Evaluating Selective Redundancy in Data-Flow Software-Based Techniques

Eduardo Chielle, José Rodrigo Azambuja, Raul Sério Barth, Felipe Almeida, and Fernanda Lima Kastensmidt, *Member, IEEE*

***Abstract*—This paper presents an analysis of the efficiency of using selective redundancy applied to registers in software-based techniques. The proposed selective redundancy chooses a set of allocated registers to be duplicated in software in order to provide detection of upsets that occur in the processor hardware and provokes data-flow errors. The selective redundancy is implemented over miniMIPS microprocessor software. A fault injection campaign is performed by injecting single event effect upsets in the miniMIPS hardware. Results show error detection capability, performance degradation and program memory footprint for many case studies. With that, designers can find the best trade-off in using selective redundancy in software.**

***Index Terms*—Fault tolerance, microprocessors, selective redundancy, soft errors, software-based techniques.**

I. Introduction

THE last decade of advances in semiconductor industry have led to the fabrication of high density integrated circuits (ICs). However, the same technology also brought new challenges. Transistor scaling reduced voltage operation and increased clock frequencies, which have made ICs more susceptible to upsets caused by radiation. Such upsets can be caused by energized particles present in space or secondary particles such as alpha particles, generated by the interaction of neutrons and materials at ground level [1]. The interaction with an off-state transistor's drain in the PN junction may charge or discharge a node of the circuit. The consequence is a transient pulse in the circuit logic, also known as Single Event Effect (SEE). SEE can be potentially destructive, known as hard errors, or non-destructive, know as soft errors [2]. In this work, we focus on soft errors.

Soft errors affect microprocessors by modifying values stored in memory elements (such as registers and data memory). Such faults may lead the processor to incorrectly execute an application by jumping or re-executing some instructions, or even by entering in a loop and never finishing the application. Such faults can also modify some computed data values, generating errors in the data results.

The use of fault detection techniques in microprocessor to tolerate soft errors is mandatory. A known approach to detect SEEs in processors is by means of software-based techniques. These techniques rely on adding instruction redundancy and comparison in software to detect faults in the microprocessor hardware that generate errors in the execution or in the data results. The software-based techniques can be divided in two main groups: (1) the ones able to detect errors in the program execution [3]–[6], also known as control-flow errors, and (2) the ones able to detect faults in the data stored in the microprocessor [7]–[10], also known as data-flow errors. Also, techniques have been proposed to combine both detections with optimizations [11], [12].

Software-based techniques are non-intrusive and therefore provide high flexibility and low development time and cost. In addition, they allow the use of commercial off-the-shelf COTS microprocessor, since no modifications internal to the microprocessor are needed. Also, new generations of microprocessors that are not radiation hardening by design can be used. However, software-based techniques require additional processing time and therefore cause performance degradation and additional program and data memory usage. Although software-based techniques usually present high detection coverage in data-flow errors, it is not possible to fully detect control-flow errors generated by faults in the processor by using purely software-based techniques. The use of watchdogs and concurrent blocks are needed to increase the detection of errors in the control flow [13], [14].

We define register hardening as the act of detecting an error in a register and leading the program flow to the appropriate error handling routine. It does not imply in error correction, but in the first step to correct and error, which is error detection and handling. Software-based techniques require the use of spare registers to store global system signatures and to duplicate all allocated registers. However, some applications once compiled require more than half of registers available in the register file in the processor. Therefore, in those cases, not all allocated registers can be duplicated in software. In such cases, a subset of registers must be chosen to be duplicated, which defines the selective redundancy technique. Consequently, the application can be recompiled using fewer registers. However, this solution may lead into performance degradation due a higher number of memory accesses and register management.

The selection of registers to be duplicated in software is an important task and affects directly the error detection rates. If the registers are randomly chosen, the program may get lower detection rates comparing to a more smart selection of registers based on the application behavior. Also, the execution time of an algorithm as well as its program memory footprint depends on the selection of the right registers. Thus, an analysis of each

Manuscript received September 17, 2012; revised January 10, 2013 and March 15, 2013; accepted March 22, 2013. Date of publication July 16, 2013; date of current version August 14, 2013. This work was supported in part by CNPq and CAPES Brazilian agencies.

The authors are with the Instituto de Informática, PPGC and PGMICRO, Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre 91509-900, Brazil (e-mail: echille@inf.ufrgs.br; jrazambuja@inf.ufrgs.br; rsbarth@inf.ufrgs.br; falmeida@inf.ufrgs.br; fglima@inf.ufrgs.br).

Digital Object Identifier 10.1109/TNS.2013.2266917

register used by the application must be done in order to improve the system performance, program memory footprint and error detection. In this work, the presented analysis and selective redundancy methodology focus in errors affecting the data (data-flow errors). Therefore, a technique to detect control-flow errors should be used with the proposed strategy to fully protect the microprocessor against soft errors.

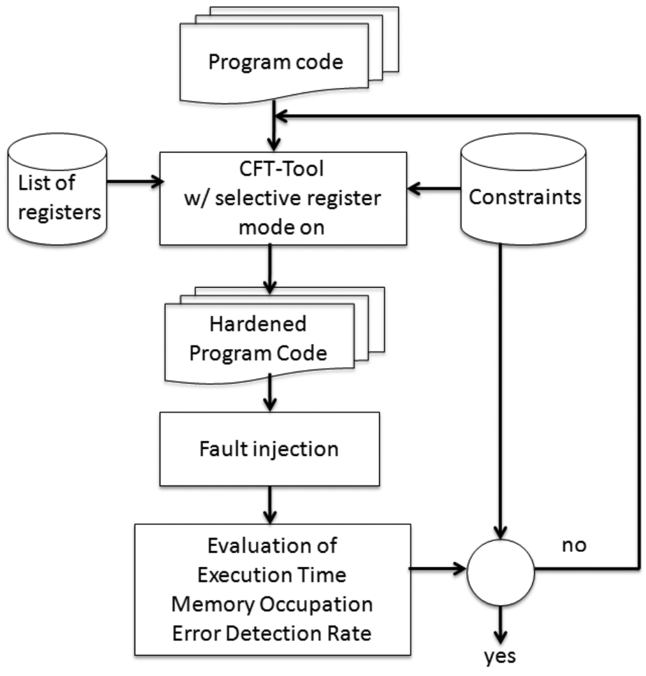
This paper performs an analysis of the efficiency of using selective redundancy applied to registers duplication in software-based techniques. The technique is implemented over miniMIPS microprocessor software. First, we verify the influence of duplicating each register according to performance degradation and memory footprint. Furthermore, registers are grouped in order to check their scalability. A fault injection campaign is performed by injecting single event effect upsets in the miniMIPS hardware. Results show error detection capability, performance degradation and program memory footprint for many case studies. With that, designers can find the best trade-off in using selective redundancy in software. Consequently, the proposed selective redundancy chooses a set of allocated registers to be duplicated in software in order to provide detection of upsets that occur in the processor that provokes data-flow errors.

II. Software-Based Hardening Techniques and Proposed Selective Redundancy Methodology

Fault tolerant software-based approaches combine techniques to detect both control and data-flow errors. Data-flow errors can be data stored in the registers or in the memory. The software-based techniques to detect such errors are based on duplicating all the registers allocated by the compiler by using the spare registers. Checkers are inserted in some parts of the code to verify the consistency of the data. Software-based techniques to detect control-flow errors aim at detecting incorrect jumps in the program execution by assigning a unique signature to each block of instructions. These techniques assign the signature during runtime to one spare register and insert invariant checking instructions into the original program code. By doing so, they are able to detect whether a branch was correctly taken or not.

An example of software-based techniques that combines different approaches to detect both control and data-flow errors is the SWIFT technique [11], [12]. It combines the Error Detection by Diverse Data and Duplicated Instructions (ED4I) technique [8] to detect the data-flow errors with Control-Flow Checking by Software Signatures (CFCSS) technique [3] to detect the control flow errors. It is well known that software-based techniques require additional processing time and therefore cause performance degradation and additional program and data memory usage. Solutions to reduce the time and memory overhead can rely on selective redundancy and combinations of software and hardware techniques proposed in the literature [15], [16].

In this work, we aim at investigating the effect of selective redundancy in software for data-flow detection in order to reduce the execution time and memory footprint overheads. As mentioned previously, software-based technique to detect data-flow errors requires a considerably larger number of spare registers


Program code
List of registers
CFT-Tool w/ selective register mode on
Constraints
Hardened Program Code
Fault injection
Evaluation of Execution Time Memory Occupation Error Detection Rate
no
yes
Fig. 1. Proposed methodology.

than techniques to detect control-flow errors. Sometimes there are no sufficient available spare registers to duplicate all the used registers and recompiling the program using fewer registers is not always a good solution due to performance degradation. The performance degradation happens when recompiling the code because of more memory accesses and more instructions to manage the smaller amount of available registers. In such cases, a subset of registers must be chosen to be duplicated, which defines the selective redundancy technique.

The selection of registers to be duplicated in software is an important task and affects directly the error detection rate. If the registers are randomly chosen, the program may get lower detection rates comparing to a more smart selection of registers based on the application behavior. Also, the execution time of an algorithm as well as its program memory footprint depends on the selection of the right registers. Thus, an analysis of each register used by the application must be done in order to improve the system performance, program memory footprint and error detection.

A correct selection of the registers to be duplicated can increase the error detection rate. In some cases, when there are time, power or area constraints, a correct selection of the registers to be duplicated can give higher error detection rate than a random selection of the registers. Furthermore, the analysis of the error detection rate, execution time and memory footprint overheads of duplicating each register for different programs can show the best trade-off among them, considering the requirements.

Fig. 1 presents the design flow of the proposed methodology. The first step of the proposed methodology consists of the analysis of the effect of each duplicated register individually. By doing so, we can quantify the gain and the cost of protecting each register by software in terms of execution time, memory footprint and error detection capability when compared to the original version of the program. In a further step, we duplicate some sets of registers according to the results, in terms of execution time, memory footprint and error detection capability, obtained by each register individually. This is made in order to check how the overhead in performance, area and the error detection scale in a group of registers protected by software. Our goal with this methodology is to obtain the best cost-benefit in performance, area and error coverage according to the constraints imposed by the needs of the system.

III. Implementation

In order to implement the described methodology, we started by choosing a target microprocessor. The chosen microprocessor is the miniMIPS [17], based on the MIPS architecture. It is a well-known microprocessor with a register file of 32 32-bit registers. We used the gcc 2.3 cross-compiler to compile the applications to the miniMIPS microprocessor.

Five applications were then chosen as case-study applications, which are: a matrix multiplication, a bubble sort, the TETRA Encryption Algorithm (TEA2), run-length encoding (RLE) and Dijkstra's algorithm.

We decided to duplicate registers, instead of variables, because the miniMIPS has sets of registers with specific purposes. Thus, we can address only part of the variable processing, i.e., it is possible to protect only the most critical part of the variable path (when its being passed as a function argument but not when its stored, for example). The same cannot be achieved by duplicating directly variables at high level code before the compilation.

In the matrix multiplication, two 3×3 matrixes are multiplied. It was chosen because there are many arithmetic instructions in it, which leads to more errors affecting the data. Around 70% of the errors are data-flow errors [6]. On the other hand, the bubble sort was chosen because it has many comparison instructions, which leads to more control-flow errors. Approximately 78% of the errors are control-flow errors [6]. TEA2 is an algorithm used to provide confidentiality to the TETRA air interface. This encryption algorithm protects against eavesdropping as well as protection of signaling [18]. The run-length algorithm is a simple algorithm for data compression. Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative edge path costs, producing a shortest path tree.

The chosen software-based technique is Variables 3 (VAR3) [10]. According to this technique, all variables used in the program are replicated. All operations performed on the original variables are also performed on their replicas, that is, the instructions are replicated, changing the original variables for their replicas, keeping both original variables and replicas with the same values. Finally, instructions to check the consistency are inserted in the program code. These instructions are inserted before instruction that load or store data from/to the memory and they compare the value in the source registers with the value in their replicas.

An example of the VAR3 technique is presented in Fig. 2. The original code, presented at the left side of Fig. 1, has three instructions: a *load* (line 2), an *add* (line 6) and a *store* (line 10). These instructions must be replicated, using the replicas of the

Original Code	Hardened Code

`2: ld r1, [r4]`	**`1: bne r4, r4′, error`**
	`2: ld r1, [r4]`
	`3: ld r1′, [r4′ + offset]`

`6: add r1, r2, r4`	`6: add r1, r2, r4`
	`7: add r1′, r2′, r4′`

`10: st  [r1], r2`	**`8: bne r1, r1′, error`**
	`9: bne r2, r2′, error`
	`10: st  [r1], r2`
	`11: st [r1′ + offset], r2′`

Fig. 2. VAR3 technique example.

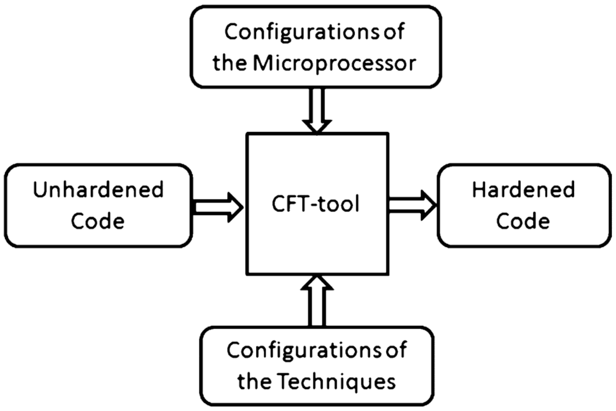

Configurations of the Microprocessor
Unhardened Code
CFT-tool
Hardened Code
Configurations of the Techniques

Fig. 3. Overview of the CFT-tool.

registers in place of the original registers, as can be seen in lines 3, 7 and 11, where they are replicas of the instructions in lines 2, 6 and 10, respectively. Before each instruction accesses the memory (*load* or *store*), the source registers are checked with their replicas, as presented in lines 1, 8 and 9.

In order to harden the case-study applications, we used the CFT-tool [19]. CFT-tool is a configurable tool capable of automatically apply some software-based fault tolerance techniques. These techniques are applied by CFT-tool to the assembly code of programs. The tool modifies the assembly code and inserts the selected techniques. The reason to use this tool is because it permits to select the techniques that will be applied to the program code and it also permits to select the registers that will be duplicated. In this way, we can customize how the program will be hardened. This enables us to create hardened codes according to the proposed methodology.

Fig. 3 presents an overview of the tool's workflow. This tool works independently of the architecture. It reads information about the microprocessor from configuration files. The microprocessor configuration file contains information about the microprocessor's architecture and organization, such as the instruction set architecture (ISA) and information about internal registers. The techniques configuration file contains information about the software-based techniques, i.e., which techniques were applied to the program's assembly code and which registers must be duplicated. After reading the configurations of the microprocessor and the configurations of the techniques, CFT-tool reads the original unhardened assembly code of the program and applies in it the selected techniques, duplicating only the selected registers and creating a hardened version of the program.

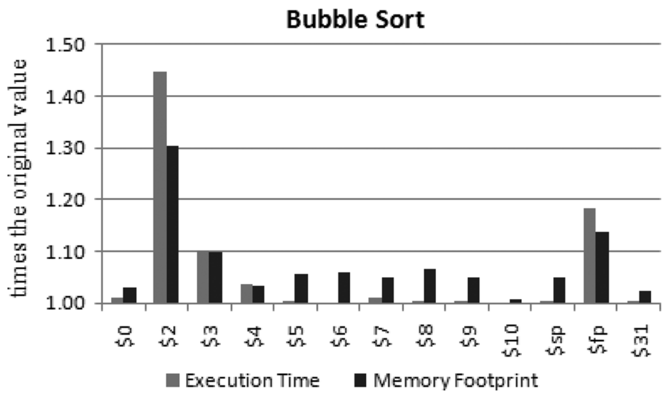


Fig. 4. Execution time and memory footprint for Bubble Sort.

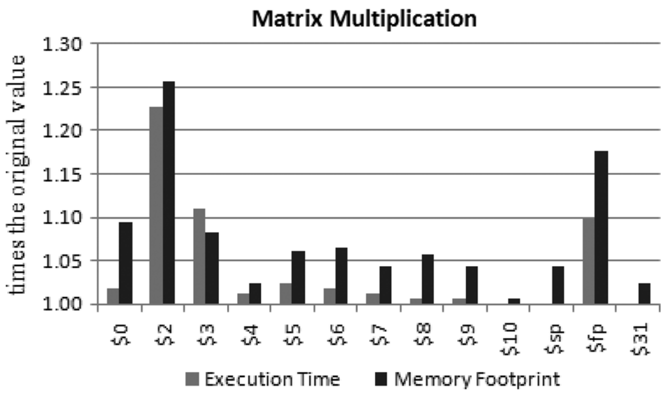


Fig. 5. Execution time and memory footprint for Matrix Multiplication.

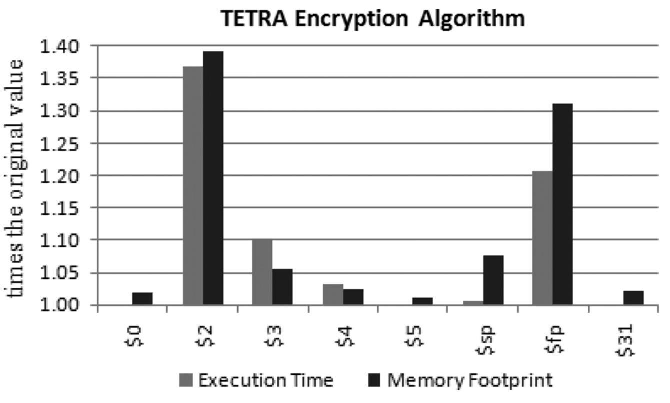


Fig. 6. Execution time and memory footprint for TETRA Encryption Algorithm.

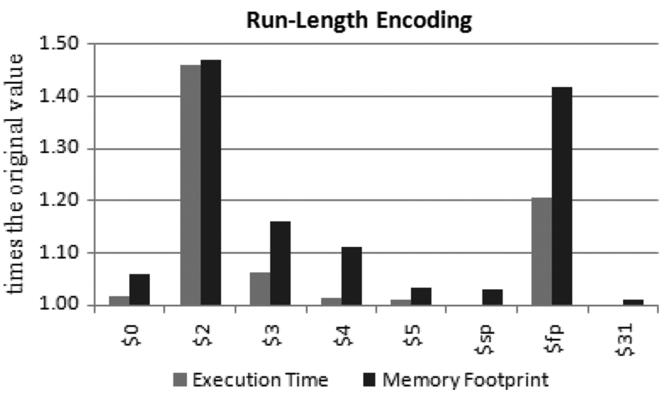


Fig. 7. Execution time and memory footprint for Run-Length Encoding.

IV. Evaluating Selective Redundancy in Registers

The programs were hardened using CFT-tool. For each used register, a hardened version of the program was created. As the overhead in execution time and memory footprint of a set of registers is the sum of the overheads of individual registers, only the versions where one register is duplicated are presented. The hardened programs were executed and compared to the unhardened version.

A. Execution Time

All the different hardened versions of the programs were executed. Their execution times were compared to the execution time of the unhardened version. Thus, it is possible to evaluate the overhead in terms of execution time for each hardened version of the programs.

Fig. 4 presents the execution time each version of the bubble sort. The register duplicated in software is indicated below the bars. As we can see, the duplication of register $2 generates an overhead of 45% in execution time, considerably greater than the duplication of the other registers. The duplication of register $fp presents 18% of overhead, the second greatest in execution time. The duplication of register $3 has a still considerably overhead in terms of execution time, 10%. The overhead in execution time caused by duplicating the other registers is very low.

The overhead in execution time for the matrix multiplication is shown in Fig. 5. The version that presents the greatest overhead is when register $2 is duplicated. The overhead in execution time in this case is 23%, considerably lower than the overhead in the bubble sort. It happens because register $2 is much more used in loops in the bubble sort than in the matrix multiplication. Instead of register $fp, it is register $3 that presents 11% of overhead in execution time, the second greatest. Register $fp is the third with 10% of overhead. Other registers present minor overheads.

In the TETRA encryption algorithm, presented in Fig. 6, the register $2 is the one that most influences the increase in execution time, 37% of overhead, followed by the register $fp, 21% of overhead. Register $3 has a still considerable influence in the execution time, 10% of overhead. These overheads are similar to the one presented by bubble sort. However, in this case, the register usage is due to a fewer number of used registers, instead of loops.

Fig. 7 presents the execution time for each version of the run-length encoding. As can be seen, the run-length encoding uses the same registers than the TETRA encryption algorithm. The overheads are 46% for register $2, 21% for register $fp and 6% for register $3. Other registers do not exceed 2% of overhead.

Dijkstra's algorithm uses less registers than the other case-study applications. Nevertheless, the duplication of register $2 continues to be the one which has more impact more the execution time, 37% of overhead, as seen in Fig. 8. As the matrix multiplication, register $3 presents the second greatest execution time, 18% of overhead, and register $fp presents the third greatest execution time, 15%. Other registers do not exceed 1%

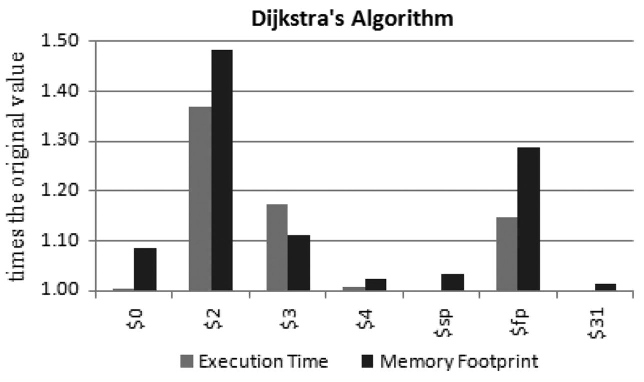


Fig. 8. Execution time and memory footprint for Dijkstra's Algorithm.

of overhead. Run-length algorithm and Dijkstra's algorithm are similar too TETRA encryption algorithm, but they have even fewer loops than TETRA.

In all case-study applications, registers \$2, \$3 and \$fp are the most critical in execution time. The reason for that is because register \$2 and \$3 are values for functional returns and expression evaluation, while register \$fp is the stack frame pointer.

B. Memory Footprint

Memory footprint refers to the amount of memory that a program uses while running. The memory footprint of each hardened version of the programs was gathered. Therewith, it is possible to quantify the overhead in terms of memory footprint, i.e., we can find how much the duplicated register increases the area occupied by the program in memory.

Fig. 4 shows the overhead in terms of memory footprint for each hardened version of the bubble sort. The duplicated register is indicated below the bars. The duplication of register \$2 creates 30% of overhead in memory footprint. The duplication of register \$fp presents 14% of overhead, the second greatest. The duplication of register \$3 has a still considerable overhead in terms of memory footprint, 10%, being the third in both cases. The overhead caused by duplicating the other registers is low, varying from 2% to 7%.

The overhead in memory footprint for the matrix multiplication is shown in Fig. 5. The greatest overhead in terms of memory footprint occurs when register \$2 is duplicated, presenting 26% of overhead. The register \$fp presents the second greatest overhead in terms of memory footprint, 18%. Register \$0 presents 9% of overhead and register \$3 presents 8%. The overhead for the other registers varies between 1% and 6%. Matrix multiplication uses the same number of registers than bubble sort and they are used by a similar number of instructions, which is why their overheads in the memory are close.

In the TETRA encryption algorithm, presented in Fig. 6, the register \$2 is the one that most influences the increase in memory footprint, 39% of overhead, followed by the register \$fp, 31% of overhead. Register \$sp presents 8% of overhead in memory footprint. Other registers do not exceed 5% of overhead.

Fig. 7 presents the overhead in memory footprint for each version of the run-length encoding. Register \$2 presents 47% of overhead. Register \$fp 42% of overhead. Also, register \$3 and register \$4 present considerably overheads, 16% of overhead for register \$3 and 11% of overhead for register \$4. Other registers present overheads varying from 1% to 6%.

The overhead in memory footprint for the Dijkstra's algorithm can be seen in Fig. 8. Register \$2 presents the greatest overhead in terms of memory footprint, 48%. Register \$fp presents 29% of overhead, register \$3 presents 11% of overhead and register \$0 presents 9% of overhead. Other registers do not exceed 3% of overhead in memory footprint. TETRA encryption algorithm, run-length algorithm and Dijkstra's algorithm present similar overheads in the memory footprint because they use the same number of registers.

V. Fault Injection Results

A fault injection campaign was performed to evaluate the error detection rate of each hardened version of the programs. Faults were injected at Register Transfer Level (RTL) in the entire miniMIPS microprocessor [17] using the ModelSim simulation tool. The location and time for each injected fault were randomly selected. One fault was injected per execution. All case-study applications were simulated, considering both individual and combined group of registers when duplicated. Only few sets of registers were combined. These sets were based on individual results of the registers.

For each hardened version of the case-study applications, 10,000 faults were injected. Faults were injected by forcing a bit-flip in the microprocessor's internal signals. Every signal of the microprocessor was considered. It includes the register file, the program counter, ULA, the control path, etc. The faults duration was set to the time of one and a half clock cycle. Thus, the fault effect reaches both clock edges (rising and falling) and increases the error probability. Only the faults that caused an error were taken in account, the faults masked by the logic of the microprocessor were ignored. The error is signaled when the result stored in the memory or the value of the PC during the execution differs from a golden execution. The results for each version were compared to the complete hardened version. Therefore, the relation between the overhead in execution time of each version and the overhead in execution time of the complete hardened version is presented. The same occurs for the memory footprint. The error detection rate of each hardened version is also compared to the error detection capability of the complete hardened version.

Fig. 9 shows the execution time, memory footprint and detection rate for each hardened version the bubble sort. The data are relative to the complete hardened version. Thus, it is possible to see the gains of the proposed methodology. As can be noticed, in the individual protection, where only one register was hardened, the best detection rate was achieved by register \$2. It reaches 76% of the complete hardened version. The overhead in execution time is 52% and in memory footprint is 32% of the obtained by the complete hardened version. Good cost-benefit in terms of error detection rate, execution time and memory footprint were obtained by the registers \$0, \$4 and \$fp in the individual protection. The register \$0 achieved 42% of error detection rate with only 1% of overhead in execution time and 3% of overhead in memory footprint. The register \$4 reaches 38% of error detection rate with a cost of 4% in execution time and

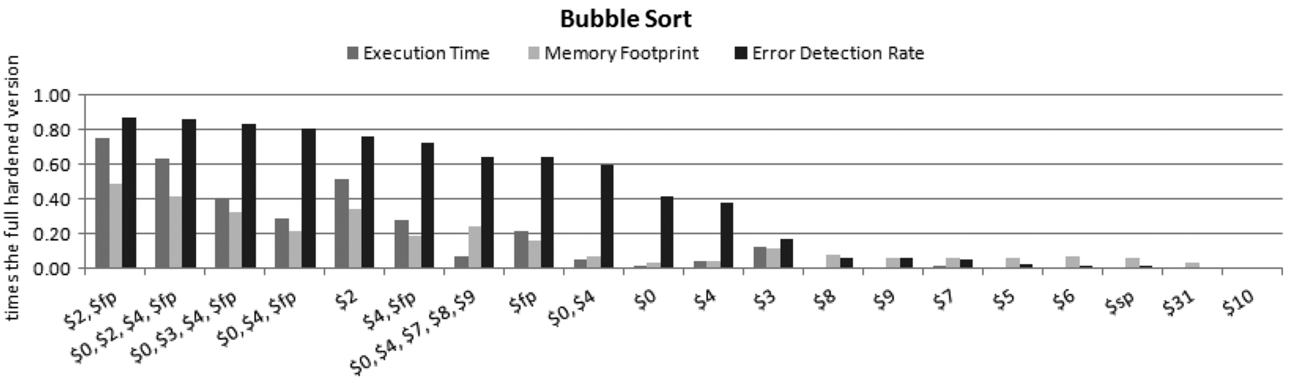


Fig. 9. Relative execution time, memory footprint and error detection rate for Bubble Sort.

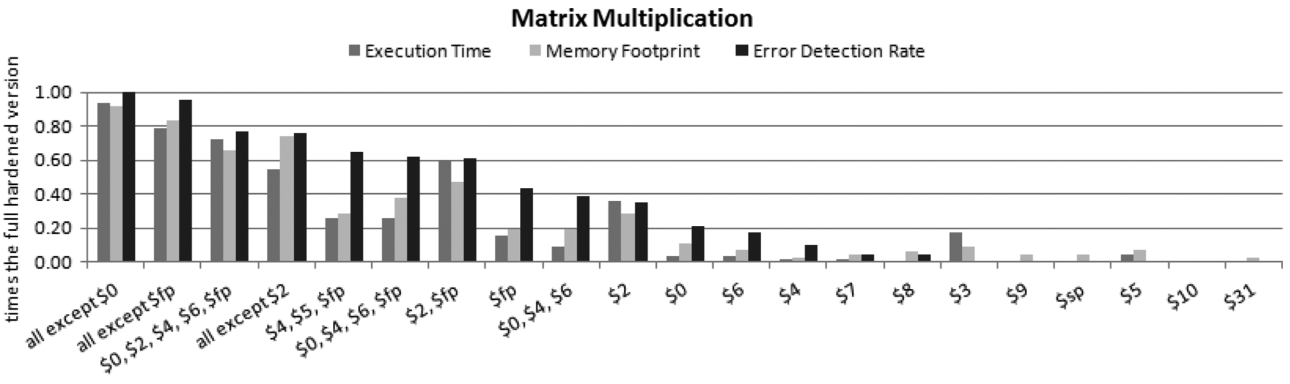


Fig. 10. Relative execution time, memory footprint and error detection rate for Matrix Multiplication.

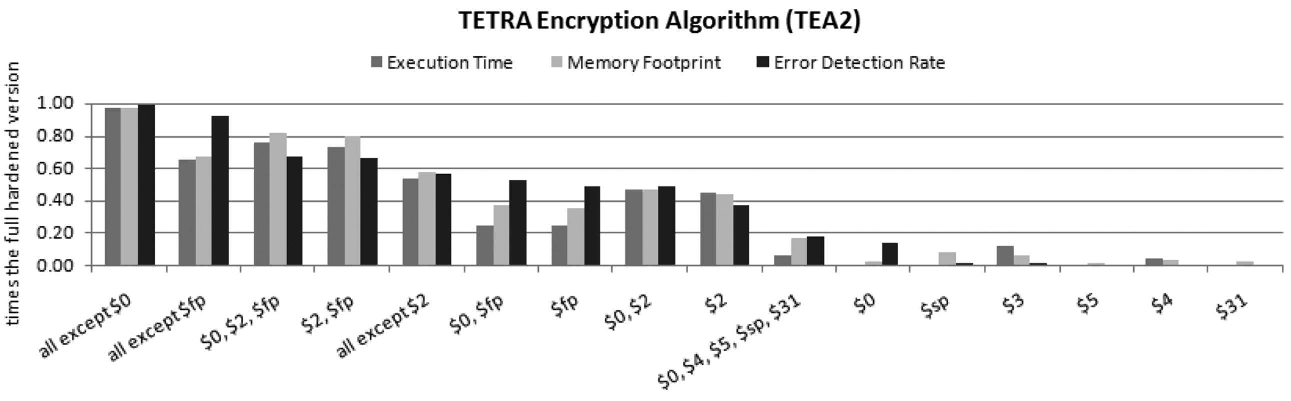


Fig. 11. Relative execution time, memory footprint and error detection rate for TETRA Encryption Algorithm.

memory footprint. And, the register $fp reaches 64% of error detection rate with a cost of 21% in execution time and 15% in memory footprint. For the group where a set of registers were duplicated in software, the combination of the registers $0, $4 and $fp reaches 80% of error detection rate with only 28% of execution time overhead and 22% of memory footprint overhead. And the combination of the registers $0 and $4 reaches 72% of error detection rate, with 28% of overhead in execution time and 19% of overhead in memory footprint.

The error detection of the individual protection is relatively lower for the matrix multiplication, as show in Fig. 10. We can highlight register $fp, where 43% of error detection rate was obtained with a cost of 16% of overhead in execution time and 20% of overhead in memory footprint. For the combined method, the set where all registers, except register $fp, were combined presented 95% of error detection rate with 78% of overhead in execution time and 84% of overhead in memory footprint. The combination of registers $4, $5 and $fp achieved 65% of error detection rate. In this case, the overhead in execution time is 25% and the overhead in memory footprint is 28%.

Fig. 11 shows the results of the fault injection campaign for the TETRA encryption algorithm. Among the individual protection, register $fp presents a good cost-benefit in terms of error detection rate, execution time and memory footprint. It achieved 49% of error detection rate, with 25% of overhead in execution time and 35% of overhead in memory footprint. In the combined method, the set where all registers, except register $fp, were duplicated, it presented 92% of error detection rate, with

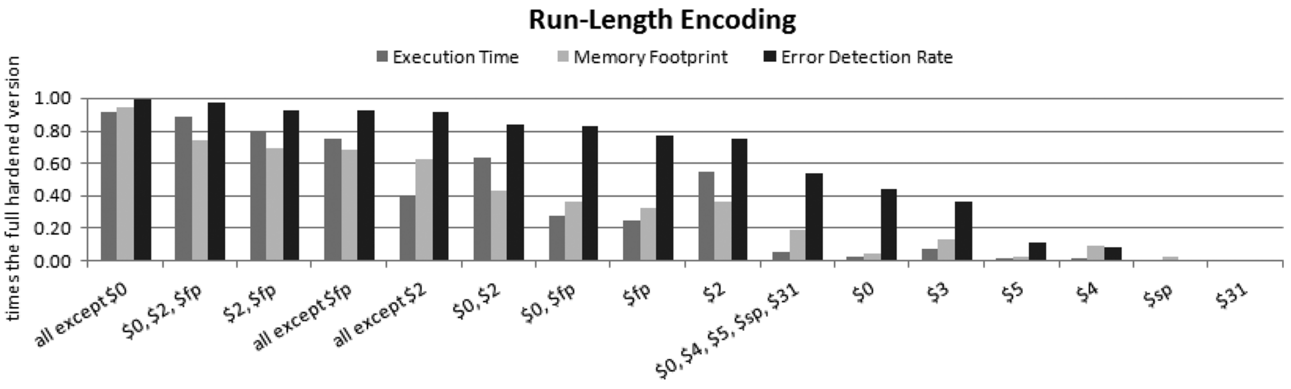


Fig. 12. Relative execution time, memory footprint and error detection rate for Run-Length Encoding.

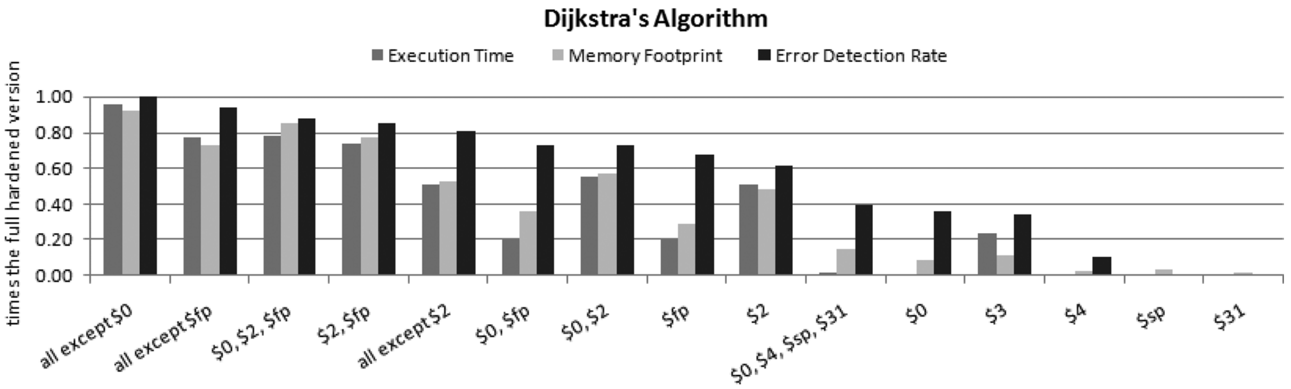


Fig. 13. Relative execution time, memory footprint and error detection rate for Dijkstra's Algorithm.

66% of overhead execution time and 68% in memory footprint. The combination of registers $0 and $fp presented 52% of error detection rate and 25% of overhead in execution time and 37% of overhead in memory footprint.

Fig. 12 shows the execution time, memory footprint and error detection rate for each version of the run-length encoding. As it can noticed, in the individual protection, the best detection rate was achieved by register $fp, where 77% of errors were detected. The overhead in this case was 25% in execution time and 33% in memory footprint. The version where register $2 was duplicated has reached a error detection rate of 75%. The overhead in execution time was 55% and in memory footprint was 37%. Register $0 and register $3 presented a good cost-benefit in terms of error detection rate, execution time and memory footprint. The register $0 achieved 44% of error detection rate with only 2% of overhead in execution time and 5% of overhead in memory footprint, when compared to complete hardened version. The register $3 reaches 37% of error detection rate with a cost of 8% in execution time and 13% in memory footprint. For the group where a set of registers were duplicated, the combination of the all registers, except register $2, the error detection rate was 92%, having only 40% of overhead in execution time and 62% of overhead in memory footprint. It shows that is possible to obtain 92% of the error detection rate achieved by the complete hardened version, having an overhead in execution time and memory footprint that less than half of the overhead achieved by the complete hardened version.

Fig. 13 shows the results of the fault injection campaign for the Dijkstra's algorithm. The characteristics presented in this application are quite similar to that one's presented in the run-length encoding. For the individual protections, the version where register $fp was duplicated presented the greatest error detection rate, 68%. Is this case, the overhead in execution time was 20% and the overhead in memory footprint was 29%. The version where register $2 was duplicated presented the second greatest error detection rate, 62%. The overhead was 51% in execution time and 49% in memory footprint. Register $0 presented 36% of error detection rate with negligible overhead in execution time and 9% of overhead in memory footprint. For the combined protection, the set of registers where all registers, except register $2, were duplicated presented 81% of error detection rate with 51% of overhead in execution time and 53% of overhead in memory footprint. The set where all registers, except register $fp, were duplicated achieved 94% of error detection rate with 77% of execution time overhead and 73% of memory footprint overhead.

VI. Conclusions and Future Work

This paper presented an evaluation of the impact of selective redundancy in software-based fault tolerant techniques to detect data-flow errors in microprocessors. Five case-study applications were chosen and hardened at software level only. Registers were individually duplicated in software and analyzed according to error detection rate, execution time and program memory footprint. A fault injection campaign was performed and 930,000 faults were injected by simulation in the miniMIPS microprocessor.

Depending on the program, all registers may not be duplicated, due to the lack of spare registers in the register file. In this case, a decision of which registers to be duplicated should be made. If the register $31 was randomly chosen, the error detection rate would be lower than if the registers $fp had been chosen by making the analysis proposed in this paper. The same principle can be extended when selecting a set of registers to adapt to performance constraints.

Results showed that a small number of registers are most required in the execution of a program. Therefore, a partial protection taking into account the registers most used, can achieve acceptable detection rates, without significant penalties of performance and area when compared to a full duplication of the registers. On the other hand, the influence of each register is not linear for the error detection rate, execution time and memory footprint, this way, permitting to substantially decrease the overheads in the execution time and memory footprint with a little loss in the error detection rate.

For example, in the case application used in this work, if only one register could be duplicated, register $fp or register $2 should be selected to maximize the error detection rate. However, if performance and area are critical constraints, so register $fp is a better choice than register $2, because that presents smaller overheads. If there are enough spare registers to duplicate all used registers and the aim is to maximize the error detection rate, independently of the execution time and memory footprint, the best choice would be to duplicate all the used registers. If a lower overhead in execution time and memory occupation is important, the version where all the registers, except register $fp, are duplicated is the best one, since it presents an error detection rate close to the complete hardened version with smaller overhead in execution time and memory footprint, as shown in the results.

REFERENCES

[1] P. E. Dodd, M. R. Shaneyfelt, J. R. Schwank, and J. A. Felix, "Current and future challenges in radiation effects on CMOS electronics," *IEEE Trans. Nucl. Sci.*, vol. 57, no. 4, pp. 1747–1763, Aug. 2010.

[2] M. O'Bryan, Nov. 15, 2000, Radiation Effects and Analysis [Online]. Available: http://radhome.gsfc.nasa.gov/radhome/see.htm

[3] N. Oh, P. P. Shirvani, and McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.

[4] L. D. Mcfearin and V. S. S. Nair, "Control-flow checking using assertions," in *Proc. IFIP Int. Working Conf. Dependable Computing for Critical Applications (DCCA-05)*, Urbana-Champaign, IL, USA, Sep. 1995.

[5] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 627–641, Jun. 1999.

[6] J. R. Azambuja, A. Lapolli, L. Rosa, and F. L. Kastensmidt, "Detecting SEEs in microprocessors through a non-intrusive hybrid technique," *IEEE Trans. Nucl. Sci.*, vol. 58, no. 3, pp. 993–1000, Jun. 2011.

[7] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante, "Soft-error detection through software fault-tolerance techniques," in *Proc. IEEE Int. Symp. Defect and Fault Tolerance in VLSI Systems*, 1999, pp. 210–218.

[8] N. Oh, S. Mitra, and E. McCluskey, "ED4I: Error detection by diverse data and duplicated instructions," *IEEE Trans. Comput.*, vol. 51, no. 2, pp. 180–199, 2002.

[9] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: Method, tools and experimental results," in *Proc. Design, Automation and Test in Europe Conf. and Exhib.*, 2003.

[10] J. R. Azambuja, A. Lapolli, M. Altieri, and F. L. Kastensmidt, "Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors," in *Proc. IEEE Latin Amer. Symp. Circuits and Systems*, 2011.

[11] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. S. Reorda, and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," *IEEE Trans. Nucl. Sci.*, vol. 47, no. 6, pp. 2231–2236, Dec. 2000.

[12] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proc. Symp. Code Generation and Optimization*, 2005, pp. 243–254.

[13] C. Bolchini, A. Miele, F. Salice, and D. Sciuto, "A model of soft error effects in generic ip processors," in *Proc. IEEE Int. Symp. Defect and Fault Tolerance in VLSI Systems*, 2005.

[14] J. R. Azambuja, S. Pagliarini, L. Rosa, and F. L. Kastensmidt, "Exploring the limitations of software-only techniques in SEE detection coverage," *J. Electron. Test.*, no. 27, pp. 541–550, 2011.

[15] S. Cuenca-Asensi, A. Martínez-Álvarez, F. Restrepo-Calle, F. R. Palomo, H. Guzmán-Miranda, and M. A. Aguirre, "A novel co-design approach for soft errors mitigation in embedded systems," *IEEE Trans. Nucl. Sci.*, vol. 58, no. 3, pp. 1059–1065, Jun. 2011.

[16] A. Lindoso, L. Entrena, E. S. Millán, S. Cuenca-Asensi, A. Martínez-Álvarez, and F. Restrepo-Calle, "A co-design approach for SET mitigation in embedded systems," *IEEE Trans. Nucl. Sci.*, vol. 59, no. 4, pp. 1034–1039, Aug. 2012.

[17] L. M. O. S. S. Hangout and S. Jan, The Minimips Project [Online]. Available: http://www.opencores.org/projects.cgi/web/minimips/overview 2010

[18] D. W. Parkinson, "TETRA security," *BT Technol. J.*, vol. 19, no. 3, pp. 81–88, 2001.

[19] E. Chielle, R. S. Barth, A. C. Lapolli, and F. L. Kastensmidt, "Configurable tool to protect processors against SEE by software-based detection techniques," in *Proc. IEEE Latin Amer. Symp. Circuits and Systems*, 2012.