Improving Error Detection with Selective Redundancy in Software-based Techniques

Eduardo Chielle, José R. Azambuja, Raul S. Barth, Fernanda L. Kastensmidt
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
{echielle, jrfazambuja, rsbarth, fglima}@inf.ufrgs.br

Abstract— **This paper presents an analysis of the impact of selective software-based techniques to detect faults in microprocessor systems. A set of algorithms is implemented, compiled to a microprocessor and selected variables of the code are hardened with software-based techniques. Seven different methods that choose which variables are hardened are introduced and compared. The system is implemented over a miniMIPS microprocessor and a fault injection campaign is performed in order to verify the feasibility and effectiveness of each selective fault tolerance approach. Results can lead designers to choose more wisely which variables of the code should be hardened considering detection rates and hardening overheads.**

I. INTRODUCTION

Integrated circuits fabricated with nanometric dimension technology operating in high frequencies with low voltage supply have become more sensitive to transient faults. Such faults 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]. Single Event Effect (SEE) faults are one of the major effects that may occur when a single radiation ionizing particle strikes the silicon. It may charge or discharge a node of the circuit by interacting with the drain in the PN junction of an off-state transistor. The consequence can be a transient pulse in the combinational logic, defined as Single Event Transient (SET), or a bit-flip in a memory element, defined as Single Event Upset (SEU).

SEEs can affect microprocessor systems by modifying values stored in memory elements (such as registers and data memory) or by changing the way the microprocessors responds to a branch instruction. Such faults can cause an error in the output registers of the microprocessor or even cause incorrect jumps inside the program execution. In order to harden the microprocessor against SEEs, the use of fault tolerance techniques is mandatory.

There are two types of fault tolerance techniques that aim to harden microprocessors, which are: hardware-based techniques and software-based techniques. The first one relies on replicating or adding hardware modules, while the second relies on the replication of information and instructions in the program code.

Hardware-based techniques usually change the original microprocessor by adding logic redundancy, error correcting codes and majority voters. They can also be based on hardware monitoring devices that exploit special purpose hardware modules, called watchdog processors [2], to monitor memory accesses. Watchdogs usually have access to data and code stored in memory. Since they only have access to the memory, watchdogs do not detect faults that are latent inside the microprocessor or faults in the register bank.

Software-based techniques rely on adding instruction redundancy and comparison to detect or correct errors. They provide high flexibility, low development time and cost, and do not require modifications in the hardware. In addition, new generations of microprocessors that do not have RadHard versions can be used. As a result, aerospace applications may use commercial off-the-shelf (COTS) microprocessors with RadHard software. However, software-based techniques require more resources, such as registers and memory.

Software-based techniques usually require spare registers to store global signatures or to duplicate all used registers. However, sometimes there are no sufficient spare registers to fully implement such techniques. In these cases, only a subset of all registers is chosen to be hardened. The selection of these registers is an important task and it affects directly the error detection rate. Also, the overheads in the execution time and the memory footprint of a program depend on the selected registers. Therefore, the set of registers to be hardened must be wisely selected in order to increase the system performance and fault tolerance.

This paper presents seven different methods with different approaches to select the registers to be hardened. Three case study applications are hardened with a software-based technique according to the methods. After that, the overheads in execution time and memory are compared. Furthermore, a fault injection campaign is performed. Detection rates are extracted from each method and then compared.

II. Related Works

Software-based techniques have been proposed in the past years. These techniques can be automatically applied to the source code of programs. They can be mainly divided in two groups: (1) aiming at detecting control flow errors [3, 4, 5, 6] and (2) aiming at detecting data flow errors [7, 8]. The first group aims to detect incorrect jumps in the program execution by assigning a unique signature to each block of instructions. These techniques then assign the signature during execution to one spare register and insert invariant checking instructions into the program code. By doing so, they are able to detect whether a branch in the program execution was correctly taken.

The second group aims to detect faults affecting the data. In order to do that, such techniques duplicate all the registers used by the application. By duplicating the registers, it is possible to compare them by adding checking. It is important to notice that every operation performed on a register must also be performed on its copy in order to keep the program consistent. The implementation of this type of technique requires a larger number of spare registers, when compared to techniques to detect control flow errors.

Besides the types of techniques presented above, there are techniques proposed in the literature as software-based techniques that aim to detect both data and control flow errors. One example is SWIFT [9], a technique that combines the Error Detection by Diverse Data and Duplicated Instructions (ED4I) [7] to protect the microprocessor against data flow errors with Control-Flow Checking by Software Signatures (CFCSS) [3] to protect against control flow errors.

III. Proposed Methodology

Software-based techniques aimed at control flow error usually required only one spare register or few spare registers, independently of application, while techniques that aim to harden the data flow require the same number of used registers. For this reason, techniques to harden the microprocessor against data errors present considerably greater overheads in execution time and memory footprint. Furthermore, sometimes there are no sufficient available spare registers to duplicate all the used registers. Therefore, the registers that are selected to be hardened influence considerably the overheads in execution time and memory footprint and the error detection rate. Considering that, we will only use data flow techniques is this work to be able to evaluate different methods to select the registers, intending to reduce overheads and keep error detection rate high even when there are few available spare registers.

The proposed methods are divided in two groups, which are: (A) static methods and (B) dynamic methods. The first group analyses the code statically, by considering only the source code. The second group considers the program execution. Data about the registers are collected. The groups and methods are described below.

A. Static Methods

The static methods consist in methods that analyze the program's source code. These methods read the program code and they count how many times each register was used, either as source register or target register. Depending on the method, source registers and/or target registers are considered. There are four different static methods, they are: Static First (SF), Static Target (ST), Static Source (SS) and Static Source and Target (SST).

The Static First method is a baseline method. This method is the way how most of the tools select registers to apply a data-flow software-based technique. They usually select the registers in the order they appear in the code. Therefore, in this method, registers are selected by the order they appear in the program code. The first registers that appears in the code has the highest priority, the second register has the second highest priority and so on. The Static Target method considers the target registers, i.e., the register that has more instructions attributing values to it is the most priority. The Static Source method counts how many times each register is used as a source registers, which means registers read by the instructions in the program code. It assigns higher priorities to registers more used as source register. The Static Source and Target method reads the source code and counts the number of times a register is used as source or as target register. So, it counts how many times each register is read or written by an instruction. Therefore, it gives higher priorities to the registers more used in the program code.

B. Dynamic Methods

Instead of analyze the source code, the dynamic methods analyze the program execution. The unhardened program is executed in the target microprocessor and data about the registers are extracted. This information consists in the number of times that each register was used as source register or target register. Counters are inserted in the code to gather these data. Two counters are assigned to each registers. One is incremented when the register is used as source register and the other is incremented when the register is used as target register. Based on this information, three methods are proposed, they are: Dynamic Target (DT), Dynamic Source (DS) and Dynamic Source and Target (DST).

The dynamic target methods are similar to the static methods. The difference consists in the origin of the information to select the register. While static registers consider the program's source code, the dynamic methods the program execution. The Dynamic Target method considers the importance of each register according to its target register counter, i.e., how many times the register was used as target register during the program execution. The Dynamic Source method set the priority of each register according to its source register counter, i.e., how many times the register was used as source register during the program execution. The Dynamic Source and Target method combines the Dynamic Source and Dynamic Target. It considers the sum of the target register counter and the source register counter, i.e., how many times the register was used during the program execution to select the registers.

IV. Implementation

In order to implement the methods described above, we started by choosing a target microprocessor, a set of

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'`

Figure 1. VAR technique example

applications and one data flow software-based fault tolerant technique. As mentioned before, we will not consider techniques to detect faults in the control flow, since they usually require only one spare register.

The chosen microprocessor is the miniMIPS [10], based on the MIPS architecture. It is a well-known microprocessor with a register bank of 32 32-bit registers. Three applications were chosen as case-study applications, which are: a 6x6 matrix multiplication, a bubble sort and the TETRA encryption algorithm. We used the gcc 2.3 cross-compiler to compile all three applications to the miniMIPS microprocessor.

The chosen software-based technique is the Variables 3 (VAR) [8]. The VAR technique duplicates every variable used by the application. Then, it performs the same operations that are performed over the original variables over the replicas, to keep both variables and replicas consistent. Finally, a consistency check is performed whenever a value is stored or loaded from the memory. An example can be seen on Fig. 1.

The original code has three instructions that operate with registers and memory elements. Instructions 1 and 3 are inserted to protect the load instruction located in position 2 (ld r1, [r4]), where the first instruction verifies the register containing the base address for the load instruction (r4) and its replica (r4'). The second instruction replicates the load instruction, using the replicated memory position (r4' + offset) and loads the value into the replicated register (r1'). Instructions 8, 9 and 11 are inserted to protect the store instruction. While instructions 8 and 9 verify values stored in the base and data registers (r1 and r2, respectively) with their replicas (r1' and r2', respectively). Instruction 11 replicates the original store instruction located in position 10 (st [r1], r2) using the replicated registers r1' and r2' in a replicated memory address (r1' + offset). The original add instruction located in position 6 (add r1, r2, r4) operates only using registers. The same operation is inserted below (line 7) using the replicated registers (r1', r2' and r4').

These transformations require the same number of spare registers as the originals one. Also, it requires double the size of the data memory. In order to harden the case-study applications, we used the CFT-Tool [11].

V. EVALUATING REGISTERS USAGE

As the dynamic methods need to know how many times each register was used during the execution, an analysis should be performed in the applications. Counters were inserted in the source code, using spare registers and the applications were simulated. Figs. 2, 3 and 4 show the register usage for bubble sort, matrix multiplication and encryption algorithms, respectively. The bubble sort and the matrix multiplication use a total of 12 registers, while the encryption algorithm uses 7 registers. It is important to mention that register $0 cannot be written and thus was not considered. For the static methods, the usage is how many times each register appears in the code and for dynamic methods is how many times each register is used during the program execution.

As can be seen in Fig. 2, register $2 is the most used as target register and also the most used as source register. Register $3 is the second one in usage as target register and the third in usage as source register. That happens due to register $fp, which is barely used as target, but it is the second most used as source register. The other registers have lower usage.

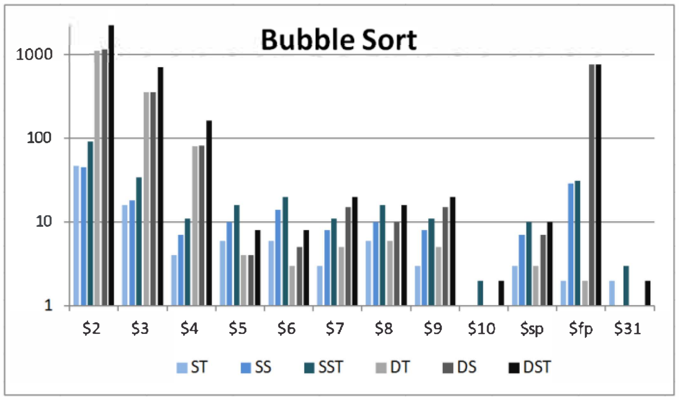

Figure 2. Register usage for the bubble sort

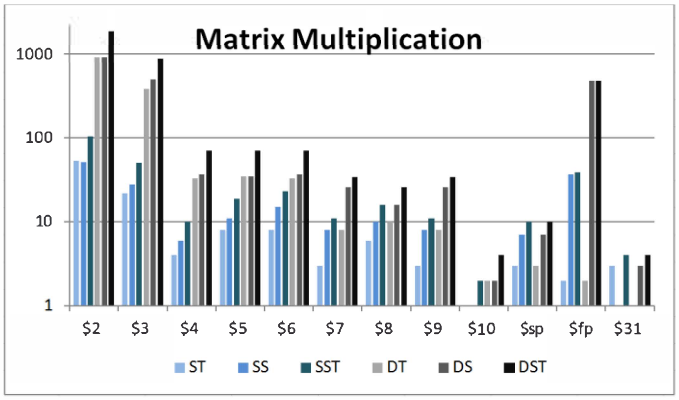

Figure 3. Register usage for the matrix multiplication

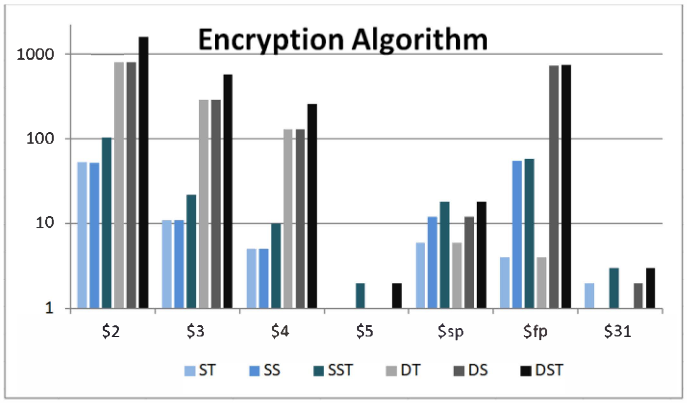

Figure 4. Register usage for the encryption algorithm

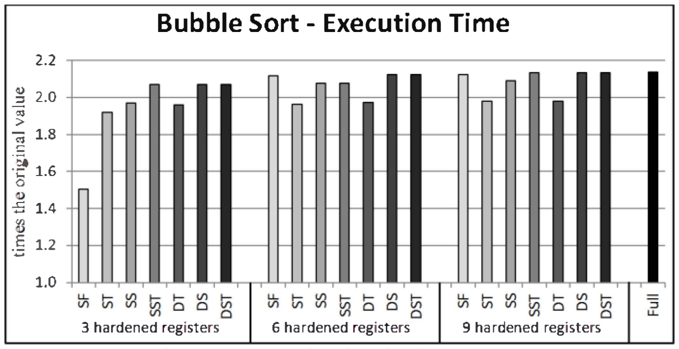

Figure 5. Execution time for the bubble sort

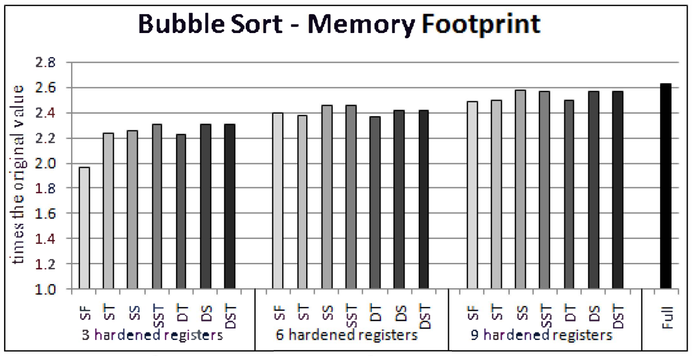

Figure 6. Memory footprint for the bubble sort

Fig. 3 shows the register usage for matrix multiplication. The usage is similar to bubble sort. Register $2 is the most used as target and source register. Registers $fp and $3 has a considerable usage. The other registers have lower usage.

The graphic presented in Fig. 4 shows the registers usage for the encryption algorithm, where 7 registers were utilized. As the bubble sort and the matrix multiplication, the most used registers are $2, $fp and $3.

These results show that only few registers have high usage, while the others are used few times. The few registers with high usage (source or target) represent almost the totality of the register usage. Therefore, we can infer that hardening a small set of registers should be enough to achieve high error detection rates. On the other hand, the same protection should result in high performance degradation and memory overheads.

VI. Simulation Results

As the target microprocessor has 32 registers, we could duplicate all used registers of the case-study applications. It allows us to fully explore the hardening design space. We created four different hardened versions of each program for each method, where the set of used registers that were hardened varies from 25% to 100% of the total of used registers.

As mentioned before, the bubble sort uses 12 registers. Thus, the four different hardened versions of the bubble sort for each method have 3, 6, 9 and 12 hardened registers, representing, respectively, 25%, 50%, 75% and 100% of the total of used registers. Fig. 5 shows the execution time overhead of all the methods implemented with the four different amounts of hardened registers. The program memory footprint can be seen on Fig. 6.

Except for SF method with 3 hardened registers, the execution times are between 1.9 and 2.2 times the execution time of the unhardened program. The methods ST and DT achieved the lower overheads. Small differences can be noticed when increasing from 3 to 6 hardened registers and from 6 to 9. That happens because the bubble sort uses 3 registers a lot more than the others in its execution. The memory footprint increases more linearly, according to the amount of hardened registers. This fact occurs because the memory footprint is related with the program's source code and not its execution.

The matrix multiplication case also uses 12 registers. Thus, we use the same amount of spare registers than the bubble sort, which are 3, 6, 9 and 12 hardened registers. Fig. 7 shows the overheads in the execution time. The overhead of most of the versions are between 1.6 and 1.8 times the unhardened version of the program. The memory footprint can be seen on Fig. 8. ST and DT methods achieved lower overheads. The results are similar to the bubble sort.

The third case-study application uses 7 registers. For this reason, the amounts of registers that were hardened in the four versions for each method were 2, 3, 5 and 7. That represents 28.6%, 42.9%, 71.4% and 100% of the total of used registers, respectively. Fig. 9 presents the overhead in the execution time of the different versions. The execution time increases from 2 to 3 hardened registers and 3 to 5 hardened registers. This fact occurs because there are four registers that are considerably used by the program. Fig. 10 shows the memory footprint overhead for each different version of the methods. As we can see, the same effects presented in the execution time are presented in the memory footprint. The memory footprint overhead increases linearly, according to the amount of hardened registers, like occur with the other case-study applications.

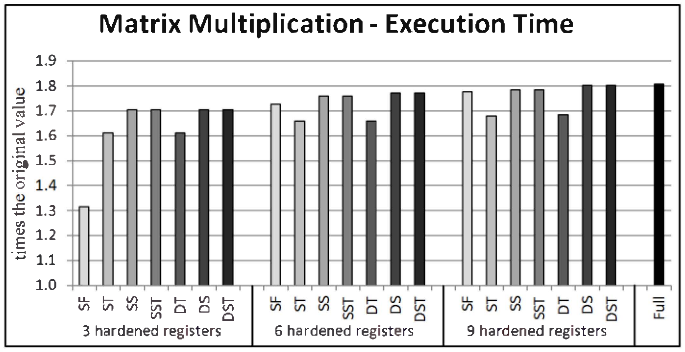

Figure 7. Execution time for the matrix multiplication

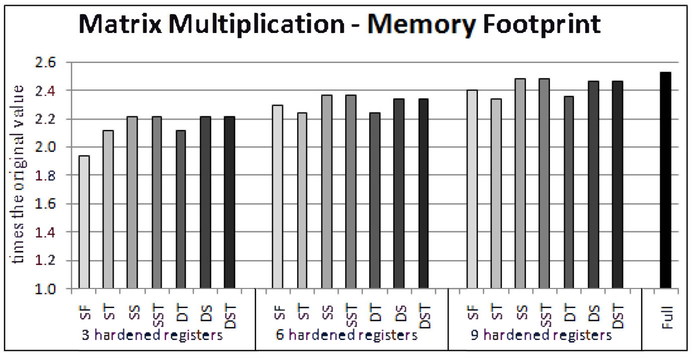

Figure 8. Memory footprint for the matrix multiplication

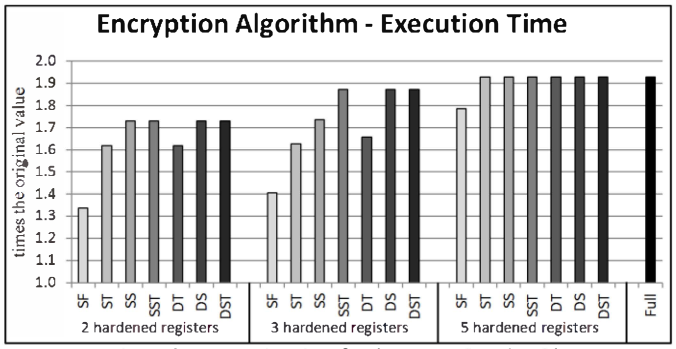

Figure 9. Execution time for the encryption algorithm

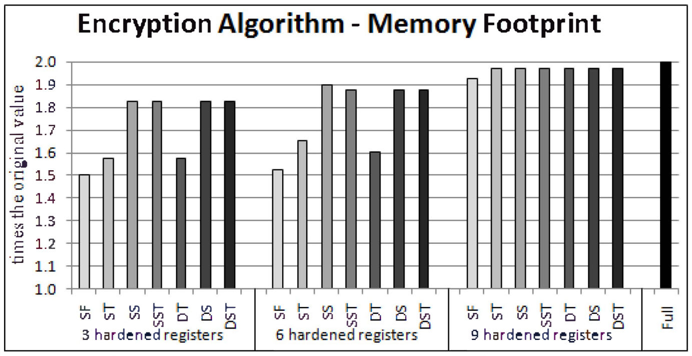

Figure 10. Memory footprint for the encryption algorithm

VII. Fault Injection Results

In order to evaluate the detection rate of each proposed method, a fault injection campaign was performed. Faults were injected at Register Transfer Level (RTL) using the miniMIPS microprocessor and the ModelSim [12] simulation tool. All versions for each proposed method for three case-study applications were simulated, were the percentage of hardened registers are between 25% and 100%.

Each version of the programs was simulated 10,000 times. In each simulation, a fault was injected by forcing a bit-flip in a microprocessor's internal signal. The faults duration were set to one and a half clock cycle. Thus, their effect can reach both clock edges (rising and falling) and increase the error probability.

Table I shows the detection rates for the bubble sort. The highest detection rate was achieved by the full hardened version, where 100% of the registers were hardened. The DS and DST methods achieved almost the same detection rate than the full hardened version, even when only half of the registers were hardened. With the dynamic methods, the error detection rate is higher than 92% even when there are only 25% of available spare registers to harden the used ones, while the SF method, that is how the tools normally select the registers, gets 75.5% of error detection rate.

Table II presents the results for the matrix multiplication. As presented in the bubble sort, the DS and DST methods achieved error detection rate close to the full hardened version, even when only half of the registers were hardened. The matrix multiplication has one more sensible register than the bubble sort. That is why the error detection rate decreases for the version where there are only three available spare registers. Comparing the proposed methods with the baseline method (SF), we can see a significant increase in the error detection rate when the same number of spare register is available to harden the used ones. For example, when there are only three available spare registers, the error detection rate for

Table I. Bubble Sort

Number of Hardened Registers	Method	Number of Errors	Detection Rate (%)
3	SF	852	75.5
	ST	911	89.4
	SS	1022	88.0
	SST	1158	93.9
	DT	1102	92.7
	DS	1169	92.2
	DST	1169	92.2
6	SF	1274	96.8
	ST	1045	92.1
	SS	1155	93.2
	SST	1141	93.9
	DT	1105	92.7
	DS	1295	97.3
	DST	1295	97.3
9	SF	1316	97.3
	ST	1070	93.2
	SS	1272	94.7
	SST	1317	97.2
	DT	1096	92.7
	DS	1311	97.6
	DST	1311	97.6
12	Full Hardening	1300	97.9

Table II. Matrix Multiplication

Number of Hardened Registers	Method	Number of Errors	Detection Rate (%)
3	SF	1022	57.6
	ST	1157	76.7
	SS	1194	72.4
	SST	1230	74.6
	DT	1230	74.6
	DS	1230	74.6
	DST	1230	74.6
6	SF	1315	82.4
	ST	1460	90.6
	SS	1633	91.1
	SST	1661	90.8
	DT	1439	88.7
	DS	1677	95.3
	DST	1677	95.3
9	SF	1558	95.2
	ST	1523	90.7
	SS	1666	92.7
	SST	1602	91.9
	DT	1557	90.8
	DS	1811	97.1
	DST	1811	97.1
12	Full Hardening	1773	97.7

the SF method is 57.6%, while the error detection rate for the other methods is above 72%.

Table III shows the error detection rate for each version of the program, where different amount of registers were hardened according to the proposed methods. The full hardened version, where all registers were hardened, achieved a 96.3% of error detection rate. The DS and DST methods achieved 96.1% of error detection rate, almost the same error detection rate than the full hardened version, when five registers were hardened. As the encryption algorithm uses only seven registers and four of them are more sensible, there is a reduction in the error detection rate when only three spare registers are used to harden the used ones. However, if compared with the baseline method (SF), the error detection rate is significant higher. While the SF method detects 56.8% of the errors using three spare registers and 56.4% using two spare registers, the dynamic methods (DT, DS, and DST) achieve 80% or more using three spare registers and around 67% using two spare registers.

VIII. Conclusions and Future Work

This paper presented an analysis of the impact of selective software-based fault tolerant techniques to detect faults in a microprocessor. Seven methods to selectively harden applications were presented, four static and three dynamic. One of these methods implements the way how the tools usually select the registers to apply a data flow software-based technique and it was used as baseline to evaluate the other methods. Furthermore, the methods using different amount of spare registers were also compared to a full hardened version of the case-study applications. Three case-study applications were chosen and hardened with different number of available spare registers using the proposed techniques. A fault injection campaign was performed and 660,000 faults were injected by simulation in the miniMIPS microprocessor.

Table III. TETRA Encryption Algorithm

Number of Hardened Registers	Method	Number of Errors	Detection Rate (%)
2	SF	1317	56.4
	ST	1156	66.4
	SS	1296	68.4
	SST	1390	67.1
	DT	1156	66.4
	DS	1390	67.1
	DST	1390	67.1
3	SF	1241	56.8
	ST	1148	65.3
	SS	1329	69.0
	SST	1419	80.0
	DT	1300	88.8
	DS	1419	80.0
	DST	1419	80.0
5	SF	1309	72.3
	ST	1589	96.2
	SS	1581	95.8
	SST	1638	94.3
	DT	1541	94.7
	DS	1564	96.1
	DST	1564	96.1
7	Full Hardening	1559	96.3

Simulations showed that few registers are responsible for the most of operations. Thus, it is possible to achieve higher error detection rate, even when there are few available spare registers. All proposed methods achieved higher error detection rate than the baseline method (SF). Dynamic methods showed an even higher error detection rate. DS and DST methods achieved error detection rates chose to the full hardened version, even when the amount of available spare registers is half of the total of used registers. Thus, the wise selection of the register to be hardened can influence significantly the fault tolerance of the application.

As future work, we intend to further analyze the influence of hardening selectively the register and identify how the hardening of each single register interfere the error detection rate. We will check the impact of partial selective fault tolerance techniques in terms of error detection rate, performance degradation and memory footprint.

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 Transactions on Nuclear Science, vol. 57, n. 4, august 2010, pp. 1747-1763.

[2] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors – a survey", IEEE Trans on Computers, vol. 37, issue 2, 1988.

[3] N. Oh, P.P. Shirvani, and McCluskey, "Control-flow checking by software signatures", IEEE Trans. on Reliability, vol. 51, Issue 1, March 2002, pp. 111-122.

[4] L.D. Mcfearin and V.S.S. Nair, "Control-flow checking using assertions", Proceedings of the IFIP International Working Conference Dependable Computing for Critical Applications (DCCA-05), Urbana-Champaign, IL, USA, September 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. on Parallel and Distributed Systems, vol. 10, Issue 6, June 1999, pp. 627-641.

[6] J. R. Azambuja, A. Lapolli, L. Rosa, F. L. Kastensmidt, "Detecting SEEs in microprocessors through a non-intrusive hybrid technique," in IEEE Transactions on Nuclear Science, Vol. 58, p. 993-1000, 2011.

[7] N. Oh, S. Mitra and E. McCluskey, "ED4I: error detection by diverse data and duplicated instructions", IEEE Transactions on Computers, vol. 51, n. 2, 2002, p. 180-199.

[8] J.R. Azambuja, A. Lapolli, M. Altieri, F.L. Kastensmidt, "Evaluating the efficiency of software-only techniques to detect SEU and SET in microprocessors", IEEE Latin American Symposium on Circuits and Systems, 2011.

[9] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan and D.I. August, "SWIFT: software implemented fault tolerance", Proceedings of the Symposium on Code Generation and Optimization, 2005, p. 243-254.

[10] L.M.O.S.S. Hangout and S. Jan. The minimips project, available online at http://www.opencores.org/projects.cgi/web/minimips/overview, 2010.

[11] E. Chielle, R.S.Barth, A.C. Lapolli and F.L. Kastensmidt, "Configurable Tool to Protect Processors against SEE by Software-based Detection Techniques", IEEE Latin American Symposium on Circuits and Systems, 2012.

[12] Mentor Graphics, http://www.model.com/content/modelsim-support, 2010.