A Compiler Extension to Protect Embedded Systems Against Data Flow Errors

Brent De Blaere, Elias Verstappe, Jens Vankeirsbilck and Jeroen Boydens
Department of Computer Science
KU Leuven
Spoorwegstraat 12, 8200 Bruges, Belgium
{Brent.DeBlaere, Jens.Vankeirsbilck, Jeroen.Boydens}@kuleuven.be

Abstract – **External disturbances such as alpha particles, electromagnetic interference, or malicious external attackers can cause erroneous bit-flips in the hardware of modern embedded systems. A broad range of software-implemented error detection techniques have been presented in the past to safeguard embedded systems against these disturbances. Two well-known state-of-the-art techniques are SWIFT and SWIFT-R. However, since those solutions must be implemented in low-level code, such as assembly language, implementing them can be time-consuming and error-prone. To solve this issue, this paper describes a GCC compiler extension in the form of a plugin that can integrate the data flow error detection of SWIFT and SWIFT-R to any ARMv7-M program. We verify that the compiler implements the techniques correctly by performing fault injection campaigns on various case studies.**

Keywords – **GCC compiler plugin; embedded systems; redundancy; reliability; soft error detection;**

I. Introduction

The increasing use of embedded systems in harsh working environments makes their reliability increasingly important, especially in safety-critical domains such as medical devices, automotive systems, and avionics. In these environments, external disturbances like electromagnetic interference or high energy particles may impact the systems by creating unintended charges in the hardware components, causing a bit-flip in a memory cell of the microcontroller. This is often referred to as a soft error or a single event upset (SEU). Such a bit-flip can result in an erroneous jump in the executing process, called a control flow error (CFE), or in a corruption of data, called a data flow error (DFE).

To increase the reliability of embedded systems, several approaches are possible. Hardware-based countermeasures such as error correcting codes (ECC) are often used as a way to protect a system. Software-based solutions, however, have grown in popularity since they provide a more versatile, generally more applicable, and low-cost solution. Since these software-implemented techniques are designed to work without the need for any hardware modification, they can also be used on commercial off-the-shelf systems.

Software-implemented techniques rely on redundancy to detect errors. This redundancy can be enforced at two levels: high-level, i.e. using processes and threads, and instruction-level, i.e. adapting the instructions or source code of the program. High-level redundancy techniques require either an operating system or a superscalar processor. This, however, excludes a big part of the embedded system market, since approximately 35% of all embedded systems operate bare-metal [1]. In contrast, instruction-level redundancy techniques can be used without the need for an operating system. Although there are instruction-level techniques that should be used at high-level code, such as C or C++, the majority of these techniques are applied in low-level code, such as assembly language. This is because the compiler might remove redundant instructions during its optimization passes when compiling the high-level code.

Manually implementing a low-level soft error detection technique is, however, a time-intensive and error-prone process that requires a profound knowledge of both assembly language and the specific error detection technique. Moreover, since these techniques are applied on low-level code, the techniques must completely be reimplemented after every change in the high-level code. This often makes manually implementing an instruction-level redundancy technique unfeasible. In this paper, we present how such instruction-level techniques can automatically be implemented using a compiler extension. This massively decreases the time-to-market and development cost of systems that want to implement these soft error detection techniques.

In this paper, two well-known state-of-the-art error detection techniques are implemented in a compiler plugin: SWIFT and SWIFT-R [2], [3]. Software Implemented Fault Tolerance (SWIFT) is one of the best known software-implemented error detection techniques proposed by Reis et al. [2]. It uses a variation on EDDI [4] (EDDI+ECC) to detect data flow errors, combined with an advanced version of CFCSS [5] (EDDI+ECC+CF) to detect control flow errors. In this paper, we will focus on the data flow error detection (DFED). SWIFT, like most soft error detection techniques, calls an error handler when it detects an error. This error handler should be used to bring the program to a safe state, which typically is application specific. SWIFT-R (Software Implemented Fault Tolerance with Recovery) proposed by Chang et al. shows how SWIFT can be extended with triplication and majority voting to create a system that can recover from SEUs without an error handler [3].

The remainder of this paper is structured as follows: In Section II, the internal operation of the compiler extension is discussed. Section III goes over the implementation of SWIFT, followed by Section IV, which discusses the implementation of SWIFT-R. Next, the future work is discussed in Section VI. Finally, in Section VII, our conclusions are summarized.

II. GCC Compiler Plugin

To automatically implement SWIFT and SWIFT-R, a GCC compiler plugin, originally proposed by Vankeirsbilck et al., is extended [6]. This compiler plugin is an extension to the GNU Compiler Collection (GCC). GCC compiles high-level source code such as C and C++ to low-level machine code through three stages, with each stage consisting of a number of passes. A pass is a series of instructions that accomplishes a particular task during the compilation process,

978-1-6654-4518-4/21/$31.00 ©2021 IEEE

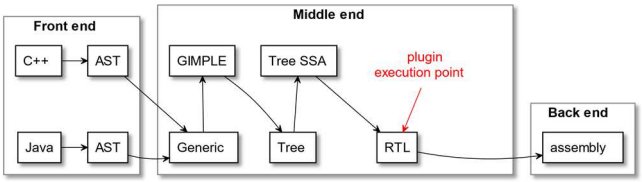

Fig. 1. Stages and intermediate representations used by GCC and the plugin execution point

such as dead code removal, inline expansion and loop optimization. The front-end deals with the high-level language itself, the middle-end deals with the intermediate language, and the back-end component deals with code specific for the target system [7]. During this process, the source code goes through several intermediate representations such as an *Abstract Syntax Tree* (*AST*) representation, *GIMPLE* and *Register Transfer Language* (*RTL*). These representations are shown in Fig. 1.

For the compiler plugin, the middle-end stage is most interesting, specifically the RTL representation at the end of its chain. This representation is very close to assembly language and corresponds to an abstract target architecture. It is used to describe the data flow at the register transfer level of the architecture. The plugin is implemented near the end of this representation to ensure that no major optimization passes, that might intervene with the instructions injected by the plugin, can follow. Currently, the plugin supports the ARMv7-M instruction set architecture, although support for other architectures will be added in the future.

A. Usage

To use the compiler plugin, a developer should first define the error handler to be used when an error is detected by an error detection technique. This function should be named *DFE_Detected*. When using C++, the handler should be defined in an *extern "C"* environment so that the name is not mangled during the compilation process [8]. Since this function is only called when an error is detected, the error detection technique should not be implemented on this handler. To be able to specify this behavior, we defined the *noProtection* function attribute. This attribute can be added to any function that does not need to be protected by the error detection technique. For some applications, it might be useful to exclude some functions to reduce unnecessary code size overhead, such as applications with a large initialization function that is only called once.

Every instruction-level soft error detection technique requires some registers to be specifically reserved for that technique. Therefore, the compiler should not use these registers for its compilation of the business logic. When using the GCC compiler, this can be done with the compiler flag *-ffixed-rN*, with *N* being the number of the register to be reserved. Some plugin-related compiler flags also have to be used. A first compiler flag indicates where the compiler can find the plugin executable. Secondly, the plugin argument *technique* indicates which DFE detection technique to implement. Finally, the optional *function* argument indicates which program function(s) to apply the technique to. When this argument is not given, all program functions will be protected.

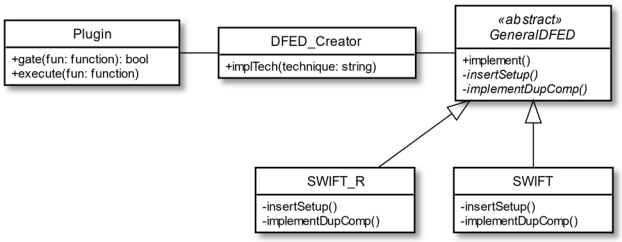

Fig. 2. UML class diagram of the DFE compiler plugin

B. GCC Plugin Implementation

The implementation of the compiler plugin is shown in Fig. 2 as a UML class diagram. This diagram shows the classes and methods needed for SWIFT and SWIFT-R. The plugin is designed according to the *Factory Method* and *Template Method* design patterns defined by Gamma et al. [9].

Upon first execution of the plugin, it registers itself as a compilation pass. Next, for each function, the compiler calls the *gate* method of the *Plugin* class. This method determines whether or not the given function should be protected by an error detection technique based on the conditions discussed in Section II.A. If the *gate* method returns true, the *execute* method is called. This method contains the compilation pass. It will evaluate the plugin-specific compiler flags discussed in Section II.B, instantiate a *DFED_Creator* object, and call its *implTech* method.

The *DFED_Creator* utilizes the *Factory Design* pattern. It knows which instruction sets are implemented in the plugin and how the implemented soft error detection techniques should be built. First, the used instruction set is determined. Next, the *DFED_Creator* object instantiates the class of the selected technique (*SWIFT* or *SWIFT_R* in Fig. 2).

Following the *Template Method* design pattern, all classes implementing DFED techniques inherit from the abstract *GeneralDFED* class and are implemented via its *implement* (template) method. This method calls the *insertError*, *insertSetup* and *implementDupComp* methods. The *insertError* method adds a call to the *DFED_error* handler at the end of the evaluated function and places a label at the position of the inserted call. The plugin can later use this label to branch to this call when an error is detected. The *insertSetup* and *ImplementDupComp* are both abstract functions that are implemented in the *SWIFT* and *SWIFT_R* classes discussed in Sections III and IV. This approach ensures that the algorithm's structure remains unchanged, while allowing the subclasses to provide implementation of the technique-specific steps.

III. SWIFT

In this section, the implementation of SWIFT in the compiler plugin is discussed. SWIFT can be split up in two parts: data flow error detection (DFED) and control flow error detection (CFED). This paper focuses on the DFED part of SWIFT.

The idea behind many DFED techniques is that either all calculations or a select set of calculations are performed twice on different registers. These registers are often referred to as shadow registers. At specific synchronization points in the program, the values of the original registers are compared to the values of the shadow registers. If a mismatch is detected, the program branches to its error handler. SWIFT uses the

TABLE 1. THE REGISTER MAP FOR SWIFT AND SWIFT-R, WHEN USING THE ARMV7-M INSTRUCTION SET

Original register	Shadow register(s)	
	SWIFT	SWIFT-R
r0	r7	r7, r8
r1	r8	r9, r10
r2	r9	r11, r12
r3	r10	-
r4	r11	-
SP (r13)	r6	r5, r6
LR (r14)	r12	r3, r4

principle of EDDI with some key refinements to improve its performance.

As mentioned previously in Section II.B, the *insertSetup* method of *SWIFT* will first be called. The method will insert some setup instructions to initialize the shadow registers. A register map is used to map original registers to the shadow registers. The register map is shown in Table 1.

Next, the *implementDupComp* method is called. The implementation of this method is shown in Algorithm 1. It first evaluates all instructions in the function and duplicates all those that should be duplicated (lines 3 and 4). This entails making a copy of the instruction, replacing the used registers with their corresponding shadow registers and placing the duplicate after the original instruction. In SWIFT, any instruction that modifies the value of a register should be duplicated. Compare instructions and control flow instructions such as branch instructions are therefore not duplicated. Exceptions are *store*, *push* and *pop* instructions. Since memory structures are often well-protected by hardware schemes like ECC and parity checking, SWIFT considers the memory to be outside of the sphere of replication. Therefore, *store* instructions are not duplicated. *Push* and *pop* instructions are also not duplicated since the duplicated instructions could push or pop a wrong value onto or from the stack. However, since *pop* instructions modify a register value, the corresponding shadow register should also be updated (lines 5 and 6 in Algorithm 1). Additionally, *pop* instructions will modify the stack pointer. The shadow register of this stack pointer should therefore also be updated after each *pop* instruction. A similar rationale holds for *push* instructions (lines 7 and 8).

Now that the instructions are duplicated, the correctness of the calculations can be verified by comparing the original registers to the shadow registers at certain synchronization points in the program code. The program's output defines program correctness. This means that, for a system that uses memory mapped I/O, any data written to memory should be correct. This is why EDDI, and consequently also SWIFT, uses *store* instructions as synchronization points. However, misdirected branches can also cause stores to be skipped. This is why EDDI also considers instructions affecting the control flow synchronization points. With SWIFT, incorrect transfer of control can be protected with its control flow error detection mechanism instead, which we will not discuss in this paper. Function calls to unprotected subroutines may also affect program output. Therefore, function calls are also considered synchronization points.

Before these defined synchronization points, the correctness of the program is checked. Each register is compared to its shadow register. If there is a mismatch, the program jumps to the error handler call discussed in Section II.A. However, in some circumstances, adding a compare instruction can break the control flow of the program. This is illustrated in the left part of Fig. 3. Instruction 8 is a conditional branch which normally depends on the compare instruction at line 2. However, by adding compare instructions at lines 3 and 5, the conditional flags are changed and the control flow of the program can be corrupted. To resolve this issue, the concept of *unsafe areas* is introduced. An unsafe area is defined as the area starting from an instruction that sets the condition flags and ending with the last instruction that relies on those specific condition flags. In the right part of Fig. 3, the unsafe area is marked. If a synchronization point resides inside an unsafe area, the compare and jump instructions are added before the unsafe area instead. The implementation of this logic can be seen in Algorithm 1 at lines 9 through 15.

One last consideration with regards to the implementation of SWIFT is call handling. When a function call occurs, the control of the program is transferred to a subroutine which may or may not be protected by the error detection technique. This means that when the subroutine returns, the original registers might have changed while the shadow registers might not have, causing the program to incorrectly transfer control to the DFE error handler at the next synchronization point. Therefore, all shadow registers are re-initialized after each function call. Because a function call is already a synchronization point, this has no major effect on the error detection capability.

Algorithm 1 *implementDupComp* function of *SWIFT*

```
1:  procedure SWIFT.implementDupComp()
2:    for each instr ∈ function do
3:      if shouldDuplicate(instr) then
4:        duplicate(instr)
5:      if isPopInstruction(instr) then
6:        updateShadowRegistersAfter(instr)
7:      if isPushInstruction(instr) then
8:        updateShadowSpRegisterAfter(instr)
9:    unsafeAreas ← findUnsafeAreas()
10:   for each instr ∈ function do
11:     if shouldCompareBefore(instr) then
12:       if instr ∉ unsafeAreas then
13:         addCmpBneBefore(instr)
14:       else
15:         addCmpBneBefore(unsafeAreas[instr])
17:   initShadowRegsAfterEachFunctionCall()
```

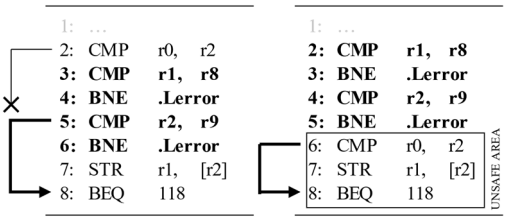

Fig. 3. (left): Corrupted control flow by the inserted instructions, (right): Using an unsafe area to avoid the control flow corruption

```
1: CMP   r8,  r7
2: BEQ   5        ; if r0' != r0"
3: MOV   r7,  r0  ;     r0'  ← r0
4: MOV   r8,  r0  ;     r0"  ← r0
5: MOV   r0,  r7  ; r0 ← r0'
```

Fig. 4. Low-level single-fault tolerant majority voting

IV. SWIFT-R

In this section, the implementation of SWIFT-R is discussed. The main difference between SWIFT and SWIFT-R is that SWIFT-R attempts to recover when a DFE is detected instead of transferring the control to the DFE error handler. This recovery is possible by using triplication. Where SWIFT duplicates a calculation, SWIFT-R triplicates the calculation, meaning that a second set of shadow registers is required. For that reason, SWIFT-R requires the reservation of two-thirds of all available registers.

The implementation of the *insertSetup* method SWIFT-R is similar to that of SWIFT. As can be seen in the last column of Table 1, the register map for SWIFT-R consists of two shadow registers instead of one.

In the *implementDupComp* method, where SWIFT would insert a comparison block that compares a register to its shadow register, SWIFT-R inserts a *recovery block* instead. These recovery blocks implement a majority voting procedure as shown in Fig.4. This implementation assumes a single event upset, meaning that only one fault can occur at a time. First, the two shadow registers are compared to each other. If they match, they must be correct, so they are copied to the original register to rectify a possible corruption. If they do not match, the original register is certainly correct, so its value is copied to both shadow registers.

V. Verification

To verify the correctness of the compiler plugin, it was used to compile five data processing case studies: *bit count*, *bubble sort*, *cyclic redundancy check*, *matrix multiplication* and *quick sort*. Each of these case studies have five different datasets to ensure that various execution paths are taken during execution. Each dataset was compiled multiple times: once without the compiler plugin, once with the compiler plugin using the SWIFT technique, and once with the compiler plugin using SWIFT-R. The ARM Cortex-M3 microprocessor was chosen as the target platform, as this 32-bit processor is widely used in industry applications.

First, the created assembly files of all the compilations were manually checked for correctness. Next, DFE fault injection campaigns were performed on a simulated ARMv7 Cortex-M3 processor provided by the Imperas instruction set simulator [10]. This allows for the experiments to be conducted at host speed, which speeds up the fault injection experiments. The fault injection procedure used in this study was previously discussed by Vankeirsbilck et al. [11].

The summarized results of the experiments are shown in Fig. 5. In this figure, the silent data corruption (SDC) is indicated in red. This is the percentage of injected faults that resulted in a corrupt output of the program but was not detected. The green boxplot (*Det*) shows the percentage of injected faults that were detected by SWIFT. The results show that, as expected, bot SWIFT and SWIFT-R reduce the amount of silent data corruptions significantly. This indicates that the

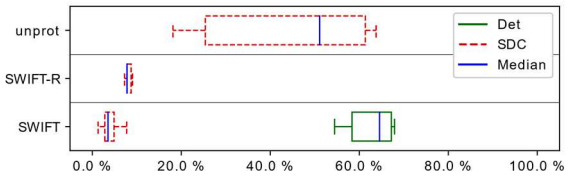

Fig. 5. Fault injection results when using the GCC extension

GCC plugin works as expected. The remaining SDCs are largely due to a corrupted control flow caused by the injected errors, which is currently not yet checked by our implementation of SWIFT and SWIFT-R.

VI. Future Work

Using the plugin makes it very convenient to implement a soft error detection technique on any algorithm in a matter of seconds. The work on SWIFT and SWIFT-R is, however, only fully complete when the CFED portion is also implemented. Additionally, further versions of SWIFT and SWIFT-R like Selective SWIFT-R could also be included for algorithms where register availability is a major concern [12].

VII. Conclusion

Manually implementing a soft error detection technique is a tedious, slow and error-prone task. Therefore, we presented our GCC compiler extension for SWIFT and SWIFT-R. This plugin works on the RTL language, a low-level intermediate representation which can directly be translated into machine code or assembly code. The internal working of the compiler plugin was shown and discussed, after which we demonstrated that the low-level implementation of SWIFT and SWIFT-R work as expected by performing fault injection experiments on five different case studies. Using the compiler extension significantly lowers the effort and time needed to implement the SWIFT and SWIFT-R techniques.

References

[1] Aspencore, “2019 Embedded Markets Study, Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments,” Mar. 2019. Accessed: Mar. 19, 2021. [Online].

[2] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, “SWIFT: Software implemented fault tolerance,” in *Proceedings of the 2005 International Symposium on Code Generation and Optimization, CGO 2005*, 2005, vol. 2005, pp. 243–254.

[3] J. Chang, G. A. Reis, and D. I. August, “Automatic Instruction-Level Software-Only Recovery,” in *International Conference on Dependable Systems and Networks (DSN’06)*, 2006, vol. 2006, pp. 83–92.

[4] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Error detection by duplicated instructions in super-scalar processors,” *IEEE Transa. Reliab.*, vol. 51, no. 1, pp. 63–75.

[5] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Control-flow checking by software signatures,” *IEEE Trans. Reliab.*, vol. 51, no. 1, pp. 111–122.

[6] J. Vankeirsbilck, H. Hallez, and J. Boydens, “Automatic Implementation of Control Flow Error Detection Techniques,” in *ACM International Conference Proceeding Series*, Sep. 2019.

[7] S. Kwong, “An Overview of GCC,” *Bitboom Technical*

Blog, 2018. http://bitboom.github.io/an-overview-of-gcc (accessed Jun. 02, 2021).

[8] D. Herity, "C++ in embedded systems: Myth and reality," *EE Times-India*, Feb. 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Software*, 37nd print. Boston: Addison-Wesley, 1996.

[10] Imperas Software, "ISS - The Imperas Instruction Set Simulator." https://www.imperas.com/iss-imperas-instruction-set-simulator (accessed Mar. 21, 2021).

[11] J. Vankeirsbilck, J. Van Waes, H. Hallez, and J. Boydens, "A New Approach to Selectively Implement Control Flow Error Detection Techniques," in *Lecture Notes in Networks and Systems*, vol. 96, Springer, 2020, pp. 704–715.

[12] F. Restrepo-Calle, A. Martínez-Álvarez, S. Cuenca-Asensi, and A. Jimeno-Morenilla, "Selective SWIFT-R: A flexible software-based technique for soft error mitigation in low-cost embedded systems," *J. Electron. Test. Theory Appl.*, vol. 29, no. 6, pp. 825–838.

978-1-6654-4518-4/21/$31.00 ©2021 IEEE