IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 26 March 2024, accepted 19 April 2024, date of publication 25 April 2024, date of current version 3 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3393844

RESEARCH ARTICLE

HW-SW Interface Design and Implementation for Error Logging and Reporting for RAS Improvement

NICASIO CANINO[1], STEFANO DI MATTEO[1,2], DANIELE ROSSI[1], (Senior Member, IEEE), AND SERGIO SAPONARA[1], (Senior Member, IEEE)

[1]Department of Information Engineering, University of Pisa, 56126 Pisa, Italy
[2]CEA-List, Grenoble Alpes University, 38000 Grenoble, France

Corresponding author: Nicasio Canino (nicasio.canino@phd.unipi.it)

This work was supported in part by European Union's Horizon 2020 Research and Innovation Program "European Processor Initiative" (EPI) under Grant 101036168 (EPI SGA2); and in part the Italian Ministry of University and Research (MUR) in the framework of the CrossLab and FoReLab projects (Departments of Excellence), Department of Information Engineering, University of Pisa.

ABSTRACT When designing a resilient computing system, the desired degree of Reliability, Availability, and Serviceability (RAS) must be assessed and guaranteed. This article presents a Hardware-Software (HW-SW) Interface for Error Logging and Reporting independent of specific Instruction Set Architectures (ISA), aiming to improve RAS in computing systems. A HW-SW Interface defines the facilities by which detected hardware errors are logged into an ad hoc set of registers (i.e., Error Record) and then reported to system software. System software will promptly address and recover from those errors, preventing system failures. Our architecture offers flexible and configurable Error Logging and Reporting features, satisfying the requirements of different application scenarios by selectively incorporating or removing specific features. After reporting the most relevant results from synthesis on FPGA (Xilinx UltraScale+ MPSoC) and Standard-Cell technologies (45nm and 7nm libraries), we discuss them to provide valuable insights on the dependency of resource utilization on error logging capability. The principal findings demonstrate that the developed module would not limit system operating frequency, and its area occupation can be readily configured to align with desired logging and reporting features to be implemented. Then, we validate the Error Logging and Reporting features of our architecture by developing a test SoC on FPGA that emulates a computing system, including a 32-bit RISC-V core and two ECC-protected (Error Correcting Code) memories. The proposed HW-SW Interface extends beyond monitoring only ECC-protected memories, yet it can monitor any system module incorporating error control logic.

INDEX TERMS Error logging, error reporting, FPGA, HW-SW interface, reliability-availability-serviceability.

I. INTRODUCTION

The three separate properties of *Reliability*, *Availability*, and *Serviceability* (RAS) in a computing system converge into one main concept, which is its ability to prevent or recover from failures and to handle undesired system states adequately [1]. A greater system RAS enables a lower downtime frequency and may provide a quick recovery from system failures. However, it should be increased with minimal impact on performance and cost to bring a real benefit to the system. This requirement can be addressed by proper adoption of error-checking hardware and redundancy on critical data paths [1]. Both solutions should provide the ability to detect and locate errors upon their first generation and, if hardware is also able to autonomously correct it, system operation continues without any noticeable loss in performance.

Since errors in a system can arise from a plethora of causes, over the years numerous solutions have been proposed to reach the desired RAS, being either hardware-based or

The associate editor coordinating the review of this manuscript and approving it for publication was Lorenzo Ciani.

software-based. Different software-based solutions can be embraced to increase the overall system RAS through *fault prediction* and *error recovery* techniques [1], [2], [3], [4], [5], [6], [7]. Fault prediction tries to prevent failures by analyzing fault statistics and effective resource usage [8], [9]. It allows for the confinement of potentially defective parts of the system and the application of predictive maintenance. Relevant information to implement this technique can be gathered by dedicated SW such as the HealthLog monitor proposed in [5]. Error recovery, instead, is applied after the occurrence of the error or failure and tries to recover the correct system functionality as effortlessly as possible. However, all software approaches need some hardware to monitor selected modules and to signal the occurrence of an error. Concurrently, there are several hardware-based solutions not only to *monitor* and *correct errors* [4], [10], [11], [12] but also to *monitor the usage* of some system modules (e.g., Performance Monitors [13]). Since many studies assess memories to be the components most susceptible to errors in a computing system (99.9% of errors corrected by ECC (Error Correcting Code) circuitry [1], [14], [15], [16]), if not explicitly specified, for the rest of the paper we will only refer to memories and their ECC circuitry.

Regarding hardware errors, they can be classified as *soft errors* and *hard errors*. Soft errors are usually caused by the emission of high-energy particles from the atmosphere, or alpha particles from the chip package (non-deterministic event) [17]. Instead, hard errors depend on physical defects or wear out of the memory (deterministic event) [18]. Therefore, technology advancement, increased miniaturization, higher operating frequencies, and lower voltage levels, make memories more susceptible to errors (whether soft or hard).

Both hardware- and software-based approaches have a detrimental impact on the availability and performance of the system. Indeed, the software needs to handle every error event even if it does not lead to a failure, e.g. erroneous data autonomously corrected by HW through its ECC code. Concurrently, the hardware cannot handle autonomously certain error states, for example, when the ECC logic is not able to directly correct the error. For this reason, a "*HW-SW Interface for Error Logging and Reporting*" is required, which can store relevant information related to the detected HW errors, calling system software only when its intervention is truly necessary to restore correct operation or contain the error effect. This way, it can significantly increase the system's availability and serviceability, affecting its performance as little as possible.

The HW-SW Interface acts as a monitoring peripheral for error messages from error-checking hardware. When an error is detected, it stores relevant error-related information such as severity and source of the error, address location (if available), and timestamp, in an *Error Record* (ad hoc set of registers). A set of error records is referred to as a *Bank*. The HW-SW Interface can also alert software about potential failures. This is crucial since such failures could potentially result in the loss of data integrity, thus discarding all the results of the affected task.

Nowadays, only ICs design companies like Intel, AMD, and Arm [19], [20], [21] provide Error Logging and Reporting features in their systems. These proprietary solutions can be divided into two main approaches, briefly described in Subsection II-C: the Intel/AMD approach [19], [20], denoted as *Machine Check Architecture (MCA)*, and the Arm approach [21], referred to as *RAS Extension*. Some processor architectures, including various generations of Intel Xeon [22] and AMD EPYC [23], implement the MCA. Likewise, several Arm platforms such as the v8-A architecture [24], include the RAS Extension. These two approaches represent the state-of-the-art in terms of HW-SW Interfaces for processor systems. Still, to the best of the authors' knowledge, no micro-architectural details of these solutions are publicly available.

Regarding our work on designing and implementing a HW-SW Interface, preliminary results have been presented at the SEEDA-CECNSM conference [25]. Considering the scenario described above, the main contributions of this paper are summarized hereafter:

- Proposal of the first public micro-architecture of an ISA-independent HW-SW Interface for Error Logging and Reporting, named ENGAGE (Error loggiNG And reportinG architecturE). Its internal architecture is characterized by a first pre-processing stage that gathers and synchronizes the error-related information from the monitored HW units, and then a log & report stage that writes the information in an error record.
- The proposed HW-SW Interface is highly configurable in the implementation stage, thus allowing designers to meet different trade-offs in terms of system resource overhead and logging/reporting requirements. Therefore, it can satisfy the needs of different application scenarios, from embedded low-cost systems, which usually rely on 32-bit CPUs, to high-performance computers, which usually exploit the processing power of 64-bit CPUs.
- Evaluation of the ENGAGE module considering power consumption and area occupation on both Standard-Cell and FPGA technologies.
- Validation of the ENGAGE module in a test SoC demonstrator implemented on FPGA, which emulates a simple computing system.

The rest of the paper is organized as follows: Section II provides some background information including the state-of-the-art in terms of HW-SW Interfaces, with a brief description of the main properties of the two existing approaches (i.e., Intel/AMD and Arm approaches). In Section III, a system-level description of the designed architecture will be provided, and a detailed discussion of the purpose and functionality of all its building blocks will be included. In Section IV, the results of the syntheses performed on both FPGA and Standard-Cell will be discussed. Then, the testing

environment in which the developed architecture has been validated will be described. Finally, Section V will conclude with a summary of the main characteristics of the developed IP and of all the major results obtained.

II. BACKGROUND

After evaluating the desired resiliency level of a system, we must consider implementing a combination of different approaches because each one can improve one or more properties within "RAS". Specifically, Reliability refers to the system's ability to avoid failures and produce correct results over time. Availability measures the system's functionality at a specific moment and is often expressed as the percentage of time it delivers the expected service. Serviceability relates to the system's capability to provide relevant information about occurred failures, with *error logging* playing a crucial role.

In the considered scenario, we will refer to an Error as the *"deviation of an external state of the system from the correct service state"* [26]. For instance, a SEU (soft error) in memory will cause an error in the affected data. It is important to observe that not all errors lead to failure because they could be corrected or be irrelevant to the system's correct operation.

A. ERROR DETECTION, CORRECTION, LOGGING AND REPORTING

Several approaches can be adopted to enhance the RAS of a system. From a hardware perspective, *Error Detection, Correction, Logging, and Reporting* mechanisms are effective ways to improve the system's RAS. Usually, system memories (either Main Memory or Caches) adopt SEC-DED (Single-Error-Correction and Double-Error-Detection) codes to correct single errors and detect double errors affecting the protected information [27], [28]. As far as RAS is concerned, their adoption provides the system with increased reliability and availability. For instance, if an error is autonomously corrected by some dedicated hardware, the system will not undergo a failure (increase in reliability) and will continue to operate correctly with no downtime. Concurrently, Error Logging and Reporting operate in synergy with ECC to increase the reliability, availability, and serviceability of the system. Indeed, error logging enables an improvement of reliability and serviceability as, after the detection of an error, relevant error-related information is stored as an error record in the ad-hoc set of registers. Therefore, in case of a recoverable error (e.g., a single-bit error or a double-bit error in a non-modified line in the data cache with SEC-DED protection), it can correct the error either by fixing the data or invalidating only the affected cache line. In case of unrecoverable errors (double bit error in a modified line in the data cache, with SEC-DED protection), the system must be halted, as there is no possible simple recovery action. Also, error reporting increases availability since, signaling immediately a critical event to software, the system can take recovery actions before the error further expands, for example, in case of erroneous data used in a task. Error logging and reporting nowadays are implemented in processor systems by proprietary HW-SW Interfaces, such as *MCA* from Intel [19] and AMD [20], or *RAS Extension* from Arm [21].

B. ERROR TAXONOMY

Error Taxonomy classifies errors in the system depending on their severity for the correct system state. Hypothesizing that the monitored units are equipped with hardware for error detection and correction (error-checking HW), the error taxonomy depicted in Figure 1 has been considered [21].

Upon its occurrence, an error is classified according to its detectability. Additionally, undetected errors cannot be handled by the HW-SW Interface because of the very nature of this type of error. Therefore, they are not further analyzed, even though they may cause a system failure.

Detected errors can either be *corrected* or *uncorrected*, depending on the error-checking HW and the type of error. Also, an uncorrected error can be further categorized as *deferred* or *urgent*. Adopting the standard description of these three error categories [19], [20], [21], detected errors are then classified into the following three classes:

- Corrected Error (CE): *if the detected error has also been corrected by specific hardware*. As an example, such hardware can implement an ECC protection that can detect and correct errors in protected data words [27], [28].
- Uncorrected Deferred Error (UDE): *if the detected error can not be corrected but has no immediate impact on the operation of the system*. In this case, the operation can continue, and dealing with the error is eventually deferred to a later point in time when the corrupted data is consumed. For instance, if an uncorrected error is detected in a memory-to-memory transfer, it will be classified as deferred since no unit is going to immediately consume the wrong data. However, such erroneous data is tagged as poisoned and monitored throughout its lifetime. Since the system can track the *poisoned data*, when it is about to be *consumed*, thus altering the state of the system, it will be escalated to *Urgent*;
- Uncorrected Urgent Error (UUE): *if the detected error can not be corrected and requires immediate action from system software*. For example, erroneous data that is going to be consumed by the processor, thus altering the correct state of the system.

Among the detected errors, a severity hierarchy can be defined in which $CE < UDE < UUE$. CEs are the least severe because the affected data has already been corrected. Instead, UUEs are the most severe from the system perspective, since they must be handled immediately to minimize the impact of the possible system failure.

C. STATE-OF-THE-ART HW-SW INTERFACES FOR ERROR LOGGING AND REPORTING

Hardware with error-checking capabilities can quickly locate and even correct some errors, independently from the

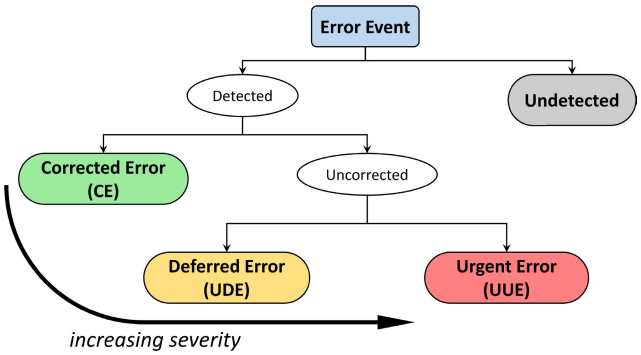

FIGURE 1. Error taxonomy adopted and severity hierarchy of the error classes.

TABLE 1. Major differences between the Intel/AMD approach, on the left, and the Arm approach, on the right. The rows compare design choices for the same characteristic.

Intel/AMD approach	Arm approach
There is only one MCA for each processor core of a system	There is no limit to the number of nodes in a system
The single MCA owns all the error records of that processor core	The error records in the system might be owned by different nodes
The error detected in a unit can be stored only in a specific record of the MCA	The error detected in a unit can be stored in each record owned by a node
The error logging features implemented are fixed and in common for each record of the MCA	Each record might have different error logging features if owned by different nodes
The error reporting signals are at the MCA level, without any distinction on the record that generates the interrupt	The error reporting signals are at the node level, providing some distinction at least from a group of records to another

software. However, to implement some post-error or failure diagnosis, error-checking HW must immediately report to system software that an error has occurred, even if the newly detected error does not lead to a failure (e.g., when errors are corrected directly by the HW). These calls to system software penalize the availability and achievable performance of the system, as it may pause important tasks to handle error events even if they are harmless.

To solve this issue, one of the possible HW add-ons is the *Hardware-Software Interface for Error Logging and Reporting*. It provides valuable information about the latest hardware errors detected by error-checking HW (error logging) and calls up software only when needed (error reporting).

In the next paragraphs, the main characteristics of the Intel/AMD approach [19], [20], [29], [30] and the Arm approach [21] for Error Logging and Reporting will be described. Table 1 reports the main characteristics of the two considered approaches. The main difference between the two approaches that is worth mentioning is the granularity of the HW-SW interface. Indeed, while there can be only one MCA per core (in single-core systems there will be only one), there can be any number of RAS Extensions, as highlighted in the first row of Table 1. In the Arm approach, each RAS Extension module is denoted as a "*node*".

The error logging feature is also implemented with different design strategies. From the first four rows of Table 1, it can be deduced that the RAS Extension has a more flexible structure, since each core of a computing system can instantiate more than one node, and each node can have more than one error record. Also, each node has its own feature and configuration registers that are implementation-defined and independent of the other nodes of the same processor. Different from the Intel/AMD approach, the Arm approach uses all the error records owned by a node without distinguishing between the monitored error-checking units. Therefore, if a node allocates a number of error records equal to or higher than two, an error can be stored in any of them. Instead, the Intel/AMD approach imposes a direct correspondence between the monitored error-checking HW and an error record. This greatly simplifies both the control logic and the routing of the error-related information to the records, at the cost of less flexibility and adaptability to the specific application in which such an architecture is to be implemented.

The last row of Table 1 compares the error reporting feature. While the MCA can send only one or two interrupt signals (depending on the Intel or AMD version), each RAS Extension node can send up to four different interrupt signals. This way, the software interrupt handling routine in the Arm approach can univocally identify which node is reporting the error, thereby eliminating the need to read all the error records. Moreover, the handling routine can be restricted to only those records owned by that node.

III. PROPOSED ERROR LOGGING AND REPORTING ARCHITECTURE (ENGAGE)

Before discussing the architectural details of our proposed solution, hereafter referred to as ENGAGE (Error loggiNG And reportinG architecturE) module, this section provides a concise overview of our approach from a system perspective.

It should be noted that the proposed architecture will be described considering, as monitored units, only ECC-protected memories. However, it could monitor other units (not only memories) with a likewise approach, as long as they have error control logic. For instance, it may also monitor bus controllers, NoC (Network on Chip), temperature and power controllers, and watchdog errors [10].

A. SYSTEM-LEVEL DESCRIPTION

From a system perspective, see Figure 2, the proposed ENGAGE module can be seen as a peripheral that gathers information about hardware errors detected by error-checking hardware through specific input ports (error monitoring). Error information is stored in a set of error records that can be accessed by the CPU via a standard memory-mapped interface (e.g., AXI [31], Avalon-MM [32], etc.). An interrupt signal is generated to provide the error reporting capability. Moreover, system software can configure at run-time which events generate an interrupt request. This capability

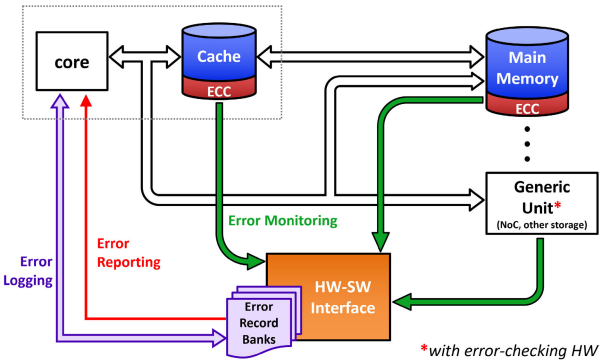

FIGURE 2. Placement of our HW-SW Interface in a generic computing system that includes units monitored by error-checking HW.

significantly increases system availability and overall performance, as the non-relevant conditions can be ignored.

Similar to the Arm approach, our developed architecture provides high flexibility and efficiency in the use of Error Records. Nevertheless, we highlight that our work provides the first public micro-architecture of an HW-SW Interface for Error Logging and Reporting. Even though Intel, AMD, and Arm provide details on their organization of the error records and the logged information format, to the best of the authors' knowledge, no public detail is provided about the architecture and implementation of their proprietary HW-SW Interfaces.

We will refer to the single ENGAGE module within a system as *"RAS hub"*. Also, a system may contain more than one RAS hub. Referring only to ECC-protected memories as "units", the single RAS hub may own one or more error records and may monitor one or more units. Since the number of error records and monitored units are independent of each other, a record can store data from one or more units, and, vice-versa, multiple records can be related to only one unit. We will refer to *Bank* as the group of error records owned by that RAS hub.

A high-level block illustration of the architecture of the developed ENGAGE module is depicted in Figure 3, which is considered to monitor M units protected by error-checking HW. It comprises the following main building blocks:

- **Error Mux**
- **Error Synchronization Interface**
- **Logging Controller**
- **Error Record Bank**
- **Interrupt Request (IRQ) Generator**

Its structure can be divided into two stages. First, *Pre-Process* stage, including Error Mux & Error Synchronization Interface, monitors error-checking HW and pre-processes information related to the detected error. This information will then be channeled to the second stage, *Error Log and Report*, composed of Logging Controller & Error Record Bank. This stage is in charge of collecting the error information and storing it in one of the error records of the Bank within a RAS hub, along with the error timestamp. The Controller also monitors the *poisoned data* throughout their lifetime in the monitored units, thus escalating its severity from UDE to UUE. Lastly, the IRQ Generator triggers an interrupt signal according to the run-time configuration of the Control register of the Bank (see Subsection III-D).

TABLE 2. Error Code implemented, characterized by different formats in which the fields have specific meanings.

`error_code[7:0]`	Type of Source	Description
`0b 000 01 T LL`	TLB	`T` = Transaction Content `LL` = Cache Level
`0b 001 RR T LL`	Memory	Errors in the cache hierarchy `RR` = Transaction Type `T` = Transaction Content `LL` = Cache Level
`0b 010 RR A LL`	Interconnects	General bus errors `RR` = Transaction Type `A` = Memory or I/O `LL` = *unused*

TABLE 3. Values that each field of the error code can assume.

T	LL	RR	A
`0b0`: Instruction `0b1`: Data	`0b00`: *undefined* `0b01`: Cache L1 `0b10`: Cache L2 `0b11`: Higher memory hierarchy	`0b00`: Generic `0b01`: Read `0b10`: Write `0b11`: *undefined*	`0b0`: Memory bus access `0b1`: I/O peripheral bus access

B. ERROR MUX AND ERROR SYNCHRONIZATION INTERFACE

As shown in Figure 3, the *Error Mux* receives the error control signals generated by the error-checking circuitry of the M monitored blocks and generates an `error_message` signal comprising of the:

- `err_m_th` signal, which indicates which one of the M error-checking HW has detected the error;
- `error_sev` signal, which contains the severity of the detected error (either CE, UDE, or UUE);
- `error_code` signal, which identifies the nature of the monitored unit affected by the error, as per the classification reported in Table 2.

The `err_m_th` field of the error message contains a binary number in the range $[0, M-1]$, allowing us to identify the m^{th} unit that has detected the error. The `error_sev` information is encoded with three bits in a one-hot encoding, `0b001` for CE, `0b010` for UDE, and `0b100` for UUE. We propose the `error_code` as reported in Table 2, which is similar to the AMD implementation [33]. Each row of the table represents a different error code format, depending on the source of the error. In detail, TLB (Translation Lookaside Buffer), Memory, and Interconnects. [1] For a given type of source, the error code is further subdivided into different fields described in the third column in the Table 2. The encoding of these fields is listed in Table 3.

The *Error Synchronization Interface* synchronizes and combines the information generated by the Error Mux

[1] Although we only refer to memories in the design description, we also included interconnects for the sake of completeness.

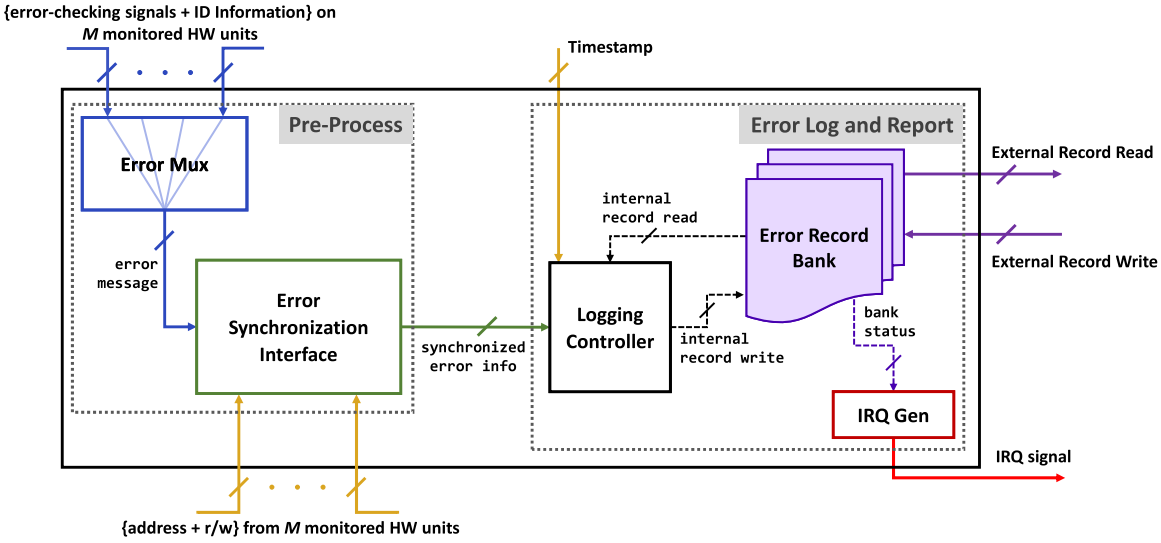

FIGURE 3. Architecture of the developed Error loggiNG And reportinG architecturE (ENGAGE).

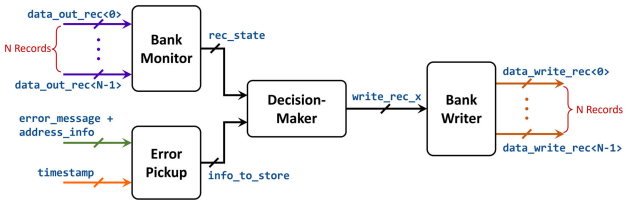

FIGURE 4. Simplified block diagram of the internal architecture of the Logging Controller.

(`error_message` in Figure 3) with the address associated with that specific error (`mem<m> address+r/w` in Figure 3), as they may not be synchronous with each other. Therefore, assume that at time t_1 a read command, along with the address, is sent to the monitored memory. If the ECC circuitry generates its result one clock cycle after t_1, they will arrive at the RAS hub in different cycles. Consequently, the Error Synchronization Interface must delay the `address` data of one clock cycle. Then, the synchronized information is forwarded to the *Logging Controller*.

C. LOGGING CONTROLLER

Once an error has been detected, its relative information must be stored in one of the error records within a Bank, whose number is limited. When there is no available error record, an overwriting algorithm should be activated to determine whether to discard the new error log or overwrite one of the previously stored error logs. The *Logging Controller* makes these decisions to resolve overwriting issues based on the state of the records (its decision criteria will be detailed below). Considering a Bank with N error records, Figure 4 illustrates a simplified view of this module, which can be divided into four main functional components.

The *Error Pickup* retrieves the information on the detected error from the first stage, together with the timestamp (if provided by the system).

TABLE 4. Action taken by the Logging Controller in the selected record, depending on the severity of the new error and the stored one in said record.

	new CE	new UDE	new UUE
-free record-	Write	Write	Write
stored CE	Count/Discard	Overwrite	Overwrite
stored UDE	Discard	Discard	Overwrite
stored UUE	Discard	Discard	Discard

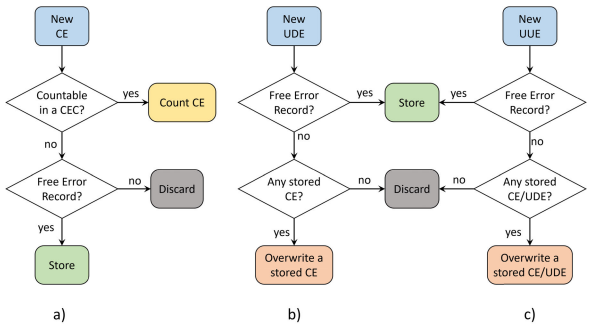

FIGURE 5. State flow of the policy adopted by the Logging Controller to handle new detected errors. It depends on the error severity: a) for CE, b) for UDE, and c) for UUE.

The *Bank Monitor* explores the current state of each record, checking if the record contains valid information and its severity. It outputs a synthesis of the state of the error records, such as the first free error record, or the presence of stored CEs. This information is then provided to the Decision-Maker.

The *Decision-Maker* is the heart of the Logging Controller. It determines which record to select and how to act (see Table 4), based on the state of the records and the new error. Figure 5 summarizes the writing/discarding policy. If one or more free error records are available, any new error-related

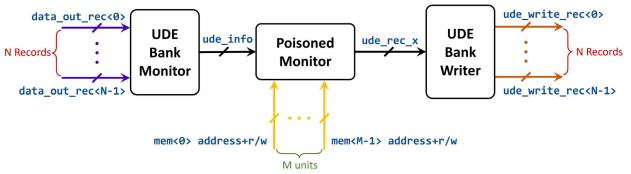

FIGURE 6. Simplified block diagram of the internal architecture of the monitoring process of UDEs carried out by the Logging Controller.

information will be stored in the lowest indexed free record within the bank. However, if there are no free error records and the new error is classified as UUE or UDE, it will overwrite the stored error log with the lowest severity. If all stored error records have higher or equal severity, the new error log will be discarded according to the policy prioritizing older errors. We have considered that an older error record may represent the first error that has led to the system failure, thus causing a series of newer errors. In this case, knowing the root cause of the failure may be more valuable than the errors resulting from it. Finally, if a new CE occurs and all records contain higher severity error logs (i.e., UDE or UUE), its relative information will be discarded. Otherwise, if specific conditions are met, as will be described in Subsection III-D, one of the two counters (CE Counter, CEC) in the record will be incremented to keep track of the number of CE detected.[2]

Lastly, the *Bank Writer* writes on the selected error record within the Bank of the RAS hub.

In parallel, the Logging Controller also monitors *poisoned data*, which is erroneous data whose error information is stored in the records of a bank and classified as UDE. A sketched view of its internal architecture is depicted in Figure 6, assuming a RAS hub that monitors more than one unit (*M* in Figure 6) and with more than one error record (*N* in Figure 6). It determines whether poisoned data are being accessed in read or write mode, by continuously monitoring address and read/write signals from the monitored units. The internal architecture of this parallel task can be split into three main stages.

The *UDE Bank Monitor* searches for logged errors with a UDE severity, among all the records owned by the ENGAGE module, and provides the information related to each poisoned data to be monitored to the next stage.

The *Poisoned Monitor* monitors each read or write transaction related to the units containing the poisoned data. Therefore, whenever poisoned data is going to be consumed or overwritten by new data, the monitor will determine how to update the corresponding error record. The following three cases may occur:

- The memory location containing the *poisoned data is being overwritten*, resulting in the deletion of the poisoned data. Thus, the corresponding error record will be invalidated;

[2]Each error record can handle two counters for CEs, namely CEC. While one counts all CEs detected in the same memory location, the other counts all CEs detected in that memory independently of the specific location.

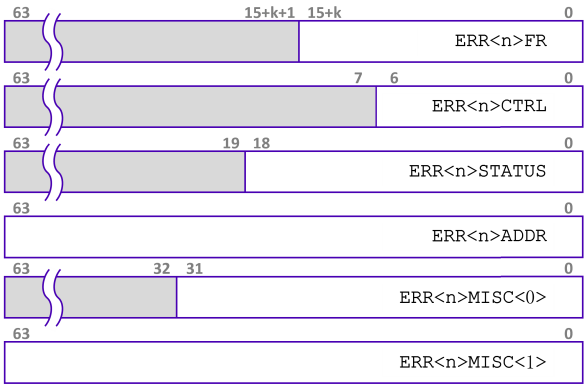

FIGURE 7. Registers composing the n^{th} error record. For each register, the maximum number of bits that might be implemented is delimited.

- The memory location containing the *poisoned data is requested by a consumer unit* (e.g., the processor). In this case, the severity of the corresponding error record will be escalated from UDE to UUE;
- The memory location containing the *poisoned data is requested by a non-consumer unit* (e.g., memory-to-memory transaction). In this case, the corresponding error record remains unchanged.

Lastly, the *UDE Bank Writer* invalidates or upgrades the specific record in the bank, according to the operations described above.

D. ERROR RECORD

The designed ENGAGE module provides the error logging feature to the system through its *Error Records*. An error record comprises an ad hoc set of 64-bit registers to store all necessary error-related information. As described in Section III, each RAS hub can own more than one record. Figure 7 depicts the set of registers within the generic n^{th} *Error Record* and the maximum number of available bits within each register (grey fields will never be used, independently of the configuration of the logging capability). A description of the characteristics of each register, from the user's point of view, and the information stored in it, is given below.

Feature Register (`ERR<n>FR`) is a Read-Only register that contains information concerning the error logging and reporting features that are implemented in that RAS hub; consequently, its content is implementation dependent. It specifies the severity of the logged error (CE, UDE, UUE); whether the counters for CEs are implemented or not and, if so, their width; if the timestamp is stored; the severity of errors that may raise an interrupt.

Control Register (`ERR<n>CTRL`) is a Read-Write register, which consists of several bits to enable the conditions under which an interrupt is generated (Error Reporting) by that RAS hub. At run-time, a user can enable an interrupt to be generated when a UDE is stored. Therefore, this register

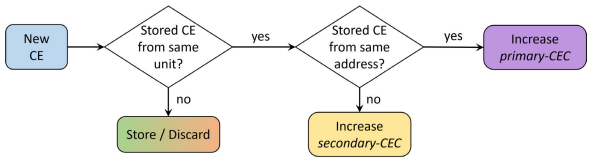

FIGURE 8. State flow of the conditions to be satisfied for incrementing one of the two CECs (CE Counters).

allows for the control of the error reporting feature at run-time, depending on the specific application requirements.

Status Register (ERR<n>STATUS) is a Read-Write register that stores all the major information about the error logged in the nth record. It contains the error message, the overflow bit of the CE Counter, and a bit representing if the error record has been overwritten. In addition, it contains a bit to validate the address contained in the Address Register and another field to validate the content of the Miscellaneous Registers.

Address Register (ERR<n>ADDR) is a Read-Only register that stores the address associated with the erroneous data, as long as an address can be associated with the detected error. For example, an error in the watchdog timer can be logged in an error record, yet it cannot be associated with an address.

Two Miscellaneous Registers (ERR<n>MISC<0> and ERR<n>MISC<1>) that store additional information about the logged error. The ERR<n>MISC<1> is a Read-Only register that stores the timestamp, i.e. date and time at which the error is stored in the nth record. Instead, the ERR<n>MISC<0> is a Read-Write register that keeps track of the number of CEs detected in a specific monitored unit through the *CE Counters* (CECs), the primary-CEC and the secondary-CEC. Figure 8 shows the criteria for the use of the CECs. The first prerequisite is to have previously logged a CE in one of the Error Records; then, when a new CE is detected, the *primary-CEC* is incremented if both the monitored unit and the address of the new CE match the ones of the stored CE. Instead, the *secondary-CEC* is incremented if the monitored unit of the new CE matches the one of the stored CE but the related address is different. This feature can be useful for statistical purposes and for determining whether the faults are localized or distributed throughout the unit.

As will be explained in Section IV, the error record Bank requires a big portion of the total area resources of the entire ENGAGE module. In the assessment of the area overhead of the error logging features, for system parameters the notation in Table 5 is adopted. Particularly, area overhead will be evaluated in terms of the number of 1-bit registers (REGs) needed only by the error records (REG_{TOT}). Firstly, we should consider that a system will never have more RAS hubs than monitored units (it would be a waste of resources) and that each RAS hub shall have at least one record; therefore, it is $N_{REC} \geq N_{HUB}$. Therefore, the total number of REGs of the error record can be estimated as:

$$REG_{TOT} = (b_{FR} + b_{CTRL})N_{HUB} + (b_{STAT} + b_{ADDR} + b_{CEC} + b_{TS})N_{REC}. \quad (1)$$

TABLE 5. Indexes that are required to define the total number of REGs enveloped by all the error records in a system, with their values and definitions.

Index	Value	Meaning
N_{HW}	—	Total number of HW units monitored in the system
N_{HUB}	$N_{HUB} \leq N_{HW}$	Total number of RAS hubs implemented in the system
N_{REC}	$N_{REC} \geq N_{HUB}$	Total number of error records implemented
b_{FR}	0	Number of REGs required by the *Feature Register*
b_{CTRL}	7	Number of REGs required by the *Control Register*
b_{STAT}	17	Number of REGs required by the *Status Register*
b_{ADDR}	[0 - 64]	Number of REGs required by the *Address Register*
b_{CEC}	[0, 12, 16] × Counter	Number of REGs required by the *Misc<0> Register*
b_{TS}	[0 - 64]	Number of REGs required by the *Misc<1> Register*

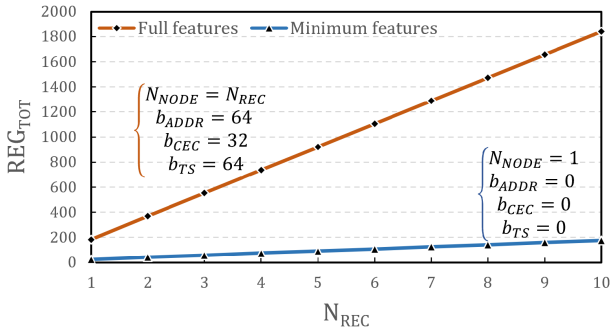

FIGURE 9. Example of progression of the REG_{TOT} depending on the configuration of the Error Logging features implemented in the system.

Since the Feature and Control Registers are only implemented in the first record of the RAS hub's Bank, they will be hardwired to logical zero from the second record onwards. Instead, each record within the Bank includes all the remaining registers. As reported in the fourth row of Table 5, the Feature register does not require actual registers since it contains only implementation-defined values that are usually synthesized as hard-wired signals; no REGs are needed for this register. Also, b_{ADDR}, b_{CEC}, and b_{TS} vary depending on the specific configuration of the error logging feature of a RAS hub (see last three rows of Table 5).

Once the configuration of the error logging feature has been set (b_{ADDR}, b_{CEC}, and b_{TS}), the remaining variables are N_{HUB} and N_{REC}. In the case of $N_{HUB} > 1$, we considered that all RAS hubs implemented in the system have the same Bank configuration. For example, Figure 9 shows the increase of REG_{TOT} as a function of the number of records in the system (N_{REC}) in the two extreme cases: *Full error logging features*, assuming the maximum values for b_{ADDR}, b_{CEC}, b_{TS}, and each RAS hub with only one error record; and *Minimum error logging features*, assuming to minimize the bits of information in the error record to only the Status register ($b_{ADDR} = b_{CEC} = b_{TS} = 0$) and including all error records within a single RAS hub, single Bank. In an actual implementation of the proposed ENGAGE module, the number of REGs used will probably lie between these two configurations since, for instance, one system may have 16-bit addresses, while another can have 64-bit ones. Furthermore, the resource overhead is strongly

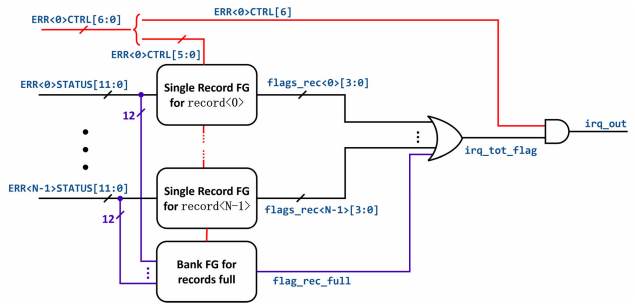

FIGURE 10. Simplified block diagram of the internal architecture of the Interrupt Request (IRQ) Generator in case of multiple records owned by the RAS hub.

correlated with the number of error records implemented in the overall system, which depends on the number of units to be monitored and the number of error logs to be stored simultaneously.

E. IRQ GENERATOR

The IRQ Generator implements the Error Reporting feature through an interrupt signal connected to an Interrupt Controller.

Assuming a total of N implemented records, the IRQ Generator module includes a *Single Record FG* (Flag-Generator) sub-module for each record, as depicted in Figure 10, which generates a 4-bit flag. For the generic `record<n>`, this sub-module can trigger a flag depending on the `ERR<n>STATUS` and on the configuration of the `ERR<0>CTRL`:

- `flags_rec<n>[0]` – UUE stored in the `record<n>`;
- `flags_rec<n>[1]` – DUE stored in the `record<n>`;
- `flags_rec<n>[2]` – Overflow of one of the CE Counters (primary- or secondary-CEC). If this error logging feature is not implemented, a CE stored in `record<n>` raises this flag;
- `flags_rec<n>[3]` – UUE or UDE has been overwritten or discarded.

In addition, all the Status Registers of the N records within the Bank are simultaneously monitored by the *Bank FG* sub-module, which will raise the flag `flag_rec_full` when there are no free error records within that RAS hub.

Every flag is channeled into a single signal through a logic-OR stage, connected to the final logic-AND that provides the global enable for the IRQ signal, which is controlled at run-time through a bit in the Control Register. This choice does not preclude refining the granularity of the error reporting feature, since the IRQ Generator can be easily modified to output multiple interrupt signals, which will then be fed into different inputs of the processor interrupt controller. For example, each flag related to a stored UUE (`flags_rec<n>[0]`) can be singled out and sent to the interrupt controller as a different signal.

The flags described above can be configured at the synthesis level. Indeed, their generation can be: *never enabled*, *always enabled*, or enabled at *run-time* via a specific bit in the Control Register.

IV. SYNTHESIS RESULTS AND FPGA DEMONSTRATOR

For evaluation purposes, the proposed ENGAGE module has been equipped with an AXI4 Slave memory-mapped interface to access its error record registers. The overall system has been designed in SystemVerilog language and synthesized on Standard-Cell and FPGA technologies. Also, we have implemented a System-on-Chip (SoC) on an FPGA board to validate the Error Logging and Reporting features that are provided by the ENGAGE module in a real system. For both synthesis and implementation, the ENGAGE module was configured with the following error logging and reporting features:

- $N_{HUB} = 1$;
- Two monitored units ($N_{HW} = 2$);
- Two error records implemented ($N_{REC} = 2$);
- Each record implements both CE Counters of 8-bit ($b_{CEC} = 16$);
- Each record includes the timestamp information, considering a 32-bit value ($b_{TS} = 32$);
- AXI4 address bus is 32-bit wide ($b_{ADDR} = 32$);

It is worth noting that different configurations of the ENGAGE module, which use different amounts of hardware resources, can be realized according to the application requirements. From (1) we can estimate the number of single-bit registers required for the error records in this implementation, which is:

$$REG_{TOT} = 7 \cdot 1 + (17 + 32 + 16 + 32) \cdot 2 = 201. \quad (2)$$

A. SYNTHESIS RESULTS ON STANDARD-CELL

The developed module has been synthesized using the Synopsys 2019.12 Design Compiler tool on two Standard-Cell technologies:

- The NanGate Open-Cell Library, designed with the 45nm FreePDK kit [34]. The synthesis has been performed with a voltage supply of 1.10 V, a working temperature of 25^{o}C, and a typical process. Moreover, the *5K_hvratio_1_1* wire load model has been used;
- The Artisan 7nm TSMC standard-cell technology [35]. In this case, the synthesis has been performed with a voltage supply of 0.90 V, a working temperature of 125^{o}C, and a slow process. Also, the *Zero* wire load model has been used.

Table 6 reports the synthesis results for the ENGAGE module in terms of maximum operating frequency, total estimated power (including both switching and leakage power), and resource utilization of the sub-modules of the ENGAGE module, according to the hierarchy of the synthesized design. The maximum operating frequency has been obtained by performing multiple syntheses with a frequency sweep of the clock signal, reaching 1.20 GHz with the 45nm technology and 5.50 GHz with the 7nm technology. The value obtained for the total power consumption has been estimated by

TABLE 6. Maximum operating frequency, total estimated power, and resource utilization for both *45nm* and *7nm* Standard-Cell technologies.

	45nm technology		*7nm technology*	
Maximum Frequency	1.20 GHz		5.50 GHz	
Power (leak + switch)	5.31 mW		8.23 mW	
Hierarchical Cell	**Comb. gate [kGE]**	**Seq. gate [kGE]**	**Comb. gate [kGE]**	**Seq. gate [kGE]**
AXI4 Slave Interface	0.83	1.05	0.84	0.84
ENGAGE module	2.04	2.54	2.31	2.09
Error Mux	0.05	0.09	0.05	0.07
Error Synch. Intf.	0.33	1.05	0.40	0.87
Err. Rec. Bank	0.68	1.34	0.80	1.10
record<0>	0.35	0.69	0.38	0.56
record<1>	0.33	0.65	0.41	0.54
Logging Controller	0.88	0.06	0.94	0.05
IRQ Generator	0.03	0.01	0.02	0.01
Total Design	6.46		6.09	

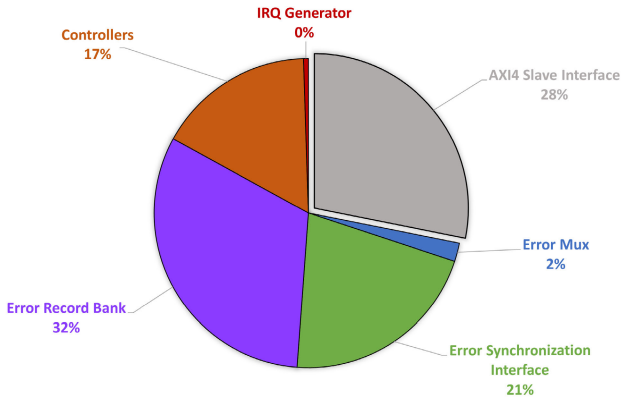

FIGURE 11. Relative resource utilization of the ENGAGE module from the synthesis results on the 7nm Standard-Cell technology.

the synthesis tool. By looking at the resource utilization results, expressed in Gate Equivalents (GE), they have been distinguished between *Combinational* and *Sequential* logic (*Comb. gate* and *Seq. gate*, respectively, in Table 6). The differences between the two Standard-Cell technologies are minimal for both categories since they depend mainly on how the respective libraries are designed. Indeed, both synthesis results report a total resource utilization of about 6.2 kGE.

Figure 11 shows a pie chart with the relative resource utilization of each sub-module. Even though the resource overhead introduced by the controller cannot be reduced, the one due to the error records can be adjusted according to the implementation requirements. Indeed, we can modify the number of records or even the amount of information stored in each error record (such as address, timestamp, and CE counters). Note that fewer records imply a higher probability that a new error is detected in the system while the records still contain information about previous errors, increasing the overwriting frequency.

To estimate the area overhead as a function of the number of error records inside a bank, and to confirm the (2), we have also performed a synthesis on the 7nm technology of the ENGAGE module with $N_{REC} = 20$. Table 7 presents the results of this analysis and compares the resource usage with

TABLE 7. Impact of error records on Resource Utilization, synthesizing on 7nm technology, evaluating N_{REC} values of 2 and 20.

	N_{REC}=2	N_{REC}=20	Resource Trend N_{REC}=20 / N_{REC}=2
Estimated REG_{TOT}	201	1947	9.70
Error Record Seq. logic [kGE]	1.10	10.50	9.54
Remaining Seq. logic [kGE]	1.83	2.05	1.12
Comb. logic [kGE]	3.16	18.22	5.77
Total logic [kGE]	6.09	29.99	4.92

$N_{REC} = 2$ and $N_{REC} = 20$: the first row shows the total number of registers REG_{TOT} calculated with (1), the second row highlights the sequential logic utilized by only the Error Record Bank module, the third includes the remaining sequential logic consumed by the ENGAGE module, and the last row shows the combinational area of the whole ENGAGE module. As can be seen in Table 7, the resource trend of the sequential logic of the Error Record module is in line with the one estimated using (1); the remaining sequential logic area (total sequential area - error record sequential area) is almost invariant. Instead, the total combinational logic area of the implementation with $N_{REC} = 20$ is around 5.77 times higher than the implementation with $N_{REC} = 2$. This is due to the circuitry needed to convey the error record registers to the AXI4 data bus, plus the more complex logic of the Controllers that monitor and update the records. It should be noted that the maximum operating frequency for the 7nm synthesis with $N_{REC} = 20$ decreases to 3.8 GHz; however, the optimization of this metric is out of the scope of this analysis, whose main goal is to provide an approximated evaluation of the area increase depending on the number of error records.

B. SYNTHESIS RESULTS ON FPGA

We synthesized and implemented, the ENGAGE module on the Xilinx Zynq Ultrascale+ MPSoC, featured on the Xilinx ZCU104 board, using Vivado 2020.2 Design Suite. Table 8 outlines the utilization report following the hierarchy of the implemented system. The first row reports the resource consumption of the CV32E40P RISC-V core, a 32-bit single-core processor that has been integrated into the test SoC that will be presented in Subsection IV-C; this indicates that the ENGAGE module consumes notably fewer resources compared to the considered core (around 10% of total resources of the CV32E40P core). It should be noted that the latter is a small area and low-performance RISC-V microcontroller. For instance, in [36] the RISC-V application processor CVA6 has been synthesized on the same FPGA family, obtaining 31170 LUTs and 19076 FFs, plus 37 BRAMs (Block RAMs) for the cache memory. Therefore, if the ENGAGE module was implemented in an application system, it would require less than 1% of the total resources of the CVA6 core.

The results in Table 8 refer to post-implementation in the target FPGA at 100 MHz; however, we found that the

TABLE 8. Absolute resource utilization of the test SoC implemented on the ZCU104 board, XCZU7EV FPGA.

Hierarchical Cell	*LUTs*	*FFs*	*CARRY8s*	*BRAMs*
CV32E40P RISC-V core	4479	2193	79	0
ENGAGE module & AXI4 Slave Interface	412	388	8	0
AXI4 Slave Interface	154	138	0	0
ENGAGE module	263	250	8	0
Error Mux	15	11	0	0
Error Synch. Interface	82	70	0	0
Error Record Bank	132	159	0	0
Logging Controller	34	9	8	0
IRQ Generator	0	1	0	0

maximum frequency of the ENGAGE module is about 320 MHz. From Table 8 we can derive that the designed module does not use BRAM resources, while CARRY8 primitives have been used to synthesize the *Logging Controller* of the ENGAGE module, which includes large comparators.

The *Error Synchronization Interface*, along with the *Error Records*, require the most resources among all the sub-modules of the ENGAGE module. However, the overhead introduced by these modules is largely configurable according to system requirements. From Table 8, we can observe that the error records require 159 FFs, in contrast to the 201 single-bit registers estimated with (1). This difference can be attributed to the optimization made by Vivado when implementing the design. For example, the ENGAGE module was configured to have $b_{ADDR} = 32$, even if the monitored modules in the test SoC have 13-bit addresses (8 KB memories, as later described). Indeed, Vivado does not implement these unused FFs.

C. VERIFICATION AND FPGA SOC DEMONSTRATOR

The proposed ENGAGE module was first verified in behavioral RTL (Register Transfer Level) simulations in Questa Sim 2022.2 simulation environment, then implemented and prototyped to validate its functionality.

An extract of the RTL verification is depicted in Figure 12, in which the timing of the internal signals can be evaluated, and the fundamental signals are highlighted. Also, the color coding of the depicted signals is the same as in Figure 3. Additionally, we have configured the verification environment to contain one RAS hub (our module) with only one Error Record; it monitors two memories with SEC-DED protection (i.e., `mem[0]` as L1 cache, and `mem[1]` as L2 cache) with no delay between address and ECC output. Moreover, the "fifo" group of signals refers to the output of a FIFO (First-In First-Out) buffer inside the *Error Synchronization Interface*, which interfaces with the *Logging Controller*. Finally, the timestamp signal is also included. By looking at the data path from the error detection to the rise of the respective interrupt (from the first to sixth clock cycle in Figure 12, respectively), the following steps can be appreciated, in which:

1° A double-bit error is detected performing a read operation at the address `0x08C0` of the second memory (`mem[1]` that emulates an L2 cache). Since this memory acts as a high-level memory, it is only supposed to perform mem-to-mem transactions, so it would never directly provide data to the processor; therefore, the *Error Mux* will classify it as UDE.

2° At the second clock cycle there is no signal variation because the output of the *Error Mux* and the address-related information are being stored in the FIFO buffer, which is internal to the *Error Synchronization Interface*.

3° The *Logging Controller* reads the error-related information and determines the action to take. In this case, since the error record is still free, the controller will write the error information in the record (see Table 4).

4° The controller performs the chosen action, thus writing in the `ERR<0>STATUS`, `ERR<0>ADDR`, and `ERR<0>MISC<1>` (containing the timestamp) registers.

5° The aforementioned registers of the first (and only) Error Record are updated with the error-related data.

6° Since the interrupt generation has been previously enabled in the `ERR<0>CTRL` register, the *IRQ Generator* raises the interrupt signal.

The number of entries of the FIFO buffer must be determined depending on the probability of errors occurring in consecutive clock cycles.

Finally, we implemented a prototype SoC on the FPGA board Xilinx Zynq ZCU104, which features the Zynq UltraScale+ MPSoC, to validate our ENGAGE module in a real system. Figure 13 shows the structure of the developed test SoC. All blocks communicate via memory-mapped accesses, exploiting the AXI4 communication protocol.

The designed SoC includes the following modules:

1) *RISC-V CV32E40P core [37]:* 4-stage in-order 32-bit RISC-V processor implementing RV32IMFC Instruction Set Architecture (ISA) [38]. It has two master AXI4 interfaces (data and instruction) that allow for access to all the available resources in the test SoC. Its interrupt controller has also been connected to the ENGAGE module, to receive the interrupt generated by the *IRQ Generator* (red blocks in the figure);
2) *ENGAGE module:* It is configured to own two records with all error logging features implemented and to have full error reporting capability, by properly setting the `ERR<0>CTRL` register. Furthermore, we included an *Error Injection* register that the processor can access to inject errors at run-time in the 2^o and 3^o *BRAMs* to validate the proposed architecture;
3) *Main Block RAM (BRAM):* It is provided as Xilinx IP (Intellectual Property), and has been used as the Main Memory of the prototyped system, acting as both instruction and data memory. Its size is 128 KB;
4) *2^o and 3^o BRAMs:* Similar to the Main BRAM, they are provided as Xilinx IP. These are two small auxiliary BRAMs (one of 8 KB and the other of 4 KB), which are configured to implement a *SEC-DED* type ECC code. This IP embeds dedicated error injection input

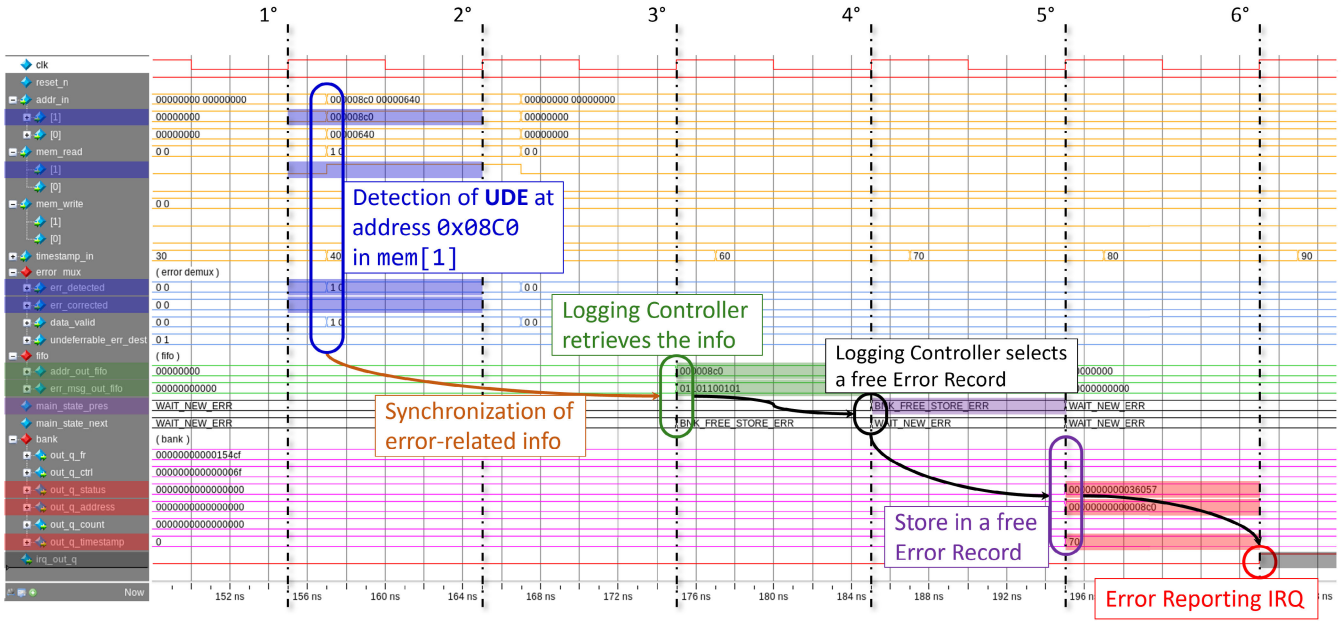

FIGURE 12. Extract of behavioral RTL simulation of the ENGAGE module. Detection and storage of a UDE, and reporting through interrupt, is shown.

ports to inject at run-time single and double errors on the ECC-protected memory words (32-bit data word protected by 7-bit additional SEC-DED code). This feature can be activated by writing in the *Error Injection* register included in the ENGAGE module;

5) *Central DMA (CDMA):* It is provided as Xilinx IP, and has two AXI4 interfaces, a master and a slave. It can transfer large groups of bytes from one memory-mapped device to another, and has been used to support memory-to-memory transfers of large blocks of data, emulating data transfers between adjacent cache layers;
6) *UART:* It is provided as Xilinx IP, and is required to display debug information via the terminal;
7) *JTAG:* It is provided as Xilinx IP, and is used to load the program binary into the Main BRAM at run time.

Since the test SoC is equipped with a 32-bit processor, the data bus width is 32-bit. This implies that each 64-bit register of an error record will be addressed and accessed as two separate 32-bit registers. Moreover, it is worth noting that our module could be implemented into 64-bit systems by simply modifying its AXI4 slave interface to support 64-bit reads and writes of the error records.

In the proposed SoC, the ENGAGE module monitors two units with error-checking HW, namely 2^o and 3^o BRAMs. The CDMA has been exploited to trigger memory-to-memory data transfers between the two ECC-protected BRAMs (2^o and 3^o BRAMs), thus emulating transfers between two adjacent cache levels (e.g., L1 and L2 caches). This configuration can emulate the following error scenarios:

- Single-bit error injections and their correction through the ECC SEC, thus causing errors with CE severity.

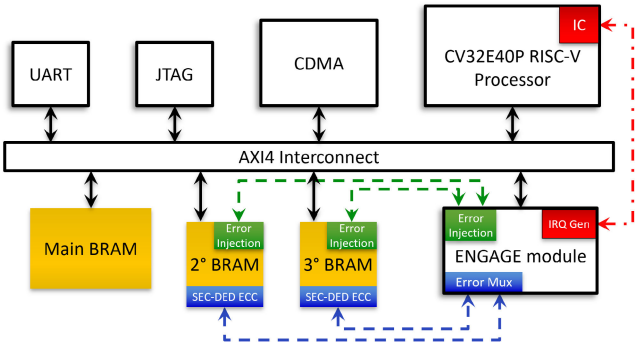

FIGURE 13. Architecture of the test SoC implemented. The colored dashed lines represent the main interconnections, independent of the AXI4 interface, to and from the ENGAGE module.

- Double-bit error injections and their detection through the ECC DED in memory-to-processor transactions that trigger errors with UUE severity; they are classified as UUE since data will be consumed immediately by the processor, potentially inducing a system failure.
- Double-bit error injections and their detection through the ECC DED in memory-to-memory transactions that trigger errors with UDE severity; they are classified as UDE errors since data may not be consumed immediately. The erroneous data will be tagged as poisoned.

To test the features of the proposed ENGAGE module, several software routines were developed in C language and run on the RISC-V processor to stimulate error conditions that exploit the error injection ports of the BRAMs. First of all, we tested the generation of errors with severity CE, UDE, and UUE and checked the correctness of the related error

information stored in the error record registers. Afterward, we tested the developed SoC for several corner cases:

- Overwrite of a stored CE by a newly detected error with severity UDE or UUE;
- Overwrite of a stored UDE by a newly detected error with severity UUE;
- Discard of a new CE, UDE, or UUE due to higher severity errors already stored (or older ones with the same severity);
- CE counting and overflow of the primary and secondary CEC;
- Escalation of a stored error record from UDE to UUE severity (consumption of poisoned data);
- Invalidation of a stored error record of UDE severity (overwrite of the poisoned data);
- Enable interrupt generation via the Control register.

By performing all those tests in the SoC, we were able to assess the ability of our module to provide all the Error Logging and Reporting capabilities to the system. We recall that, even if we have considered only memories as HW units to monitor, the ENGAGE module may monitor every HW unit that embeds error control logic.

V. CONCLUSION

This work presented the design and implementation of a HW-SW Interface for Error Logging and Reporting called ENGAGE (Error loggiNG And reportinG architecturE), which can be adopted by 32- and 64-bit systems to improve system RAS independently of their specific ISA. An Error Record interface is also proposed, which stores the error-related information. The developed ENGAGE module is highly flexible and configurable, allowing designers to implement only the desired features, thus optimizing the hardware overhead introduced. Although only ECC-protected memories were considered as monitored HW units, the proposed architecture also allows other system modules to be monitored, including TLBs and interconnections. Indeed, as long as a system module implements error control and/or correcting logic, it can be monitored by our HW-SW Interface. Our ENGAGE module has been synthesized on both a Xilinx UltraScale+ FPGA device and two Standard-Cell technologies, 45nm and 7nm. The synthesis results have shown that the complexity of the module heavily depends on parameters such as the number of records implemented and the amount of information stored in them. Also, the module is not a bottleneck for the system operating frequency. Therefore, depending on the logging and reporting features to be implemented in the system, the resource overhead of our module can be adjusted accordingly. We then developed a test SoC to validate the proposed ENGAGE module, featuring also the RISC-V CV32E40P core, which has been implemented on a ZCU104 board.

To the best of the authors' knowledge, this is the first public micro-architectural proposal of a HW-SW Interface for Error Logging and Reporting, since proprietary solutions do not share such detail. As it is also ISA-independent, it can monitor any HW unit within a system that has error control logic. Therefore, it may be taken as a reference to provide such features in any computing system.

REFERENCES

[1] R. Canal, C. Hernandez, R. Tornero, A. Cilardo, G. Massari, F. Reghenzani, W. Fornaciari, M. Zapater, D. Atienza, A. Oleksiak, W. PiĄtek, and J. Abella, "Predictive reliability and fault management in exascale systems: State of the art and perspectives," *ACM Comput. Surv.*, vol. 53, no. 5, pp. 1–32, Sep. 2021.

[2] J. Brandt, F. Chen, V. De Sapio, A. Gentile, J. Mayo, P. Pèbay, D. Roe, D. Thompson, and M. Wong, "Quantifying effectiveness of failure prediction and response in HPC systems: Methodology and example," in *Proc. Int. Conf. Dependable Syst. Netw. Workshops (DSN-W)*, Jun. 2010, pp. 2–7.

[3] F. Cappello, H. Casanova, and Y. Robert, "Checkpointing vs. migration for post-petascale supercomputers," in *Proc. 39th Int. Conf. Parallel Process.*, Sep. 2010, pp. 168–177.

[4] O. Khan and S. Kundu, "Hardware/Software codesign architecture for online testing in chip multiprocessors," *IEEE Trans. Dependable Secure Comput.*, vol. 8, no. 5, pp. 714–727, Sep. 2011.

[5] A. Chatzidimitriou, G. Papadimitriou, and D. Gizopoulos, "HealthLog monitor: Errors, symptoms and reactions consolidated," *IEEE Trans. Device Mater. Rel.*, vol. 19, no. 1, pp. 46–54, Mar. 2019.

[6] Z. Zheng, L. Strigini, N. Antunes, and K. Trivedi, "Editorial: Software reliability and dependability engineering," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 4, pp. 2674–2676, Jul. 2023.

[7] I. Kaitovic and M. Malek, "Impact of failure prediction on availability: Modeling and comparative analysis of predictive and reactive methods," *IEEE Trans. Dependable Secure Comput.*, vol. 17, no. 3, pp. 493–505, May 2020.

[8] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "Fault injection analytics: A novel approach to discover failure modes in cloud-computing systems," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 3, pp. 1476–1491, May 2022.

[9] V. Sridharan, H. Asadi, M. B. Tahoori, and D. Kaeli, "Reducing data cache susceptibility to soft errors," *IEEE Trans. Dependable Secure Comput.*, vol. 3, no. 4, pp. 353–364, Dec. 2006.

[10] J. K. Park, D. Kim, and J. T. Kim, "Efficient error-resilient bus coding method using bit-basis orthogonal integrative multiplexing," *IEEE Trans. Emerg. Topics Comput.*, vol. 10, no. 2, pp. 1178–1191, Apr. 2022.

[11] J. Li, P. Reviriego, L. Xiao, and H. Wu, "Protecting memories against soft errors: The case for customizable error correction codes," *IEEE Trans. Emerg. Topics Comput.*, vol. 9, no. 2, pp. 651–663, Apr. 2021.

[12] Y. Sazeides, A. Bramnik, R. Gabor, and R. Canal, "A real-time error detection (RTD) architecture and its use for reliability and post-silicon validation for F/F based memory arrays," *IEEE Trans. Emerg. Topics Comput.*, vol. 10, no. 2, pp. 524–536, Apr. 2022.

[13] F. Cosimi, F. Tronci, S. Saponara, and P. Gai, "Analysis, hardware specification and design of a programmable performance monitoring unit (PPMU) for RISC-V ECUs," in *Proc. IEEE Int. Conf. Smart Comput. (SMARTCOMP)*, Jun. 2022, pp. 213–218.

[14] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory errors in modern systems: The good, the bad, and the ugly," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 297–310, 2015.

[15] S.-H. Lim, R. G. Miller, and S. S. Vazhkudai, "Understanding the interplay between hardware errors and user job characteristics on the Titan supercomputer," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2020, pp. 180–190.

[16] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2014, pp. 610–621.

[17] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Trans. Electron Devices*, vol. ED-26, no. 1, pp. 2–9, Jan. 1979.

[18] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2014, pp. 467–478.

[19] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corp., Santa Clara, CA, USA, 2022.
[20] *AMD64 Architecture Programmer's Manual: System Programming*, Adv. Micro Devices, Inc., Santa Clara, CA, USA, 2021.
[21] *Arm Architecture Reference Manual Supplement. Reliability, Availability, and Serviceability (RAS), for Armv8-A*, Arm Holdings Plc, Cambridge, U.K., 2021.
[22] *Intel Xeon Processor E7 Family: Reliability, Availability, and Serviceability—White Paper*, Intel Corp., Santa Clara, CA, USA, 2011.
[23] *AMD EPYC Brings New RAS Capability—White Paper*, Moor Insights Strategy, Austin, TX, USA, 2017.
[24] *Arm Reliability, Availability, and Serviceability (RAS) Specification Armv8, for the Armv8-A Architecture Profile—White Paper*, Arm Holdings plc, Cambridge, U.K., 2022.
[25] D. Rossi, N. Canino, S. Di Matteo, S. Saponara, and V. Tenentes, "Design and evaluation of a peripheral for integrity checking to improve RAS in RISC-V architectures," in *Proc. 8th South-East Eur. Design Automat., Comput. Eng., Comput. Netw. Social Media Conf. (SEEDA-CECNSM)*, Nov. 2023, pp. 1–6.
[26] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.
[27] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, Apr. 1950.
[28] M. Y. Hsiao, "A class of optimal minimum odd-weight-column SEC-DED codes," *IBM J. Res. Develop.*, vol. 14, no. 4, pp. 395–401, Jul. 1970.
[29] X Project. *Xen's Machine Check Architecture Implementation for Intel Processors*. Accessed: Feb. 27, 2024. [Online]. Available: https://lists.xenproject.org/archives/html/xen-devel/2012-07/pdfuVm2TRqnjV.pdf
[30] A. Kleen. *Machine Check Handling on Linux*. Accessed: Feb. 27, 2024. [Online]. Available: https://www.halobates.de/mce.pdf
[31] *AXI (Advanced EXtensible Interface) Memory-mapped Interface*. Accessed: Jan. 15, 2024. [Online]. Available: https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification
[32] *Avalon Memory-Mapped Interface*. Accessed: Jan. 15, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683091/20-1/introduction-to-the-interface-specifications.html
[33] *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 16h Models 30h-3Fh Processors*, Adv. Micro Devices, Inc., Santa Clara, CA, USA, 2016.
[34] *FreePDK45 and the Nangate Open-Cell Library*. Accessed: Feb. 5, 2024. [Online]. Available: https://mflowgen.readthedocs.io/en/latest/stdlib-freepdk45.html
[35] *7 nm Cell Library*. Accessed: Feb. 5, 2024. [Online]. Available: https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l7nm
[36] P. Nannipieri, S. Di Matteo, L. Zulberti, F. Albicocchi, S. Saponara, and L. Fanucci, "A RISC-V post quantum cryptography instruction set extension for number theoretic transform to speed-up CRYSTALS algorithms," *IEEE Access*, vol. 9, pp. 150798–150808, 2021.
[37] OpenHW-Group. *Cv23e40p RISC-V Processor Core*. Accessed: Jan. 8, 2024. [Online]. Available: https://github.com/openhwgroup/cv32e40p
[38] *RISC-V International*. Accessed: Feb. 27, 2024. [Online]. Available: https://riscv.org/

NICASIO CANINO received the M.Sc. degree (cum laude) in electronic engineering from the University of Pisa, where he is currently pursuing the Ph.D. degree in information engineering. His research interests include security in automotive and IoT domains, the design of resilient computing systems, and the RISC-V community.

STEFANO DI MATTEO received the M.Sc. degree in electronic engineering and the Ph.D. degree (cum laude) in information engineering from the University of Pisa. He leads a Chaire in secure hardware implementation of post-quantum cryptography as a Research Engineer with CEA Leti/List, Grenoble. His research interests include hardware security, VLSI design, and embedded systems for cybersecurity and cryptography in different application fields.

DANIELE ROSSI (Senior Member, IEEE) received the M.Sc. degree in electronic engineering and the Ph.D. degree in electronics and computer engineering from the University of Bologna. He is currently an Associate Professor in electronics with the University of Pisa. He has coauthored over 100 articles. His research interests include energy-efficient and reliable electronic design and hardware security. He served as an associate editor for several journals and a TPC member for international conferences.

SERGIO SAPONARA (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in electronic engineering from the University of Pisa. He was a Marie Curie Research Fellow with IMEC. He is currently a Full Professor in electronics with the University of Pisa. He is also the Director of the I-CAS Laboratory, Summer School Enabling Technologies for IoT, and the leader of many funded projects by the EU and companies. He has coauthored about 400 scientific publications and 18 patents.

Open Access funding provided by 'Università di Pisa' within the CRUI CARE Agreement