Electronic Testing: Theory and Applications manuscript No.
(will be inserted by the editor)

Multi-threaded mitigation of radiation-induced soft errors in bare-metal embedded systems

Alejandro Serrano-Cases · Felipe Restrepo-Calle · Sergio Cuenca-Asensi · Antonio Martínez-Álvarez ✉

Received: date / Accepted: date

Abstract This article presents a software protection technique against radiation-induced faults which is based on a multi-threaded strategy. Data triplication and instructions flow duplication or triplication techniques are used to improve system reliability and thus, ensure a correct system operation. To achieve this objective, a relaxed lock-step model to synchronize the execution of both, redundant threads and variables under protection on different processing units is defined. The evaluation was performed by means of simulated fault injection campaigns in a multi-core ARM system. Results show that despite being considered techniques that imply an evident overhead in memory and instructions (*Duplication With Comparison and Re-Execution* – DWC-R and *Triple Modular Redundancy* – TMR), spreading the replicas in different instruction flows not only produce similar results than classic techniques, but also improves the computational and recovery time in presence of soft-errors. In addition, this paper highlights the importance of protecting memory-allocated data, since the instruction flow triplication is not enough to improve the overall system reliability.

Keywords Fault tolerance · reliability · thread replication · lock-step · soft errors · bare-metal

Alejandro Serrano-Cases, Sergio Cuenca-Asensi, Antonio Martínez-Álvarez (✉)
Dept. of Computer Technology, Ctra. San Vicente del Raspeig s/n, 03690, San Vicente del Raspeig - Alicante, Spain, E-mail: aserrano@dtic.ua.es, sergio@dtic.ua.es, amartinez@dtic.ua.es

Felipe Restrepo-Calle
Dept. of Systems and Industrial Engineering, Universidad Nacional de Colombia, Bogotá, Colombia E-mail: ferestrepoca@unal.edu.co

1 Introduction

The continuous miniaturization of electronic devices makes them increasingly vulnerable to the effects of radiation, and therefore, less reliable. Radiation affects the different components present in electronic devices. For instance, incident particles may cause several disruptions to the transistors nodes. As a result, nodes voltage may alter their properties and, consequently, the system expected behavior. Also, heavy ions may degrade the transistor substrate by leaving an ionization track. These effects known as Single-Event Effects (SEE) may suppose undesirable effects in any node of the electronic device [1,2]. For example, in a microprocessor, a SEE may cause either unexpected/erroneous results (Silent Data Corruption - SDC) or may cause a state in which no further operations can be carried out (HANG) [3]. A special type of SEE called SEU (Single-Event Upsets) is characterized by being caused by alterations in the memory bits of a device (whose effect is modeled as a non-transient bit-flip). Also, SET (Single-Event Transient) are signal alterations occurring during a short period of time (which are modeled as transient bit-flips). Depending on the number of affected bits in the memory, these errors can be classified as Single-Bit Upset (SBU) or Multiple-Bit-Upset (MBU). Finally, the effect of a fault affecting the configuration bits of an electronic device, sticking it at an unknown state, is known as Single-Event Functional Interrupt (SEFI).

As a consequence, designers of mission-critical systems, such as satellites, aviation systems or autonomous driving vehicles have to modify their designs with redundant components, and perform exhaustive verification tests to improve the system reliability and thereby ensure that electronic devices continue their operation without errors in presence of the aforementioned events. In this sense, the literature offers a wide range of techniques to deal with the problem. Depending on where they are applied, these tech-

niques can be divided into hardware, software and hybrids. Most of these techniques are based on components replication to detect and/or correct radiation-induced faults. The most widespread mitigation technique is called Triple Modular Redundancy (TMR). These approaches are also called hardening techniques.

Hardware techniques are mostly based on replication of different components of microprocessors, such as registers, memories or even entire processing units. Although they have proven to be effective and elicit the highest mitigation rates, these techniques have a high economic cost, as well as a long development time aimed at a specific product. Some examples of these techniques are those based on *Dual-redundant Core Lock-Step* (DCLS) and *Triple-redundant Core Lock-Step* (TCLS) [4], which replicate each processor and compare the system status every clock-cycle to detect discrepancies during the execution of a program. Another example is defined by those techniques based on *Error Correction Codes* (ECC), such as those using Hamming code, which are known to be highly effective techniques with high correction rates, due to the fact that they can easily mask multi-bit errors with low redundancy. However, massive implementations may require large power consumption due to continuous checks and area overhead, resulting in high cost devices.

Software replication techniques, also known as SIHFT (Software-Implemented Hardware Fault Tolerance) techniques [5], are aimed to enable a reliable computing in general-purpose devices such as micro-controllers and embedded devices. These techniques introduce replication at different levels of the software components of a computer program (functions/methods, loops, assembly code, etc.), or even can replicate the whole program (process replication). Although their development is faster and less expensive when compared with hardware techniques, their application present additional costs in the execution time and the resource usage that should be taken into consideration. An example of a hardening software technique called *Trikaya* can be found in [6], where the program to be protected is executed twice in the absence of errors, and a third time if some discrepancy is found in the outputs of the two preceding replicas.

Finally, hybrid techniques attempt to integrate and improve the results obtained by software and hardware-only techniques and, at the same time, mitigate some of their deficiencies. An example of a hybrid technique is found in [7], where the authors show a co-design methodology and a hardening infrastructure that permits an easy exploration of the design space between hardware-only and software-only mitigation techniques.

The latest technological advances in electronic manufacturing processes allow multi-core systems to be a ubiquitous reality in embedded computing. The extra processing units of these systems can be used to accelerate software computation strategies aimed at mitigating or detecting soft errors, nevertheless, the majority of the existing software protection techniques are not designed to take advantage of these resources. Recent works, such as [8], point to distribute the replica computation across different processing units as a way to gain efficiency without losing reliability. Simultaneous Muti-Threading (SMT) is a long ago defined concept that has been used along with the concept of Sphere-of-Replication (SoR) to achieve reliability [9]. A SoR defines the set of resources of a program that are replicated. Several proposals have made use of these two concepts. For instance, Simultaneous and Redundantly Threaded (SRT) processors [9], Chip-Level Redundant Multithreading (CRT) [10], and Triple/Dual Core Lock-Step (TCLS / DCLS) [4] solutions make use of custom hardware approaches to achieve fault tolerance.

Software approaches, instead, try to replicate the original program code into several instruction flows or threads (Software Redundant Multi-Threading - SRMT) by using custom compilers which are also based on the SoR concept [11]. Other software approaches try to achieve reliability improvements by launching several instances (processes) inside an operating system that checks the replicas' outputs (PLR) [12]. Other proposals make use of general purpose compilers and Commercial-of-the-Shelf (COTS) devices without any modifications to enable programmatically a lock-step functionality. For example, a multi-threading strategy can be found in [13], which makes use of threading libraries and take advantage of resources present in the majority of operating systems. Whereas [14, 15] use reconfigurable hardware as FPGAs to achieve a lockstep solution. These software approaches introduce large size and performance overheads that have to be taken into consideration. In this sense, [13], [16] and [17] use high-level libraries such as *OpenMP* and *PThreads* to generate software redundancy, or [18], where a modification of *PThreads* (*RedThreads*) is used to generate reliability-oriented redundancy with performance overheads close to $3\times$. However, all of them emphasize a clear performance overhead due to the complexity introduced by the mentioned libraries. Another variable to take into account is the effect of using an operating system, which can be considered as another source of errors because it is not protected [19] and supposes an additional cost that may be unaffordable. For instance, authors in [15] proved that RTOS (Real-Time Operating System) has an impact on performance and reliability, achieved by their DCLS technique, which is similar to the bare-metal results. Also, authors in [16] conclude that the usage of parallelization APIs (Application Programming Interfaces) increases the occurrence of unexpected terminations that are caught by the operating system. In this sense, recent works such as [20] and [21] have presented multi-threaded techniques in environ-

ments without operating systems with promising results. In [20], authors show a multi-threaded data triplication technique that improves the fault coverage by 26× while presenting execution overheads less than 8× on average. In [21], a triplication technique is evaluated showing a positive influence on the reduction of error rates.

Unlike related works, this proposal tries to reduce the impact of the unavoidable non-reliable software to achieve a reliable and performant computation. In fact, no operating system nor threading libraries are used at all. Also, our proposal exploits the threads capability of modern microprocessors by means of spreading the same program execution to each available processing core. This leads multiple instances of the same program to be run in parallel without any communication between them, excluding a little piece of code for stall and synchronization purposes. Preliminary work presented in LATS [21] evaluates the threaded TMR technique, and puts in relevance that the storage data were not protected because lifetime of several variables goes beyond the technique protection. Therefore, the present work compares the use of TMR and DWC-R techniques and also introduces the application of TMR at the storage data to improve the results of previous preliminary work. As a result, a slightly instrumented version of the original program by means of synchronization blocks is proposed. This way, not only the performance overheads of each solution is investigated, but also a characterization of the impact of the synchronization routines, which are explored in conjunction with the storage data triplication.

The rest of this paper is organized as follows: Section 2 presents and describes the proposed strategies; Section 3 shows the tools designed to evaluate the different protection techniques and their configurations; Section 4 discusses the results obtained after evaluation of a set of configurations that covers all proposed techniques; finally, Section 5 presents the conclusions.

2 Multi-threaded mitigation in bare-metal

Our improved Multi-threaded mitigation techniques: *Duplication With Comparison and Re-Execution* (DWC-R) and Threaded-*Triple Modular Redundancy* (Threaded-TMR), are based on the concept of Sphere-of-Replication (SoR) first presented in [22]. SoR makes use of redundant items to make an independent computation. Therefore, the redundant computation is able to mask any fault occurrence if it occurs when the redundant items are processed. Usually, faults are masked when replicated items are compared at the end of the defined replicate block. This proposal distributes each replica across the different available computation units (microprocessor cores) to gain performance and reliability at once. Our proposal extends both techniques by adding protection (Data-TMR) to data allocated in the persistent storage (memory) during the execution of the program. This way, our technique can be considered as a high level or coarse grain mitigation technique by design. Therefore, a multi-level mitigation approach taking into consideration low level mitigation techniques, as those based on TMR with registers from the register file is also possible [23].

2.1 DWC-R: Duplication With Comparison and Re-Execution

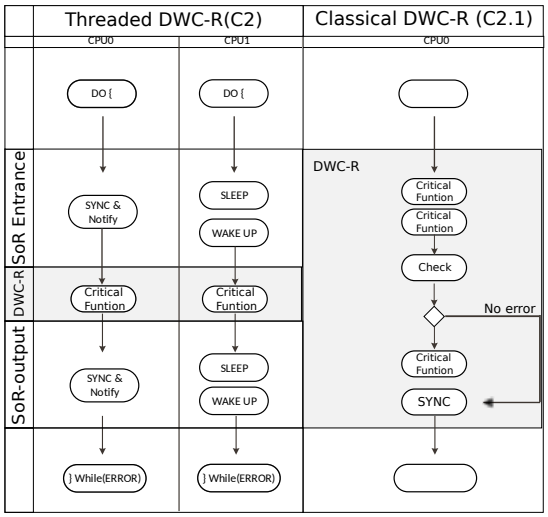


Figure 1 Threaded (left) and Classical (right) DWC-R protection techniques.

Figure 1 shows both, the parallelized and the classical approach of the DWC-R techniques. The proposed parallelized DWC-R (on the left) is based on two replicas running simultaneously on two independent shared memory processing units (CPU0 and CPU1). Before performing the hardening of the critical computation, a checkpoint to test each input variable to be hardened must be performed to create an execution context restoration point by saving the value of each variable (See SoR-entrance in Figure 1). Subsequently, the program executes the protected section which typically defines a critical calculation which involves the previous variables. After the execution of the protected section, a second checkpoint point (SoR-output) is reached. At this point, it is decided whether to go back to the first checkpoint and perform a context restoration (rollback) to re-execute the protected section due to the presence of a fault, or continue the normal program execution flow in absence of faults.

In contrast, the classical technique (on the right) executes twice the critical function before looking for discrep-

ancies in results. Only if a fault is detected, the third execution is performed in order to mask the fault.

2.2 Threaded-TMR (Triple Modular Redundancy)

The functioning of threaded (parallelized) and single-threaded approaches for TMR mitigation techniques are showed in Figure 2. The TMR parallelized technique (on the left) presents a similar structure as the previous DWC-R but instead two replicas executed simultaneously, now the three replicas are working simultaneously on different processing units.

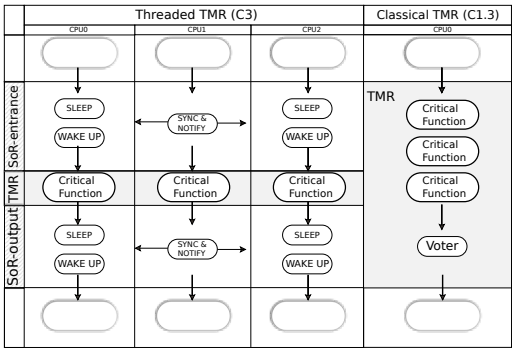


Figure 2 Threaded (left) and Classical (right) TMR protection techniques.

Similarly to DWC-R, this technique proposes a first checkpoint zone (labeled as SoR-entrance in the figure) which is optional because a desynchronization of input variables may lead to an erroneous output in a replica, which will be masked at the end of the protected zone (labeled SoR-output in the figure). However, even being optional, its use increase the system consistence and reduce the probability of faults in those scenarios where multiple bit upsets (MBU) are a concern.

Subsequently, the protected code is executed by each independent processing unit until a last synchronization zone (SoR-output). In contrast to the previous technique, the correct value is obtained by majority voting, thus, no re-execution of the protected section is needed before proceeding with the normal flow of the program. On the other hand, the classical TMR approach (on the right) executes three times the critical section to be protected to finally obtain the correct result by majority voting.

2.3 Data triplication: Data-TMR

The aforementioned techniques do not take into account the data stored in memory (persistence) and their lifetimes, so they only manage to reduce the exposition time to soft errors and correct faults that affect the current calculation and thus, prevent them to be spread to the system. A data protection technique that can be used jointly and selectively on program variables with independence of the memory sections they are allocated is presented in this subsection. Data allocated in read-only sections (*rodata*), initialized data sections (*data*), uninitialized data sections (*bss*), and automatic variables (*stack*) may have different detection or mitigation policies.

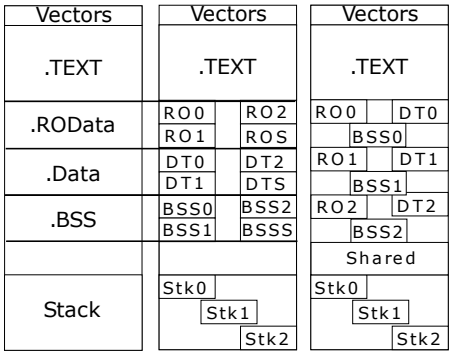


Figure 3 Example of different memory maps with and without program section replication. The first configuration shows the most widely used memory section distribution. The second configuration shows how the section are replicated for 3 processing units. The last configuration shows how the different replicas can be easily isolated in different memory sections to perform a parallel computation over them.

In addition to the data triplication, also a replica dissemination over the all available processing units is also proposed. This ensures that faults affecting a component of the device do not interfere with the computation and further processing of other replicas. In this sense, separated memory areas are requested for each replica to ensure memory isolation. For this purpose, the memory maps used by the compiler during the linking phase of the executable have been modified conveniently in such a way that it can be ensured that each replica storage is isolated in a different memory component or section.

To illustrate this, Figure 3 shows an example of data triplication and a possible mapping in a device with two Double Data Rate *DDR* chips and three processing units. As shown in the figure, the first memory map is composed by the *text* section, where the code and the *reset vectors* reside, the data sections composed by *rodata* (read-only date), *data* (initialized data), *bss* (uninitialized data), and the *stack*. The second one shows how each data section and the stacks have been triplicated, and also a new area has been added to store shared variables (e.g. variables used for exclusive access to avoid data races or perform synchronizations). In addition, the third configuration shows an example on how the different replicas can be isolated in different memory sections to perform a parallel computation over them.

2.4 Low-intrusive Hardening Using Code Annotations

All the aforementioned techniques have been applied following a strategy based on source code annotation, so that code instrumentation is reduced to a minimum by design. This means that code sections and variables to be hardened have to include only essential C/C++ compatible artifacts containing the semantics for hardening, and therefore, a complete rewriting of code is not required. Moreover, this annotation-based approach is designed to restore the behavior of the original code when no hardening is required by means of custom C/C++ definition flags that control how the hardening tasks are implemented. To achieve this objective, different strategies are followed depending on the element to be protected (variable or code section): 1) flow replication techniques, 2) replication of global scope data and 3) replication of automatic variables.

The first one of the mentioned techniques (flow replication) requires starting the program in each available processing unit simultaneously. Thus, depending on the number of available processing units, one of the replication strategies mentioned above is applied (i.e., *DWC-R*, *threaded-TMR* or classical single-threaded techniques when only one is available). The second approach (replication of global variables) has been carried out with the substitution of the declaration of each variable by hardening macros. This way, depending on the hardening level of the selected variable, the implementation of the variables are protected or left unchanged. Finally, the third approach (replication of automatic variables) is carried out transparently for the user due to the separation of stacks in multi-threaded based techniques.

An example of the application of each strategy can be seen in the C++ code showed in Figure 4. Note the 'C/C++ *defines*' designed to trigger each hardening behavior: letters D and C stands for data and code respectively. The next digit is the number of threads, so `C3` means *using 3 threads*. The remaining characters, if any, represent the detection or mitigation technique (see `C1.2` and `C1.3`). In the example, a simple C/C++ program containing 3 global variables that are used by a simple function (`void foo`) is shown. Note how the three variables to be hardened are redefined to add the appropriate hardening technique semantics (*Data-TMR*). Macros `RODATA_HARD`, `DATA_HARD` and `BSS_HARD` are in charge of triplicate variables in the read-only, `.data` or `.bss` program sections, respectively. The macro (`PTR`) gives each processing unit the ability to automatically access its own replica using its original variable name. Finally, note that variable `autoVar` is allocated in the `stack` section of each thread and thus, it is independent of the other replicas and synchronization is needed. In the case of having only a single processing units, this variable is forced to have a read-only behavior since successive executions must have the same input value in order to perform the same calcula-

```
#define D3        //Data-TMR
//#define C2      //Thread Instruction DWC-R
#define C3        //Thread Instruction TMR
//#define C1.2    //Classic DWC-R
//#define C1.3    //Classic TMR

#include "MTH.h" //Macro definitions

/** Original definitions of variables: **
 ** constVar, initVar and noInitVar.   **/
//const int constVar = 3;
//float initVar  = 5.0;
//char noInitVar;

/** Anotated versions of variables:  **
 ** constVar, initVar and noInitVar. **/
RODATA_HARD(int, constVar, 3)
#define constVar  PTR(constVar)
//const int constVar_0 = 3; //Stored in RO_0
//const int constVar_1 = 3; //Stored in RO_1
//const int constVar_2 = 3; //Stored in RO_2
//#defines constVar *constVar_ptr

DATA_HARD(float, initVar, 5.)
#define initVar   PTR(initVar)
//float initVar_0 = 5.; //Stored in DT_0
//float initVar_1 = 5.; //Stored in DT_1
//float initVar_2 = 5.; //Stored in DT_2
//#defines initVar *initVar_ptr

BSS_HARD(char, noInitVar)
#define noInitVar PTR(noInitVar)
//char noInitVar_0; //Stored in BSS_0
//char noInitVar_1; //Stored in BSS_1
//char noInitVar_2; //Stored in BSS_2
//#defines noInitVar *noInitVar_ptr

void foo(int autoVar){
    int entryVar;
    int outVar;
    ...
    SYNC(entryVar, autoVar, ...)
    THREAD_REPLICA_CONST(int, constVar)
//const int* constVar_ptr = &constVar_COREID;
    THREAD_REPLICA_VAR(float, initVar)
//float* initVar_ptr = &initVar_COREID;
    THREAD_REPLICA_VAR(char, noInitVar)
//char* noInitVar_ptr = &noInitVar_COREID;
        ...
        ///
        ///  SoR section
        ///
        int fooVar = 4;
        noInitVar = 3 * constVar << fooVar;
        outVar = noInitVar * DataVar;
        ...
    SYNC(outVar)
    ...
}
```

Figure 4 Example of a C/C++ code instrumentation for hardening three variables and a code section using TMR and three threads. C-line comments show the internal functioning of the techniques. Also, it is shown how each core access to its own private replica.

tion on the replicated data. Also, a code section contained by the mentioned *foo* function is annotated to be hardened. Variables which control the input and output to the *Sphere of Replication* are indicated with `FC_HARD` and `SYNC` respectively; in this case `entryVar`, `autoVar` and `outVar`. Variables `entryVar` and `autoVar` are synchronized (protected) using the first call to the `SYNC` macro, whereas protection of `outVar` is indicated in the second `SYNC`. Therefore, computation diverges in these points and changes its normal behavior depending on the protection technique and the number of available processing units.

The annotation proposal has the limitation of dealing with variables allocated in the HEAP section. In fact, the HEAP memory section may present a random allocation behavior during a normal program execution (e.g., using `malloc`). Therefore, it is difficult to track changes on this section and replicate them on the remaining replicas. This behavior makes it difficult to plan a static checking due to the unknown replicas allocation.

3 Experimental Setup

To assess and validate different combinations of the aforementioned multi-threaded fault mitigation techniques, we have selected the matrix multiplication from the project BEEBS [24] to harden the multiplication of two 20×20 square matrices. As a result, a reference and eight hardened versions of the program have been generated to test the suitability of the strategies, as well as to study the impact of data triplication strategies on program performance.

This classical matrix multiplication algorithm is defined by means of three nested loops. The two outermost ones select an index for each element and the innermost loop computes the multiplication for this index. Each hardened version corresponds to harden the computation of one of these loops with one of the flow replication techniques, so we have 8 possibilities. Therefore, protecting the outermost loop has the effect of applying the protections to the whole function, while protecting the innermost loop (`AccLoop`) is equivalent to protect each cell from the resultant matrix.

Figure 5 represents how the hardening instrumentation was done. Data protection is applied to each matrix defined by the algorithm (`ArrayA`, `ArrayB`, and `ResultArray`). By doing this, it is ensured that each replica is computed inside an exclusive memory area.

Hardened versions are denoted by `C2` and `C3`, which refers to DWC-R and TMR flow protections, respectively; and a suffix to denote the protection level applied (`Outer`, `Inner` or `Acc`). The version used as reference is denoted by `C0-NH` (NH for non-hardened) and has no data or flow replication technique was applied. To complete the study, two versions with the classical single-threaded hardening

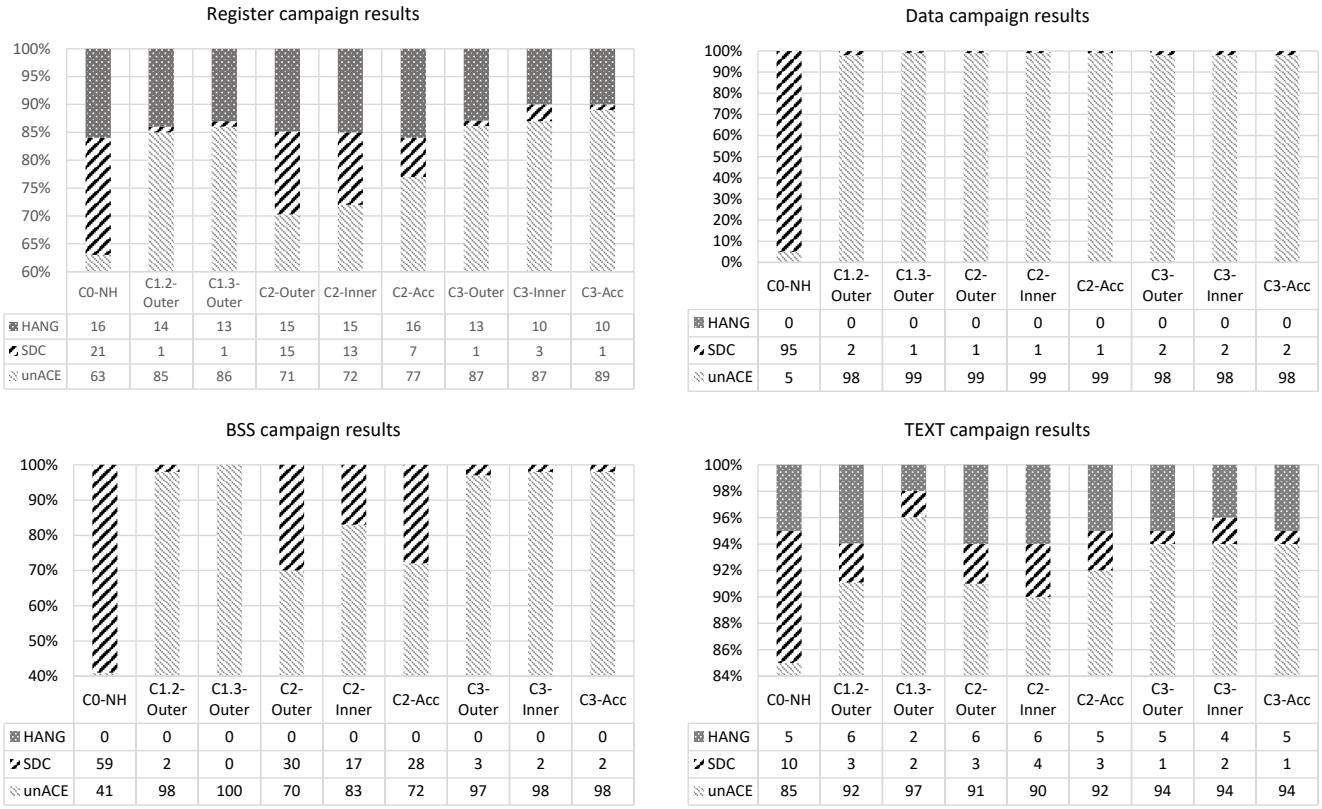
```
BSS_HARD(matrix, ArrayA)     //matrix ArrayA;
#define ArrayA PTR(ArrayA)
BSS_HARD(matrix, ArrayB)     //matrix ArrayB;
#define ArrayB PTR(ArrayB)
BSS_HARD(matrix, Result)     //matrix Result;
#define Result PTR(Result)
//matrix expected = {...};
DATA_HARD(matrix, expected, {...})
#define expected PTR(expected)

void initMatrices(int seed);
int verify();
/*
 * Multiplies arrays A and B
 * and stores the output in Result.
 */
void Multiply(matrix A, matrix B, matrix Res)
{
register int Outer, Inner, Index;
#ifdef OuterLoopProtection
SYNC()
#endif
for (Outer = 0; Outer < UPPERLIMIT; Outer++)
{
#ifdef InnerLoopProtection
SYNC(Outer)
#endif
for (Inner = 0; Inner < UPPERLIMIT; Inner++)
{
#ifdef AccLoopProtection
SYNC(Outer, Inner)
#endif
Result [Outer][Inner] = ZERO;
for (Index = 0; Index < UPPERLIMIT; Index++)
{
Result[Outer][Inner] +=
A[Outer][Index] * B[Index][Inner];
}
#ifdef AccLoopProtection
SYNC(Result[Outer][Inner])
#endif
}
#ifdef InnerLoopProtection
SYNC(Result[Outer])
#endif
}
#ifdef OuterLoopProtection
SYNC(Result)
#endif
}
```

Figure 5 Matrix multiplication program selected from the project *BEEBS*. The code shows the different points where the multi-threading and classical strategies may introduce redundancy to gain reliability. Also, it is shown how to protect uninitialized variables (`.bss` section) using *TMR*, except for the *expected* variable, which is saved in the `.data` section.

flow and data triplication were used (`C1.2` and `C1.3`, respectively).

Simics simulator has been used to evaluate all the above program versions. This tool has been configured to simulate the board *Versatile Express*, characterized by offering



Register campaign results

	C0-NH	C1.2-Outer	C1.3-Outer	C2-Outer	C2-Inner	C2-Acc	C3-Outer	C3-Inner	C3-Acc
HANG	16	14	13	15	15	16	13	10	10
SDC	21	1	1	15	13	7	1	3	1
unACE	63	85	86	71	72	77	87	87	89

Data campaign results

	C0-NH	C1.2-Outer	C1.3-Outer	C2-Outer	C2-Inner	C2-Acc	C3-Outer	C3-Inner	C3-Acc
HANG	0	0	0	0	0	0	0	0	0
SDC	95	2	1	1	1	1	2	2	2
unACE	5	98	99	99	99	99	98	98	98

BSS campaign results

	C0-NH	C1.2-Outer	C1.3-Outer	C2-Outer	C2-Inner	C2-Acc	C3-Outer	C3-Inner	C3-Acc
HANG	0	0	0	0	0	0	0	0	0
SDC	59	2	0	30	17	28	3	2	2
unACE	41	98	100	70	83	72	97	98	98

TEXT campaign results

	C0-NH	C1.2-Outer	C1.3-Outer	C2-Outer	C2-Inner	C2-Acc	C3-Outer	C3-Inner	C3-Acc
HANG	5	6	2	6	6	5	5	4	5
SDC	10	3	2	3	4	3	1	2	1
unACE	85	92	97	91	90	92	94	94	94

Figure 6 Results from the fault injection campaigns. Each sub-figure corresponds to a different resource being tested, such as the register file (top left), initialized data *.data* (top right), the non-initialized *.bss* (bottom left) and the *.text* section (bottom right) where the program resides. Tables show the resulting unACE, SDC and HANG rates (%) obtained from the simulation-based fault injection campaigns performed against each resource separately.

the possibility of using different multi-core processors of the family *Cortex-A9* from *ARM*.

To provide the simulator with reliability assessment capabilities, a non-intrusive fault injection *ad hoc* plugin has been developed. This means that during each simulation, a bit-flip is introduced in a random bit of a random resource under evaluation (mainly a register from the register file or memory location) without using any interruption or instrumentation routine.

The injection is directed by the tool *Fault Injection Manager* (FIM) [25], which is in charge of calling the platform that inserts the bit-flips, restarting it after each injection and labeling the result. Thus, faults are labeled as *unACE* (unnecessary for Architectural Correct Execution) when an injection is made, and they do not affect the result of the program's output, *SDC* (Silent Data Corruption) when the result is not correct but the program ends, and *HANG* if it does not end or exceeds a time limit. The programs have been evaluated having as a reference a faultless (ground truth) execution of themselves and adding a recovery time equal to the faultless execution duration. If this restriction is exceeded, the program is considered that does not meet the valid requirements and is labeled as *HANG*.

4 Experimental Results and Discussion

The fault injector has been configured to perform 1800 injections per core at the register file (100 · 18 registers), 800 injections per memory section and core (200 · {copy 0, copy 1, copy 2, shared} at `.rodata`, `.data`, `.bss` and `.stack`) and 200 injection at `.text` section per core. Thus, 5200 injections of faults have been injected at the single-core versions, 10400 at multi-threaded *DWC-R* versions and 15600 at multi-threaded *TMR* versions.

Raw event rates (unACE, SDC and HANG) from simulated fault injection campaign are shown in Figure 6, where each plot shows the results obtained after performing a fault injection campaign within a single resource over the different program versions. For sake of simplicity, only most relevant sections are shown (`registers, .data, .bss and .text`).

The fault injection campaign for registers (Figure 6 first plot - top left) shows the reference version (`C0-NH`), which shows an elevated rate of SDC and HANG (21% and 16% respectively). The HANG rate is mostly constant at around 10-15% across the different versions because the majority of the faults are caused by the stack pointer, the frame pointer, and the program counter. These registers present a partic-

ularity, they are more prone to get a HANG state when a bit-flip is produced. It is worth to mention that hardened versions also achieve the same HANG rate due to the fact that no timeouts nor reconfiguration methods have been provided in case of one of the replicas hangs. As a result, hangs faults affect the whole system by freezing it in a dead-lock due to the fact that synchronization routines are expecting that all cores enter the routine before resuming the computation. In contrast, the SDC rates show a general reduction with rates around 1-5%, except for the threaded DWC-R version denoted with `C2`, which is only able to reduce to rates around 7-14%.

The second plot from Figure 6 (top right) shows the injection results performed over the `.data` section, in which is stored the expected results used to compare with the program computed results. As can be seen, reference point shows a higher error rate of SDC (95%) due to the fact that the variable `exp` has the highest lifetime, whereas, hardened versions barely present errors (1%). As expected, data triplication could mask mostly all the errors presented in this memory section and, as a result, they present the highest correction rates. Also, the presence of erroneous output indicates that data triplication technique has a reduced time when it still remains vulnerable.

The third plot (bottom left) from Figure 6 shows that `.bss` sections follows the same trend as `.data` section: SDC fault rates around 60% in the reference version while hardened versions are around 1%, except for the threaded DWC-R, which are only able to reduce the rate to 20-30%.

Finally, the `.text` section (fourth plot from Figure 6 - bottom right) presents similar trends when hardened versions are compared against the reference version. All hardened versions present improvements in SDC rate, and they manage to reduce rates from 10% to 3%, while HANG rate remains constant across all hardened builds (5%), except `C1.3` that achieves the best results (2%).

To evaluate the hardening performance and estimate the inherent trade-offs between speed, memory footprint and fault coverage, a compact mathematical model presented in [26] has been used. This model evaluates the effects of high-level C++ code hardening. With this model a quick and effective radiation tolerance optimization of different hardening techniques is possible. Moreover, it is proven with real radiation experiments over the same hardware than ours. Two different assessment concepts are defined in this model, the *equivalent size* measured in Bytes ($\beta_\psi = Size_\psi \cdot \frac{\#SDC_\psi}{\#Injections_\psi}$ and $\gamma_\psi = Size_\psi \cdot \frac{\#HANG_\psi}{\#Injections_\psi}$), which takes into account the size (in Bytes) of each section (ψ) among the raw error event rates obtained from the simulated injection campaigns; and the *size-time* figure ($\chi_{SDC} = T_E \cdot \sum K_\psi \cdot \beta_\psi$ and $\chi_{HANG} = T_E \cdot \sum K_\psi \cdot \gamma_\psi$), which are used to estimate the overheads and predict the measures from real accelerated radiation test of the application under evaluation. Also, the model offers a way to approximate the obtained *size-time* figure to the well-known Mean-Work-To-Failure (MWTF) metric presented in [27] as $MWTF = \frac{1}{\Phi \cdot \alpha \cdot \chi}$, which offers a value of the amount of work that the application is able to do before a fault occurs.

Table 1 Equivalent size metric in Bytes for each section evaluated. Lower values are better (bold).

	REGISTER		DATA		BSS		Text	
	β (B)	γ (B)	β (B)	γ (B)	β (B)	γ (B)	β (B)	γ (B)
C0-NH	531	403	1520	0	2834	0	869	**435**
C1.2-O	25	342	88	0	288	0	1040	1906
C1.3-O	**18**	**326**	48	0	**0**	0	494	**494**
C2-O	749	736	56	0	4300	0	1080	2250
C2-I	657	731	36	0	2462	0	1349	2410
C2-A	343	799	**32**	0	4071	0	1330	1841
C3-O	56	962	85	0	440	8	**465**	1627
C3-I	227	752	88	0	352	0	746	1493
C3-A	99	752	109	0	256	0	528	1979

Table 1 shows the equivalent size (β and γ) of each version (nine in total). As can be seen, the register file shows that classical TMR versions (`C1.2-O`, `C1.3-O`) offer better results than multi-threading ones due to the resource triplication. This effect, which produces a worsening of 3× in the HANG rates can be also appreciated. `.data` section reveals a significant fault probability reduction, which states that data triplication offers good protections to long-lifetimes variables; also the associated HANG rate is negligible. The `.bss` section exposes that classical approaches achieve better corrections rates which is correlated with the register section data. Again, it can be seen the DWC-R routine weaknesses, which doubles the reference program (`C0-NH`). Also, table shows that multi-threading techniques with bigger synchronization blocks are more prone to faults. Finally, the `.text` section data exposes that TMR involves less complex synchronization code, and as a consequence, it is less prone to SDC; on the contrary, the HANG rate is higher than expected, because no recovery routine, in case of one thread *hangs*, is provided.

Time-Size and MWTF metrics are showed in Table 2, where better applications are considered those with higher reliability and lower execution time (lower χ and higher MWTF). As in the cited paper [26], the proposed model was evaluated against real radiation measurements with protons and neutrons during two radiation campaigns described in it. As can be seen, the reference version presents a MWTF of 1.21 in protons and 0.35 in neutrons, which is only improved by TMR versions. Focusing at DWC-R, all of them show a worsening in their results compared to the reference version due to an elevated execution time overhead (around 1.5× and 2×) and the aforementioned synchronization vulnerabilities (around 1.5× and 2× χ_{TOT}). Focusing at TMR,

Table 2 Execution time in *microseconds* (T_E), Time-Size metric measured in $Bytes \cdot milliseconds$ (χ) and MWTF in number of programs executions until a failure is produced. MWTF columns has been calculated to an estimated proton/neutron irradiation campaign with radiation flux of $1 \times 10^9 particles/cm^2/s$ for protons and $7.5 \times 10^5 particles/cm^2/s$ for neutrons. Best values are in bold.

	T_E (μs)	χ_{SDC} (B*ms)	χ_{HANG} (B*ms)	χ_{TOT} (B*ms)	p^+ MWTF RUN (10^1)	n^0 MWTF RUN(10^3)
C0-NH	**886**	5145	**816**	5961	1.21	4.61
C1.2-O	2090	3096	4786	7883	0.91	3.49
C1.3-O	3115	1869	**2666**	4535	1.59	6.06
C2-O	1221	7612	3732	11344	0.63	2.42
C2-I	1259	5744	3997	9741	0.74	2.82
C2-A	1730	10079	4661	14741	0.49	1.86
C3-O	**1200**	**1315**	3173	**4488**	**1.60**	**6.13**
C3-I	1233	1812	2838	4650	1.55	5.91
C3-A	1591	1652	4419	6071	1.19	4.53

all versions improve the reference version except `C3-Acc`, which presents the highest number of synchronizations (one per each calculated element) and thus it slows-down the system performance by a factor of 2×. Also, `C1.3-Outer` shows one of the highest MWTF rates, even having the highest overhead in time (3.5× T_E), due to the higher reliability gain (2.5× χ_{HANG}). On the other side, if `C1.3-Outer` is compared with multi-threaded versions (3.8× χ_{HANG} over the reference), the best T_E is determinant to obtain the higher MWTF.

5 Conclusion

A software protection technique based on a multi-threaded strategy with memory protection based on triplication against radiation-induced faults is presented in this paper. Also, this work shows the behavior of the protection with different program and microprocessor resources (memory sections and processor registers) and identifies the vulnerabilities of multi-threading protection techniques. Results offer an overview of the overheads introduced when applying different techniques using several configurations, and protection levels.

Experimental results show that raw fault rates are not enough to evaluate the system reliability, because the resource usage and exposure time are factors that need to be taken into account jointly to test the suitability of a solution. For instance, non-threaded DWC-R offers good fault coverage rates, however its associated resource usage and high latencies make it worse than the reference version. In case of TMR (both, threaded and non-threaded), it can be seen that application achieves higher improvements (33% MWTF in average).

Also, it has been demonstrated that significant reliability improvements are achieved when replicas are spread on each processing unit. In addition, performance improvements can be obtained when large software resources are eliminated, such as complex threading libraries or the operating system itself. Without this software, the application resources and the overall system complexity is reduced to a minimum.

Acknowledgements 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*', (Ref: `ESP2015-68245-C4-3-P`, MINECO/FEDER, UE).

References

1. J. M. Benedetto, P. H. Eaton, D. G. Mavis, M. Gadlage, and T. Turflinger, "Digital single event transient trends with technology node scaling," *IEEE Transactions on Nuclear Science*, vol. 53, pp. 3462–3465, dec 2006.
2. R. Gaillard, "Single Event Effects: Mechanisms and Classification," in *Soft Errors in Modern Electronic Systems* (M. Nicolaidis, ed.), vol. 41 of *Frontiers in Electronic Testing*, pp. 27–54, PO BOX 17, 3300 AA DORDRECHT, NETHERLANDS: SPRINGER, 2011.
3. R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, pp. 305–316, Sept. 2005.
4. X. Iturbe, B. Venu, E. Ozer, and S. Das, "A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications," in *Proc. 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pp. 246–249, IEEE, June 2016.
5. O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-implemented hardware fault tolerance*, vol. XIV. Springer, 2006.
6. H. Quinn, Z. Baker, T. Fairbanks, J. L. Tripp, and G. Duran, "Software Resilience and the Effectiveness of Software Mitigation in Microcontrollers," *IEEE Transactions on Nuclear Science*, vol. 62, pp. 2532–2538, Dec. 2015.
7. S. Cuenca-Asensi, A. Martinez-Alvarez, F. Restrepo-Calle, F. R. Palomo, H. Guzman-Miranda, and M. A. Aguirre, "A Novel Co-Design Approach for Soft Errors Mitigation in Embedded Systems," *IEEE Transactions on Nuclear Science*, vol. 58, pp. 1059–1065, June 2011.
8. I. Oz and S. Arslan, "A Survey on Multithreading Alternatives for Soft Error Fault Tolerance," *ACM Computing Surveys*, vol. 52, pp. 27:1–27:38, Mar. 2019.
9. S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 25–36, May 2000.
10. S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," *ACM SIGARCH Computer Architecture News*, vol. 30, pp. 99–110, May 2002.
11. C. Wang, H. seop Kim, Y. Wu, and V. Ying, "Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection," in *Proc. International Symposium on Code Generation and Optimization (CGO2007)*, pp. 244–258, IEEE, Mar. 2007.
12. A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors, "PLR: A software approach to transient fault tolerance for multicore architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, pp. 135–148, Apr. 2009.
13. G. S. Rodrigues, F. Rosa, F. L. Kastensmidt, R. Reis, and L. Ost, "Investigating parallel TMR approaches and thread disposability

in Linux," in *Proc. 2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 393–396, IEEE, Dec. 2017.
14. A. B. de Oliveira, L. A. Tambara, and F. L. Kastensmidt, "Applying lockstep in dual-core ARM cortex-a9 to mitigate radiation-induced soft errors," in *2017 IEEE 8th Latin American Symposium on Circuits & Systems (LASCAS)*, pp. 1–4, IEEE, Feb. 2017.
15. Á. B. de Oliveira, G. S. Rodrigues, and F. L. Kastensmidt, "Analyzing lockstep dual-core ARM cortex-a9 soft error mitigation in freeRTOS applications," in *Proceedings of the 30th Symposium on Integrated Circuits and Systems Design Chip on the Sands - SBCCI 2017*, SBCCI '17, (New York, NY, USA), pp. 84–89, ACM Press, 2017.
16. G. Rodrigues, F. ROSA, A. de Oliveira, F. L. Kastensmidt, L. Ost, and R. Reis, "Analyzing the Impact of Fault Tolerance Methods in ARM Processors under Soft Errors running Linux and Parallelization APIs," *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 2196–2203, 2017.
17. G. S. Rodrigues, F. L. Kastensmidt, R. Reis, F. Rosa, and L. Ost, "Analyzing the impact of using pthreads versus OpenMP under fault injection in ARM cortex-a9 dual-core," in *2016 16th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, pp. 1–6, IEEE, Sept. 2016.
18. S. Hukerikar, K. Teranishi, P. C. Diniz, and R. F. Lucas, "RedThreads: An Interface for Application-Level Fault Detection/Correction Through Adaptive Redundant Multithreading," *International Journal of Parallel Programming*, vol. 46, pp. 225–251, Feb. 2017.
19. J. S. Monson, M. Wirthlin, and B. Hutchings, "Fault Injection Results of Linux Operating on an FPGA Embedded Platform," in *Proc. 2010 International Conference on Reconfigurable Computing and FPGAs*, pp. 37–42, IEEE, Dec. 2010.
20. H. So, M. Didehban, A. Shrivastava, and K. Lee, "A software-level Redundant MultiThreading for Soft/Hard Error Detection and Recovery," in *Proc. 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1559–1562, IEEE, Mar. 2019.
21. A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martinez-Alvarez, "Softerror mitigation for multi-core processors based on thread replication," in *Proc. 2019 IEEE Latin American Test Symposium (LATS)*, pp. 1–5, IEEE, Mar. 2019.
22. S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 25–36, May 2000.
23. A. Martinez-Alvarez, S. Cuenca-Asensi, F. Restrepo-Calle, F. R. Palomo Pinto, H. Guzman-Miranda, and M. A. Aguirre, "Compiler-directed soft error mitigation for embedded systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, pp. 159–172, march 2012.
24. J. Pallister, S. J. Hollis, and J. Bennett, "BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms," *CoRR*, vol. abs/1308.5174, 2013.
25. J. Isaza-Gonzalez, A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martinez-Alvarez, "Dependability evaluation of COTS microprocessors via on-chip debugging facilities," in *Proc. 2016 17th Latin-American Test Symposium (LATS)*, pp. 27–32, IEEE, Apr. 2016.
26. L. M. Reyneri, A. Serrano-Cases, Y. Morilla, S. Cuenca-Asensi, and A. Martínez-Álvarez, "A Compact Model to Evaluate the Effects of High Level C++ Code Hardening in Radiation Environments," *Electronics*, vol. 8, p. 653, June 2019.
27. G. Reis, J. Chang, N. Vachharajani, R. Rangan, D. August, and S. Mukherjee, "Design and Evaluation of Hybrid Fault-Detection Systems," in *Proc. 32nd International Symposium on Computer Architecture (ISCA2005)*, pp. 148–159, IEEE, 2005.