Embedded Software Resilience

Piet Cordemans, Edouard Thyebaut, Davy Pissoort and Jeroen Boydens

Abstract – **Soft errors are faults which are not caused by defective hardware, rather they are induced due to noise or transient events. In this paper we describe defensive programming and redundancy techniques to detect and deal with soft errors. These techniques are categorized according to data and control flow errors.**

Keywords – **Embedded Software, Software Resilience, Soft Errors**

I. Introduction

Electronic systems, more specifically microcontroller systems, are susceptible to transient events and noise due to electromagnetic interference or radiation. These issues might introduce a sudden change of state in a single bit. Errors induced this way lead to erratic behavior. Due to the non-deterministic nature of these events, it is impossible to predict where and when these errors will occur.

In this paper we focus on software-based techniques for embedded systems to detect and deal with bit flips due to such issues.

In the introduction, we define soft errors, identify critical areas in a typical microcontroller based system and introduce basic concepts in software-based resilience. Then, in Section 2 we describe control flow errors and detection techniques. Finally, we describe data flow errors and detection in Section 3.

A. Soft errors

Soft errors are erroneous signals or states, which are not caused by defect of a particular component or a mistake in design or implementation [1]. Rather, a soft error, also known as single-event upset, is the change of an instruction or a data value in a program, which leads to an erroneous state. Soft errors are non-deterministic, as such it cannot be ascribed to a particular system state or state transition. Root causes of soft errors are due to externally triggered events caused by electromagnetic, electrical and radiation issues, such as bit flips, noise and transient events. Transient events are temporary bursts of energy which are typically undesired. For instance overshoot at the rising edge of a block signal. Noise and transient events induce bit flips in registers and memory locations, which in turn affect system state, control flow or lead to incorrect data results.

While rebooting the system typically solves the problem, this only lasts until a new soft error occurs. Furthermore, rebooting interrupts the continuous operation of the system and might result in the loss of data or temporary control. Moreover, soft errors are typically hard to detect, as it is impossible to predict which registers or memory locations will be affected.

Typically, hardware-based solutions are proposed to deal with soft errors, such as shadow registers, watchdog timers, triple mode redundancy, etc [2]. However, when hardware is in production, it is costly to make changes to its design. Software solutions are typically less costly, especially when a solution is introduced after the hardware is produced [3].

B. Critical areas

Soft errors can be classified according to the resulting effect. This effect depends on the place of the fault in the underlying hardware of the microcontroller-based system.

On the one hand, when a soft error affects the data memory, it typically results in corruption of data. Data flow errors are also introduced when values from (or pointers to) the data memory region are loaded in registers and a bit flip occurs on that particular register.

On the other hand, when the instruction pointer becomes corrupted, soft errors have introduced a fault in the control flow of the microcontroller. Instruction pointer corruption has multiple potential causes:

(1) a bit flip might occur at the instruction pointer register.

(2) When calling a function, the current instruction pointer is pushed onto the stack. When the function returns, the instruction pointer is popped from the stack and placed in the instruction pointer register. This is called the return address and this address might be corrupted when that particular memory region on the stack undergoes a bit flip.

(3) If the stack pointer, this is the register containing the address of the current stack frame, is corrupted, the current function context becomes corrupted. Furthermore, as the value of the address of the previous frame is also stored on the stack, the entire stack call chain becomes invalid.

(4) When calling into a function, this is effectively implemented as jumping to the memory address where the function has been defined. However, when the value of the address of the function has been altered in the program memory, this will result in executing arbitrary instructions. These instructions might be part of the function, or lie outside the function scope. When an arbitrary instruction in the function is executed first, the function itself might not be correctly initialized or missed some critical operations. Outside the function scope, another distinction can be made in executing instructions defined by the programmer or executing in an undefined region of the program memory. For instance when jumping to an unintended function. On the other hand, when the address lies outside the defined program memory range, the instruction pointer will execute undefined behavior.

P. Cordemans and J. Boydens are with the Technology cluster Computer Science,

D. Pissoort, is with the Technology Cluster ESAT,

E. Thyebaut serves as an intern at the Technology cluster Computer Science,

Faculty of Engineering Technology, KU Leuven campus Ostend, Zeedijk 101, 8400 Ostend, Belgium, e-mail: {piet.cordemans, jeroen.boydens, davy.pissoort}@kuleuven.be

(5) When the destination address is computed, a jump might have an incorrect destination address when one of the operands is altered by a bit flip.

(6) Conditional branches are susceptible to soft errors in two manners. One the one hand, as with (4) the destination address of the alternative condition might have been corrupted. On the other hand, the condition on which the appropriate branch has to be decided might be corrupted. This can be ascribed to a bit flip in the register which contains the arithmetic and logic operation flags.

C. Software-based resilience

When dealing with soft errors, the particular nature of these errors requires a meticulous implementation of software-based techniques to obtain resilience. As any register, in particular the instruction pointer and stack pointer, or an arbitrary value in data memory and program memory regions are susceptible to soft errors, the problem might manifest in any value or instruction.

Consequently, when techniques dealing with these issues are applied on the level of individual instructions, functions, modules or the system, they should be applied to any given instruction, function or module. Or else these techniques will not cover the scope of the complete system and soft errors will remain undetected and uncorrected. Embedded software techniques dealing with soft errors can be put into three categories: defensive programming techniques, redundancy and hybrid solutions involving hardware.

Defensive programming is a design technique at function level to ensure correct function execution under unforeseen conditions. In general, the programmer reduces the number of assumptions and tries to handle all possible error states. Assumptions on data are implemented as assertions in preconditions and postconditions.

Preconditions are conditions which should hold true when the function is entered. For instance these include valid ranges of input values, the necessary memory reservation, particular locks obtained or the system is in a specific global state, which is required to execute the function.

Postconditions are conditions which will hold true when the function exits and returns to its calling context. For instance these include particular ranges of output values, freeing or obtaining a specific memory chunk, obtaining or releasing a certain lock or putting the system in a particular state.

In defensive programming, preconditions and postconditions are defined in a set of assertions [4], which are appropriately checked when entering and exiting a function. If any of these assertions fail, an error is raised.

Redundancy is a general concept, which can be applied at the level of values, functions, modules or even the system.

At the level of values, redundancy can be implemented on the one hand as maintaining copies of values. Multiple storage is effective as the non-deterministic nature of soft errors implies that the probability of a common error in a value is virtually non-existent. However, the cost of redundancy is reflected in multiplying the size of the memory needed every time for every additional copy. On the other hand, storing the entire value is unnecessary as techniques such as parity or cyclic redundancy checks allow to check the integrity of a particular value with a minimal overhead in storage.

Redundancy at the function level can be applied as a function can be identically implemented or implemented with similar externally observable behavior. By executing both functions and comparing the results it is possible to detect a mismatch caused by a soft error between the two implementations of the functions. Alternatively, instead of duplicating the entire function, function arguments and return values can be duplicated in the function header itself. By checking the values of the original parameter and the copied parameter, soft errors will be detected.

II. CONTROL FLOW ERRORS

Control flow errors are soft errors which result in a corrupted instruction pointer or lead to unexpected execution paths of the system.

A. Functions

When using defensive programming at function scope by validating preconditions, starting to execute the function in a faulty state is signaled. Furthermore by asserting the postconditions, soft errors on function completion will also be detected. Combined, these predicates will signal the execution of unexpected paths in the embedded software.

Function redundancy can be implemented by replicating the function as a whole or duplicating function parameters and return values. By asserting the correspondence of the results, soft errors introduced in a single function scope are detected. An alternative is to use a different implementation of the function with similar external behavior to avoid common-mode errors, i.e. errors with a common cause.

In order to detect an astray instruction pointer, function tokens can be introduced [5]. A function token is a constant value assigned to the function at compile-time. Before the function is called, the value of the function token is pushed onto a stack data structure. When entering and leaving the function scope, the top value is compared with the function token. If they do not correspond, the instruction pointer was not supposed to contain an address of the function scope. Mark that, when leaving the function the value is popped from the stack.

A proof of concept of these three techniques at function level has shown that they are complementary to each other. Defensive programming is able to signal a faulty state while executing the function. Whereas function redundancy is able to detect soft errors within the range of expected values. Finally, function tokens allow to detect erratic behavior of the instruction pointer. The same stack is used to push the function address which serves as the function token, as well as the arguments and return values. Defensive programming is implemented as assertions on the expected boundary conditions when entering and leaving the function.

B. Conditional branches

Conditional branches allow to execute different paths in the program. When soft errors affecting the decision are

introduced, these lead to a faulty state and execution path for the rest of the program. Therefore special consideration is given to the registers, variables and addresses affecting the condition. Introducing redundancy at condition checking covers the condition code register [3]. In Figure 1, the decision tree is shown. By duplicating decision and address variables and comparing their result the entire conditional branch is covered.

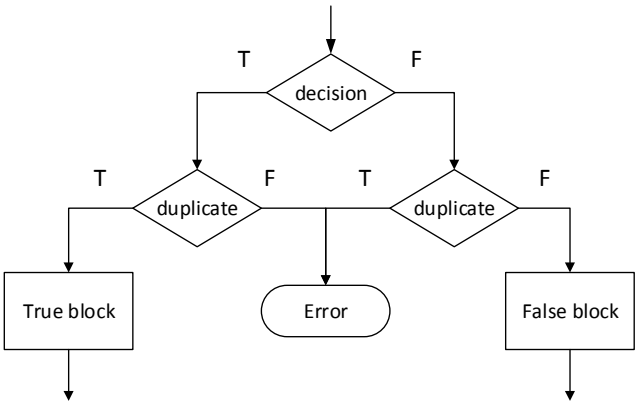

Fig. 1. Conditional branch duplication

C. Program memory

Although the techniques in Subsection B and C are profound, redundancy techniques and explicit assertions introduce overhead which is typically a criterion to minimize in embedded systems. There are two programming techniques which deal with an astray instruction pointer. They affect the program memory directly without affecting the run-time behavior or required memory.

On the one hand, unused locations in the program memory are potentially dangerous when an astray instruction pointer contains such an address. Filling these locations with "*no operation*" instructions avoids unwanted behavior [5]. Before the end of the program memory is reached an error handling routine can be provided to catch the lost instruction pointer and take an appropriate corrective action.

On the other hand, immediately calling the error handler at every unused memory location is an improvement to the previous technique. However, this requires that microprocessor architecture supports branches to every program memory location in a single program word.

Both techniques will not affect the run-time behavior in a normal situation nor require extra memory allocation.

D. Interrupts

An interrupt is an external signal which results in changing the value of the instruction pointer. It is an asynchronous event, which is hard to differentiate from a soft or transient error. Two complementary precautions deal with unintentionally invoked interrupts by soft errors.

Keeping the interrupt service routine as short as possible is a design directive in order to properly detect as many interrupts as possible and avoid interrupt masking of a lower or equal priority level. However, this design directive also has its merit in order to mitigate the effects of a soft error induced interrupt. When an interrupt is raised, the interrupt is only registered and the normal execution is resumed as soon as possible. When the state machine checks if any interrupts have been registered, it can perform additional checks, to see if an interrupt effectively was raised by an intended external signal rather than a soft error induced interruption.

Microcontroller systems include a number of interrupt vectors, which might not be used in an application. However, if a soft error is introduced these interrupt vectors might be called and subsequently unexpected instructions are executed. In order to prevent the unexpected behavior, unused interrupt vectors can be implemented with a call to an error handling routine.

As in Subsection II.C these techniques do not affect the normal run-time behavior and do not have an impact on memory usage.

III. Data Flow Errors

Data flow errors are soft errors which affect data values or data pointers. These can lead to erroneous data or missing data. Furthermore if the data is used to decide upon a transition to a new state, data flow errors might lead to a faulty state. Data flow error solutions typically involve redundancy, either in space (duplication, error correction codes) [6] or in time (multiple input read).

A. Data memory

Variables can be duplicated and synchronized during execution. Data flow errors are detected when the values are out of sync.

When storing data, data can be duplicated in the same physical memory system or duplicated across different non-volatile memory systems. Though some hardware technologies are more susceptible to soft errors of a specific kind than other non-volatile memory systems, the random nature of soft errors ensures there is a very low probability that the duplicated data will also be affected. Therefore there is no reason related to soft errors to duplicate the data across multiple memory systems. However, if hardware would be damaged permanently, data integrity of both original and duplicated values is compromised.

B. Error detection codes

Variable duplication has a severe impact on the usage of memory. Namely, a multiple of the minimal amount is needed depending on the number of duplicated values.

In contrast, error detection codes use less memory, with guarantees that a specific number of bit errors depending on the algorithm can be detected. For instance a parity bit is able to detect any odd number of bit errors. On the other hand, a Cyclic Redundancy Check (CRC) is used extensively in data networks and is suited to detect multiple bit errors in a single data word.

C. Input and output

When capturing a digital signal, the current voltage level on the input pin might be the result of a transient error. In

order to prevent reading a faulty level, a burst of reads is needed to capture the signal. The majority of levels read can be considered as the correct level to capture.

Considering a continuous analog signal, a burst of reads might give an average level of the analog value. However, when interested in the continuous behavior, it might be more interesting to check if the next value is in the range of the maximum slew rate when considering the previous value.

Finally when generating an output, the output value is stored in a register resembling the value of the output pin. These I/O registers can be regularly overwritten in order to prevent any soft errors occurring at that register.

IV. Future Work

A. Design patterns

A first direction of future work is to introduce design patterns for the strategies in this paper. A design pattern is an informal way of documenting a solution to a design problem. A pattern does not only describe why the proposed solution to a particular problem is considered a good one, it also gives an example and explains the relationship of the pattern to other patterns. A pattern language for software-based embedded resilience techniques collects these patterns.

B. Measurements

A second direction of future work is located in the field of the effect of the strategies proposed onto the embedded system. Redundancy is a common solution in both control and data flow errors. However, a typical characteristic of redundancy is that it requires more memory. This also increases the probability of the occurrence of a soft error. This trade-off should be careful considered and more data is needed to decide upon the extent of duplication needed, its cost and problems associated with it. Currently, implementations of the various techniques are developed which will be evaluated.

C. Corrective actions

Finally, when detecting a soft error, a corrective action must be taken. There are several options including, resetting the system, putting the system in a generic safe state or performing error recovery. These actions can be performed by techniques such as voting mechanisms, applying correction codes, introducing transactional behavior or reverting to a golden reference.

D. Invariants

In defensive programming, the concept of a condition, which should hold true at specific moments during execution is called an invariant. As it is impossible to continuously monitor all these assumptions pragmatic approaches should be adopted when dealing with soft errors. For instance, loop invariants could be asserted every time the loop condition is checked.

V. Conclusion

Hardware-based solutions to deal with soft errors are costly and must be introduced during design. Alternatively, software-based solutions can be implemented afterwards and do not require any additional resources.

This paper describes software techniques to detect soft errors in data and control flow. By applying these techniques, embedded software resilience can be improved.

First, following guidelines concerning unused program memory locations and interrupt vectors do not require any investment, while providing a basic solution for astray instruction pointers.

Secondly, redundancy techniques can be applied at any given level, both in control flow and data. However, these techniques have an impact on the memory footprint.

Finally, when defensive programming is applied at the function level, it allows to detect erroneous behavior early on. Consequently, appropriate corrective actions could be undertaken.

References

[1] S. A. Asghari, O. Kaynak and H. Taheri, "An Investigation into Soft Error Detection Efficiency at Operating System Level," *The Scientific World Journal*, vol. 2014, 2014.

[2] R. Carlton, G. Racino and J. Suchyta, "Improving the Transient Immunity Performance of Microcontroller-Based Applications," 2005.

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

[4] R. Venkatasubramanian, J. P. Hayes and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, 2003.

[5] K. Wilken and J. P. Shen, "Continuous signature monitoring: low-cost concurrent detection of processor control errors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 9, no. 6, pp. 629-641, 1990.

[6] M. Maghsoudloo, H. R. Zarandi and N. Khoshavi, "An efficient adaptive software-implemented technique to detect control-flow errors in multi-core architectures," *Microelectronics Reliability*, vol. 52, no. 11, pp. 2812-2828, 2012.

[7] "Microcontroller in a Harsh Environment,", Technical report at Atmel, 2007.

[8] A. Li and B. Hong, "A low-cost correction algorithm for transient data errors," *Ubiquity*, vol. 7, no. 21, pp. 2-15, 2006.