a fine-grained analysis. It is based on dynamic code analysis (profiling), using the low-level (assembly) source code of the program. In addition, the proposed metric facilitates the selection (and prioritization) of registers to be hardened when a selective approach is required.

As in the original work of Bergaoui et al. [12], our metric is based on the combination of three different criteria: *lifetime*, *weight in conditional branches*, and *functional dependencies*. The novelty and contributions of our proposal can be summarized as follows:

- *Dynamic code analysis*: unlike the compilation-time method used in [12] (static code analysis), we propose to use dynamic measurements for the computation algorithms during run-time (simulation-time). This kind of assessment does not represent any inconvenience in the usual design-flow of embedded systems, and alternatively, improves significantly the accuracy of the estimations.
- *Effective lifetime:* the modulation of the role of the register lifetime using diverse considerations not taken into account in previous works. These include the consideration of the total lifetime as the sum of several lifetime intervals, and the correct calculation of those intervals (Section 3.1).
- *Normalization*: all criteria have to be normalized with respect to the total number of executed instructions. For instance, in the case of the weight in conditional branches criterion, this consideration permits to assign a real contribution to the criticality estimation.
- *Direct descendants*: counting dynamically only its direct descendants during the program execution to avoid counting erroneously n-levels descendants.

Two case studies have been explored and discussed. To validate the applicability of the proposed metric, each case study targets a different microprocessor and benchmark. To corroborate the accuracy of the results, criticality estimations have been analyzed and compared with the results obtained during fault injection campaigns. The experimental outcomes show that the proposal improves related works accuracy results.

This paper is organized as follows. Next section provides background information about related works. Section 3 presents the criticality criteria and defines the proposed metric. Section 4 validates the proposal by means of a comprehensive experimental study. Finally, Section 5 summarizes the concluding remarks of this work.

2 Related Works

The most commonly used vulnerability metric is the Architectural Vulnerability Factor (AVF) [13]. The AVF of a hardware structure is defined as the probability that a fault in that structure will result in a visible error in the final output of a program. AVF-derived works evaluate the vulnerability based on the micro-architecture features of processors; however, they do not take into account detailed characteristics of the executed programs.

In addition, it is known [14-15] that different programs and even different functions in an application are not equally critical due to different data and control flow properties, internal error masking effects, etc. These programs/functions exhibit distinct resilience to hardware-level faults. In this work, therefore, we focus on the criticality of the registers in the register file for a given executed program.

Among proposals analyzing specific features of the executed programs to estimate criticality, two groups can be found: proposals based on static code analysis, and those based on dynamic measurements.

Static analysis does not require simulation or actual usage of the system. Instead, it relies on statically-determined criteria (e.g., lifetime, functional dependencies between variables) [12, 16-17]. Static code analysis is a very important task in modern compilers because it makes possible to find bugs and perform live variable analysis in order to allocate registers. However, it is not very useful in the context of this work, since no compiler can statically know all the program dynamic properties to estimate criticality. Therefore, we propose to compute the criticality based on dynamic code analysis during simulation-time.

Dynamic techniques use information gathered during simulation or operation of the system to estimate criticality factors [18-19]. Unlike approaches based on static code analysis, dynamic analysis takes into account conditional branches, loop iterations, recursive functions, values only known in run-time, and in general, the execution progress of the application.

In addition, authors in [26-27] have proposed a control flow protection technique known as Optimized Embedded Signature Monitoring (OESM). The first step in this approach consists on the application profiling (dynamic code analysis) to optimize the number of checkpoints introduced into the application code, which are inserted in the second step by means of the application of a well-known control flow technique such as CFCSS [28]. Unlike this control-flow protection technique, our metric is aimed at identifying the most critical registers from the microprocessor register file for a further application of data protection schemes (out of the scope of the present work).

As summarized in the introduction, the present work extends the proposal presented in [12] by re-defining the calculation of the criticality criteria improving their accuracy in the estimation, by means of considering new parameters and performing dynamic analyses of the executed programs, instead of the static code analysis originally proposed.

3 Application-based Criticality Metric

We focus on the criticality of each individual register in the microprocessor register file for a given program (assembly code). Criticality can be expressed in terms of 3 criteria: *effective lifetime*, *weight in conditional branches*, and *functional dependencies*.

3.1 Effective Lifetime

Register *lifetime* represents the time when useful data is present in the register. Any fault occurring to the register during that time destroys data integrity. Therefore, the higher the *lifetime* is, the longer the register is prone to faults.

Register *lifetime* is expressed as the sum of clock cycles of all the register living intervals during the program simulation/execution. A *living interval* starts with a generic write operation and ends with the last read operation, which precedes the next write operation or the end of the program execution.

However, it is important considering that a new interval is created every time there is a write operation to the register, and during the execution of that instruction (*n* clock cycles), there are *k* clock cycles (*k*<*n*) at the beginning of the write execution in which the register has not yet stored the value, and any fault affecting it during that *k* time will be overwritten when it finally stores the written value. The remaining time in the interval (*n* - *k*) is called *partial effective lifetime (pelt)*. Hence, the register *effective lifetime* is the sum of all the *partial effective lifetimes*. These terms are illustrated in Fig. 1.

In the simplest case, these cycles are due to the number of stages in the instruction pipeline, and in case of more complex processorsthey are consequence of using different techniques that delay the effective write in the register file, these include: forwarding, speculative execution, reorder buffers, etc.

This consideration is very important because it implies that in cases of registers having the same/similar *lifetime*, criticality is lower for those registers whose *lifetime* presents a larger number of *living intervals*, i.e., their *effective lifetime* is lower.

According to this, we propose to adjust the *lifetime* (measured in clock cycles) by subtracting the *non-effective lifetime* (*k*) of each write operation from the total *lifetime* of a register, i.e., the *effective lifetime*. Depending on the microprocessor complexity, the *k* number of cycles can be either calculated or estimated. Finally, for estimating the register criticality the *effective lifetime* is normalized with respect to the duration of the program (clock cycles).

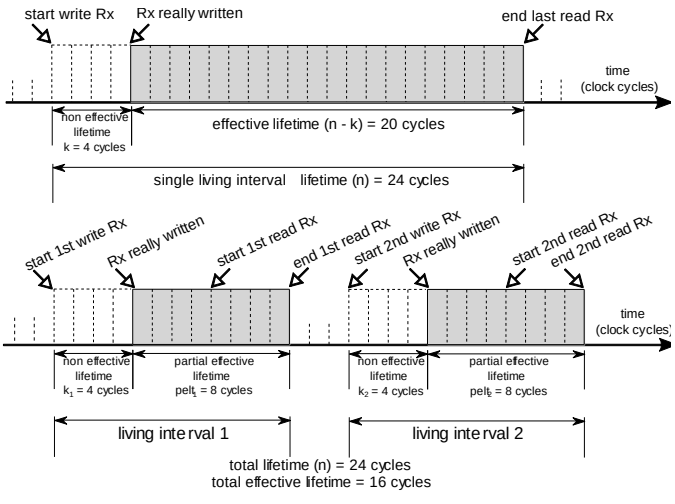

Fig. 1.Effective lifetime

3.2 Weight in Conditional Branches

The second criterion is *weight in conditional branches*. As it was first proposed in [12], it should be given more attention to the registers taking part in branch conditions. Erroneous data stored in these registers may lead the program control flow to take an incorrect path. We introduce two improvements to this criterion based on the dynamic analysis.

Firstly, real influence of registers on criticality is taken into account depending on the statement where the conditional branch is located. For instance, it is more critical a register involved in the evaluation of a loop condition, whose condition is repeated *n* times, than another that participates in a single conditional statement (executed once). Dynamic counting of registers that take part in branch conditions is able to capture this influence. Static code analysis, in contrast, may incorrectly indicate the same level of criticality for both registers in the previous example.

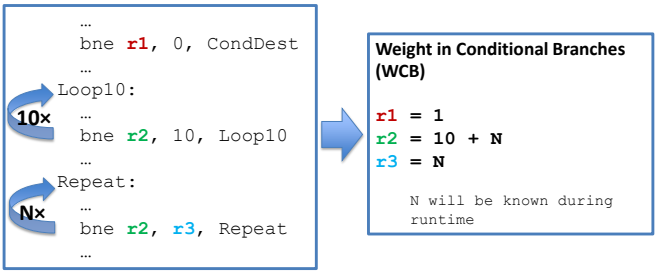

Fig. 2. Example of the weight in conditional branches criteria

Fig. 2 presents an example of different variations that can occur during the dynamic counting of the registers involved in conditional branches: *r1* is present in a single conditional statement executed once; *r2* is firstly present in a loop condition with a static loop limit (executed 10 times) and then it also participates in a second dynamic loop (executed *N* times); *r3* is involved in the last loop statement (executed *N* times). In terms of *weight in conditional branches* for this example, the most critical register is *r2* and the least critical *r1*.

The second improvement to this criterion consists on the normalization procedure. The dynamic measurement for each register taking part in conditional branches has to be normalized with respect the total number of executed instructions of the program, instead of the total number of effectively conditional branches. Otherwise, a register involved in conditional branches might have a high value for this criterion, regardless that number of conditional branches did not represent a significant portion of the executed instructions. For instance, a register participating in most conditional branches may be seen as critic for this criterion if only effectively conditional branches are taken into account on the normalization; however, let's suppose that those branches only represent the 1-2% of the executed instructions of the program, in which case the real register vulnerability is considerably lower. This consideration not only permits to assign a real contribution to the conditional branch criterion, but also facilitates to combine this criterion with other criticality criteria.

3.3 Functional Dependencies

The third criterion is the count of functional dependencies between registers. A register depends on a given register *r* (or is a *direct descendant* of *r*) if it takes a value that is the result of an expression involving the value of *r*. This criteri-

on remarks the criticality of those registers having a lot of descendants, as erroneous values will be propagated widely.

We propose to measure functional dependencies for each register by counting dynamically only its *direct descendants* during the program execution. Other descendants different to the register *direct descendants* (*n*-level descendants where $n \geq 2$) are not taken into account. Considering other levels without including dependency time intervals could lead to consider some *n*-level dependencies erroneously.

To compute functional dependencies, it is built an $N \times N$ matrix M (where N is the number of registers in the register file). A cell $M_{i,j}$ means that register i is descendant of register j. For each instruction simulated, matrix M is updated with the dynamic count of direct descendants. Once the matrix M is completely calculated, each cell is normalized in the same way as for the conditional branches criterion, i.e., with respect to the total number of executed instructions.

3.4 Criticality Metric

Criticality C of a register r can be estimated as expressed in (1).

$$C(r) = W_{lt} \cdot C_{lt}(r) + W_{cb} \cdot C_{cb}(r) + W_{fd} \cdot C_{fd}(r) \quad (1)$$

Where: W_{lt}, W_{cb}, and W_{fd} are weight coefficients assigned to each criterion; $C_{lt}(r)$ is the normalized effective lifetime; $C_{cb}(r)$ is the normalized conditional branch value; and $C_{fd}(r)$ is the normalized functional dependencies value, which can be calculated using (2).

$$C_{fd}(r) = \sum_{i=0}^{N-1} M_{i,r} \cdot \left(W_{lt} \cdot C_{lt}(i) + W_{cb} \cdot C_{cb}(i)\right) \quad (2)$$

Where: M is the dynamic functional dependencies matrix, and N is the number of registers in the register file. Notice that (2) includes the *effective lifetime* and *conditional branches* criteria of all *direct descendants* of r in the calculation of $C_{fd}(r)$.

4 Experimental Results and Discussion

4.1 Experimental setup

The weight coefficients used in (1) and (2) for the register criticality calculation were equal to 0.33 each. This configures an equal weight for each criterion. These values were chosen for demonstration, and at the same time, for comparison purposes, as these were the same weights used by authors in [12]. However, these coefficients could have been modified according to each application dependability requirements.

To validate the applicability of the proposal, several target applications have been studied. The experimental setup is divided in two different case studies that permit to demonstrate that the metric is hardware-agnostic. The first set of target applicationsis based on the PicoBlaze microprocessor [20] (case study 1), whereas the second group targets the miniMIPS microprocessor [21] (case study 2).

To corroborate the accuracy of the results in both case studies, criticality estimations have been analyzed and compared with results obtained during fault injection test. We focus on the type of transient fault known as *Single Event Upset* (*SEU*), which is characterized by the logic state alteration of a single memory element in the system. For each studied application in the case studies, a fault injection campaign has been carried out against each one of the registers in the register file. Each campaign consisted in 10,000 faults injected (one fault per run). Each fault consisted of a *bit-flip* in a randomly selected bit from the target register in a randomly selected clock cycle from all the workload duration.

Besides to validate the accuracy of the estimated criticality results comparing with those obtained by means of fault injection tests, the case studies were aimed to verify that our proposal improves related works accuracy results. Therefore, we have also implemented the criticality metric based on static code analysis proposed in [12] to compare the results using the same processor and the same test programs.

To evaluate the goodness of the estimations, we check how well the expected rank of critical registers (based on the error rate obtained in the fault injection tests) matches to the estimated rank, i.e., match level. Ranks are expressed in ascending order from the most critical to the least critical register. The match level determines the distance between each value of the estimated criticality rank compared to its respective expected value. A match level equals to 0 means an exact match between them, which is the desirable value.

Moreover, the analysis of the code necessary for the criticality estimation is performed previously to the final deployment. To do so, specific features of the simulator or code instrumentation can be used. In case of code instrumentation, it is removed from the program after the estimation. Therefore, there is no memory overhead or performance penalty introduced to the deployed program in any case.

4.2 Case Study 1: PicoBlaze

PicoBlaze is a widely used IP (intellectual property) core. The main features of this processor are: 16 byte-wide general-purpose data registers, 1K instructions of programmable on-chip program store, byte-wide Arithmetic Logic Unit (ALU) with Carry and Zero indicator flags, 64-byte scratch-pad RAM, 256 input and 256 output ports, 31-location Call/Return stack.

The benchmark software suite used for PicoBlaze is made up of 11 example programs. Some of them are representative programs used in embedded systems: proportional-integral-derivative controller (PID), finite impulse response filter (FIR), and advanced encryption standard (AES) or Rijndael. The rest of them consist of typical small computations: bubble sort (BUB), Fibonacci (FIB), greatest common divisor (GCD), matrix addition (MADD), matrix multiplication (MM), scalar multiplication (MULT), exponentiation (POW), and quick sort (QSORT). In this case, fault injection tests were performed using the software-based simulation tool presented and validated in [22].

Table 1 presents the obtained results for the PicoBlaze case study. For each studied application and for each register (RX) used within the program source code, it is presented the register error rate obtained in the fault injection tests, the estimated criticality calculated using the static criticality

metric, and the estimated criticality calculated using our proposal. In all the cases only 5 (or less) registers were used to code the applications.

Notice that the estimations of criticality ranks obtained by the dynamic criticality metric are quite approximate to the ranks obtained by fault injection. An 82.0% of the estimated positions in the rank correspond to the respective expected value in the criticality rank (match level = 0).

Furthermore, in case of giving an acceptable error margin to the estimations, i.e., considering match levels equal to 0 and 1 as well, this percentage goes up to 98.0%. That is, the estimated positions in the criticality rank can be at a maximum distance of one position to its expected value based on the fault injection results.

Comparing these results with the originally published results (from Tables VII and VIII in [12]) using the match level, it can be seen that only 25.0% of the estimations match to the respective expected value in the criticality rank (match level = 0). Including results from match levels equal to 0 and 1, this percentage only increases to 62.5%.

Nevertheless, for the sake of comparison, it is necessary to analyze the estimated results obtained by the static criticality metric for the same microprocessor and the same benchmark (Table 1). In this case, from Table 1 one can see that only a 46.0% of the estimated positions in the rank match the expected value in the criticality rank. This percentage is increased to 78.0% when considering as acceptable match levels equal to 0 and 1. These results are 36.0% and 20% less accurate than our proposal, respectively.

Fig. 3 represents the match level results presented in Table 1. This premits to compare, at a glance, estimation results obtained by the state-of-the-art static approach [12] with results achieved using our proposal. As commented above, the experimental outcomes show that our proposal improves significantly the related work accuracy results.

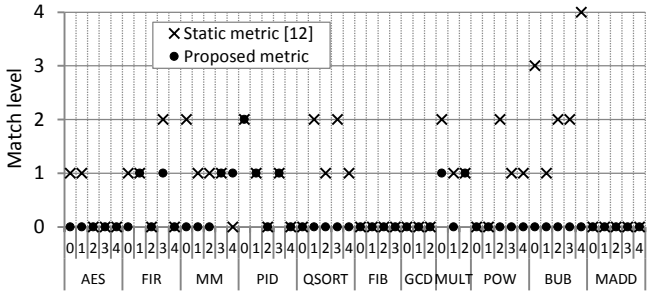

Fig. 3. Proposal Vs. Static metric match levels in PicoBlaze

Fig. 4 represents the correlation coefficient between the estimations (static metric and dynamic metric) and the error rate obtained in the fault injection campaigns for each test program, which shows the strength of the linear association between the variables. All the correlation coefficients from the proposed metric are greater than the coefficients obtained in the static metric (closer to 1.0). This means that the criticality estimations obtained using our metric represent the expected criticality in a much better way than the static metric.

TABLE 1
CRITICALITY RESULTS FOR THE PICOBLAZE CASE STUDY

Prog.	RX	Fault Injection		Static criticality metric [12]			Dynamic Criticality Metric (proposed)		
		Error Rate (%)	Crit. Rank	Est. Crit.	Est. Crit. Rank	Match Level [a]	Est. Crit.	Est. Crit. Rank	Match Level [a]
AES	0	48.91	2	0.264	1	1	0.277	2	**0**
	1	69.94	1	0.180	2	1	0.314	1	**0**
	2	31.55	3	0.136	3	0	0.205	3	**0**
	3	12.15	5	0.108	4-5	0	0.107	5	**0**
	4	15.40	4	0.108	4-5	0	0.114	4	**0**
FIR	0	30.56	4	0.240	3	1	0.118	4	**0**
	1	38.35	3	0.247	2	1	0.313	2	**1**
	2	86.02	1	0.272	1	0	0.331	1	**0**
	3	45.08	2	0.233	4	2	0.291	3	**1**
	4	1.26	5	0.042	5	0	0.006	5	**0**
MM	0	39.81	5	0.145	2	2	0.173	5	**0**
	1	76.16	3	0.061	4	1	0.308	3	**0**
	2	74.95	4	0.022	5	1	0.290	4	**0**
	3	85.01	2	0.079	3	1	0.337	1	**1**
	4	96.64	1	0.174	1	0	0.321	2	**1**
PID	0	65.01	3	0.124	5	2	0.241	5	**2**
	1	31.54	5	0.143	4	1	0.246	4	**1**
	2	80.32	1	0.172	1	0	0.311	1	**0**
	3	40.69	4	0.156	3	1	0.269	3	**1**
	4	70.94	2	0.170	2	0	0.275	2	**0**
QSORT	0	82.09	1	0.221	1	0	0.306	1	**0**
	1	66.95	2	0.083	4	2	0.253	2	**0**
	2	14.70	4	0.022	5	1	0.073	4	**0**
	3	8.65	5	0.102	3	2	0.057	5	**0**
	4	28.12	3	0.111	2	1	0.165	3	**0**
FIB	0	57.82	3	0.192	3	0	0.222	3	**0**
	1	79.79	2	0.243	2	0	0.306	2	**0**
	2	55.08	4	0.147	4	0	0.218	4	**0**
	3	81.51	1	0.259	1	0	0.347	1	**0**
GCD	0	86.22	1	0.336	1	0	0.423	1	**0**
	1	81.81	2	0.254	2	0	0.400	2	**0**
	2	1.59	3	0.041	3	0	0.008	3	**0**
MULT	0	87.11	3	0.203	1	2	0.364	2	**1**
	1	99.32	1	0.198	2	1	0.437	1	**0**
	2	99.31	2	0.165	3	1	0.328	3	**1**
POW	0	99.36	2	0.178	2	0	0.336	2	**0**
	1	99.77	1	0.242	1	0	0.370	1	**0**
	2	16.84	5	0.134	3	2	0.152	5	**0**
	3	17.30	4	0.110	5	1	0.185	4	**0**
	4	19.05	3	0.113	4	1	0.191	3	**0**
BUB	0	76.06	4	0.292	1	3	0.263	4	**0**
	1	79.31	3	0.111	2	1	0.313	3	**0**
	2	23.71	5	0.043	3	2	0.148	5	**0**
	3	99.07	2	0.042	4	2	0.327	2	**0**
	4	99.15	1	0.028	5	4	0.343	1	**0**
MADD	0	88.40	1	0.240	1	0	0.332	1	**0**
	1	49.66	2	0.166	2	0	0.189	2	**0**
	2	49.51	3	0.151	3	0	0.167	3	**0**
	3	9.00	4	0.049	4-5	0	0.043	4-5	**0**
	4	8.96	5	0.049	4-5	0	0.043	4-5	**0**

[a] ***Match level:*** *distance between each value of the estimated criticality rank compared to its respective expected value (based on fault injection results). 0 means an exact match between them (desirable).*

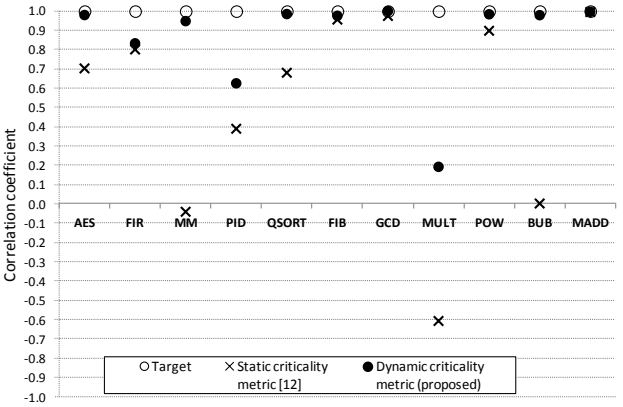

Fig. 4. Correlation coefficients between estimations and error rate in PicoBlaze

The only two programs with significant discrepancies for our proposal were PID and MULT, i.e., their correlation coefficients were less than 0.8. One possible reason for this divergence is that the criticality estimations in these cases are under the sensitivity of the proposed metric.In both cases the standard deviation of the results is less than 0.060, which means that the estimations are very close to each other and the criticality rank could vary slightly. The standard deviation for the rest of the test programs is greater than 0.060.

4.2 Case Study 2: miniMIPS

MiniMIPS is a 32 bits core based on MIPS I architecture. It has a pipeline of 5 stages and 32 general purpose registers ($0-$28, $sp, $fp and $31). All miniMIPS instructions take five cycles to be executed and the peak throughput is 1 instruction per cycle. Register $0 is constant, so it is not considered in the analysis.

Five case-study applications are selected as benchmak for miniMIPS: a bubble sort (BS), a matrix multiplication (MM), a non-recursive Depth-Fist Search (nDFS), a recursive Depth-First Search (rDFS) and the Tower of Hanoi (TH). The number of used registers ranges from 7 to 16 depending of the application.

A Register Transfer Level (RTL) description of the microprocessor is submited to a fault injection campaign performed by means of a VHDL simulator [23]. The faults are injected by forcing a bit-flip in the registers' signals. The duration of a fault is set to one clock cycle to increase the probability of the fault causes an error. The fault injection results are then compared to the results obtained by the proposed metric.

Table 2 presents the obtained results for the miniMIPS case study. For each studied application and for each register (RX) used within the program source code, it is presented the register error rate obtained in the fault injection tests, the estimated criticality calculated using the static criticality metric, and the estimated criticality calculated using our proposal.

Only 33.3% of the estimated positions in the rank correspond to the expected criticality rank (match level = 0), but it is still better than the 29.8% presented by the static criticality metric. The low correspondence for the miniMIPS is due to the higher number of registers used by the programs and also to the close error rates presented by many registers. That makes the probability of ranking them correctly more unlikely.

Furthermore, in case of giving an acceptable error margin to the estimations, i.e., considering match levels equal to 0 and 1 as well, the percentage goes up to 66.7% for the proposed metric. It is an improvement when compared with the state-of-the-art because the static criticality metric reaches 47.4%. Match level results obtained by the state-of-the-art approach [12] and the results achieved using our proposal are presented in Fig. 5.

One fact that must be pointed out is about the register 6 in the matrix multiplication. Our approach says it is the second most critical when it is not that much (match level = 9). The reason for this is that register 6 has actually no much effect in the program outputs, but as it has a long lifetime, our metric see it as critical. There is a loop (whose limits are unknown in compilation-time) between the write and read of register 6. In this way, the lifetime of this register in the static approach is short because the positions of write and read are close but due the loop between such positions our dynamic approach sees a long lifetime, and consequently, a high criticality is assigned to this register. Anyway, as this kind of situation is not common, the average match level for all case-study applications is smaller in our proposal than in the static approach as shown in Table 3, which clearly shows the improvements of our approach in the state-of-the-art. However, the influence of the registers in the outputs is a topic for a future work to avoid counting these *dummy* registers within the resultant criticality estimation.

TABLE 3
AVERAGE MATCH LEVEL FOR THE MINIMIPS CASE STUDY

Program	Static Criticality Metric [12]	Dynamic Criticality Metric (Proposed)
BS	1.83	1.50
MM	1.88	1.75
nDFS	1.86	0.57
rDFS	2.60	1.00
TH	1.27	1.09
Average	**1.89**	**1.18**

Fig. 6 shows the correlation coefficient between the estimations and the error rate obtained in by fault injection. Four of five correlation coefficients from the proposed metric are greater than the coefficients obtained in the static metric, and the other one differs only by 3% from the static metric. The miniMIPS results support that the criticality estimations obtained using our metric represent better the expected criticality.

TABLE 2
CRITICALITY RESULTS FOR THE MINIMIPS CASE STUDY

Prog.	RX	Fault Injection		Static Criticality Metric [12]			Dynamic Criticality Metric (proposed)			Prog.	RX	Fault Injection		Static Criticality Metric [12]			Dynamic Criticality Metric (proposed)		
		Error Rate (%)	Crit. Rank	Est. Crit.	Est. Crit. Rank	Match Level [a]	Est. Crit.	Est. Crit. Rank	Match Level [a]			Error Rate (%)	Crit. Rank	Est. Crit.	Est. Crit. Rank	Match Level [a]	Est. Crit.	Est. Crit. Rank	Match Level [a]
BS	2	47.32	3	0.141	3	0	0.330	1	**2**	*nDFS*	1	9.11	5	0.010	7	2	0.012	6	**1**
	3	2.82	7	0.090	4	3	0.164	6	**1**		2	89.00	2	0.079	4	2	0.310	3	**1**
	4	61.00	2	0.267	1	1	0.315	4	**2**		3	89.62	1	0.117	3	2	0.318	1	**0**
	5	0.32	8	0.179	2	6	0.303	5	**3**		4	38.93	4	0.166	2	2	0.280	4	**0**
	6	62.90	1	0.076	5	4	0.330	2	**1**		5	5.75	6	0.049	6	0	0.112	5	**1**
	7	0.00	9-12	0.013	8-10	0	0.002	11	**0**		6	1.27	7	0.079	4	3	0.006	7	**0**
	8	0.00	9-12	0.013	8-10	0	0.014	10	**0**		31	61.62	3	0.204	1	2	0.314	2	**1**
	9	0.00	9-12	0.000	12	0	0.000	12	**0**	*rDFS*	1	8.73	8	0.007	10	2	0.008	9	**1**
	10	3.88	6	0.000	11	5	0.328	3	**3**		2	68.15	2	0.056	7	5	0.232	3	**1**
	12	8.64	4	0.064	6-7	2	0.137	7	**3**		3	69.66	1	0.083	6	5	0.239	2	**1**
	13	6.37	5	0.064	6-7	1	0.124	8	**3**		4	46.60	3	0.201	3	0	0.303	1	**2**
	14	0.00	9-12	0.013	8-10	0	0.025	9	**0**		5	13.60	7	0.035	8	1	0.105	6	**1**
MM	2	69.51	5	0.159	7	2	0.259	9	**4**		6	25.97	5	0.228	2	3	0.087	7	**2**
	3	85.53	3	0.263	1	2	0.321	3	**0**		7	18.96	6	0.133	5	1	0.204	5	**1**
	4	80.77	4	0.215	4	0	0.308	8	**4**		8	4.03	9	0.034	9	0	0.073	8	**1**
	5	47.31	9	0.110	8	1	0.228	10	**1**		sp	0.00	10	0.269	1	9	0.003	10	**0**
	6	6.28	11	0.018	13	2	0.326	2	**9**		31	34.39	4	0.144	4	0	0.211	4	**0**
	7	86.83	2	0.220	3	1	0.327	1	**1**	*TH*	2	60.08	3	0.242	2	1	0.316	3	**0**
	8	94.56	1	0.110	9	8	0.317	5	**4**		3	10.06	9	0.218	4	5	0.180	6	**3**
	9	0.00	13-16	0.018	14	0	0.021	15	**0**		4	27.87	4	0.197	5	1	0.208	5	**1**
	10	0.00	13-16	0.000	16	0	0.000	16	**0**		5	17.21	7	0.194	6	1	0.211	4	**3**
	12	64.74	8	0.196	6	2	0.314	7	**1**		6	92.21	1	0.301	1	0	0.330	1	**0**
	13	64.80	7	0.202	5	2	0.315	6	**1**		7	0.00	10-11	0.073	10	0	0.059	10	**0**
	14	65.45	6	0.257	2	4	0.321	4	**2**		8	20.46	5	0.122	7	2	0.081	9	**4**
	15	8.94	10	0.025	12	2	0.066	11	**1**		9	15.31	8	0.103	8	0	0.085	8	**0**
	16	0.00	13-16	0.006	15	0	0.024	14	**0**		10	62.99	2	0.241	3	1	0.321	2	**0**
	17	1.61	12	0.037	10	2	0.046	12	**0**		sp	0.00	10-11	0.054	11	0	0.000	11	**0**
	18	0.00	13-16	0.025	11	2	0.030	13	**0**		31	17.52	6	0.102	9	3	0.105	7	**1**

[a] ***Match level:*** *distance between each value of the estimated criticality rank compared to its respective expected value (based on fault injection results). 0 means an exact match between them (desirable).*

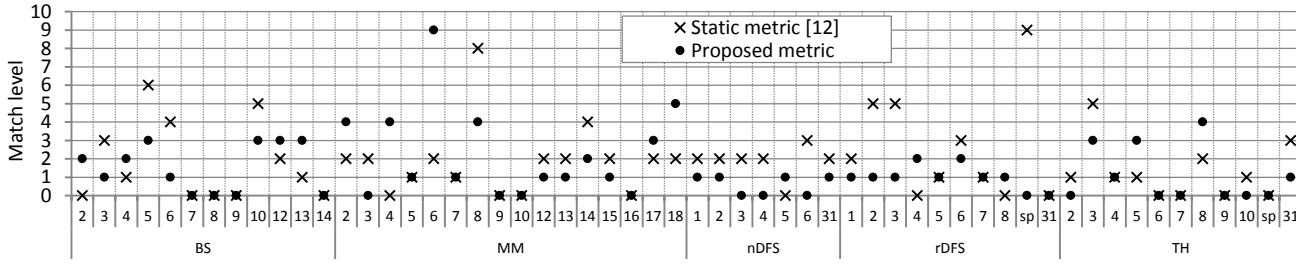

Fig. 5. Proposal vs. Static metric match levels in miniMIPS

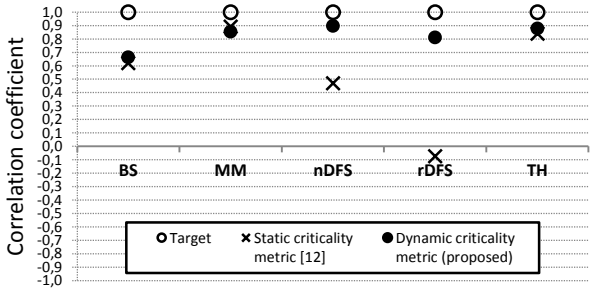

Fig. 6. Correlation coefficients between estimations and error rate in mini-MIPS

7 Conclusion

Given that providing early estimations of the register file criticality is a key task within the fault tolerant design of embedded systems, this paper presents an application-based metric to estimate the criticality of each register from the register file in microprocessor-based systems.

This metric facilitates the selection of the hardened set of registers when a selective hardening strategy is required in software, avoiding costly design space explorations guided by costly brute force strategies. Furthermore, early estimation of the register file criticality permits to perform preliminary reliability assessments before the system is fully implemented and fault injections can be carried out.

Experimental results demonstrate the applicability and accuracy of the proposed metric. Criticality estimations obtained using our metric represent the expected criticality (based on fault injection campaigns). Moreover, results showed that criticality estimations based on dynamic code analysis improve significantly the accuracy of results compared to metrics based on static code analysis.

This work opens up interesting new boundaries in the criticality analysis of resources in the design of fault tolerant embedded systems. Beside application-based criticality criteria, we will investigate new metrics taking into account more factors for the proper selection of the hardened set of registers, e.g., the influence of the registers in the output, expected time execution overheads.

Acknowledgment

This work was funded in part by the Spanish Ministry of Education, Culture and Sports with the project “Developing hybrid fault tolerance techniques for embedded microprocessors” (PHB2012-0158-PC).

References

[1] S. Hamdioui, M. Nicolaidis, D. Gizopoulos, A. Grasset, G. Guido, and P. Bonnot (2013) Reliability challenges of real-time systems in forthcoming technology nodes. Proc. *Design, Automation and Test in Europe (DATE '13)*, EDA Consortium, Pp. 129-134.

[2] M. Nicolaidis (2011) *Soft Errors in Modern Electronic Systems*. 1st Ed. *Frontiers in Electronic Testing Series*, vol. 41. Springer.

[3] M. Nicolaidis (2005) Design for soft error mitigation.*IEEE Trans. Device Mater. Rel.* 5(3):405-418.

[4] B. Nicolescu, Y. Savaria, and R. Velazco (2004) Software detection mechanisms providing full coverage against single bit-flip faults.*IEEE Trans. Nucl. Sci.*51(6):3510-3518

[5] J.R. Azambuja, S. Pagliarini, L. Rosa, and F. Lima Kastensmidt (2011) Exploring the Limitations of Software-based Techniques in SEE Fault Coverage, *J. Electron. Test.*27(4):541-550

[6] B. Pratt, M. Caffrey, J.F. Carroll, P. Graham, K. Morgan, M. Wirthlin (2008) Fine-Grain SEU Mitigation for FPGAs Using Partial TMR, *IEEE Trans. Nucl. Sci.*, 55(4):2274-2280

[7] O. Ruano, J.A. Maestro, P. Reviriego (2009) A Methodology for Automatic Insertion of Selective TMR in Digital Circuits Affected by SEUs, *IEEE Trans. Nucl. Sci.*, 56(4):2091-2102

[8] F. Restrepo-Calle, A. Martínez-Álvarez, S. Cuenca-Asensi and A. Jimeno (2013) Selective SWIFT-R: A Flexible Software-Based Technique for Soft Error Mitigation in Low-Cost Embedded Systems, *J. Electron. Test.,* 29(6):825-838

[9] E. Chielle, J.R. Azambuja, R.S. Barth, F. Almeida, F. Lima Kastensmidt (2013) Evaluating Selective Redundancy in Data-flow Software-based Techniques, *IEEE Trans. Nucl. Sci.*, 60(4):2768-2775

[10] S. Cuenca-Asensi, A. Martínez-Álvarez, F. Restrepo-Calle, F.R. Palomo, H. Guzmán-Miranda, M.A. Aguirre (2011) A Novel Co-Design Approach for Soft Errors Mitigation in Embedded Systems, *IEEE Trans. Nucl. Sci.*, 58(3):1059-1065

[11] A. Lindoso, L. Entrena, E. San Millan, S. Cuenca-Asensi, A. Martínez-Álvarez, F. Restrepo-Calle (2012) A Co-Design Approach for SET Mitigation in Embedded Systems, *IEEE Trans. Nucl. Sci.*, 59(4):1034-1039

[12] S. Bergaoui, P. Vanhauwaert, and R. Leveugle (2010) A New Critical Variable Analysis in Processor-Based Systems, *IEEE Trans. Nucl. Sci.*, 57(4):1992-1999

[13] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, T. Austin (2003) A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor, in Proc. *36th Int. Symp. on Microarchitecture., MICRO-36*. pp. 29-40

[14] S.Rehman, M.Shafique, F.Kriebel, and J. Henkel, (2011) Reliable software for unreliable hardware: embedded code generation aiming at reliability, In Proc.*7th IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '11),*pp. 237-246

[15] P. Giacinto, N. Wang, Z.Kalbarczyk, S. Patel, and R.Iyer (2005) An experimental Study of Soft Error in Microprocessors, *IEEE MICRO*, 25(6):30-39

[16] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri (2000) A C/C++ source to source compiler for dependable applications, in Proc. *IEEE Int. Conf. on Dependable Systems and Networks (DSN)*, pp. 71-78

[17] J. Lee, and A. Shrivastava (2011) Static Analysis of Register File Vulnerability, *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 30(4):607-616

[18] K. Pattabiraman, Z. Kalbarczyk, R.K. Iyer (2005) Application-based metrics for strategic placement of detectors," in Proc. *11th Pacific Rim Int. Symp. on Dependable Computing*, pp.8, 12-14

[19] V. Sridharan, and D.R. Kaeli (2008) Quantifying Software Vulnerability, in Proc. *Workshop Radiation Effects and Fault Tolerance in Nanometer Tech. WREFT*, pp. 323-328

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

[21] L. M. O. S. S. (2010) Hangout and S. Jan, TheMinimips Project [Online]. Availiable: http://www.opencores.org/projects.cgi/web/minimips/overview 2010.

[22] A. Martínez-Álvarez, S. Cuenca-Asensi, F. Restrepo-Calle, F.R. Palomo Pinto, H. Guzmán-Miranda, M.A. Aguirre (2012) Compiler-Directed Soft Error Mitigation for Embedded Systems, *IEEE Trans. Dependable and Secure Computing*, 9(2):159-172

[23] Mentor Graphics (2014) http://www.model.com/content/modelsim-support

[24] Antonio Martínez-Álvarez, Felipe Restrepo-Calle, Luis Alberto Vivas Tejuelo, Sergio Cuenca-Asensi (2013) Fault tolerant embedded systems design by multi-objective optimization, *Expert Systems with Applications*, 40(17):6813-6822

[25] Goloubeva, O., Rebaudengo, M., Reorda, M. S., & Violante, M. (2006). Software-Implemented Hardware Fault Tolerance (Vol. XIV). Springer.

[26] M. Portela-Garcia, A. Lindoso, L. Entrena, M. Garcia-Valderas, C. Lopez-Ongil, N. Marroni, B. Pianta, L. Bolzani Poehls, and F. Vargas (2012) Evaluating the Effectiveness of a Software-Based Technique Under SEEs Using FPGA-Based Fault Injection Approach. J. Electron. Test. 28(6): 777-789.

[27] Vargas F, Rocha CA, Farina A, de Alecrim AA Jr (2007) Embedded signature monitoring based on profiling deployed software technique. IEEE Int East–west Des Test Symp, Yerevan, Armenia, pp. 230–236.

[28] Oh, N.; Shirvani, P.P.; McCluskey, E.J. (2002) Control-flow checking by software signatures, *IEEE Trans. Rel.*, 51(1):111,122.