Comparing Software-Based Fault Detection Techniques Applied at Different Abstraction Levels

Eduardo Chielle, Daniel Henrique Grehs, José Rodrigo Azambuja and Fernanda Lima Kastensmidt

Abstract— **This paper presents an analysis of software-based fault tolerance techniques applied at different abstraction levels. Four case-study applications are partially hardened using software-based fault tolerance techniques. Different partial hardened versions of the case-study applications are generated with varying levels of hardening. In the high level programming language, variables are individually hardened, while in the low level programming language, registers are individually hardened. A fault injection campaign is performed. Results show the performance, memory footprint and error coverage of each hardened version. Both approaches are compared and presented in detail.**

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

I. Introduction

THE recent advances in the semiconductor industry have led in the development of more complex components and systems' architectures by allowing fabrication processes to place a higher number of transistors per area of silicon die. The CMOS technology has developed at a pace where the number of transistors on Integrated Circuits (ICs) doubles every two years. Nowadays, with factories fabricating transistors with 32nm, we are reaching the physical limits of a couple atoms to form the transistor's gate [1][2]. However, the higher quantity of transistors per die combined with reduced voltage threshold and increased operating frequencies have made ICs more sensitive to faults caused by radiation [3]. Such faults can be caused by energized particles present in space or secondary particles such as alpha particles, generated by the interaction of neutron and materials at ground level [4].

Transient ionization may occur when a single radiation ionizing particle strikes the silicon, creating a transient voltage pulse, or a Single Event Effect (SEE). This effect affects microprocessors by modifying values stored in the sequential logic or by changing the function of a circuit in the combinational. Such faults may lead the system to incorrectly execute an application or even to enter a loop and never finish the application. In such cases, the use of fault tolerance techniques to protect the microprocessor against SEEs is mandatory.

Software-based techniques are a known approach to protect the system against SEEs, by modifying to program code, without having to change the underlying hardware. These 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) processors, since no modifications to the hardware are necessary. Also, new generations of processors without a radiation hardening by design can be used. On the other hand, they require processing time and therefore cause performance degradation and more program memory usage.

The software encompasses all the compiling phases, from the high level to the low level programming language and then to the executable program. At low level, microprocessors have special groups of registers to address different architecture functions, such as stack management and function arguments, while the high level has only variables. When protecting a variable at high level, the actual result in the low level is the protection of a group of registers during a time frame when they are used. By protecting a given register at low level, part of many variables will be hardened.

The literature presents different works to harden a microprocessor system by software at only at high level [5][6] or only at low level [7][8], but no work so far has presented software-based techniques applied considering high and low levels of abstraction. This work innovates by comparing both approaches and starting the discussion on combining both levels to achieve higher detection fault rates at lower costs on performance and memory footprint.

This paper performs an analysis of selective redundancy by means of software-based fault tolerance techniques applied at different abstraction levels. It verifies the influence of software based-techniques applied to a high level of abstraction (C code) and compares it to the same approach at a low level of abstraction (assembly code). In the high level, variables are hardened, while the in the low level, registers are hardened. Results are compared according to execution time overhead and memory footprint overhead. A fault injection campaign is then performed to evaluate the detection rates. Conclusions show that designers must analyze the program code in different level in order to achieve the best performance and error detection rates.

Manuscript received April 08, 2013. This work was supported in part by CNPq and CAPES Brazilian agencies.

Eduardo Chielle, Daniel Henrique Grehs and FernandaLima Kastensmidt are with the Instituto de Informática, PPGC and PGMICRO at Universidade Federal do Rio Grande do Sul (UFRGS), PortoAlegre, Brazil. (phone: +55 (51) 3308 7036) e-mail: {echille, daniel.grehs, fglima}@inf.ufrgs.br.

José Rodrigo Azambuja is with the Centro de Ciências Computacionais at Universidade Federal do Rio Grande (FURG), Rio Grande, Brazil. (phone: +55 (53) 3233 6500) e-mail: jrazambuja@furg.br

II. Methodology and Implementation

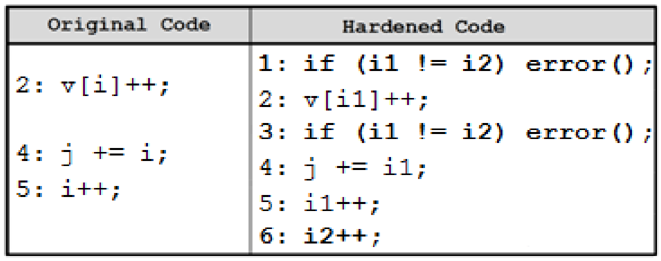
The proposed methodology consists of an analysis of the effects of hardening an application in a high level programming language and in a low level programming language. In the high level, the variables are individually hardened, using a software-based fault tolerance technique presented in Fig. 1. The variables are duplicated. Every operation performed in the original variable is also performed in its copy. To check the consistency, variables are tested before being read by an operation. If they present the same value, the execution is correct, otherwise, an error is detected and a subroutine to handle the error is called. The C language is used as the high level programming language in this work.

In the example presented in Fig. 1, only the variable *i* is duplicated. It is renamed to *i1* and a new variable called *i2* is created. All write operation in *i1* is replicated in *i2* (line 6). A comparison between the original variable *i1* and its copy *i2* is inserted before *i1* is read for any operation, as can be seen in lines 1 and 3.

Assembly was used as the low level programming language. In the assembly language approach, each register used by the program is individually hardened. The program is hardened using the CFT-tool [9]. CFT-tool is a configurable tool capable to automatically apply software-based fault tolerance techniques in assembly codes. It modifies the program's assembly code. Instructions are inserted in order to apply the selected techniques. This tool was chosen because it permits to select the registers that will be duplicated. By doing so, we can customize the protection of the program and perform analysis in the generated codes.

The software-based technique applied in the assembly codes is the *Variables 3* (VAR3) [10]. This technique duplicates the registers used in the program code. It then duplicates all instructions performed on the original registers to their replicas to keep both variables and replicas consistent. Finally, it performs consistency checks whenever a value is stored or loaded from the data memory.

An example of the VAR3 technique is presented at Fig. 2. The original code, presented at the left side of Fig. 2, has three instructions, a *load*, an *add* and a *store*. The instructions 3, 7 and 11 are replicas of the instructions 2, 6 and 10, respectively. Before loads and stores the value of the registers used by the instruction must be checked with their replicas. Only source registers are checked. These instructions to check the registers are presented in lines 1, 8 and 9. As the CFT-tool allows us to select the registers that will be duplicated, we can apply the technique only to one register, thus, protecting it individually.



Original Code	Hardened Code

`2: v[i]++;`	`1: if (i1 != i2) error();`
	`2: v[i1]++;`
	`3: if (i1 != i2) error();`

`4: j += i;`	`4: j += i1;`

`5: i++;`	`5: i1++;`
	`6: i2++;`

Fig. 1. High level language example.

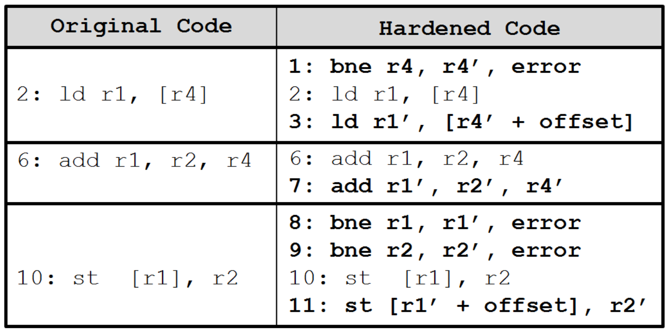
The target microprocessor is the miniMIPS [11]. It is a well-known microprocessor, based on the MIPS architecture, with a register bank of 32 32-bit registers. Four applications were chosen as case-study applications, which are: a matrix multiplication, a bubble sort, the TETRA Encryption Algorithm (TEA2) and the run-length encoding (RLE). These are simple application. Thus, we can easily map variables into registers and present a clearer comparison. The matrix multiplication has a lot of arithmetic operations, so it tends to present more errors in the data-flow. On the other hand, the bubble sort has many branch instructions, which leads to more control-flow errors [8]. 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 [12]. The run-length algorithm is a simple algorithm for data compression. We used the gcc 2.3 cross-compiler to compile the applications to the miniMIPS microprocessor.

III. Execution Time and Memory Footprint Evaluation

Partially hardened versions of the case-study application were created. Regarding the C language approach, a different hardened version of each application was created for each variable existing in the program, according to the methodology described in section II. Concerning the assembly approach, a different hardened version of each program was created for each used register. In each version only one register was hardened.

The matrix multiplication has 6 variables in the C language code (*a*, *b*, *c*, *i*, *j* and *v*) and it uses 13 registers in assembly (*$0*, *$2-$10*, *$sp*, *$fp* and *$31*). The bubble sort has 5 variables in the C language code (*aux*, *i*, *j*, *k* and *v*) and it uses 13 registers in assembly, same amount as the matrix multiplication. TEA2 has 11 variables in the C language code (*delta, i, k, k0, k1, k2, k3, sum, v, v0* and *v1*) and it uses 8 registers in assembly (*$0, $2-$5, $sp, $fp* and *$31*). And the run-length encoding has 11 variables in the C language code (*aux_count, byte1, byte2, count, histogram, i, in, inpos, insize, marker* and *outpos*) and it uses 8 registers, the same one that TEA2 uses.

Figs. 3(a-d) present the execution times for the matrix multiplication, bubble sort, TEA2 and run-length encoding, respectively. As one can see, for the matrix multiplication the highest overhead presented by the assembly approach occurs when register *$2* is duplicated. It is the most used register by



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.

this program. The execution time for this version is 1.23 times the execution time of the unhardened version. On the other hand, the highest overhead presented by the C language approach occurs when variable *c* is duplicated. Variable *c* is the output matrix, so it has a considerable impact in the execution time. The execution time in this case is 2.05 times the execution time presented by the unhardened version, which is more than when all used registers are duplicated [13]. Furthermore, register *$3* and *$fp* also present non negligible overheads. The execution time of the version that register *$3* is duplicated is 1.11 times the execution time of the unhardened version. And the execution time of the version that register *$fp* is hardened is 1.10 times the execution time of the unhardened version. The duplications of the other registers present overheads less than 2%. The overheads presented by the C language approach are higher. The lowest one is present by the duplication of variable *i*, where the execution time is 1.08 times the execution time of the unhardened version.

We can see similarities in the bubble sort overheads. But in this case, registers overheads are higher than registers overheads for the matrix multiplication and the variables overheads are lower than matrix multiplication variables overheads. The highest overhead for the assembly approach occurs when register *$2* is duplicated. The execution time is 1.45 times the execution time of the unhardened version. With regards to the C language approach, variable *j*, which is the variable of a internal loop, presents the highest execution time when hardened, it is 1.72 times the execution time of the unhardened version.

For the TETRA Encryption Algorithm the highest overhead presented by the assembly approach also occurs when register *$2* is duplicated. The execution time for this version is 1.37 times the execution time of the unhardened version. With regards to the C language approach, the highest overhead for this program occurs when variable *v0* or when variable *v1* is duplicated. They are not the output of the encryption algorithm, but they are part of the main calculation to define this output. That is the reason why they have a high impact on the execution time. The execution time in these cases are 1.52 times the execution time presented by the unhardened version. Furthermore, register *$3* and *$fp* also present non negligible overheads. The execution time of the version that register *$3* is duplicated is 1.11 times the execution time of the unhardened version. And the execution time of the version that register *$fp* is hardened is 1.21 times the execution time of the unhardened version. The duplications of the other registers present overheads less than 3%. The overheads presented by the C language approach are higher. Only variables *k* and *v* present very low overheads, 3% and 2%, respectively. The execution time of the variable *sum* is 1.35 times the execution time of the unhardened version. The other variables present overheads ranging from 9% to 14%.

The run-length encoding uses the same registers than the TETRA Encryption Algorithm and it also has the same amount of variables in the C language code. As one can see, the characteristics of the overhead in the execution time are similar. The highest overhead for the assembly approach is

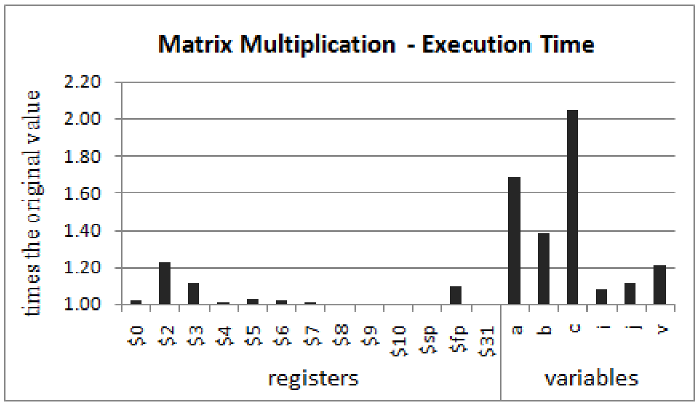


Fig. 3(a). Execution times for the matrix multiplication.

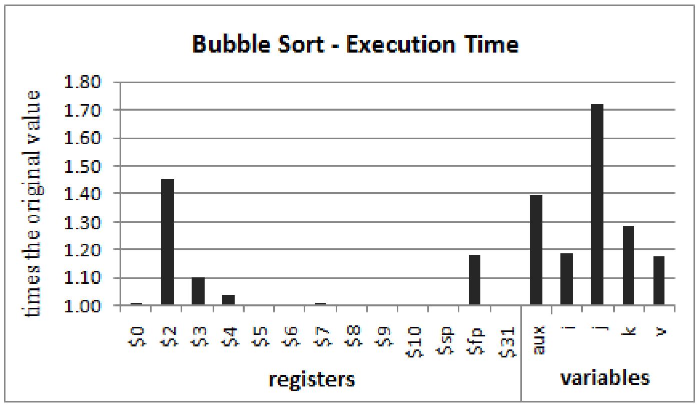


Fig. 3(b). Execution times for the bubble sort.

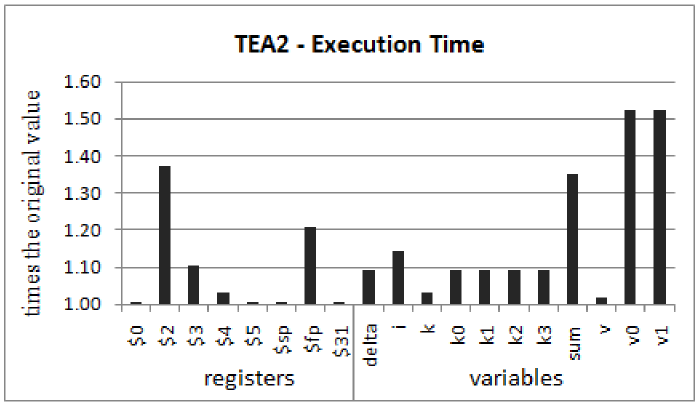


Fig. 3(c). Execution times for the TETRA Encryption Algorithm.

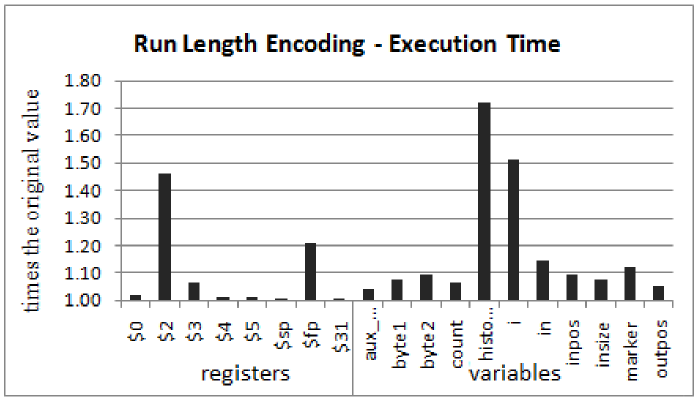


Fig. 3(d). Execution times for the run-length encoding.

also presented by register *\$2*. Its execution time is 1.46 times the execution time of the unhardened version. For the C language approach, the highest overhead occurs when variable *histogram* is duplicate. The execution time is this case is 1.72 times the execution time of the unhardened version. Registers *\$3* and *\$fp* also present non negligible execution time overheads. The execution time for the version when register *\$3* is duplicate is 1.06 times the execution time of the unhardened version. And for register *\$fp*, it is 1.21 times the execution time of the unhardened version. The other registers present overhead inferior to 2%. Concerning to the C language approach, variable *i* also present a high overhead in the execution time. Its execution time is 1.51 times the execution time of the unhardened version. The lowest overhead for the C language approach is presented by variable *aux_count,* 4%. The overhead of the other version range until 14%.

Figs. 4(a-d) present the memory footprints for the matrix multiplication, bubble sort, TEA2 and run-length encoding, respectively. With regards to the matrix multiplication, the highest overhead is presented by variable *c*, 31%. Concerning the assembly approach, register *\$2* presents the highest memory footprint overhead, 26%. Register *\$fp* presents 18% of overhead. The other registers present overheads between 1% and 9%. All the variables, except variable *c*, present overheads between 10% and 13%. Regards to the memory footprint, the overheads are more similar between C language approach and assembly approach than the overheads presented in the execution time. Concerning the bubble sort, the highest memory footprint overhead for the assembly approach is presented by register *\$2*, 30%. Other registers present overheads less or equal to 10%. For the C language approach, the highest overhead is presented by variable *v*, which is the output of the program. In this case, the overhead is 38%. Variable *j* presents 19% of overhead. The other variables present overheads between 7% and 9%. For the TEA2, the highest overhead is presented by register *\$2* in the assembly approach. It reaches 40% of overhead. Register *\$fp* presents 32% of overhead in the memory footprint. The other registers present overheads between 2% and 8%. With regards to the C language approach, variable *k* presents the highest memory footprint overhead, 27%, followed by variables *v0* and *v1*, both 20%. The other variables present overheads ranging from 6% to 18%. The individual overheads for the variables in this case is lower because there is more variables than registers. However, the total overhead present by duplicating all registers is 97%. If all variables were duplicated, the overhead in the memory footprint will be 143%. As the run-length encoding and TEA2 use the same registers and the same amount of variables, the memory footprints for them are quite similar to TEA2. Concerning the assembly approach, the highest overheads are presented by registers *\$2* and *\$fp,* 47% and 42%, respectively. Register *\$3* presents 16% of overhead in the memory footprint and register *\$4* presents 11%. The other registers overhead are inferior to 6%. For the C language

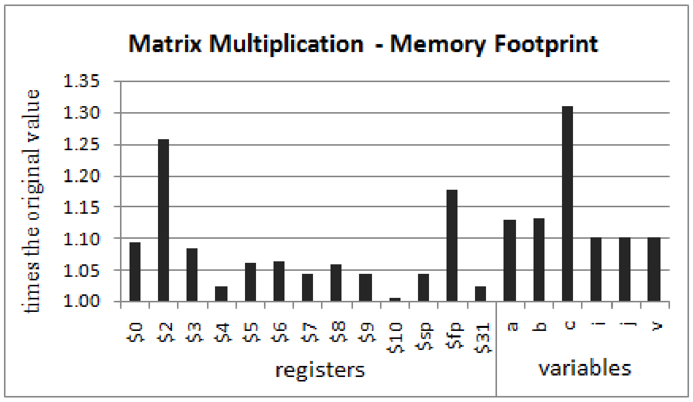


Fig. 4(a). Memory footprints for the matrix multiplication.

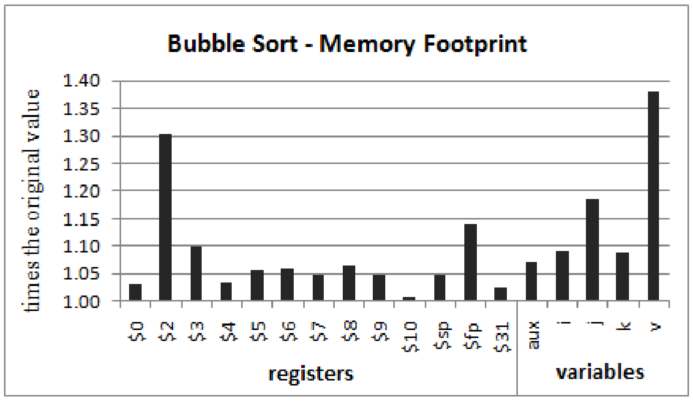


Fig. 4(b). Memory footprints for the bubble sort.

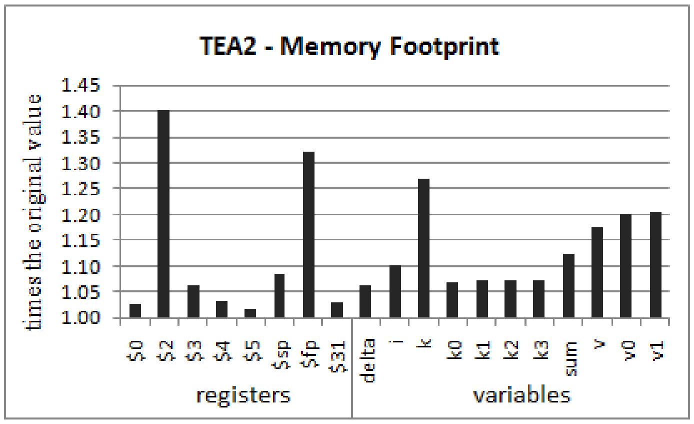


Fig. 4(c). Memory footprints for the TETRA Encryption Algorithm.

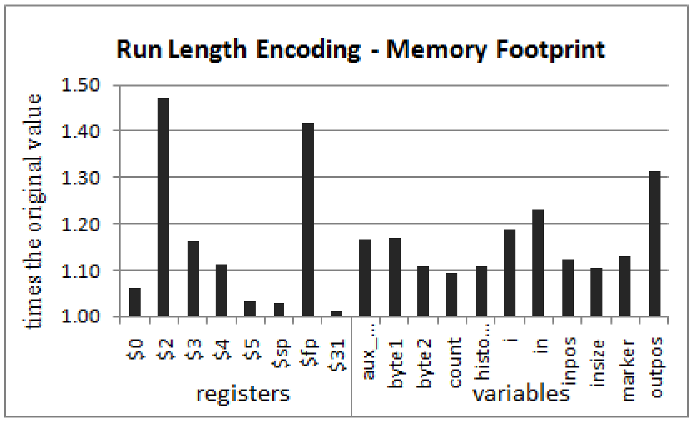


Fig. 4(d). Memory footprints for the run-length encoding.

approach, the highest overhead is presented by variable *outpos*, 31%, followed by variable *in*, 23%. Other variables overheads range from 9% to 19%. However, like for TEA2, the memory footprint overhead of duplicating all registers, 130%, is smaller than the overhead of protecting all variables, 175%. It shows that protecting in C language is more costly than protecting in assembly.

IV. Fault Injection Results

A fault injection campaign was performed to evaluate the error detection rate of each approach. Faults were injected at Register Transfer Level (RTL) in the miniMIPS microprocessor using the ModelSim simulation tool. The location and time for each injected fault were randomly selected. Only one fault was injected per program execution.

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. The faults duration was set to one clock cycle to force their effect to increase the error probability. As we are using only techniques developed to detect data-flow errors, only the errors that affected the data-flow were considered. The faults that caused control-flow errors were ignored because they are not in the scope of this work. Control-flow techniques should be used to detected them. Also faults that were masked by the logic of the microprocessor were ignored. The error is signaled when the result stored in the memory differs from a golden execution.

Fig. 5(a) shows the error detection rate for the matrix multiplication. The highest error detection rate for the assembly approach was obtained by the version where register *$fp* was duplicated. In this case, 41.6% of the data-flow errors were detected. The trade-off between error detection and performance overhead is 4.16 (41.6 / 10). Registers *$2*, *$0*, *$6* and *$4* achieved considerable error detection rates (34.1%, 20.7%, 16.7% and 10%, respectively) and trade-offs between error detection and performance overhead greater than 1 (1.37, 10.4, 8.4 and 10, respectively). Concerning the C language approach, the version where variable *c* was duplicated achieved 57% of error detection rate. However, in this case, the overhead in the execution time is quite high, 105%. The trade-off between the error detection rate and the execution time overhead is 0.54 (57 / 105). Even so, it is the best trade-off obtained by the C language approach. The versions where variables *i*, *j* and *v* were duplicated practically have not detected any data-flow error. With regards to variables *a* and *b*, the error detection rates were 9.2% and 8.6%, respectively.

Fig. 5(b) presents the error detection rate for the bubble sort. The highest error detection rate for the assembly approach was presented by the duplication of register *$2*. It reaches 73.8%. The trade-off between error detection and execution time overhead is 1.64 (73.8 / 45). Register *$0*, *$3* and *$4* presented non negligible error detection rate (40.6%, 15.9% and 36.9%, respectively) and trade-offs greater than 1 (39.83, 1.56 and 10.33, respectively). The highest error detection rate for C language approach was presented by variable *v*, 68.2%. The trade-off between error detection rate

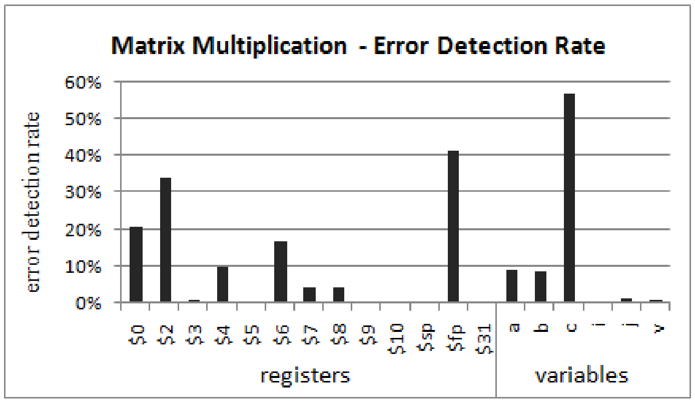


Fig. 5(a). Error detection rate for the matrix multiplication.

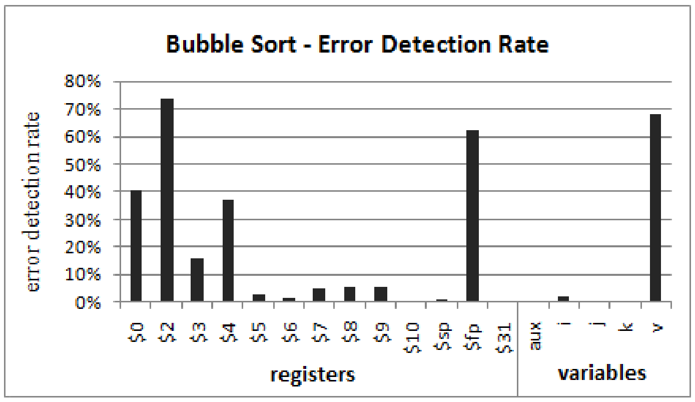


Fig. 5(b). Error detection rate for the bubble sort.

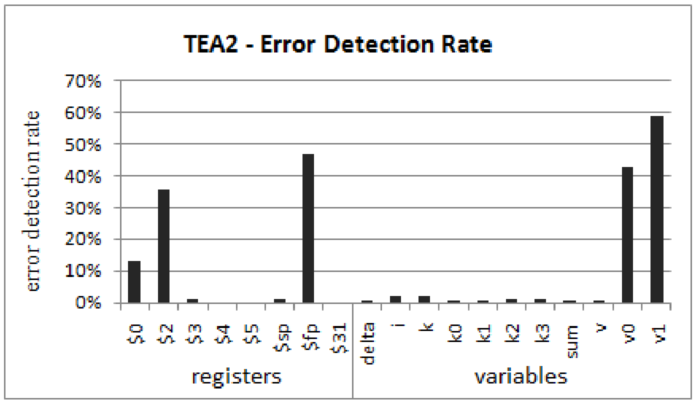


Fig. 5(c). Error detection rate for the TETRA Encryption Algorithm.

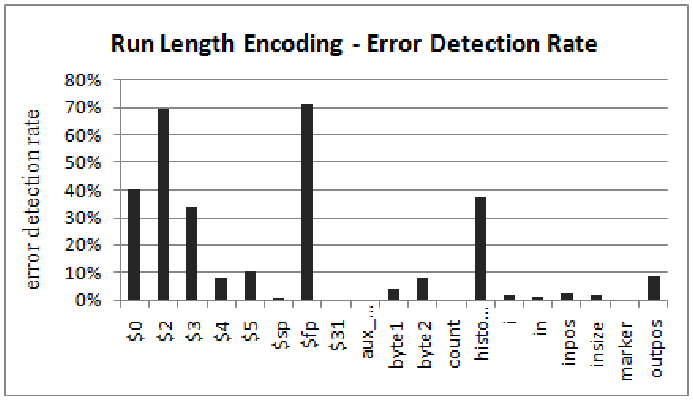


Fig. 5(d). Error detection rate for the run-length encoding.

and execution time overhead is 3.93 (68.2 / 17. The other registers have not presented considerable error detection rate.

The error detection rate for TEA2 is presented in Fig. 5(c). Register *$fp* reached 46.9% of data-error detection rate. Its trade-off between error detection and execution time overhead is 2.23 (46.9 / 21). Registers *$2* and *$0* also presented considerably error detection rates, 35.7% and 13.5%, respectively. The trade-off between error detection and execution time overhead is 0.96 (35.7 / 37), slight inferior to 1. As register *$0* present only 1% of overhead in the execution time, its trade-off is the highest, 13.5. Regards to the C language approach, variable *v0* and *v1* are the only that present considerably error detection rate, 42.7% and 58.9%, respectively. Variable *v1* is the only that has a trade-off greater than one, 1.13 (58.9 / 52). The trade-off between error detection and execution time overhead for variable *v0* is 0.82 (42.7 / 52).

Fig. 5(d) presents the error detection rate for the run-length encoding. With regards to the assembly approach, registers *$fp, $2, $0, $3, $5* and *$4* presented non negligible error detection rate, 71.3%, 69.7%, 40.4%, 33.8%, 10.5% and 7.8%, respectively. They also presented trade-offs between error detection and execution time overhead greater than 1, they are 3.4, 1.51, 20.2, 5.63, 10.5 and 7.8, respectively. On the other hand, the C language approach only present non negligible error detection rates for variable *histogram* (37.1%), variable *outpos* (8.5%) and variable *byte2* (8.2%). Their trade-offs were 0.52, 1.7 and 0.91 respectively. We can see that the trade-offs for the close error detection rates are quite different for both approaches. Assembly approach presents considerably better trade-offs, usually obtained due to smaller overheads.

V. Conclusions and Future Work

This paper presented a comparison between hardening by software in a high level programming language (C language) and in a low level programming language (assembly). The analysis was done using software-based fault tolerant techniques to detect data-flow errors in microprocessors. Four case-study applications were chosen and hardened. Regarding the C language approach, variables were individually duplicated and analyzed according to error detection rate, execution time and memory footprint. For the assembly implementation, registers were hardened instead of the variables. A fault injection campaign was performed where 750,000 faults were injected by simulation in the miniMIPS microprocessor.

Results showed considerable differences between both approaches. The overheads in the execution time presented by duplicating the registers in assembly are considerably lower than the overheads presented by duplicating the variables in the high level language. It makes the difference in the trade-off between error detection and performance overhead to be quite expressive when comparing the assembly approach with the high level language approach. Even in cases where there are more variables in the high level language code than used registers in the assembly code, the assembly seems to be a better approach because it presents better trade-offs between the error detection rate and the execution time overhead. Concerning the C language approach, when the error detection rate is high, the overheads are also high, which makes this approach not to be a good option. On the other hand, registers from the assembly approach that presented higher error detection rates did not present higher overheads, which makes this approach more interesting. This work shows the advantages of software hardening applied at assembly level, rather than applied at C level.

As future work, we intend to include control-flow techniques in order to analyze its effect when implemented at different abstraction levels. Also, we are interested in checking how techniques aimed at control-flow an data-flow combined themselves when applied at different abstraction levels.

References

[1] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.Hu, M. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: moore's law meets static power", IEEE Computer Society, Los Alamitos, USA, v. 36, p 68-75, 2003.

[2] S. Thompson et al., "In search of forever: continued transistor scaling one new material at a time", IEEE Transactions on Semiconductor Manufacturing, New York, USA, v. 18, n. 1, p 26-36, 2005.

[3] R. Baumann, "Soft errors in advanced semiconductor devices-part I: the three radiation sources", IEEE Transactions on Device and Materials Reliability, Los Alamitos, USA, v. 1, n. 1, p. 17-22, 2001.

[4] International Technology Roadmap for Semiconductors: 2005 Edition, Chapter Design, 2005, pp. 6-7.

[5] M. Rebaudengo, M. Sonza Reorda, M. Torchiano and M. Violante, "Soft-error Detection through Software Fault-Tolerance techniques", International Symposium on Defect and Fault Tolerance in VLSI Systems, p. 210-128, 1999.

[6] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: method, tools and experimental results", Design, Automation and Test in Europe Conference and Exhibition, p. 57-62, 2003.

[7] 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.

[8] J. R. Azambuja, A. Lapolli, L. Rosa and F. L. Kastensmidt, "Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique", IEEE Transactions on Nuclear Science, v. 58, n. 3, p. 993-1000, 2011.

[9] 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.

[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", IEEE Latin American Symposium on Circuits and Systems, 2011.

[11] 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.

[12] D. W. Parkinson, "TETRA Security", BT Technology Journal, vol. 19, n. 3, 2001, p. 81-88.

[13] E. Chielle, J. R. Azambuja, R.S. Barth, F. Almeida and F. Lima Kastensmidt, "Evaluating Selective Redundancy in Data-flow Software-based Techniques", in Proc. 13th European Conf. on Radiation Effects on Components and Syst., RADECS, Biarritz, France. Sept. 24-28, 2012.