Hardening Development Environment
for Embedded Systems

F.Restrepo-Calle¹, A.Martínez-Álvarez¹, F.R.Palomo², S.Cuenca-Asensi¹, and
M.A.Aguirre²

¹ Computer Technology Department, University of Alicante, 03690 Alicante, Spain
² Department of Electrical Engineering, University of Sevilla, 41092 Sevilla, Spain

Abstract. This paper presents a novel low cost development environment for automatic hardening against Single Event Effects (SEE) of embedded systems through software redundancy. The environment is based on a generic architecture to handle multiple targets and is comprised of an automatic hardening compiler and an instruction set simulator. As a case study, it is developed a compiler back-end for the PicoBlaze soft-micro and implemented several fault tolerance techniques.

1 Introduction

Typically, reliability issues in mission critical embedded systems have been mitigated using redundant hardware [1]. Although these solutions are effective in fulfilling reliability requirements, these methods have become difficult to be implemented today. Firstly, the development of a custom hardened microprocessor can be very costly. Secondly, electronic components have become more sensitive to Single or Multiple Event Effects induced by radiation. As technology shrinks, the voltage source level and the noise margins are reduced [2] [3]. Therefore, during recent years several proposals based on redundant software have been developed, providing detection and error correction capabilities to applications. These works are especially motivated by the need for low cost solutions (using Commercial Off The Shelf — COTS hardware), ensuring an acceptable level of reliability [4] [5].

Among all software implemented hardware fault tolerance (SIHFT) techniques, those based on redundancy of instructions achieve better fault detection/correction results [1]. Rebaudengo et al. [6] proposed a high level instruction redundancy technique based on automatic ruled transformations that reports detection of 63% of the injected faults to the program data. On the other hand, using low level (assembler) instruction redundancy, Oh et al. [7] presented the EDDI technique (Error Detection by Duplicated Instructions) improving the detection rates and reducing considerably the overhead. The same authors proposed CFCSS (Control-Flow Checking by Software Signatures) [8] extending fault detection to the program execution flow. Reis et al. [9] presented an extension of these approaches called SWIFT (Software Implemented Fault Tolerance).
The above mentioned techniques are aimed to faults detection, but only few of them have been extended to allow the recovery of the system from those faults. Rebaudengo et al. [10] made an approach based on high level instruction redundancy, reporting fault coverage of 99.50%. Besides, Reis et al. [11] proposed the so called SWIFT-R, a technique based on instructions triplication and addition of majority voters before critical instructions, working with low level instructions.

Results from studied techniques show that those ones based on low level instruction redundancy offer lower code and data overheads, which is a critical characteristic for embedded systems. Furthermore, considering the execution time overhead, it is noticed that it can be reduced using the Instruction Level Parallelism (ILP) of superscalar [7] [8] and Very Long Instruction Word (VLIW) [9] [11] architectures.

The environment presented in this paper is based on a generic architecture that allows to handle multiple microprocessors. It is made up of an automatic hardening compiler and an instruction set simulator. The first one works using a generic instruction flow which is hardened by means of several automatic code transformations at low level (assembler). The second one allows to validate that the hardened version of the program is functionally equivalent to its original one.

As a case study for validating the environment, we have developed a compiler back-end for PicoBlaze (an 8 bit soft-micro) and several basic fault tolerance techniques were implemented. Thanks to their simplicity it was possible to fully understand the problem and to be prepared to apply this methodology subsequently to more powerful microprocessors such as MicroBlaze and LEON 3. In this way, we have chosen to work with soft-core microprocessors because they facilitate co-design hardening tasks.

The presented environment will allow the exploration of hybrid hardware/software solutions to obtain fault tolerant systems. By using our environment jointly with co-design techniques, it will be possible to achieve several trade-offs between reliability, performance and device area. Besides, by means of this environment it will be offered a low cost solution to incorporate fault tolerance features in embedded systems.

The remaining of the paper is organized as follows. Section 2 presents the proposed hardening development environment. Section 3 includes a case of study using the environment. Section 4 presents the experiments and their results. Finally, Section 5 concludes the paper and shows the plans for the future work.

2 Hardening Development Environment

According to the studied SIHFT techniques, it was possible to identify the main functionalities that the hardening development environment must supply: insertion of code transformations during compilation time, control flow analysis of the programs, management of architecture’s resources, etc. It was also noticed that better results were obtained using low level instruction redundancy.
We propose the use of a generic architecture to implement the hardening tasks. This generic architecture is useful to provide a uniform hardening core compatible with usual microprocessors by means of automatic code transformations at low level (assembler). Therefore, to provide all the needed tools to develop and evaluate fault tolerance techniques, it is proposed the environment scheme showed in Fig. 1.

Every code from a supported architecture is processed by a back-end compiler that generates a generic instructions flow. This flow represents a high level abstraction of a program that allows to perform a platform independent implementation of the hardening routines.

Using this scheme, the environment is prepared to take a code written for a supported architecture, to perform its compilation and generic hardening, and finally, to generate the output of the hardened code. The output can be re-targeted to the same architecture or to a different one.

Moreover, it is worth to mention that all the developed tools (generic hardening core and case study tools) are multiplatform, and have been successfully tested in Debian GNU/Linux (Kernel 2.6.30) and Windows XP SP3/Vista.

2.1 Generic Architecture

We took into account three topics to develop the generic architecture proposed in this paper: generic instructions, memory management and control flow graph.

Generic Instruction. The generic architecture is defined by means of generic instructions. Each generic instruction is composed of the following fields.

Address. Memory address where the instruction has been assembled by the back-end compiler.

Mnemonic. Original mnemonic of the instruction.

Generic Operator List. Generic operators that are present in the instruction. Each operator is a member of the list and each one has three fields, they are: operator type, addressing mode and real name. The Operator Type defines the kind of operator, such as: Register, Literal, Address or Flag. The Addressing Mode can be: Absolute, PC Relative, Register Indirect, Register, Immediate Literal, Absolute Direct, among others. Finally, the Real Name is the target’s operator original name.
Affected Generic Flag List. Generic flags affected by the execution of the instruction. Each generic flag is composed of a type and a name. The Generic Flag Type can be: Zero, Not Zero, Carry, Not Carry, Interrupt Enable, etc. The Real Name is the original flag name of the target architecture.

Instruction Type. This is used to classify the instructions. It is very important because the hardening process depends on this type. For example, it is different to handle an arithmetic instruction and a control flow instruction during hardening. Some of the supported types are: interrupt, directive, control flow, scalar arithmetic, scalar logic, scalar storage, scalar input output. When applicable generic instructions support 1, 2 or 3 operators. These are the common types for scalar architectures, but vectorial architectures are also supported.

Tool Message. This is a log that the environment tools use to register events.

Memory Management. A common task in the SIHFT techniques is the insertion of instructions into the original code during compilation time. Therefore, it is necessary to supply the following memory management facilities within the hardener tool: identification of the memory map, extraction of the code sections and memory map update. The following three possibilities were developed to update the memory map: dilation, displacement and reallocation.

Dilation. When one or more instructions are inserted into a memory section, it grows and the affected instructions addresses should be reassigned.

Displacement. If some instructions are inserted into a previous memory section, it is possible having an overlapping with the following section. Then this section must be completely moved, updating all its instructions addresses.

Reallocation. If there is a memory overflow caused by previous dilations and displacements, then it is needed to perform a complete reallocation of all memory sections. During this process, free memory space among memory sections is fully used. This situation may happen because of the reduced memory size in embedded systems.

Flow Control Graph. The generic architecture allows to identify the flow control graph from a given generic instruction flow. It is important to mention that the flow control graph is the key for most SIHFT techniques.

The flow control graph is represented by a directed graph. In order to build it, first we must identify the program’s basic blocks. A basic block is a group of instructions that are executed sequentially, without any jump instruction nor function call, excepting possibly the last instruction. Also, a basic block does not contain instructions being the destination of a call or jump instruction, excepting the first instruction. Each basic block represents a node in the graph. The flow control changes are represented in the graph as links among the nodes. Fig. 2(a) shows an example of a flow control graph.

Additionally, as it was proposed by Reis et al. [12], only store instructions can modify the system’s final state (generate real faults), then it is necessary to perform special verifications before the execution of these instructions. Therefore,
in this paper we propose that the nodes (basic blocks) of the flow control graph should be subdivided into subnodes after each store instruction (Fig. 2(b)).

2.2 Hardening Generic Core

**Hardening compiler.** This tool receives the generic instructions flow from the back-end compilers. Then, according to the options selected by the user, it applies the hardening routines. Finally, a new hardened generic instruction flow is produced, which is re-targeted to one of the supported architectures. These are the most important hardening options.

```
--method. Selects the fault tolerance technique to be applied.
--mcpu. Target microprocessor to generate the output.
--replicationRegisterLevel. Defines the register redundancy level, such as: 0 - less possible number of registers are redundant, for example in the instruction ADD s0, s1 only s0 register will be redundant; 1 - every register is redundant, for example in the instruction ADD s0, s1 both registers will be redundant, s0 and s1.
--replicationTimes. Defines the number of copies of each redundant instruction. Possibilities are: 0 - none (without instructions redundancy); 1 - one copy per instruction (duplicate); 2 - two copies per instruction (triplicate).
--voter. Used to define voter and recovery routines.
--NOlookAheadAvailableRegs. Disable the advanced registers search. This is an optimization that consists of finding available registers for replication purposes looking forward than the current node in the flow control graph.
```

**Instruction Set Simulator - ISS.** This tool simulates the generic instruction flow. It presents information about the state of the architecture’s resources during and after the simulation process.

Likewise, the ISS has the ability to verify if the functionality of the hardened programs matches the original non-hardened programs. For this purpose, a compiler pragma (called Output) was developed. It allows to insert directly into the programs source code their expected results. Using this compiler pragma, an option of the simulator called check-hardening was developed. It permits to the simulator to compare results of both original and hardened versions.

Moreover, after the simulation process, the ISS presents a brief summary to inform the code and execution time overheads of the applied hardening tech-
nique. Also, it performs a characterization of the simulated programs, informing the percentage of executed instructions by its type: arithmetic instructions, logical instructions, control flow instructions, etc.

Finally, the ISS is also able to emulate Single Event Upsets - SEU faults during the simulation by means of bitflips in the registers. The effect of the fault on the final results of the program are classified as: correct results, when correct results are obtained despite the fault; incorrect results, when incorrect results are obtained due to the fault; and hanged, if the fault causes a crash or infinite loop in the execution of the program.

Although, injection faults functionality is used to preliminary assess the fault coverage (FC) offered by the technique under test, SEU simulation at this level does not consider faults in user-hidden registers such as the ones in pipeline. Therefore, in order to assess more realistic FC results, it is highly recommended the utilization of other SEU injection tools (fault emulation tools).

3 Case Study

A compiler back-end for PicoBlaze has been developed. It takes KCPSM3 source code [13], performs lexical, syntactical and semantical analyses and finally generates a generic instruction flow as output.

This is a multiplatform compiler that provides a very accurate error localization, compared with any other PicoBlaze compilers.

Using the features of the hardening environment, several fault tolerance techniques (detection and correction fault techniques) were implemented. All of them based on the well known Triple Modular Redundancy (TMR) approach.

First implemented strategy (TMR1) can be summarized as follows.

1. Identification of nodes (basic blocks) and subnodes in the program.
2. Build the flow control graph of the program.
3. Triplication of the operation.
4. Insertion of majority voters and recovery procedures for protected registers at the following points: just before the last instruction of each node/subnode and also, just before any instruction being the destination of a jump or function call.
5. During the hardening process, majority voters and recovery procedures are dynamically injected when there are not enough available registers to replicate. By means of this, registers copies will be released to continue with the hardening process.

Second implemented strategy (TMR2) consists of detect and correct faults in the program data by computing the values twice and recomputing a third time if a discrepancy between the first two values occurs.

Fig. 3 shows an example of the hardening of a simple program (KCPSM3 syntax) using TMR1 and TMR2 applied to arithmetic instructions.

For the remaining of the paper, we will call arithTMR1 to the technique used when the first hardening strategy is applied to arithmetic instructions. When applied to logic instructions, it will be named logicTMR1. So, if it is applied to
both arithmetic and logic instructions, it will be named $arithmetic_{TMR1} + logic_{TMR1}$. This naming scheme also applies when using the second hardening strategy ($arithmetic_{TMR2}, \ldots$).

4 Experiments and Results

Verification of the developed environment has been done from three different aspects: correctness of the compiler back-end for PicoBlaze; validation of correct functionality of the hardening strategies; evaluation of the implemented hardening techniques (overheads and FC).

The functionality of the compiler back-end for PicoBlaze has been verified using regression tests. This is a technique of software testing which search uncover errors by running a set of tests when new functionalities are added. It allows to identify when a software functionality working previously, stops working due to the new changes. This strategy has been adopted as follows. Firstly, an incremental set of test programs with known compilation results has been written. Secondly, for each major change, the compilation process of all the test programs is performed in our compiler to verify that the obtained compilation results are equal to the expected results. Nevertheless, each test program can be well formed according to the language rules or may contain identified errors (lexical, syntactical and semantical errors).

Presently, the set of regression test has 477 programs. Some of them are simple programs to verify implemented functionalities of the compiler (lexical, syntactical and semantic rules of the language), others are used to check special features, and also, there are some real PicoBlaze programs found over the Internet. Regarding the results of the regression test, it has been obtained that 100% of the compilation processes for every test programs produce the expected results. Without this success we do not publish a new version of the compiler.

The hardening validation has been made through the option implemented for this purpose in the ISS (check-hardening option, mentioned before). That option is used to verify the equivalent functionality between the original and hardened programs. The 100% of the original and hardened checked programs obtained the same final expected results.

In order to evaluate the fault tolerance techniques implemented using the environment, a benchmark suite has been developed. It is made up of the follow-
ing programs: bubble sort (bubble), scalar division (div), Fibonacci (fib), greatest common divisor (gcd), matrix addition (madd), matrix multiplication (mmult), scalar multiplication (mult) and exponentiation (pow).

Fig. 4 shows the results offered by the ISS in terms of the code and time overheads. These results are normalized to a baseline built with the original version programs. Using these results the user can evaluate the impact of every technique and make decisions about the most appropriate hardening strategy. As can be seen the tools allow the study of different combination of SIHFT algorithms in order to meet the system constraints.

The following experimental setup has been configured to preliminary assess the FC of the implemented techniques. For each benchmark program (original and hardened versions), there have been performed 10,000 executions of the program in the ISS. According to the SEU fault model, there was only one SEU simulated during each program’s run. Fault was simulated randomly selecting one bit from the internal microprocessor registers or flags. Fig. 5 presents the obtained FC results for the example fault tolerance techniques, comparing original and hardened versions for the benchmark suite.

Despite arithTMR1+logicTMR1 and arithTMR2+logicTMR2 cause quite high overheads, they do not improve FC in a very significantly way. Overheads and FC depends on the instruction types that the program is most made up of, for example: when protection is applied to arithmetic instructions in mmult, the FC presents an important increase, while if protection is applied to logic
instructions the FC increase is far less, because \textit{mmult} is composed of a high percentage of arithmetic instructions.

5 Conclusions and Future Work

In this paper a hardening development environment for embedded systems was presented. A revision of the most important software based fault tolerance techniques and their instrumentation has been done to identify the main functionalities to offer. A hardening core has been designed and implemented. It is based on a generic architecture allowing to handle multiple targets.

As an initial case study, a KCPSM3 compiler back-end for \textit{PicoBlaze} has been developed. Furthermore, using the environment, several software fault tolerance techniques have been implemented and evaluated demonstrating the flexibility of the environment, which encourages reuse of previously implemented routines like: memory management procedures, control generic flow analyses, and even other hardening synthetic tasks.

The presented environment provides a set of tools which ensure the development and implementation of software based fault tolerance techniques in a flexible and architecture independent way. Likewise, it offers an automatic low cost solution to incorporate fault tolerance features in embedded systems. Moreover, using the results offered by the tools, designers can evaluate the possibility to translate protection of some components to hardware, resulting in a hybrid (HW/SW) hardening strategy.

As future work, the hardening development environment will be extended to support 32-bit microprocessors, such as \textit{MicroBlaze} from Xilinx and the family of \textit{LEON} soft-micros. Finally, looking forward to obtain more realistic FC results, we will use the \textit{SEU} emulation tool presented in [14]. The incorporation of such an emulation facility to the presented tool-chain will allow the co-design of hybrid HW/SW fault tolerant embedded systems.
Acknowledgment

This work makes part of RENASER project, Radiation Effects on Semiconductors for Aerospace Systems - Emulation Research (ESP2007-65914-C03-03), from the Space program of the Innovation and Science Ministry in Spain. The work presented here has been carried out thanks to the support of the research projects "Aceleración de algoritmos industriales y de seguridad en entornos críticos mediante hardware: Aplicación al sector calzado" (GV/2009/098) (Generalitat Valenciana) and "Aceleración hardware de algoritmos industriales para el sector calzado" (GRE08-P11) (University of Alicante).

References