electronics

MDPI

Article

A Compact Model to Evaluate the Effects of High Level C++ Code Hardening in Radiation Environments

Leonardo Maria Reyneri [1], Alejandro Serrano-Cases [2], Yolanda Morilla [3], Sergio Cuenca-Asensi [2] and Antonio Martínez-Álvarez [2,*]

1 Department of Electronics, Politecnico di Torino, Corso Duca d. Abruzzi 24, 10129 Turin, Italy; leonardo.reyneri@polito.it
2 Department of Computer Technology, Ctra. San Vicente del Raspeig s/n, 03690 San Vicente del Raspeig, Spain; aserrano@dtic.ua.es (A.S.-C.); sergio@dtic.ua.es (S.C.-A.)
3 Centro Nacional de Aceleradores (Universidad de Sevilla, CSIC, JA). Avda. Tomás Alba Edison 7, 41092 Sevilla, Spain; ymorilla@us.es
* Correspondence: amartinez@dtic.ua.es; Tel.: +34 965-903-400

Received: 30 April 2019; Accepted: 6 June 2019; Published: 10 June 2019

Abstract: A high-level C++ hardening library is designed for the protection of critical software against the harmful effects of radiation environments that can damage systems. A mathematical and empirical model to predict system behavior in the presence of radiation induced faults is also presented. This model generates a quick evaluation and adjustment of several reliability vs. performance trade-offs, to optimize radiation hardening based on the proposed C++ hardening library. Several simulations and irradiation campaigns with protons and neutrons are used to build the model and to tune it. Finally, the effects of our hardening approach are compared with other hardened and non-hardened approaches.

Keywords: fault tolerance; single event upset; proton irradiation effects; neutron irradiation effects; soft errors

1. Introduction

Progressive technological down-scaling is reducing the natural resilience of circuits, implying greater susceptibility to radiation faults [1]. In the past, fault-tolerant microprocessors were required for systems working in harsh environments, such as satellites, aircraft, autonomous vehicles, or any kind of autonomous decision-making systems, but today they are increasingly in demand, even at ground level [2] where radiation induced soft errors can frequently occur. Soft-error radiation faults are produced by the effect of incident particles on circuits where, as a consequence, the digital state of a node can be modified (bit-flipping). The developers of critical systems are constantly searching for ways to improve and/or to maximize the reliability of critical applications, due to the presence of soft errors, that can lead to catastrophic failure situations.

Many approaches are shown in the literature to minimize the effect of soft errors. Conventional approaches improve reliability by introducing redundancy at different hardware, software or hardware-software structures, in order to mask the wrong results by majority voting [3] or other redundancy-exploiting methods. For instance, it is common for hardware approaches to apply triple modular redundancy (TMR), to achieve reliability by replicating some physical components (rad-hard processors) [4]. Software-implemented hardware fault tolerance (SIHFT) techniques also introduce redundancy at instruction level by replicating several blocks of code [5] or several critical instructions [6]. Hardware–software hardening techniques, which reduce some weakness of the

hardware or software-only techniques, are also possible [7–9]. Other recent approaches represent attempts to gain reliability improvements by introducing no modifications in either the application (code instrumentation) or in the system (specific components). These techniques seek to achieve improvements during the transformation from high level code (source code) to machine code (executable) by altering the code compilation method [10]. Each approach has its own advantages and disadvantages; for example, the disadvantage of producing unwanted overheads in processing time and storage needs can be achieved by applying software hardening techniques.

When comparing the different approaches from the user perspective, there are two that require either high user intervention levels (most TMR-based approaches), or very little intervention (such as the approach proposed in this paper), or no user intervention and the delegation of hardening to some form of Artificial Intelligence (such as MOOGA [10]). The first approaches require lot of human effort, for instance, to change the focus of hardening. The third set requires a lot of CPU time to compare a large number of software versions, while the proposed approach can quickly explore several alternatives simply by changing the type and definition of each variable of interest, in a very fast operation.

In this article we focus on the SIHFT techniques, because they can be implemented in commercial off-the-shelf (COTS) microprocessors, thereby avoiding any internal modification to the microprocessor. More precisely, we are interested in high-level instrumentation techniques capable of deriving the inherent trade-offs, while maintaining flexibility and usability.

In view of the above, the key issues considered during the development of the new SIHFT approach presented in this article are:

1. The approach should be applicable to protect the largest possible amount of software, particularly the intellectual properties (IPs) commonly available on the Internet.
2. Post-compilation interventions must be as limited as possible (possibly none), in order to make software update and optimization fast and reliable.
3. The approach should apply to any COTS processor. It should not rely on any intrinsic radiation hardness of the processor except, obviously, the capability to withstand the total ionizing dose (TID).

The chosen language, C/C++, is compatible with commonly used software development techniques, leaving aside the domain of modern iconic programming. The idea which addressed and solved all the above issues is based on developing a set of C++ classes aimed at protecting program variables and processor registers, mostly by means of TMR.

In the following sections, a new high-level SIHFT technique will be presented, together with a reliability estimation model, to evaluate the impact of the system configuration parameters on program execution and radiation sensitivity. The model was developed from the results of two accelerated radiation campaigns conducted at the National Centre for Accelerators (CNA)—Spain, and Los Alamos Neutron Science Center (LANSCE)– USA.

2. Automatic Hardening Approach Based on C++ Classes

We propose a method that is intended for the protection of software code on COTS processors. In particular, it addresses the following elements of a COTS microcontroller system:

1. Numeric data stored in temporary and long-term storage locations. As it is a C/C++ level approach, there is no explicit distinction between registers and memory, although it provides overall data protection to data stored in the C/C++ variables, regardless of how and where these are allocated by the compiler;
2. Microcontroller configuration registers such as those for interrupts, ports, universal asynchronous receivers-transmitters (UARTs), analog-to-digital converters (ADCs), etc.
3. Program memory, mostly for situations where the program is stored in volatile or radiation-sensitive memory;

4. Protection against single-event functional interrupts (SEFIs).

Under certain circumstances induced by radiation, the microcontroller program may occasionally restart, which is considered acceptable, provided that the results produced at the end of execution are correct. In particular, some aspects of protection rely on inducing an automatic program reset when a SEFI is detected.

It is worth noting that most benefits of the proposed approach may also apply to temporary faults induced by other causes, such as electromagnetic interferences, allowing technological transfer to other ground-based activities, such as functional safety in automotive electric/electronic systems, and detect and correct errors in high performance computing (HPC), among others.

2.1. Using C++ Classes for Data Protection

The proposed methodology is based on a C++ template class called `TD<DataType>` (standing for "triple data") which can be applied to any numeric `DataType` (e.g., `TD<char>`, `TD<int>`, `TD<float>`). A `TD<DataType>` class transparently protects, by means of TMR, a numeric variable of any given `DataType`. This class has been designed to allow total reuse of existing code, only changing the definition of all the variables to be protected, while maintaining the rest of the code unchanged.

The internal architecture of a `TD<DataType>` (see Table 1) class contains three private variables (i.e., concealed from the user) of the type `DataType`, storing as many replicas of the same data (`d1`, `d2`, `d3`).

Table 1. Internal architecture of `TD<DataType>`.

```
template <typename DataType> class TD {
private: volatile DataType d1, d2, d3; }
```

A seamless use of the proposed class requires: (i) the appropriate re-definition of all possible numeric, comparison and logical operators; (ii) writing the code to implement each of them in a redundant way.

For instance, for the assignment operator (=), the kernel of the code and its usage are showed in Table 2.

Table 2. Definition and functioning for the = operator.

definition	operator=(TD<DataType>& val) { d1 = val.d1; d2 = val.d2; d3 = val.d3; }
usage	TD<int> a, b; a = b;
equivalent to	int a1, a2, a3, b1, b2, b3; a1 = b1; a2 = b2; a3 = b3;

Which implies that, despite the apparently identical usage of the assignment (`a = b`) to standard C variables, the usage of operator = of class `TD<DataType>` implies (in a transparent way) that the value of each replica of `b` is assigned to the corresponding replica of `a`. A similar approach applies to all algebraic operators (e.g., `+`, `-`, `*`, `/`), comparison operators, logical operators, etc.

In our library, the casting operators to/from TMR data and plain data have been overloaded for transparent conversion between data types. Conversion from TMR to plain data implicitly implements majority voting, while conversion from plain to TMR implicitly implements triplication.

The following simple example compares a simple piece of C code which sums up two variables and stores the result in a third variable (See Table 3). The same code is written in fully unprotected and partially protected ways, respectively, together with a possible manual protection.

Table 3. Comparison of how to sum up two integer variables using different protection levels: original code, protection using our technique and manual TMR protection.

Unprotected	Protected		Manual TMR

int a, b;
int c;
c = a + b;
	TD<int> a, b; \\protected
	TD<int> c; \\protected
	c = a + b;
		TD<int> a, b; \\protected
		int c; \\unprotected
		c = a + b;
			int a1, a2, a3, b1, b2, b3;
			int c1, c2, c3;
			c1 = a1 + b1;
			c2 = a2 + b2;
			c3 = a3 + b3;

As a consequence, by writing, `c = a + b;` the compiler automatically generates the code that will sum up and store each corresponding replica of the `DataType` in a completely transparent way and will preserve (by construction) the correspondence of each replica. A process that is quite unlike the TMR manual approach, which would be quite prone to coding errors.

The `TD<DataType>` class is designed to support any operator and constructor (e.g., vectors and structures) commonly used inside C programs. The potential risks of pointers are normally to be avoided and they can be applied with greater safe by using, for instance, a TMR-protected `TD<int*>`, as redundancy significantly reduces the risk of pointer corruption.

In the case of single-event upsets (SEUs) (or any other transient fault) affecting one of the three replicas, the original value can be recovered by majority voting, again in a transparent way. For instance, the simple piece of code of Table 4

Table 4. Example of triplicating and voting automation for two hardened and non-hardened variables.

```
TD<int> a; int b;
b = a; \\ majority vote a's replicas into b
a = b; \\ triplicate b into a's replicas
a = (int) a; \\ compact form: vote+TMR
```

First converts and copies a redundant variable, `a`, into a non-redundant variable, `b`, by enforcing majority voting, and it then stores the three replicas of the voted value, `b`, back into variable `a`. In other words, it re-synchronizes the replicas by majority voting. The last line is a compact form which does exactly the same thing.

Any existing program can therefore be hardened, by a mere redefinition of the variables used, while the active part of the code requires no single modification. This idea per se is not novel, as TMR is widely used to achieve data protection, but the way it has been implemented and optimized with respect to radiation tolerance is new and easy to use.

2.2. Protecting Other Elements of a Program

A complex program not only relies on data memory, which can be protected by means of the `TD<...>` class. Other elements have also been considered.

Configuration registers cannot be triplicated in the same way as normal memory location, as they are unique in hardware and their TMR would require redesigning the manufacturing masks. Protecting the configuration registers is therefore supported by another type of class, called `TDreg<...>`, which automatically stores two other copies of the register in data memory and periodically re-synchronizes the hardware register by majority voting with the other two stored replicas.

Periodical refresh can be implemented in different ways, depending on system and mission requirements. For instance: (i) a timer-driven interrupt routine which refreshes all variables, set at, for instance, every minute; (ii) at the beginning or at the end of each program loop (if any); (iii) by voting whenever a critical variable is used.

Since configuration registers of commercial processors mix read-only and write-only bits, the definition of `TDreg<...>` supports this feature and synchronization is automatically limited to writable bits.

Program memory should nominally be read-only, as it only contains machine code and numeric constants. We explicitly omit consideration of self-modifying codes, as those are considered too dangerous for critical applications.

As a consequence of a nominally constant program code, its protection is limited to computing a "signature" of the code area, on a periodic basis, and verifying it against a golden sample. As soon as a SEU affects the program area, its signature will no longer match and the program will automatically be reset, downloading the program again from a more rad-tolerant ROM. This is implemented by means of the `TDcode` class.

Internal control registers (namely internal state machines, program counter and stack pointer) are more difficult to protect and are the most common cause of program hang, therefore causing SEFIs. Our approach offers periodical verification of stack-pointer consistency, but the other control registers (e.g., program counter and status register) can be protected only to a very limited extent.

The only means available to compensate SEFIs is the use of a watchdog timer (or equivalent methods) already commonly used in these situations. Yet Section 4.1 shows that protection of program counter and stack pointer will not usually improve hardness significantly.

Interrupt handlers are normal routines that can be protected with the same techniques described above. In addition, interrupts also rely on interrupt enable bits, which are part of configuration registers; these can be protected by means of the `TDreg<...>` class described above.

2.3. Performance Issues

The use of TMR, on the one hand, significantly increases the hardness of a program to single-event effects (SEEs) but, on the other hand, it also impacts on aspects of performance, particularly speed and memory size. In theory, execution time should increase by a factor of three at most (the same as redundancy), although the increased flexibility and safety made available by the use of the C++ classes causes an additional overhead by another factor of two, on average, mostly due to the periodic necessity of majority voting. This overhead has been strongly optimized by means of the many features of state-of-the-art optimizing compilers, although it cannot be completely removed for several reasons.

As a consequence, program execution, for a program with variables that are totally triplicated will take six times more time to execute, on average. An appropriate selection of which variables to protect and which ones need no protection significantly reduces the impact on program speed. Section 4.1 gives some hints on both how to select storage blocks and which specific variables to protect and which ones need not be protected, allowing a quick performance trade-off customized for specific mission requirements.

3. A Compact Reliability Estimation Model

During the process of hardening a piece of code (or even a complete HW/SW design), it is of the utmost importance to analyze a number of different configurations and to evaluate the impact of configuration parameters (e.g., data triplication, register refresh, error checking, register optimization, inlining, interrupts, etc.) on program execution and radiation sensitivity.

We developed a mathematical and empirical model, for quick evaluation of several reliability vs. performance trade-offs and for the optimization of radiation hardening without excessively compromising performance. The model offers valuable advance information on system behaviour in the presence of radiation induced faults. Firstly, it predicts the occurrence frequency of faults that affect program execution for any combination of processor, high level language, compilation parameters, hardening techniques and selection of protected variables. Secondly, it estimates the impact of each storage area, variable or data structure on the overall reliability, to concentrate hardening efforts on the block that has the highest impact on radiation sensitivity.

It is worth highlighting other works that either compare different methods with real radiation data (making the approach quite expensive and time consuming, and therefore ruling out the possibility of comparing large numbers of alternatives) and with simulated campaigns (cheaper and faster approach, but of lower reliability). The proposed approach is, instead, based on a compact parametric mathematical model, the parameters of which are first evaluated once and for all on real radiation measurements, then the model is applied in an iterative way, thereby permitting a wider search in the space of hardening alternatives.

3.1. Model Preliminaries

The model is based on cycle-accurate simulations using the OVPsim simulator [11] while randomly corrupting: registers (**R**), data memories (**D**), and program memory (**P**). Each storage block may have a different hardware implementation, so it may therefore have its own cross section per byte $\alpha_{\mathtt{R}}$ (respectively, $\alpha_{\mathtt{D}}$ and $\alpha_{\mathtt{P}}$); in addition, data storage may be distributed between a number of memories, each one having a different cross section $\alpha_{\mathtt{D1}}$, $\alpha_{\mathtt{D2}}$, $\alpha_{\mathtt{D...}}$ (e.g., FLASH, ferroelectric, static and dynamic RAMS).

We assume that the cross section is different for each type of storage, and we relate each one to the basic cross section of main processor RAM (D1), that is $\alpha_{\mathtt{D1}} \equiv \alpha$. We therefore state that: $\alpha_{\mathtt{R}} = \mathtt{K_R} \cdot \alpha$, $\alpha_{\mathtt{D2}} = \mathtt{K_{D2}} \cdot \alpha$, $\alpha_{\mathtt{P}} = \mathtt{K_P} \cdot \alpha$, ..., where $\mathtt{K_X}$ are appropriate coefficients and, by definition, $\mathtt{K_{D1}} = 1$. In particular, $\mathtt{K_P}$, is the coefficient of either ROM or RAM, depending on where the program is executed.

For each given processor, algorithm, language, compilation flags, hardening effort, etc., OVPsim simulations are set up to induce one random SEU per run of the compiled program, in either of the aforementioned storage blocks (R, D, P). The fault injection can be performed in each memory block at different abstraction levels. It means, for example, that we can induce an error in an SRAM or a DRAM device on any possible address from their available addressing space or only induce faults on single C/C++ variables of interest (vectors, matrices, ...).

Injected faults are classified according to their effect on program behavior, in a similar way to the first proposals of Mukherjee et al. [12]. Faults which neither hang program execution nor affect expected program output are called unnecessary for architecturally correct execution (unACE). On the other hand, faults which visibly affect program execution are called architecturally correct execution (ACE), which comprise the two categories specifically considered in this paper: (i) faults which allow the program to terminate normally, but produce corrupted results, called silent data corruption (SDC); and, (ii) faults which cause abnormal program termination or infinite execution loops, called HANG.

Each simulation set was configured to inject 1000 faults per register in the register file and 18,000 faults in the memory section allocated by the benchmark. This arrangement implies a total of at least 72,000 faults injected per program version, achieving a statistical error of $\pm 1\%$ at a 99% confidence level, according to the statistical model proposed by Leveugle et al. [13].

3.2. Model Description

Simulations provide, as an output, the number of SDCs (respectively $\mathtt{SD_R}$, $\mathtt{SD_{D1}}$, $\mathtt{SD_{D2}}$, $\mathtt{SD_{D...}}$, $\mathtt{SD_P}$) and HANGs (respectively $\mathtt{HG_R}$, $\mathtt{HG_{D1}}$, $\mathtt{HG_{D2}}$, $\mathtt{HG_{D...}}$, $\mathtt{HG_P}$) out of $\mathtt{R_R}$ program executions (respectively, $\mathtt{R_{D1}}$, $\mathtt{R_{D2}}$, $\mathtt{R_{D...}}$, $\mathtt{R_P}$). The size of each storage area being $\mathtt{S_R}$ words (respectively, $\mathtt{S_{D1}}$, $\mathtt{S_{D2}}$, $\mathtt{S_{D...}}$, $\mathtt{S_P}$).

For every configuration we define, for each storage area (where Z is either R, D1, D2, P, ...) the equivalent block size for SDCs of that area, expressed in bytes, as:

$$\beta_{\mathtt{Z}} \triangleq \mathtt{K_Z} \cdot \frac{\mathtt{SD_Z}}{\mathtt{R_Z}} \cdot \mathtt{S_Z} \tag{1}$$

We also define the equivalent block size for HANGs of that area, expressed in bytes, as:

$$\gamma_{\mathtt{Z}} \triangleq \mathtt{K_Z} \cdot \frac{\mathtt{HG_Z}}{\mathtt{R_Z}} \cdot \mathtt{S_Z}. \tag{2}$$

The two formulas provide two factors (β_Z, γ_Z) which are proportional to: the increased sensitivity of the specific memory area (K_Z) to radiation; the failure probability as estimated by simulations ($\frac{SD_Z}{R_Z}$, $\frac{HG_Z}{R_Z}$) and the size S_Z of the memory block, which is proportional to the probability of a particle hitting that block.

From these factors, we can find the total equivalent size for SDCs and for HANGs of the whole program, respectively:

$$\beta_{TOT} = \beta_R + \beta_{D1} + \beta_{D2} + \beta_{D...} + \beta_P + \beta_X \tag{3}$$

$$\gamma_{TOT} = \gamma_R + \gamma_{D1} + \gamma_{D2} + \gamma_{D...} + \gamma_P + \gamma_X. \tag{4}$$

The two additional parameters, β_X and γ_X, are the equivalent block size for SDCs and HANGs of the internal control unit and the state machines, which cannot be simulated by the OVPsim simulator and are therefore empirically estimated.

3.3. System Reliability

Given β_{TOT} and γ_{TOT}, our model predicts the probability of SDCs and HANGs per execution:

$$P(SD) = \Phi \cdot \alpha \cdot (T_E \cdot \beta_{TOT}) \tag{5}$$

$$P(HG) = \Phi \cdot \alpha \cdot (T_E \cdot \gamma_{TOT}), \tag{6}$$

where Φ is the radiation flux (particles/s/cm^2), while α is the cross section per byte (cm^2/byte) of storage, and T_E is nominal program execution time (s).

The two expressions between brackets are called the size-time figures for SDCs ($\chi_{SDC} = T_E \cdot \beta_{TOT}$) and HANGs ($\chi_{HANG} = T_E \cdot \gamma_{TOT}$), respectively, of the given configuration.

An innovative aspect of the proposed approach is that the size-time figures, χ_{SDC} and χ_{HANG}, mean that the impact of each data storage, each data structure, and even each individual variable on overall radiation performance can be easily assessed and the hardening efforts may be therefore be concentrated where the effect is highest and to reduce the impact of hardening to a minimum.

Depending on the application, we can estimate, in the first place, the mean work to failure (MWTF) (i.e., the average number of program executions between two failures), in the following way:

$$MWTF = \begin{cases} \frac{1}{P(SD)} = \frac{1}{\Phi \cdot \alpha \cdot \chi_{SDC}} & \text{for SDCs} \\ \frac{1}{P(HG)} = \frac{1}{\Phi \cdot \alpha \cdot \chi_{HANG}} & \text{for HANGs} \\ \frac{1}{P(ERR)} = \frac{1}{\Phi \cdot \alpha \cdot (\chi_{SDC} + \chi_{HANG})} & \text{for any error} \end{cases}, \tag{7}$$

which depends on: (i) radiation flux Φ; (ii) processor's cross section α; and (iii) size-time figures of given program configuration (χ_{SDC} or χ_{HANG}).

In second place, for time-sampled systems, where the program starts every T_S (sample time), executes over a certain time, T_E, then stops until the next sample, we can compute the mean time to failure (MTTF), which is the average time between two failures:

$$MTTF = T_S \cdot MWTF, \tag{8}$$

which also depends on sample time T_S.

3.4. Model Validation under Radiation

The proposed model was evaluated against real radiation measurements. Table 5 shows the relevant model parameters for protons and neutrons measured during the two radiation campaigns described below. The model showed a good accuracy for the estimation of reliability. In fact, the figures

shown in the last two columns of Table 6 have an error of −30% + 50% with respect to radiation measurements (not shown in the table).

The device under test (DUT) selected for the irradiation experiments was the ZYBO board. The DUT is equipped with a 28nm CMOS Xilinx ZYNQ XC7Z010 system on chip (SoC). This SoC is divided into two parts, an FPGA area (programmable logic—PL) and a 32-bit ARM cortex A9 microprocessor (processing system—PS). The processor has a 13-stage instruction pipeline that includes a branch prediction block and support for two levels of cache. In addition, the microprocessor has a little built-in memory called on chip memory (OCM), where the bootloader or the program under test can be loaded.

The DUT was controlled by an external computer, the RaspberryPi 3 Model B, the main task of which is to receive and log all the messages sent by the DUT. The DUT was configured to send a state message every five seconds in the absence of errors, otherwise the message is notified instantly and the external computer resets and reprograms the DUT.

Tested programs present a rich variety of flow structures and data. For example, BubbleSort (BB) is a well-known sorting algorithm that achieves its objective by making use of several nested loops. The second algorithm considered here is the Dijkstra algorithm (DK), also known as the shortest path problem, which uses an adjacency matrix that is stored in the memory where the weights of all paths are located.

3.4.1. Proton Irradiation Campaign

The test campaign was carried out in mid-2018 at the National Centre for Accelerators (CNA), in Spain [14]. Irradiation tests were performed using the external beam line, installed in the cyclotron laboratory. Although the proton energy delivered by this cyclotron was fixed to 18 MeV, the beam was extracted to the air up to reach the DUT (device under radiation) position with 15.2 MeV energy. The flux fluctuated within ±5% during each run. Beam uniformity under these experimental conditions was better than 90% in the area of interest.

3.4.2. Neutron Irradiation Campaign

The neutron SEE campaigns were performed at the Los Alamos Neutron Science Center (LANSCE) in September 2018 [15,16]. The neutron beam was provided by a tungsten spallation source at approximately 30 degrees to the left of the main beam. During the campaign, the DUT remained at 23 m from the neutron source, and the beam was collimated, so that a spot was obtained in the order of 30 mm of diameter. This size covers the active area with uniformity better than 90%. A constant neutron flux of $1.7 \cdot 10^5$ n/(s · cm^2), above 10MeV, was obtained.

4. Reliability Issues

In the last step of this activity, our model has been used to identify the most critical storage areas, variables and data structures, that is, those which most affect reliability, in order to concentrate hardening efforts on the most relevant areas. In addition, the performance of the proposed C++ classes against other optimization techniques proposed by the same authors in [10] was compared.

The proposed C++ classes have been used to protect a variety of programs on an ARM Cortex-A9 processor and our model has identified the most critical storage areas which deserve more hardening effort. Some results are shown in Table 6, namely for a BubbleSort sorting algorithm and a Dijkstra shortest path finder algorithm, with both on-chip memory (OCM), an external rad-hard memory (EXT), as well as neutron and proton irradiation. All these results were also verified during the two radiation campaigns briefly described in Section 3.4

Table 5. ARM Cortex-A9 parameters estimated during two radiation campaigns.

Particles	α (cm^2/byte)	K_R	K_P	β_X (byte)	γ_X (byte)
proton	$1.39 \times 10^{-14} \pm 10\%$	35 ± 5	1.6 ± 0.2	40 ± 10	25 ± 25
neutron	$4.85 \times 10^{-14} \pm 10\%$				

4.1. Performance Considerations

We draw a few considerations here, which can be found by analyzing the results shown in Table 6, where a few C++ hardening configurations are compared with other configurations with hardening on specific aims [10]: mean work to failure (MWTF) maximization, fault coverage maximization (Max-ACE), trade-off optimization among execution time, memory size and fault coverage (Pareto), baseline compilation (O0) and code optimization (O3). All C++ versions were compiled using the -O3 optimization flag. We can observe that:

Table 6. Execution time, T_E,(for 666MHz clock) plus equivalent block sizes for SDCs and HANGs and total time-size figures for a BubbleSort and a Dijkstra program, for different compilation flags, use of C++ classes vs. other hardening techniques, for four storage blocks (registers, data memory, stack, and code memory, taken as examples, for an ARM Cortex-A9 processor, using either on-chip memory (OCM) and external rad-hard memory (EXT). Highlighted values are those referenced in the text for the sake of clarity.

	Configuration				T_E	REG		Data+BSS		STACK		CODE		TIME-SIZE	
	Hardening Strategy	**ID**	**MEM**	**Part**	**(µs)**	β_R **(B)**	γ_R **(B)**	β_{D1} **(B)**	γ_{D1} **(B)**	β_{D2} **(B)**	γ_{D2} **(B)**	β_P **(B)**	γ_P **(B)**	χ_{SDC} **(B·ms)**	χ_{HANG} **(B·ms)**
BubbleSort	Max-ACE	BB-C1	OCM	prot	67	778	12.9	401	13.8	0	5	420	516	110	37
	O0	BB-C2	OCM	prot	335	53	12.6	256	145	3	3	153	594	169	252
	MWTF	BB-C3	OCM	prot	58	81	16.2	395	24.8	0	4	535	553	61	**35**
	Pareto	BB-C5	OCM	prot	60	64	15.6	377	14.3	0	6	99	461	35	**29.8**
	O3	BB-C10	EXT	prot	980	90	15.6	19.9	0.7	0.0	0	4.5	30	151	**46**
	C++ (O3)	BB-C14	OCM	prot	522	16.8	17.6	4.1	8	0	11	0	807	**32**	440
	C++ (O3)	BB-C11	EXT	prot	12821	16.8	17.6	0.2	0.4	0	1	0	40	731	755
	C++ (O3)	BB-L4	OCM	neut	517	6.6	9	2.7	9.6	0	18	0	568	**25.5**	312
Dijk.	O0	DK-L1	OCM	neut	2377	31	4.8	5297	1986	11.8	18.7	205	2052	13,279	9656
	C++ (O3)	DK-L3	OCM	neut	9676	226	18.9	13.1	707	16.8	22.9	742	2890	10,042	35,216

- in the BubbleSort algorithm the influence of stack (β_{D2} and γ_{D2}) is close to zero, therefore negligible with respect to the influence of other storage blocks (β_R, γ_R, β_P and γ_P); in this situation, it is useless to protect the stack. In the Dijkstra algorithm, the influence of stack on SDCs (β_{D2}) is comparable to that of data storage (β_{D1}), at least for one configuration (DK-L3); in this situation, it might also be worth protecting the stack;
- the use of C++ classes (BB-C14, BB-L4, DK-L3) increases execution time by a factor of between 2 and 10 times, depending on configuration (without considering BB-C11 which runs on an external, slower, rad-hard memory), but it reduces the influence of data memory on SDCs (β_{D1}) by a factor of 100 and almost nullifies the influence of data memory on SDCs (β_P); the effects of the C++ classes on HANGs are negligible;
- for configurations not protected by means of the C++ classes, the effect of registers on SDCs and HANGs is negligible despite the register's very high cross section (see K_R in Table 5); when protecting the program by means of the C++ classes, the effect of registers (mostly for SDCs)

is almost the only relevant one, therefore increasing protection requires an additional effort to protect the registers, which are not protectable by means of the C++ classes;

- using an external, slower, rad-hard memory (configuration BB-C11, based on the proposed C++ classes, without cache) offers the lowest equivalent block sizes for all data storage (except obviously registers), despite it increasing execution time, T_E, by a factor of 20 to 25.
- by looking at the total size-time figures (two last columns), which are the most relevant overall parameter directly affecting MWTF and MTTF, the reduction of equivalent program size often counteracts an increase in execution time. The best performance for SDCs was achieved using the proposed C++ classes, while the best performance for HANGs was achieved with configurations BB-C3 and BB-C5.

4.2. Optimization Process

This section shows how an appropriate use of the compact model can rapidly optimize the usage of the C++ classes. We took as an example an optimized BubbleSort algorithm (different from the one used for Table 6) running at 666MHz on a Cortex A9 processor and irradiated by protons. We simulated the few configurations shown in Table 7, both for SDCs and for HANGs.

Each row shows different configurations: first and second configurations are plain C code with no optimization (-O0) and highest optimization (-O3), respectively. Each column shows the equivalent size of : registers (REG); whole data memory (β_{D}); only the first, the second, and the third C variables of the program ($\beta_{\mathrm{D,V1}}$, $\beta_{\mathrm{D,V2}}$, $\beta_{\mathrm{D,V3}}$, respectively); the other five variables were less relevant, taken together ($\beta_{\mathrm{D,V4}}$); program memory (PROG); the other three columns show the equivalent size, the execution time and the size-time figure of the whole program; the last two columns show the expected MWTF and MTTF for a given irradiation level (see caption of Table 7).

From the table, it is, for instance, clear that the variable V2 for SDCs has by far the highest relevance (namely, highest size, 261B/388B) among all the C variables. It would therefore be worth hardening only that variable by means of the C++ classes. The hardening of other variables would add significantly to the execution time while reducing total equivalent size by a negligible amount.

Consequently, one variable, V2, when hardened (by changing the data type to the proposed C++ class), yields the results shown in the third line of the table, which shows the lowest size-time figure χ_{SDC} from among all the configurations. We also evaluated the fourth configuration of the table, for comparative purposes, by applying the C++ classes to all the program variables.

It is clear that the configuration with only one hardened variable, V2, showed the best equivalent size (135 B) and size/figure performance (29 B·ms) from among all of them, despite the higher execution time (215 μs). The same configuration also shows the highest MWTF (about ten times higher than the -O0 and two times higher than the -O3 without the C++ classes; slightly lower for HANG) and MTTF (also about ten times higher than the -O0 and two times higher than the -O3 without C++ classes), proving the effectiveness of the proposed method. Table 8 shows the global MWTF and MTTF metrics (including both SDC and HANG). As can be seen, the configurations hardened by C++ classes provide the best overall reliability.

Table 7. Equivalent sizes ($\beta_{\mathtt{TOT}}$ and $\gamma_{\mathtt{TOT}}$), size-time figures ($\chi_{\mathtt{SDC}}$ and $\chi_{\mathtt{HANG}}$) and reliability metrics (MWTF and MTTF) of a selected program (optimized BubbleSort) in a few different configuration. The individual impact of Registers (R), total data memory (D), individual memory variables (V1 through V3), other variables (V4), code area (P) and total (TOT), for an ARM Cortex A9 processor, with on-chip memory (OCM) tuning at 666MHz clock frequency. The last two columns refer to the estimated proton irradiation results with radiation flux of 5.45×10^5 particles/cm^2/s and sample time $T_S = 20\,\mathtt{ms}$. Highlighted values are those referenced in the text for the sake of clarity.

	Configuration	**REG**	**MEM**					**PROG**	**TOTAL**			**MWTF**	**MTTF**
		$\beta_{\mathtt{R}}$ **(B)**	$\beta_{\mathtt{D}}$ **(B)**	$\beta_{\mathtt{D,V1}}$ **(B)**	$\beta_{\mathtt{D,V2}}$ **(B)**	$\beta_{\mathtt{D,V3}}$ **(B)**	$\beta_{\mathtt{D,Vx}}$ **(B)**	$\beta_{\mathtt{P}}$ **(B)**	$\beta_{\mathtt{TOT}}$ **(B)**	T_E **(µs)**	$\chi_{\mathtt{SDC}}$ **(B·ms)**	**runs** $(\times 10^3)$	**hrs**
SDC	-O0	56.0	261	1.67	**261**	1.92	2.45	210	527	562	296	446	2.5
	-O3	102	401	0.04	**388**	0.00	0.00	105	608	**87.1**	53.0	2493	13.8
	C++ -O3 (V2)	81.2	0.00	0.01	0.00	0.00	0.00	53.5	**135**	215	**29.0**	**4552**	**25.3**
	C++ -O3 (all)	82.6	0.00	0.04	0.00	0.00	0.00	68.7	151	208	31.5	4184	23.2
		$\gamma_{\mathtt{R}}$ (B)	$\gamma_{\mathtt{D}}$ (B)	$\gamma_{\mathtt{D,V1}}$ (B)	$\gamma_{\mathtt{D,V2}}$ (B)	$\gamma_{\mathtt{D,V3}}$ (B)	$\gamma_{\mathtt{D,Vx}}$ (B)	$\gamma_{\mathtt{P}}$ (B)	$\gamma_{\mathtt{TOT}}$ (B)	T_E (µs)	$\chi_{\mathtt{HANG}}$ (B·ms)	runs $(\times 10^3)$	hrs
HANG	-O0	455	152	0.3	**133**	0.1	1.2	714	1321	562	742	178	1.0
	-O3	679	17.8	0.0	0.0	0.0	0.0	434	1130	**87.1**	98.4	1341	7.5
	C++ -O3 (V2)	386	24.1	0.0	0.0	0.0	0.0	281	691	100	**69.1**	**1910**	**10.6**
	C++ -O3 (all)	274	14.6	0.0	0.0	0.0	0.0	371	660	100	**66.0**	**2000**	**11.1**

Table 8. Total MWTF and MTTF for different configurations. Highlighted values are those referenced in the text for the sake of clarity.

Configuration	**-O0**	**-O3**	**C++ -O3 (V2)**	**C++ -O3 (All)**
MWTF runs $(\times 10^3)$	127	872	**1345**	**1353**
MTTF hours	0.7	4.8	**7.5**	**7.5**

4.3. Further Improvements

It is clear from Table 7 that the proposed C++ classes significantly reduced the influence of data storage for SDCs and slightly reduced the influence of data storage for HANGs, although they significantly increased the execution time.

The reason is that all the variables were protected for the configurations shown in Table 6. Nevertheless, the proposed approach can be individually used to address the effect of each variable, by splitting data storage into smaller blocks (D1, D2, D…), namely one per variable or group of variables, and to evaluate the effect of each of them on execution time and equivalent program sizes. From this analysis, the best trade-off between what to protect and what not to protect can be assessed.

Another parameter that can be addressed is the rate of data recovery; each data verification in the C++ classes takes time and data recovery takes even longer. Frequent verifications and recovery increase execution times, while less frequent verifications can increase the risk of double faults. A trade-off may also be established in this case, by means of the proposed approach.

5. Conclusions

A new hardening approach has been proposed on the basis of a set of C++ classes, to ease the protection of existing and new software programs.

A simple though accurate reliability model has also been proposed, to support the optimization of the usage of C++ classes, and to compare the performance of those classes with the performance of other hardening methodologies.

A relevant feature of this model is that it provides two compact figures (namely the size-time figure $\chi_{\texttt{SDC}}$ and $\chi_{\texttt{HANG}}$) that directly relate to the reliability figures (MWTF and MTTF for SDCs and HANGs, respectively), by taking into account both the increased computation time-typical of SIHFT-and the improvement in robustness-typical of TMR.

The basic results showed that programs protected with the C++ classes were slower, but less subject to radiation-induced effects. Yet the two effects partially canceled out when considering the mean time or mean work between consecutive program HANGs, while the lower sensitivity to radiation was more relevant than the increase in execution time when considering the mean time or mean work between consecutive SDCs. It has been shown that a straightforward usage of C++ classes improved the reliability of a software system against corrupted results, but had less effect on program HANGs. A targeted application of the proposed C++ classes to specific variables significantly improved both effects.

In conclusion, the use of appropriate C++ classes shown in this paper has greatly facilitated the use of TMR. Also, the availability of an easy-to-use performance estimation model could be used for quick and effective radiation tolerance optimization of the COTS microcontroller systems.

Author Contributions: Conceptualization, L.M.R. and A.M.-Á.; Methodology, L.M.R., A.S.-C. and A.M.-Á.; Validation, L.M.R., A.S.-C., Y.M., S.C.-A. and A.M.-Á.; Writing—Original Draft Preparation, L.M.R., A.S.-C., S.C.-A. and A.M.-Á.

Funding: This work was funded by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund through the following projects: '*Evaluación temprana de los efectos de radiación mediante simulación y virtualización. Estrategias de mitigación en arquitecturas de microprocesadores avanzados*' and '*Centro de Ensayos Combinados de Irradiación*', (Refs: `ESP2015-68245-C4-3-P` and `ESP2015-68245-C4-4-P`, MINECO/FEDER, UE).

Acknowledgments: Thanks to the the National Centre for Accelerators (CNA)—Spain, and Los Alamos Neutron Science Center (LANSCE)– USA for all the support in the the irradiation campaigns.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Baumann, R.C. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Trans. Device Mater. Reliab.* **2005**, *5*, 305–315. [CrossRef]
2. Michalak, S.E.; Harris, K.W.; Hengartner, N.W.; Takala, B.E.; Wender, S.A. Predicting the number of fatal soft errors in Los Alamos national laboratory's ASC Q supercomputer. *IEEE Trans. Device Mater. Reliab.* **2005**, *5*, 329–335. [CrossRef]
3. Nicolaidis, M. Design for soft error mitigation. *IEEE Trans. Device Mater. Reliab.* **2005**, *5*, 405–418. [CrossRef]
4. Ruano, O.; Maestro, J.A.; Reviriego, P. A Methodology for Automatic Insertion of Selective TMR in Digital Circuits Affected by SEUs. *IEEE Trans. Nucl. Sci.* **2009**, *56*, 2091–2102. [CrossRef]
5. Quinn, H.; Baker, Z.; Fairbanks, T.; Tripp, J.L.; Duran, G. Software Resilience and the Effectiveness of Software Mitigation in Microcontrollers. *IEEE Trans. Nucl. Sci.* **2015**, *62*, 2532–2538. [CrossRef]
6. Martínez-Álvarez, A.; Cuenca-Asensi, S.; Restrepo-Calle, F.; Palomo, F.R.; Guzmán-Miranda, H.; Aguirre, M.A. Compiler-Directed Soft Error Mitigation for Embedded Systems. *IEEE Trans. Dependable Secur. Comput.* **2012**, *9*, 159–172. [CrossRef]
7. Martínez-Álvarez, A.; Restrepo-Calle, F.; Cuenca-Asensi, S.; Reyneri, L.M.; Lindoso, A.; Entrena, L. A Hardware-Software Approach for On-Line Soft Error Mitigation in Interrupt-Driven Applications. *IEEE Trans. Dependable Secur. Comput.* **2016**, *13*, 502–508. [CrossRef]
8. Reyneri, L.M.; Roascio, D.; Passerone, C.; Iannone, S.; de los Rios, J.C.; Capovilla, G.; Martínez-Álvarez, A.; Hurtado, J.A. Modularity and Reliability in Low Cost AOCSs. In *Advances in Spacecraft Systems and Orbit Determination*; Ghadawala, R., Ed.; IntechOpen: Rijeka, Croatia, 2012; Chapter 5.
9. Peña-Fernandez, M.; Lindoso, A.; Entrena, L.; Garcia-Valderas, M.; Philippe, S.; Morilla, Y.; Martin-Holgado, P. PTM-based hybrid error-detection architecture for ARM microprocessors. *Microelectron. Reliabil.* **2018**, *88–90*, 925–930. [CrossRef]

10. Serrano-Cases, A.; Morilla, Y.; Martín-Holgado, P.; Cuenca-Asensi, S.; Martínez-Álvarez, A. Non-intrusive automatic compiler-guided reliability improvement of embedded applications under proton irradiation. *IEEE Trans. Nucl. Sci.* **2019**. [CrossRef]
11. Open Virtual Platforms. OVPsim Simulator. Available online: www.ovpworld.org/technology_ovpsim (accessed on 1 April 2019).
12. Mukherjee, S.S.; Weaver, C.; Emer, J.; Reinhardt, S.K.; Austin, T. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, San Diego, CA, USA, 5 December 2003; pp. 29–40. [CrossRef]
13. Leveugle, R.; Calvez, A.; Maistri, P.; Vanhauwaert, P. Statistical fault injection: Quantified error and confidence. In Proceedings of the 2009 Design, Automation & Test in Europe Conference & Exhibition Nice, France, 20–24 April 2009; pp. 502–506. [CrossRef]
14. Centro Nacional de Aceleradores. Seville, Spain. Available online: http://www.cna.us.es (accessed on 1 April 2019).
15. Wender, S.A.; Lisowski, P.W. A white neutron source from 1 to 400 MeV. *Nucl. Inst. Methods Phys. Res. B* **1987**, *24–25*, 897–900. [CrossRef]
16. Lisowski, P.W.; Schoenberg, K.F. The Los Alamos Neutron Science Center. *Nucl. Instrum. Methods Phys. Res. Sect. A Accel. Spectrometers Detect. Assoc. Equip.* **2006**, *562*, 910–914. [CrossRef]