Journal of Aerospace Technology and Management

ISSN: 1948-9648

secretary@jatm.com.br

Instituto de Aeronáutica e Espaço

Brasil

Rodrigues Ferreira, Ronaldo; Baldiati Parizi, Rafael; Carro, Luigi; Freitas Moreira, Álvaro

Compiler Optimizations Impact the Reliability of the Control-Flow of Radiation-Hardened Software

Journal of Aerospace Technology and Management, vol. 5, núm. 3, junio-septiembre, 2013, pp. 323-334

Instituto de Aeronáutica e Espaço

São Paulo, Brasil

Available in: http://www.redalyc.org/articulo.oa?id=309427928008

doi: 10.5028/jatm.v5i3.224

Compiler Optimizations Impact the Reliability of the Control-Flow of Radiation-Hardened Software

Ronaldo Rodrigues Ferreira[1], Rafael Baldiati Parizi[1], Luigi Carro[1], Álvaro Freitas Moreira[1]

ABSTRACT: This paper discusses how compiler optimizations influence software reliability when the optimized application is compiled with a technique to enable the software itself to detect and correct radiation-induced control-flow errors. Supported by a comprehensive fault-injection campaign using an established benchmark suite in the embedded systems domain, we show that the compiler is a non-negligible source of noise when hardening the software against radiation-induced soft errors.

KEYWORDS: Compilers, Radiation effects, Single event upsets, Software reliability, Software engineering.

INTRODUCTION

Compiler optimizations are taken for granted in modern software development, enabling applications to execute more efficiently in the target hardware architecture. Modern architectures have complex inner structures designed to boost performance, and if the software developer were to be aware of all those inner details, performance optimization would jeopardize the development processes. Compiler optimizations are transparent to the developer, who picks the appropriate ones to the results s/he wants to achieve, or, as it is more common, allowing this task to be performed by the compiler itself by flagging if it should be more or less aggressive in terms of performance.

Industry already offers microprocessors built with 22 nm transistors, with a prediction that by 2026, the size of the transistor will reach 5.9 nm (ITRS, 2012). This aggressive technology scaling creates a big challenge concerning the reliability of microprocessors using newest technologies. Smaller transistors are more likely to be disrupted by transient sources of errors caused by radiation, known as *soft-errors* (Borkar, 2005). Radiation particles originated from cosmic rays when strikes a circuit induces bit flips during software execution, and because transistors are becoming smaller in size, there is a higher probability that these transistors will be disrupted by a single radiation particle with smaller transistors requiring a smaller amount of charge to disrupt their stored logical value. The newest technologies are so

1.Universidade Federal do Rio Grande do Sul – Porto Alegre/RS – Brazil

Author for correspondence: Ronaldo Rodrigues Ferreira | Instituto de Informática, Universidade Federal do Rio Grande do Sul | Avenida Bento Gonçalves, 9500, Campus do Vale – Bloco IV – Agronomia | CEP 91.509-900 Porto Alegre/RS – Brazil | Email: rrferreira@inf.ufrgs.br

Received: 08/01/13 | **Accepted:** 02/05/13

sensitive to radiation that their usage will be compromised at the sea level, as predicted in the literature (Normand, 1996). Rech *et al.,* (2012) have shown that modern graphics processing unit (GPU) cards are susceptible to such an error rate that makes their usage unfeasible in critical embedded systems. However, industry is already investing in GPU architectures as the platform of choice for high performance and low power embedded computing, such as the ARM Mali® embedded GPU (ARM, n.d.).

The classical solution to harden systems against radiation is the use of *spatial redundancy*, i.e., the replication of hardware modules. However, spatial redundancy is prohibitive for embedded systems, which usually cannot afford extra costs of hardware area and power. The increase on power is a severe problem, because it is expected that 21% of the entire chip area must be turned off during its operation to meet the available power budget, and an impressive chip area of 50% at 8 nm (Esmaeizadeh, 2011). This creates the *dark silicon* problem (Esmaeizadeh, 2011), i.e., a huge area of the circuit cannot be used during its lifecycle. This problem gets worse when the microprocessor has redundant units, because system's reliability could be compromised if redundant units were turned off. The current solution to this problem is to use radiation-hardened microprocessors, which are designed to endure radiation. The problem with this approach is the low availability, high unit pricing, and International Traffic in Arms Regulations (ITAR) restrictions of those radiation-hardened components. For instance, a 25 MHz microprocessor has a unitary price of $ 200,000.00 (Mehlitz and Penix, 2005). This high unit pricing makes the use of radiation-hardened microprocessors unfeasible for embedded systems used in aircrafts, not to say about cars and low-end medical devices, such as pacemakers. For these critical embedded systems, where cost and ITAR restrictions are hard constraints, a cheaper, but yet effective approach for reliability against radiation, is necessary.

Software-Implemented Hardware Fault-Tolerance (SIHFT) (Goloubeva, 2006) is an approach for radiation reliability that adds redundancy in terms of extra instructions or data to the application, keeping the hardware unchanged. SIHFT techniques work by modifying the original program by adding *checking mechanisms* to it. SIHFT techniques are classified either as *control-flow* or as *data-flow*. The former is designed to detect when an illegal jump has occurred during application execution to possibly proceed with the resolution of the correct jump address or at least signaling that such an error has occurred. The latter checks if a data variable being read is correct or not. While the effects of data-flow SIHFT methods are clear (usually, the duplication of program variables or the addition of variable checksums solves the problem), the impacts of the control-flow ones, are yet not well understood. Because the control-flow methods modify the program's control-flow graph (CFG), which happens to be the same artifact used by compiler optimizations, the efficiency of control-flow reliability techniques might be influenced by the optimizations in an unpredictable way.

In this paper, we evaluate how the cumulative usage of compiler optimizations influence reliability of applications hardened with the state-of-the-art Automatic Correction of Control-flow Errors (ACCE) (Vemu, Gurumurthy and Abraham, 2007) control-flow SIHFT technique, which is selected, because it is the current most efficient method in terms of reliability, attaining an error correction rate of ~70%. The application set we use in this paper is drawn from the MiBench suite (Guthaus *et al.*, 2001).

RADIATION EFFECTS ON SOFTWARE RELIABILITY

Highly energized radiation particles are known hazard sources in electronics since the 1970s (Binder *et al.,* 1975), as well as the mitigation schemes for such sources. Single-Event Transient (SET) is the observed physical effect of radiation on electronics, corresponding to voltage glitches in circuitry, which by itself does not incur on system hazards. System hazards originate when SETs are caught up by memories (e.g., SRAM's) and sequential logic (e.g., registers), thus, becoming a Single-Event Upset (SEU). An SEU is a non-permanent damage to the systems (i.e., transient) that results in a bit with logical value 1 that flips to 0 and vice-versa. Mitigation approaches might be as follows:

- *Substrate and gate-level*: The reduction of charge generation and collection.
- *Hardware design*: The modification of circuit response through the addition of logical elements or even the storage of data on spatially separated nodes.

- *System level*: The addition of redundancy at system level, e.g., software hardening.

Definitely, the system level mitigation approach is the most feasible for components off-the-shelf and to overcome ITAR restrictions.

A bit flip caused by a radiation-induced SEU can compromise the software in two different ways. Firstly, an SEU can corrupt *data*, i.e., the values of program variables. Secondly, an SEU can corrupt *control*, i.e., the program flow of execution. To illustrate these two situations, consider the program presented in Fig. 1, which corresponds to the Bubble Sort algorithm. Bubble Sort is a naïve $O(n^2)$ solution for sorting an array of arbitrary numbers.

The Bubble Sort algorithm is divided in labeled regions named basic blocks (identified by the gray and white regions in the Bubble Sort source code). A *basic block* is a region of a program where the contained program instructions does not contain any branch, i.e., iteration loops (e.g., for and while commands), if-conditionals, function call, and return. Therefore, a basic block only contains *variable assignments* and *logical evaluations*. The CFG of a program P is a graph G_p=(V, E), where the set V of vertices contains the program's basic blocks and the set E of edges contains the transitions in the execution flow. To illustrate this, the CFG of the Bubble Sort algorithm is presented in Fig. 2.

An executed branch of the program P (represented by an arrow in Fig. 2) is said to be *legal*, if and only if, it is an element of the set E of G_p and the condition to execute it is satisfied (e.g., the blue arrow in Fig. 2); it is *wrong* if the executed branch is an element of the set E, but its condition to execute cannot be satisfied (e.g., the purple arrow in Fig. 2); and an executed branch is *wrong* if it is not an element of the set E (e.g., the red arrow in Fig. 2).

```
Algorithm Bubblesort(input: n, V)
def n : number of values to sort;
def V[n] : array of size n;
def temp, i, j: integer variables;
1. i := n - 1;
2. while ( i >= 1 ) do
3.     j := 0;
4.     while ( j < i ) do
5.           if ( V[j] < V[j+1] )
6.                 temp   := V[j];
7.                 V[j]   := V[j+1];
8.                 V[j+1]:= temp;
9.           end if
10.          j := j + 1;
11.    end while
12.    i := i - 1;
13. end while
14. return V;
```

Figure 1. Bubble Sort algorithm with explicit basic-blocks, which are represented by the grouped numbered lines.

A control-flow error (CFE) occurs when either a wrong or illegal branch is executed. Notice that in these two cases, a CFE cannot exist if the program execution is not corrupted, i.e., an illegal branch cannot exist in a correct program execution, because it only executes branches from the set E; a wrong branch cannot exist, because it is always possible to satisfy the logical conditions of all branches if program execution is correct. A CFE can be created by a radiation-induced SEU in any of the following three scenarios:

- A non-branch instruction being executed changes into a non-valid branch, i.e., the operation code data is corrupted.
- The target address of a valid branch is corrupted.
- One of the variables composing a logical expression that activates a branch is corrupted.

Scenarios (1) and (2) leads to an illegal branch, and scenario (3) leads to a wrong branch.

A data-flow error (DFE) is caused by a radiation-induced SEU that corrupts variables within a basic block. A DFE might lead to erroneous results or even to a CFE, in case the corrupted variable is used in a logical expression controlling a branch. The focus of this paper is CFE's. For an extensive review of mitigation techniques of CFE and DFE, interested readers may refer to the work of Goloubeva *et al.,* (2006).

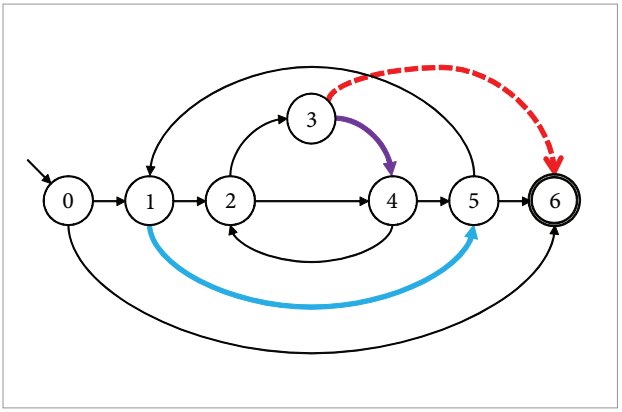

Figure 2. Control-flow graph of the Bubble Sort algorithm. The blue arrow is a legal branch (together with the black arrows), the purple arrow is a wrong branch, and the red arrow is an illegal branch.

The detection of transient CFE was established in the literature with techniques that check assertions during runtime. The general idea is to compute signatures identifying each basic block, and checking the signatures generated during compilation and runtime. If they do not match, an error is signaled. CFE's were first identified by the usage of watchdog processors, which are intrusive in the hardware design (Saxena and McCluskey, 1990). Lately, techniques based on the signature checking scheme in software, such as the Control-flow Checking Approach (CCA) (Kanawati *et al*., 1996), were identified, but with a coverage rate of only 38% and a performance overhead of 50%.

Advances in the signature checking method offered some improvements on coverage and performance, such as the Control-Flow Checking by Software Signatures, which incurs in 50% of overhead in execution time and program size (Oh *et al.,* 2002). The most efficient technique of signature checking capable of correcting errors is the ACCE (Vemu *et al.,* 2007), which incurs in approximately 20% of overhead in execution time to produce an average 70% of correct answers in fault-injection campaigns. However, ACCE is not capable of correcting errors that occur within a basic block, i.e., in the data flow; hence, the use of complementary techniques is required. When ACCE is enhanced with data-flow correction, its coverage rate achieves the average of 91.6% (Vemu *et al.,* 2007).

Because the CFG is one of the most important software artifacts used by compilers when analyzing and modifying programs, it is important to measure how the compiler impacts the reliability of the software mitigation techniques for radiation-induced CFE. The understanding of these impacts is imperative to employ software mitigation techniques in real systems. This paper presents a study using the ACCE mitigation technique, which is briefly reviewed in the next section.

AUTOMATIC CORRECTION OF CONTROL-FLOW ERRORS

ACCE (Vemu *et al.,* 2007) is a software technique for reliability that detects and corrects CFE's due to random and arbitrary bit-flips that might occur during software execution. The hardening of an application with ACCE is done at compilation, because it is implemented as a transformation pass in the compiler. ACCE modifies the applications' basic blocks with the insertion of extra instructions that perform the error detection and correction during software execution. In this section, we briefly explain how the ACCE works in two separate subsections, one dedicated to error detection and the other to error correction are discussed in the subsequent subsections. The reader should refer to the ACCE article for a detailed presentation and experimental evaluation (Vemu *et al.,* 2007). The fault model that ACCE assumes is further described in the "Fault Model and Methodology" section.

Control-Flow Error Detection

ACCE performs online detection of CFE-s by checking the signatures in the beginning and in the end of each basic block of the CFG, thus, ACCE is classified as a *signature checking* SIHFT technique as termed in the published literature. The basic block signatures are computed and generated during compilation; the signature generation is critical, because it requires computing non-aliased signatures between the basic block, i.e., each block must be unambiguously identified. In addition, for each basic block found in the CFG, two additional code regions are added, the *header* and the *footer*. The signature checking during execution takes place inside these code regions. Figure 3 shows two basic blocks (labeled as **N2** and **N6**) with the additional code regions. The top region corresponds to the header and the bottom to the footer. Still, at compilation, the ACCE creates two additional blocks for each function, namely the function entry block and the Function Error Handler (FEH). For instance, Fig. 3 depicts a portion of two functions, *f1* and *f2*, both owning entry blocks labeled as **F1** and **F2**, and FEHs, labeled as **FEH_1** and **FEH_2**, respectively. Finally, ACCE creates a last extra block, the Global Error Handler (GEH), which can only be reached from a FEH block. The role of these blocks will be presented soon.

At runtime, the ACCE maintains a global signature register (represented as S), which is constantly updated to contain the signature of the basic block that the execution has reached. Therefore, during the execution of the *header* and *footer* code regions of each basic block, the value of the signature register is compared with the signatures generated during compilation for those code regions, and if those values do not match, a CFE is detected and then the control should be transferred to the corresponding FEH block of the function where the execution takes place at that time. The ACCE also maintains the current function register

(represented as **F**), which stores the unique identifier of the function currently being executed. The current function register is only assigned at the extra entry function block. This process encompasses the detection of an illegal and erroneous branch due to a soft error.

Figure 3 depicts an example of the checking and update of signatures performed in execution time that occurs in a basic block. In this example, the CFE occurs in the block **N2** of function **F1**, where an illegal jump incorrectly transfers the control flow to the basic block **N6** of function **F2**. When the execution reaches the footer of the block **N6**, the signature register **S** is checked against the signature generated at compilation. In this case, **S** = 0111 (i.e., the previous value assigned in the header of the block **N2**). Thus, the branch test in the **N6** footer will detect that the expected signature does not match with the value of S, and thus, the CFE must be signaled (step 1 in Fig. 3). In this example, the application branches to the address *f2_err*, making the application enter the **FEH_2** block (because the error was detected by a block owned by the function **F2**, the FEH invoked is the **FEH_2**). At this point, the CFE is detected and ACCE can proceed with the correction of the detected CFE.

Control-Flow Error Correction

The correction process starts as soon as an illegal jump is detected by the procedure described in the last subsection,

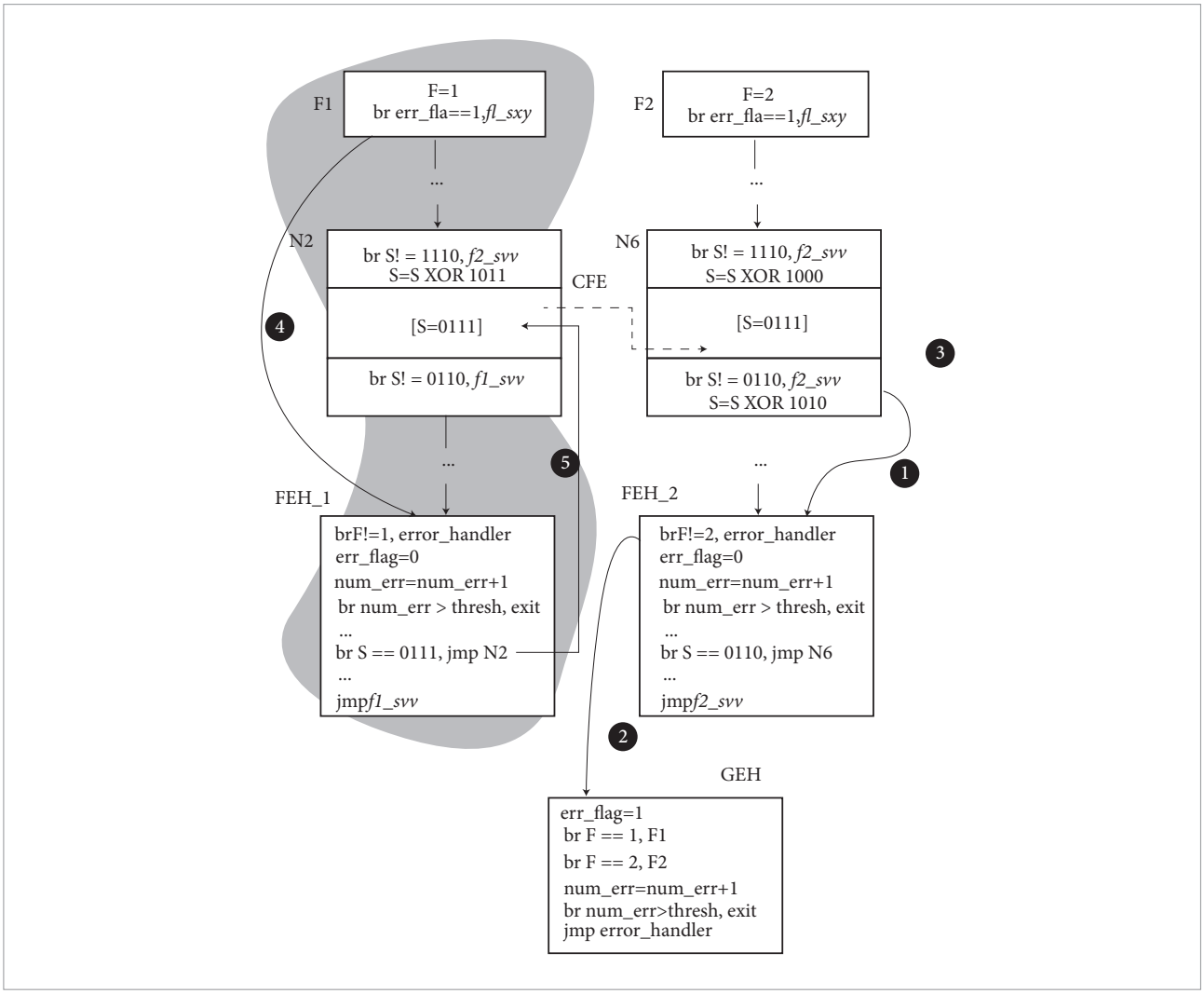

Figure 3. Depiction of how the control is transferred from a function to the basic blocks that ACCE has created when a control-flow error occurs during software execution. In this figure, there is a control-flow error (dashed arrow) causing the execution to jump from the block **N2** of function **F1** to the block **N6** of function **F2**.

with the control flow transferred to the FEH corresponding to the function where the CFE is found. The FEH checks if the illegal jump was originated in the function it is responsible to handle its detected errors by comparing the value of the function's identifier (**F1** or **F2**, in the example of Fig. 3) with the current function register **F**. If the error occurred in the function stored in the **F** register, FEH evaluates the current value of the signature register and then transfers the control to the basic block that is the origin of the illegal jump (this origin is stored in the **S** register). On the other hand, if the illegal jump is not originated in the function where the detection has occurred, the FEH then transfers the control flow to the GEH. In this case, the GEH is responsible for identifying the function where the CFE has occurred and to transfer the control flow back to this function, so that the error is correctly treated by the function's FEH. The GEH searches the function where the error has occurred and transfers the control to its entry block, which then sends the control flow to the proper FEH so that the error can be corrected, i.e., branching the control to the basic block where the CFE has occurred.

Recalling the example depicted in Fig. 3, after the CFE is detected and the control is transferred to **FEH_2** (step 1), the **F** register is matched against the function identifier of the function from where the control originated. However, because the CFE originated in the basic block **N2** of the function **F1**, **F** = 1. Therefore, **FEH_2** is not capable of finding the basic block where the CFE originated, and then it transfers the control to the GEH so that the correct FEH can be found (step 2). The GEH searches for the function identifier stored in **F**, until it finds that it should branch to **F1** (step 3). Upon reaching the entry block **F1**, the variable *err_flag*=1, because it is assigned to 1 in the GEH, meaning that there is an error that should be fixed, thus, the control branches to **FEH_1** (step 4). Now, because **F**=1, **FEH_1** knows that it is the FEH capable of handling the CFE and, as such, it sets the variable *err_flag* to 0. Finally, it searches for the basic block that has the signature equal to the register **S**. Upon finding it, the control branches to this basic block, i.e., **N2** in Fig. 3 (step 5). This last branch restores the control flow to the point of the program right before the occurrence of the CFE. Notice that inside all the FEH and the GEH, there is the variable *num_error*, counting how many times the control has passed through a FEH or a GEH. This acts as a threshold for the number of how many times the correction must be attempted, which is necessary to avoid an infinite loop in case the registers **F** or **S** get corrupted for any reason. This process concludes the correction of a CFE with the ACCE.

FAULT MODEL AND METHODOLOGY

The fault model we assume in the experiments is the *single bit flip*, i.e., only one bit of a word is changed when a fault is injected. The ACCE is capable of handling multiple bit flip as long as the bits flipped is within a same word. Because the fault injection, as it will be discussed later, guarantees that the injected fault ultimately turned into a manifested error, it does not matter how many bits are flipped, i.e., there is no silent data corruption, meaning the faults that cause a word to change its value neither change the behavior of the program nor its output. This could happen in the case that the fault flipped the bits of a dead variable.

The ACCE technique was implemented as a transformation pass in the Low Level Virtual Machine (LLVM; Lattner and Adve, 2004) production compiler, which performs all the modifications in the CFG using the LLVM Intermediate Representation (LLVM-IR). The LLVM was selected as our compilation platform, because of its increased use in the industry, accompanied with a very detailed documentation and quality of its source code. The ACCE transformation pass was applied *after* the set of compiler optimizations, because executing in the opposite order, a compiler optimization could invalidate the ACCE generated code and semantics. Table 1 presents the LLVM optimization passes used in the experiments.

Because the ACCE is a SIHFT technique to detect and correct CFE-s, the adopted fault model simulates three distinct control-flow disruptions that might occur due to a CFE. Remember that a CFE is caused by the execution of an illegal branch to a possibly wrong address. The branch errors considered in this paper are as follows:

- *Branch creation*: The program counter is changed, transforming an arbitrary instruction (e.g., an addition) into an unconditional branch.
- *Branch deletion*: The program counter is set to the next program instruction to execute independently if the current instruction is a branch.
- *Branch disruption*: The program counter is disrupted to point to a distinct and possibly wrong destination instruction address.

We implemented a software fault injector, using the GDB (GNU Debugger), in a similar fashion as implemented by Krishnamurthy *et al.,* (1998), which is an accepted fault-injection methodology in the embedded systems domain, to perform the fault-injection campaigns. The steps of the fault-injection process are the following:

- The LLVM-IR program resulting from the compilation with a set of optimization and with ACCE is translated to the assembly language of the target machine.
- The execution trace in assembly language is extracted from the program execution with GDB.
- A branch error (branch creation, deletion, or disruption) is randomly selected. On an average, each branch error accounts for 1/3 of the amount of injected errors.
- One of the instructions from the trace obtained in step 2 is chosen at random for fault injection. In this step, a histogram of each instruction is computed because instructions that execute more often have a higher probability to be disrupted.
- If the chosen instruction in step 4 executes n times, choose at random an integer number k with $1 \leq k \leq n$.
- Using GDB, a breakpoint is inserted right before the k-th execution of the instruction selected in step 4.
- During program execution, upon reaching the breakpoint inserted in step 6, the program counter is intentionally corrupted by flipping one of its bits to reproduce the branch error chosen in step 3.
- The program continues its execution until it finishes.

A fault is only considered valid, if it has generated a CFE, i.e., silent data corruption and segmentation faults were not considered to measure the impacts of the compiler optimizations on reliability. All the experiments in this paper were performed in a 64-bit Intel Core i5 2.4 GHz desktop with 4 GB of RAM and the LLVM compiler version 2.9. For all program versions, where each version corresponds to the program compiled with a set of optimizations plus the ACCE pass, 1,000 faults were injected using the aforementioned fault-injection scheme. In the experiments we considered ten benchmark applications from the MiBench (Guthaus *et al.*, 2001) embedded benchmark suite as follows: *basicmath*, *bitcount*, *crc32*, *dijkstra*, *fft*, *patricia*, *quicksort*, *rijndael*, *string search*, and *susan* (comprising *susan* corners, edge, and smooth).

IMPACT OF COMPILER OPTIMIZATIONS ON SOFTWARE RELIABILITY

This section studies the impacts on software reliability when an application is compiled with a set of compiler optimizations and further hardened with the ACCE method. Throughout this section, the baseline for all comparisons is an application compiled with the ACCE method without any other compiler optimization. The ACCE performs detection and correction of CFE-s,

Table 1. Set of Low Level Virtual Machine optimization passes used for experimental evaluation in this paper.

-adce	-loop-reduce
-always-inline	-loop-rotate
-argpromotion	-loop-simplify
-block-placement	-loop-unroll
-break-crit-edges	-loop-unswitch
-codegenprepare	-loweratomic
-constmerge	-lowerinvoke
-constprop	-lowerswitch
-dce	-mem2reg
-deadargelim	-memcpyopt
-deadtypeelim	-mergefunc
-die	-mergereturn
-dse	-partial-inliner
-functionattrs	-prune-eh
-globaldce	-reassociate
-globalopt	-reg2mem
-gvn	-scalarrepl
-indvars	-sccp
-inline	-simplifycfg
-instcombine	-simplify-libcalls
-internalize	-sink
-ipconstprop	-sretpromotion
-ipsccp	-strip
-jump-threading	-strip-dead-debug-info
-lcssa	-strip-dead-prototypes
-licm	-strip-debug-declare
-loop-deletion	-strip-non-debug
-loop-extract	-tailcallelim
-loop-extract-single	-tailduplicate

thus all data discussed in this section considers the *correction rate* as the data to compute the efficiency metric. In this analysis, we use 58 optimizations provided by the LLVM production compiler. Finally, the results were obtained using the fault model and fault injection methodology described in the section "Fault Model and Methodology".

The impact of the compiler optimizations when compiling for reliability is measured in this paper using the metric Relative Improvement Percentage (RIP; Pan and Eigenmann, 2006). The RIP is presented in Eq. 1, where F_i is a compiler optimization, $E(F_i)$ is the error correction rate obtained for a hardened application compiled with F_i, and E_B is the error correction rate obtained for the baseline, i.e., the application compiled only with ACCE and without any optimization.

$$RIP_B(F_i) = \frac{E(F_i) - E_B}{E_B} \cdot 100\% \quad (1)$$

Figure 4 shows a scatter plot of the obtained RIP for each application, with each of the 58 LLVM optimizations being a point in the *y*-axis. Each point represents the hardened application compiled with a single LLVM optimization at a time, with each application compiled with 58 distinct optimizations. Figure 4 shows that several optimizations increase the RIP considerably, sometimes reaching a RIP of ~10%. This is a great result, which shows that reliability can be increased for free by just picking appropriate optimizations that facilitates for ACCE the process of error detection and correction. However, we also find that some optimizations totally jeopardize reliability, reaching a RIP of -73.27% (bottom filled red circle for *bitcount*).

It is also possible to gather evidence that the structure of the application also influences how an optimization has an impact on the RIP of reliability. Let us consider the *block-placement* optimization, which is represented by the white diamond in Fig. 4. In the case of the *qsort* application, *block-placement* has a RIP of -42.75% and a RIP of +11.68%. The reader can notice that other optimizations also show this behavior (increasing RIP for some applications and decreasing it for others). It also happens that some hardened applications are less sensitive to compiler optimizations, as it is the case of the *crc_32* one, where the RIP is within the ±5% interval around the baseline.

Figure 5 depicts the RIP of a selected subset of the 58 LLVM optimizations, making it clear that even within a small subset, the variation in the RIP for reliability is far from negligible. For instance, the *always-inline* LLVM optimization has an error correction RIP interval of [-4.55%, +9.24%].

Usually compiler optimizations are applied in bulk, using several of them during compilation. Therefore, it is also important to examine if successive optimization passes could compromise or increase software reliability of a hardened application. Figure 6 presents the error correction rate RIP, where the hardened application was compiled with a subset of the 58 LLVM optimizations. In this experiment, we used six sizes of subsets: 10, 20, 30, 40, 50, and 58. The RIP shown in Fig. 6 is the average of five random subsets, i.e., it is an average of distinct subsets of the same size. Taking the average and picking the optimizations at random, reproduces the effects of indiscriminately picking the compiler optimizations, or at least, selecting optimizations with the object of optimizing performance without previous knowledge of how the selected optimizations together influences the software reliability.

It is possible to see that the cumulative effect of compiler optimizations in the error correction RIP is in most of the cases deleterious, but for a few exceptions. Figure 6 confirms that some applications are less sensitive to the effects of compiler optimizations, e.g., the *crc32* has its RIP within the interval [-1.11%–0.73%]. On the other hand, *basicmath*, *bitcount*, and *patricia*, are jeopardized. It is interesting to notice that the RIP in case of picking a subset of optimizations is not subject to the much severe reduction that was measured when only a single optimization was used (Fig. 4), providing an evidence that the composition of distinct optimization may be beneficial for reliability.

Based on the data and experiments discussed in this section, it is clear that selection of compiler optimizations requires the software designer to take into the consideration that some optimizations may not be adequate in terms of reliability for a given application. Moreover, data also shows that a given optimization is not only by itself a source of reliability reduction; reliability is also dependent of the application being hardened, and how a given optimization facilitates or not the work of the ACCE technique.

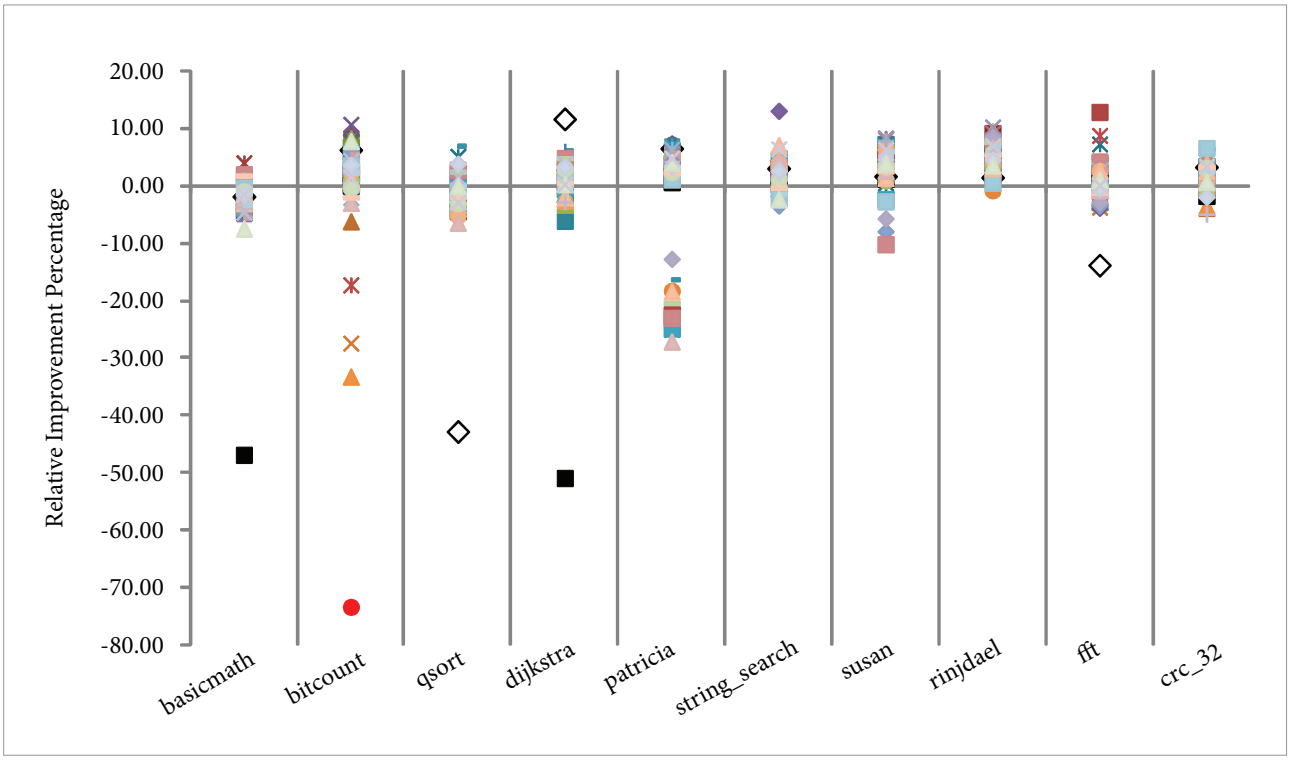

Figure 4. Relative Improvement Percentage for the error correction rate of applications hardened with ACCE under further compiler optimization. Each hardened application was compiled with a single optimization at a time, but all applications were compiled with the 58 Low Level Virtual Machine optimizations, thus, each hardened application has 58 versions. The baseline (Relative Improvement Percentage = 0%) is the error correction rate of the hardened application compiled without any Low Level Virtual Machine optimization.

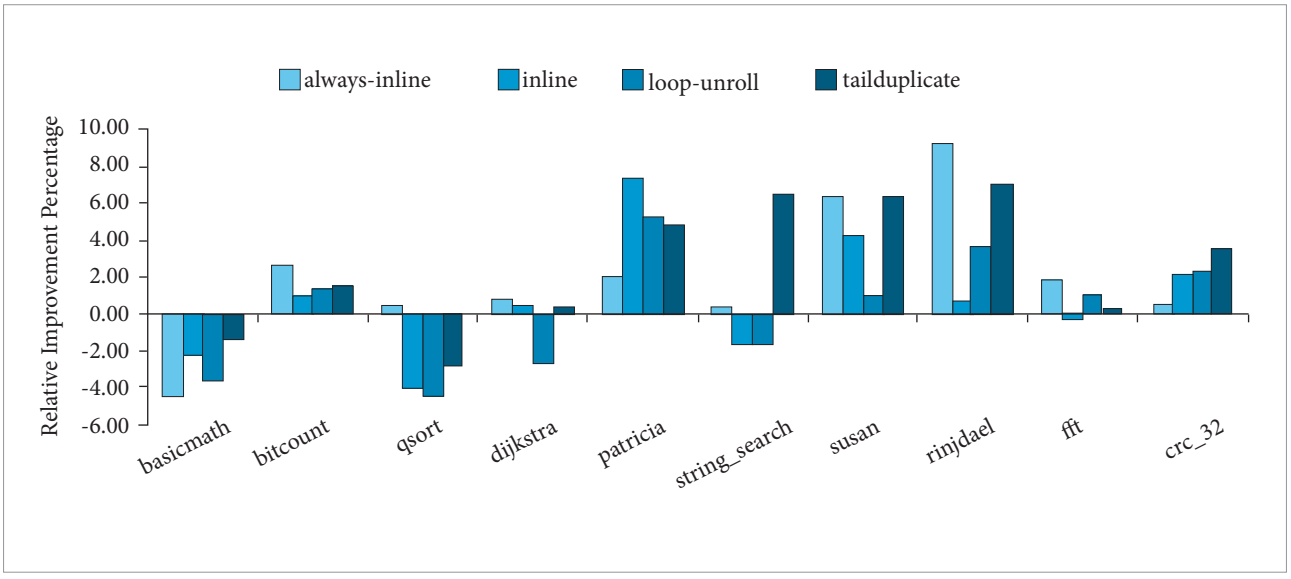

Figure 5. Relative Improvement Percentage of a selected subset of the 58 Low Level Virtual Machine optimizations. The baseline (Relative Improvement Percentage=0%) is the error correction rate of the hardened application compiled without any Low Level Virtual Machine optimization.

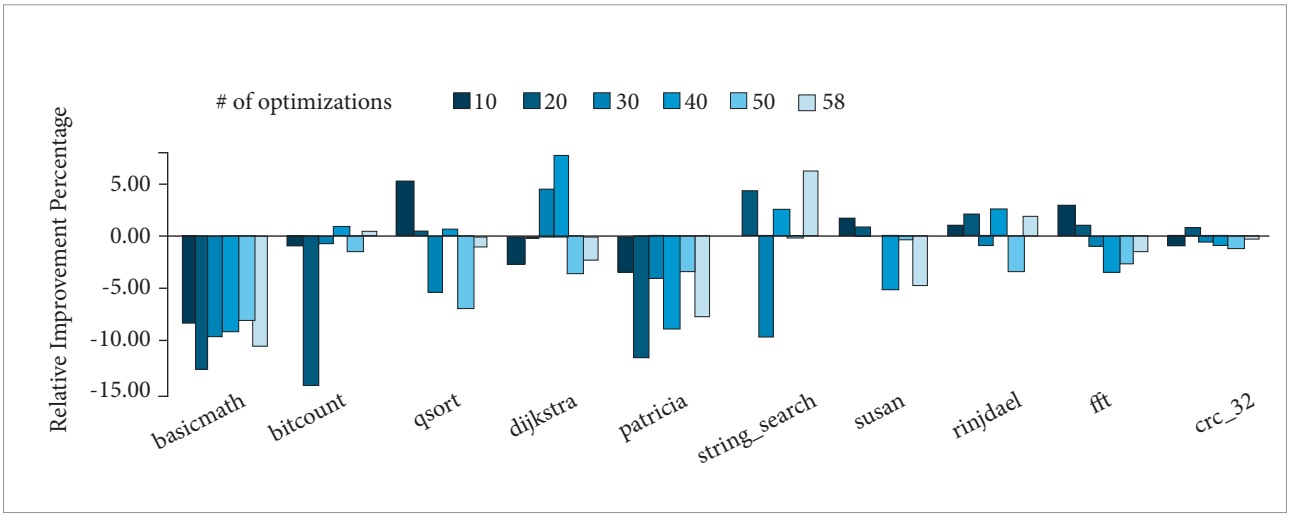

Figure 6. RIP of random subsets of the 58 Low Level Virtual Machine optimizations with a varying number of optimizations for each different subset: 10, 20, 30, 40, 50, and 58 optimizations. The Relative Improvement Percentage for each subset was measured taking the average of six random subsets for each subset size. Hence, distinct possible optimizations for subsets were considered. The baseline (Relative Improvement Percentage=0%) is the error correction rate of the hardened application compiled without any Low Level Virtual Machine optimization.

RELATED WORK

Much attention has been devoted to the impact of compiler optimizations on program performance in the published literature. However, the understanding of how those optimizations work together and how they influence each other is a rather recent research topic. The Combined Elimination (CE) (Pan and Eigenmann, 2006) is an analysis approach to identify the best sequence of optimizations for a given application set using the GNU Compiler Collection (GCC). The authors discuss that simple *orchestration* schemes between the optimizations can achieve near-optimal results as if it has performed an exhaustive search in all the design space created by the optimizations. CE is a greedy approach that first compiles the programs with a single optimization, using this version as the baseline. From those baseline versions, the set of RIP is calculated, which is the percentage that the program's performance is either reduced or increased. With the RIP at hand for all baselines, the CE starts removing the optimizations with negative RIP, until the total RIP of all optimizations applied into a program do not reduce. CE was evaluated in different architectures, achieving an average RIP of 3% for the SPEC2000, and up to 10% in case of the Pentium IV for the floating point applications.

The Compiler Optimization Level Exploration (COLE) (Hoste and Eeckhout, 2008) is another approach to achieve performance increase by selecting a proper optimization sequence. COLE uses a population-based multi-objective optimization algorithm to construct a Paretto optimal set of optimizations for a given application using the GCC compiler. The data found with COLE give some insightful results about how the compiler optimizations behave when they are applied with several of them at the same time. For instance, 25% of the GCC optimizations appear in at most one Paretto set, and some of them appear in all sets. Therefore, 75% of all the optimizations do not contribute to improve the performance, meaning that they can be safely ignored! COLE also shows that the quality of an optimization is highly tied with the application set.

The Architectural Vulnerability Factor (AVF) (Mukherjee *et al.*, 2003) is a metric to estimate the probability that the bits in a given hardware structure will be corrupted by a soft error when executing a certain application. The AVF is calculated as the total time the vulnerable bits remains in the hardware architecture. For example, the register file has a 100% AVF, because all of its bits are vulnerable in case of a soft error. The AVF metric is highly influenced by the application due to liveness of program's variables. For instance, a dead variable has a 0%

AVF, because it is not used in a computation. The impact of the GCC optimizations in the AVF metric is evaluated by trying to reduce the AVF-delay-square-product (ADS), introduced by the authors (Jones *et al*., 2008). The ADS considers a linear relation of the AVF between the square of the performance in cycles, clearly prioritizing performance over reliability. It is reported that the -O_3 optimization level is detrimental both to the AVF and performance, because the benchmarks that are considered (MiBench) have increased the number of loads executed. Again, it was found that the *patricia* application was the one with the highest reduction in the AVF at 13%.

Bergaoui and Leveugle (2011) analyzed the impact of compiler optimizations on data reliability in terms of variable liveness. *Liveness* of a variable is the time period between the variable that is written and it is last read before a new write operation. The authors concluded that the liveness is not related only with the compiler optimization, but it also depends on the application being compiled, which is in accordance with the discussion of this paper. This paper shows that some optimizations tend to extend the time a variable is stored in a register instead of memory. The goal behind this is obvious, i.e., it is much faster to fetch the value of a variable when it is in the register than in the memory. However, the memory is usually more protected than registers because of cheap and efficient error correction code (ECC) schemes, and thus, thinking about reliability, it is not a good idea to expose a variable in a register for a longer time. The solution to that could be the application of ECC, such as Huffman to the program variables itself. Decimal Hamming (DH) (Argyrides *et al*., 2011) is a software technique that performs this for a class of programs where the program's output is a linear function of the input. The generalization of the efficient data-flow SIHFT techniques, such as DH (i.e., ECC of program variables) is still an open research problem.

CONCLUSION

In this paper, we characterized the problem of compiling embedded software for reliability, given that compiler optimizations impact the coverage rate. The study presented in this paper makes clear that selecting optimizations indiscriminately, can decrease software reliability to unacceptable levels, probably avoiding the software to be deployed as originally planned. Embedded software and systems deployed in space applications must always be certified with evidence that they support harsh radiation environments, and given the increasing technology scaling, other safety critical embedded systems might have to tolerate radiation-induced errors in a near future. Therefore, the embedded software engineers must be very careful while compiling the safety critical embedded software.

Future research work is focused on the formalization of the ACCE transformation pass to generate automatic proofs about the correctness of programs compiled with the ACCE. This step is important to allow the certification of software hardened with ACCE.

ACKNOWLEDGEMENTS

This work is supported by the CAPES foundation of the Ministry of Education, CNPq research council of the Ministry of Science and Technology, and FAPERGS research agency of the State of Rio Grande do Sul, Brazil. R. Ferreira was supported with a doctoral research grant from the Deutscher Akademischer Austauschdienst (DAAD) and from the Fraunhofer-Gesellschaft, Germany.

REFERENCES

Argyrides, C., Ferreira, R., Lisboa, C. and Carro, L., 2011, "Decimal Hamming: A Novel Software-Implemented Technique to Cope with Soft Errors", Proceedings of the 26th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, pp. 11-17, doi: 10.1109/DFT.2011.35

ARM, n.d. 2012, "ARM Mali Graphics Hardware", Retrieved in December 21, 2012, from http://www.arm.com/products/multimedia/mali-graphics-hardware/index.php.

Bergaoui, S. and Leveugle, R., 2011, "Impact of Software Optimization on Variable Lifetimes in a Microprocessor-Based System", Proceedings of the 6th IEEE International Symposium on Electronic Design, Test and Application, pp. 56-61, doi: 10.1109/DELTA.2011.20

Binder, D., Smith, E.C. and Holman, A.B., 1975, "Satellite Anomalies from Galactic Cosmic Rays", IEEE Transactions on Nuclear Science, Vol. 22, No. 6, pp. 2675-2680, doi: 10.1109/TNS.1975.4328188

Borkar, S., 2005, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation", IEEE Micro, Vol. 25, No. 6, pp. 10-16, doi: 10.1109/MM.2005.110

Esmaeizadeh, H., Emily, B., Renee, A. and Sankaralingam, K., 2011, "Dark Silicon and the End of Multicore Scaling", IEEE Micro, Vol. 32, No. 3, pp. 122-134, doi: 10.1109/MM.2012.17

Goloubeva, O., Rebaudengo, M., Sonza Reorda, M. and Violante, M., 2006, "Software-Implemented Hardware Fault Tolerance", Ed. Springer, New York, NY, USA, p 228.

Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T. and Brown, R.B., 2001, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite", Proceedings of the IEEE International Workshop of Workload Characterization, pp. 3-14, doi: 10.1109/WWC.2001.990739

Hoste, K. and Eeckhout, L., 2008, "Cole: Compiler Optimization Level Exploration", Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 165-174, doi: 10.1145/1356058.1356080

ITRS, 2012, "ITRS 2009 Roadmap", International Technology Roadmap for Semiconductors.

Jones,T.M., O'Boyle, M.F.P. and Ergin, O., 2008, "Evaluating the Effects of Compiler Optimisations on AVF", Proceedings of the Workshop on Interaction Between Compilers and Computer Architecture, 6p.

Kanawati, K., Krishnamurthy, N., Nair, S. and Abraham, J.A., 1996, "Evaluation of Integrated System-level Checks for On-Line Error Detection", Proceedings of the 2nd International Computer Performance and Dependability Symposium, pp. 292-301, doi: 10.1109/IPDS.1996.540230

Krishnamurthy, N., Jhaveri, V. and Abraham, J.A., 1998, "A Design Methodology for Software Fault Injection in Embedded Systems", Proceedings of the Workshop on Dependable Computing and its applications, pp. 12

Lattner, C. and Adve, V., 2004, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", Proceedings of the International Symposium on Code Generation and Optimization, pp. 75-86, doi: 10.1109/CGO.2004.1281665

Mehlitz, P.C. and Penix, J., 2005, "Expecting the unexpected – radiation hardened software", NASA Ames Research Center, pp. 10.

Mukherjee, S.S., Shrewsbury, M.A., Weaver, C., Emer, J. and Reinhardt, S.K., 2003, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor", Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 29-40, doi: 10.1109/MICRO.2003.1253181

Normand, E., 1996, "Single Event Upset at Ground Level", IEEE Transactions on Nuclear Science, Vol. 43, No. 6, pp. 2742-2750, doi: 10.1109/23.556861

Oh, N., Shirvani, P.P. and McCluskey, E.J., 2002, "Control-flow Checking by Software Signatures", IEEE Transactions on Reliability, Vol. 51, No. 1, pp. 111-122, doi: 10.1109/24.994926

Pan, Z. and Eigenmann, R., 2006, "Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning", Proceedings of the International Symposium on Code Generation and Optimization, pp. 319-332, doi: 10.1109/CGO.2006.38

Rech, P., Aguiar, C., Ferreira, R., Silvestri, M., Griffoni, A., Frost, C. and Carro, L., 2012, "Neutron-Induced Soft Errors in Graphic Processing Units", IEEE Radiation Effects Data Workshop, pp. 1-6, doi: 10.1109/REDW.2012.6353714

Saxena, N. and McCluskey, E., 1990, "Control Flow Checking using Watchdog Assists and Extended-Precision Checksums", IEEE Transactions on Computers, Vol. 39, No. 4, pp. 554-559, doi: 10.1109/12.54849

Vemu, R., Gurumurthy, S. and Abraham, J.A., 2007, "ACCE: Automatic Correction of Control-Flow Errors", IEEE International Test Conference, pp. 1-10, doi: 10.1109/TEST.2007.4437639