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

Rafael B. Parizi, Ronaldo R. Ferreira, Luigi Carro, and Álvaro F. Moreira

Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
{rbparizi, rrferreira, carro, afmoreira}@inf.ufrgs.br

Abstract. This paper characterizes how compiler optimizations impact software control-flow reliability when the optimized application is compiled with a technique to enable the software itself to detect and correct radiation induced soft-errors occurring in branches. Supported by a comprehensive fault injection campaign using an established benchmark suite in the embedded systems domain, we show that the careful selection of the available compiler optimizations is necessary to avoid a significant decrease of software reliability while sustaining the performance boost those optimizations provide.

Keywords: compiler optimization, compiler orchestration, embedded systems, fault tolerance, LLVM, radiation, reliability, soft errors, tuning.

1 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, letting this task to the compiler itself by flagging if it should be less or more aggressive in terms of performance.

Industry already offers microprocessors built with 22 nm transistors, with a prediction that transistor's size will reach 7.4 nm by 2014 [1]. 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* [2]. Radiation particles originated from cosmic rays when striking a circuit induce bit flips during software execution, and since transistors are becoming smaller there is a higher probability that 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 sensitive to radiation that their usage will be compromised even at the sea level, as predicted in the literature [3]. In [4] it is shown that modern 22nm 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 [5].

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 [6]. This creates the *dark silicon* problem [6]: 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 and high pricing of those radiation hardened components. For instance, a 25 MHz microprocessor has a unitary price of U$ 200,000.00 [7]. This high 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 is the major constraint a cheaper but yet effective approach for reliability against radiation is necessary.

Software-Implemented Hardware Fault-Tolerance (SIHFT) [8] 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 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 solve the problem), the impacts of the control-flow ones is yet not well understood. Because 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) [9] control-flow SIHFT technique, which was chosen 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 [10] suite. For the sake of clarity, the ACCE technique is briefly reviewed in Section 2. Section 3 presents the fault model we assume and the methodology used in this paper. Finally, Section 4 presents the impact of individual and cumulative optimization passes using the LLVM [11] as the production compiler.

2 Automatic Correction of Control-flow Errors

ACCE [9] is a software technique for reliability that detects and corrects control-flow errors (CFE) due to random and arbitrary bit flips that might occur during software execution. The hardening of an application with ACCE is done at compilation, since 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 ACCE works in two separate subsections, one dedicated to error detection and the other to error correction in the subsections 2.1 and 2.2, respectively. The reader should refer to the ACCE article for a detailed presentation and experimental evaluation [9]. The fault model that ACCE assumes is further described in Section 3.

2.1 Control-Flow Error Detection

ACCE performs online detection of CFEs by checking the signatures in the beginning and in the end of each basic block of the control-flow graph, thus, ACCE is classified as a *signature checking* SIHFT technique as termed in the literature. The basic block signatures are computed and generated during compilation; the signature generation is critical because it needs to compute 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. Fig. 1 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 ACCE creates for each function in the application two additional blocks, the function entry block and the *Function Error Handler* (FEH). For instance, Fig. 1 depicts a portion of two functions, *f1* and *f2*, both owning *entry blocks* labeled as **F1** and **F2**, and function error handlers, 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 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 control flow error has just been detected and the control should be transferred to the corresponding FEH block of the function where execution currently is at. 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 due to a soft error.

Fig. 1 depicts an example of the checking and update of signatures performed in execution time that occurs in a basic block. In this example, the control-flow error occurs in the block **N2** of function **F1**, where an illegal jump incorrectly transfers the

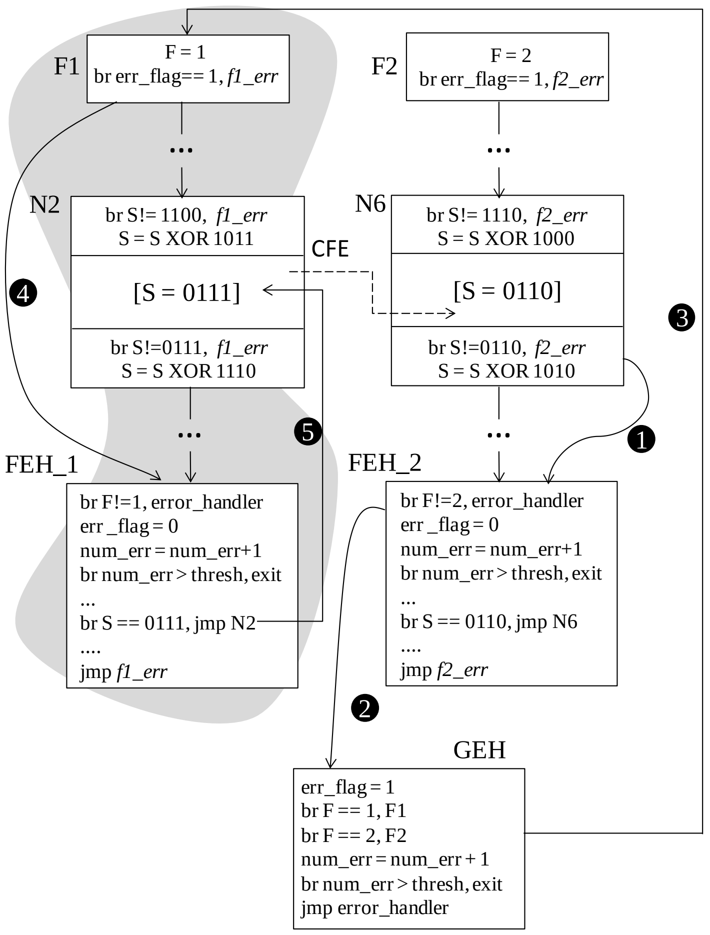

Fig. 1. Depiction of how the control is transferred from a function to the basic blocks that ACCE has created when a control-flow error occurs during software execution. In this figure, there is a control flow error (dashed arrow) causing the execution to jump from the block N2 of function F1 to the block N6 of function F2.

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 error must signaled (step 1 in Fig. 1). In this example, the application branches to the address *f2_err*, making the application enter the **FEH_2** block (since the error was detected by a block owned by the function **F2**, the function error handler invoked is the **FEH_2**). At this point, the CFE was detected and ACCE can proceed with the correction of the detected CFE.

2.2 Control-Flow Error Correction

The correction process starts as soon as an illegal jump is detected by the procedure described in subsection 2.1, with the control flow transferred to the FEH corresponding to the function where the CFE was 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. 1) with the current function register **F**. If the error happened 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 was 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 will 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. 1, 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 came. However, since the CFE originated in the basic block **N2** of 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 was 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 since **F** = 1, **FEH_1** knows that it is the FEH capable of handling the CFE and, as such, sets the variable *err_flag* to 0. Finally, it searches for the basic block that has the signature equals the register S. Upon finding it, the control branches to this basic block, i.e. **N2** in Fig. 1 (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 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 ACCE.

3 Fault Model and Experimental 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. ACCE is capable of handling multiple bit flip as long as the bits flipped is within a same word. Since 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: faults that cause a word to change its value that does not change the behavior of the program nor its output. This could happen in the case the fault flipped the bits of a dead variable.

The ACCE technique was implemented as a transformation pass in the LLVM [11] production compiler, which performs all the modifications in the control-flow graph described in section 2 using the LLVM Intermediate Representation (LLVM-IR). The ACCE transformation pass was applied *after* the set of compiler optimizations, since doing in the opposite order a compiler optimization could invalidate the ACCE generated code and semantics.

Since ACCE is a SIHFT technique to detect and correct control-flow errors, the adopted fault model simulates three distinct control flow disruptions that might occur due to a control flow error. Remind 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:

1. *Branch creation*: the program counter is changed, transforming an arbitrary instruction (e.g. an addition) into an unconditional branch;
2. *Branch deletion*: the program counter is set to the next program instruction to execute independently if the current instruction is a branch;
3. *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 [12], which is an accepted fault injection methodology in the embedded systems domain, in order to perform the fault injection campaigns. The steps of the fault injection process are the following:

1. 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;
2. The execution trace in assembly language is extracted from the program execution with GDB;
3. A branch error (branch creation, deletion or disruption) is randomly selected. In average each branch error accounts for 1/3 of the amount of injected errors;
4. 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;
5. If the chosen instruction in step 4 executes n times, choose at random an integer number k with $1 \leq k \leq n$;
6. Using GDB, a breakpoint is inserted right before the k-th execution of the instruction selected in step 4;
7. 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;
8. 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 programs 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 [10] embedded benchmark suite: *basicmath*, *bitcount*, *crc32*, *dijkstra*, *fft*, *patricia*, *quicksort*, *rijndael*, *string search*, and *susan* (comprising *susan* corners, edge, and smooth).

4 Impact of Compiler Optimizations on Control-Flow Reliability of Embedded Software

This section looks at 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. ACCE performs detection and correction of control-flow errors, 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 section 3.

The impact of the compiler optimizations when compiling for reliability is measured in this paper using the metric *Relative Improvement Percentage* (RIP) [13]. 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.

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

Fig. 2 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. Thus, for each application have 58 different versions (points in the chart). Fig. 2 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 just picking appropriate optimizations that facilitates for ACCE the process of error detection and correction. However, we also see 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 impacts on the RIP of reliability. Let us consider the *block-placement* optimization, which is represented by the white diamond in Fig. 2. In

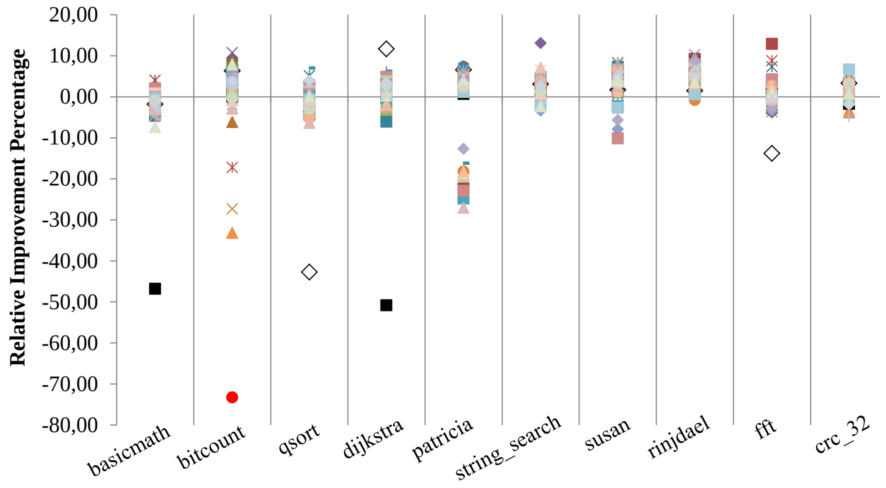

Fig. 2. Relative Improvement Percentage for the error correction rate of applications hardened with ACCE under further compiler optimization. Each hardened application was compiled with a single optimization at a time, but all applications were compiled with the 58 LLVM optimizations, thus, each hardened application has 58 versions. The baseline (RIP = 0%) is the error correction rate of the hardened application compiled without any LLVM optimization. Each point in the chart represents the application with one optimization protected with ACCE.

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 have this behavior (increasing RIP for sovme 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.

Fig. 3 depicts the RIP of a selected subset of the 58 LLVM optimizations, making it clear that even within a small subset the variation in RIP for reliability is far from

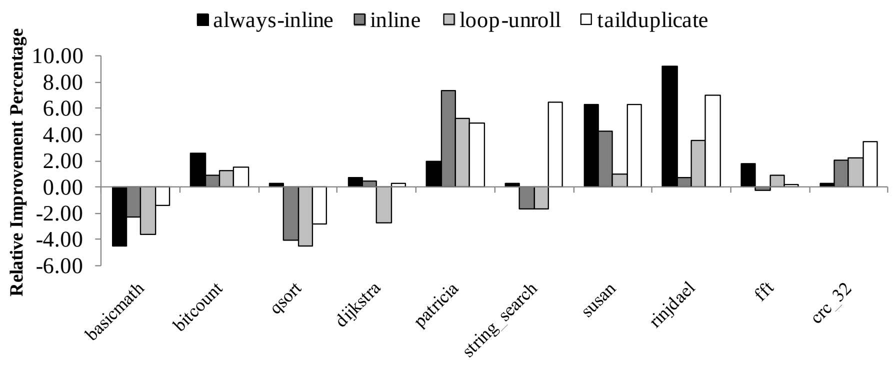

Fig. 3. Relative Improvement Percentage of a selected subset of the 58 LLVM optimizations. The baseline (RIP = 0%) is the error correction rate of the hardened application compiled without any LLVM optimization.

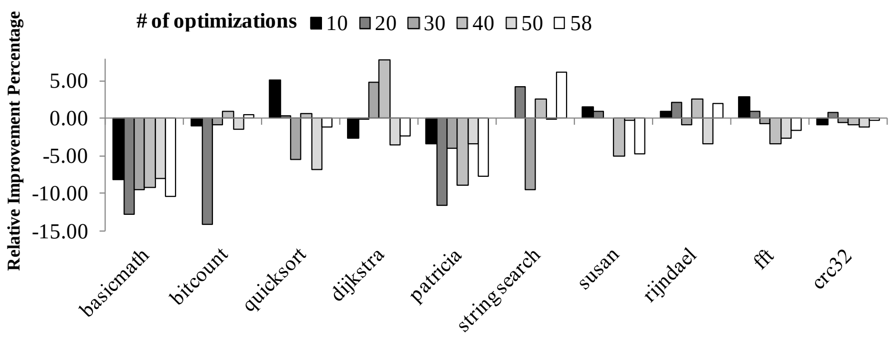

Fig. 4. Relative Improvement Percentage of random subsets of the 58 LLVM optimizations with a varying number of optimizations for each different subset: 10, 20, 30, 40, 50, 58 optimizations. The RIP for each subset was measured taking the average of 6 random subsets for each subset size. Hence, distinct possible optimizations subsets were considered. The baseline (RIP = 0%) is the error correction rate of the hardened application compiled without any LLVM optimization.

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 important to also examine if successive optimization passes could compromise or increase software reliability of a hardened application. Fig. 4 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. 4 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, choosing optimizations with the object of optimizing performance without previous knowledge of how the chosen optimizations influence together 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. Fig. 4 confirms that some applications are less sensitive to the effects of compiler optimizations, e.g. the *crc32* has its RIP within the [−1.11%, 0.73%]. On the other hand, *basicmath*, *bitcount*, and *patricia* are jeopardized. 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. 2), evidencing 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 choosing of compiler optimizations requires the software designer to take into consideration that some optimizations may not be adequate in terms of reliability for a given application. Moreover, data 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.

5 Related Work

Much attention has been devoted to the impact of compiler optimizations on program performance in the 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) [13] is an analysis approach to identify the best sequence of optimizations of for a given application set using the GCC compiler. The authors discuss that simple *orchestration* schemes between the optimizations can achieve near-optimal results as if it was performed an exhaustive search in all the design space created by the optimizations. CE is a greedy approach that firstly compiles the programs with a single optimization, using this version as the baseline. From those baseline versions the set of *Relative Improvement Percentage* (RIP) is calculated, which is the percentage that the program's performance is reduced/increased (section 4 discussed RIP in details). 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) [14] 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 optimization. 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 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) [15] 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. This metric is influence by the application due to liveness: for instance, a dead variable has a 0% AVF because it is not used in a computation. The authors in [16] evaluate the impact of the GCC optimizations in the AVF metric by trying to reduce the *AVF-delay-square-product* (ADS) introduced by the authors. The ADS relates 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 –O3 optimization level is detrimental both to the AVF and performance, because for the benchmarks considered (MiBench) have increased the number of loads executed. Again, the *patricia* application was the one with the highest reduction in the AVF at 13%.

In [17] the authors analyze the impact of compiler optimizations on data reliability in terms of variable liveness. *Liveness* of a variable is the time period between the variable is written and it is last read before a new write operation. The authors conclude 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 we made in section 4. The 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: it is much faster to fetch the value of a variable when it is in the register than in 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) [18] is a software technique that does that for a class of programs where the program's output is a linear function of the input. The generalization of efficient data-flow SIHFT techniques such as DH (i.e. ECC of program variables) is still an open research problem.

6 Conclusions and Future Work

In this paper we characterized the problem of compiling embedded software for reliability, given that compiler optimizations do impact the coverage rate. The study presented in this paper makes clear that choosing 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 evidencing 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 engineer must be very careful when compiling safety critical embedded software.

Design space exploration (DSE) for embedded systems usually considers "classical" non-functional requirements, such as energy consumption and performance. However, this paper has shown the need for automatic DSE methods to consider reliability when pruning the design space of feasible solutions. This could be realized with the support of compiler orchestration during the DSE step. As future work we are studying how to efficiently extend automatic DSE algorithms to implement compiler orchestration for reliability against radiation induced errors.

Acknowledgments

This work is supported by 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

1. ITRS. ITRS 2009 Roadmap. *International Technology Roadmap for Semiconductors, Tech. Rep*, 2009.
2. S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. 2005. *Micro*, 25(6):10–16, Nov.
3. E. Normand. Single event upset at ground level. 1996. *IEEE Trans. on Nuclear Science*, 43(6): 2742–2750, Dec.
4. P. Rech et al. Neutron-induced soft-errors in graphic processing units. 2012. In *IEEE Radiation Effects Data Workshop.* (REDW '12) IEEE, 6 pp.
5. ARM Mali Graphics Hardware, http://www.arm.com/products/multimedia/mali-graphics-hardware/index.php.
6. H. Esmaeizadeh, et al, "Dark silicon and the end of multicore scaling," in ISCA '11: Proc. of the 38th Int. Symp. on Comp. Arch., 2011, 365–376.
7. P. C. Mehlitz and J. Penix. Expecting the unexpected – radiation hardened software. 2005. In Infocom @ American Inst. of Aeronautics and Astronautics.
8. O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, Software-Implemented Hardware Fault Tolerance. New York, NY, USA: Springer, 2006.
9. R. Vemu, S. Gurumurthy, and J. Abraham. ACCE: Automatic correction of control-flow errors. In *ITC '07. IEEE Int. Test Conf.*, pages 1–10, 2007.
10. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC-4. '01: Proc. of the IEEE Int. Workshop of Workload Characterization*, 3–14, 2001. IEEE.
11. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proc. of the int. symp. on Code generation and optimization*, pages 75–, Washington, DC, USA, 2004. IEEE.
12. N. Krishnamurthy, V. Jhaveri, and J. A. Abraham. A design methodology for software fault injection in embedded systems. In *DCIA '98: Proc. of the Workshop on Dependable Computing and its applications*, IFIP.
13. Zhelong Pan and Rudolf Eigenmann. 2006. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In Proceedings of the International Symposium on Code Generation and Optimization (CGO '06). IEEE, 319-332.
14. Kenneth Hoste and Lieven Eeckhout. 2008. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (CGO '08). ACM, 165-174.
15. S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. of the 36th annual IEEE/ACM Int. Symp. on Microarchitecture*, (MICRO 36). IEEE, 29–41.
16. T. M. Jones, M.F. P O'Boyle, O. Ergin, 2008. Evaluating the Effects of Compiler Optimisations on AVF. In *Workshop on Interaction Between Compilers and Computer Architecture* (INTERACT-12).
17. S. Bergaoui and R. Leveugle. 2011. Impact of Software Optimization on Variable Lifetimes in a Microprocessor-Based System. In Proceedings of the 2011 Sixth IEEE International Symposium on Electronic Design, Test and Application (DELTA '11). 56-61.
18. C. Argyrides, R. Ferreira, C. Lisboa, and L. Carro. Decimal hamming: a novel software-implemented technique to cope with soft errors. In *Proc. of the 26th IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nanotech. Sys*., DFT '11. IEEE, 2011, 11-17.