International Journal of Safety and Security Engineering
Vol. 10, No. 4, August, 2020, pp. 467-474
Journal homepage: http://iieta.org/journals/ijsse

An Enhanced Algorithm for Memory Systematic Faults Detection in Multicore Architectures Suitable for Mixed-Critical Automotive Applications

Abdullah El-Bayoumi[1,2]

[1] Group of Electronic Expertise and Development Services (GEEDS), Valeo, Giza 12577, Egypt
[2] Electronics and Electrical Communications Engineering Department, Cairo University, Giza 12613, Egypt

Corresponding Author Email: abdullah.elbayoumi@pg.cu.edu.eg

https://doi.org/10.18280/ijsse.100405

Received: 8 May 2020
Accepted: 10 July 2020

Keywords:
functional safety, real-time operating system, multicore processor, memory protection, freedom from memory interference, fault-tolerance, safety mechanism, reliability

ABSTRACT

Evolution to multicore architectures has been trending, as a result of the shift towards the nanoscale semiconductor industry. This lets multicore processor challenges arise in a way limiting their features. In present, mixed safety-critical systems in the automotive industry utilize multicore processors. These systems include a software code may reach millions of line-of-code needed for an emerging autonomy level. This implies more design complexity. Complying with ISO 26262 safety standard increases the complexity. This work proposes new safety mechanisms that overcome memory interferences that affect an Automotive Safety Integrity Level (ASIL) multicore architecture. New optimized double inverse redundant storage algorithms are presented to mitigate systematic memory data faults. Other safety mechanisms are introduced to overcome random faults in a memory. The proposed safety mechanisms have been investigated and evaluated for Aurix Tri-core and Renesas RH850 targets with lots of suggestions to have a fully compliant architecture with principles and methods of ISO 26262. Monte Carlo analysis has been performed for the proposed safety mechanisms diagnostic coverage which exceeds 99% considered as high.

1. INTRODUCTION

Over the decades, functional safety became a crucial aspect in the development of mixed-critical systems for various industrial applications that include aerospace applications [1] complied with DO 178 standard, and automotive applications [2] complied with ISO 26262 standard for road vehicles [3]. ISO 26262 regulates mixed safety-critical systems with the Automotive Safety Integrity Level (ASIL), to enhance systems reliability, modularity, maintainability, portability and flexibility and for cost-reduction purposes [4]. These critical systems practically operate on real-time single-core/ multicore processors, as represented in Figure 1, in which they must meet their critical deadline tasks, otherwise a hazardous event would reach the end user. They have been evolving, as a result of the semiconductor evolution of the gigahertz era to the nanoscale level for the sake of a desired performance-power ratio [5].

Mixed safety critical systems have been affecting with the coexistence, where software components (SWCs), even their sub-functions, usually have mixed-ASIL. Hence, interferences from Quality Management (QM) or lower-ASIL SWCs will be introduced [6-9]. They may corrupt higher-ASIL safety-related SWCs by means of information exchange interference, memory interference, real-time interference, and shared peripheral interference as represented in Figure 2.

Consequently, safety mechanisms as single/ multi order detection and reaction mechanisms of the safety-critical data, have been leveraged and trending to prevent interferences that might lead to a safety goal violation. Their main purpose is for faults detection and for controlling system failures, enough to achieve and/ or maintain a safe state at a predefined time, less than fault time tolerant interval (FTTI) [10]. The unwise development of such mechanisms leads to a high development cost, and an overhead on the system performance as well.

Multicore processors face many challenges, presented in the study [9], to overcome. Optimizing inter-core resource sharing distributed among SWCs minimizes the computing power by avoiding wait-states concurrent accesses to the shared resources with the expense of independent data processing and parallelization losses [11]. Software applications run with different criticality such as: scheduling, sharing computation, concurrent resource sharing, memory inter-core communication, communication delays, communication links, and communication resources. These issues become challenges at an operating system (OS) level in today's multicore architectures [12, 13].

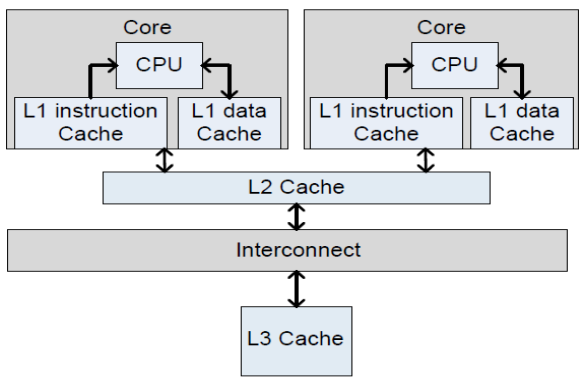

Figure 1. Multicore architecture block diagram

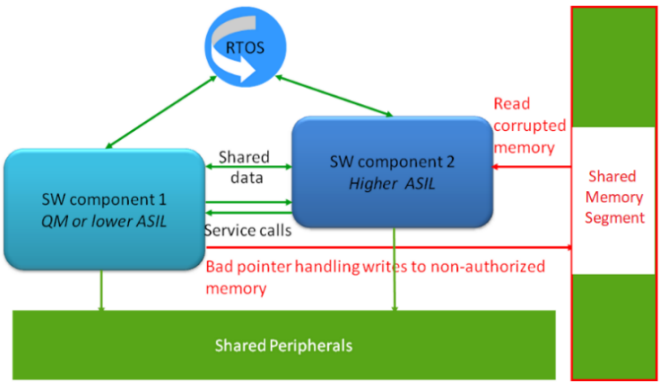

Figure 2. Example of freedom from interferences due to information exchange interference, memory interference and shared peripheral interference

Although multicore processors have clear potentials to produce real-time efficiently and more parallelized features with a high return of their investment [11], ISO 26262 arises functional safety methods and mechanisms at the expense of the functional system efficiency especially if the examined system includes a multicore architecture. ISO 26262 provides many architectural and design requirements methods assure that the examined system operates in a safe state (i.e. degraded mode) by aborting various propagation faults, if erroneous values affect critical signals (even related to calibration data) [14].

ISO 26262 guarantees Freedom from Interference (FFI), in which the separation mechanism must always adhere to the involved highest ASIL SWC. The main goal is that a safety code execution cannot be corrupted by a non-safety code. This means assuring the critical signals flow through SWCs with the desired protection from lower ASIL or QM interfering SWCs that would affect the data correctness. Software architectures including communication interfaces must be developed with this required protection level, accordingly.

In general, there is at least one critical path represents the data flow of a critical signal, from input conditions to the output root-cause, in a safety-related software architecture. It is represented in the critical path analysis of a software design that also includes mixed-ASIL SWCs interferences. In the critical path, it is sufficient to have specific ASIL SWCs that detect and react to means of cascading software/ hardware failures to mitigate inevitably an undesirable event propagation that might lead to a safety goal violation. ASIL SWCs data contain interfering QM SWCs failures. Hence, no safety efforts are needed in QM SWCs with the expense of a CPU overhead and an architecture optimization.

Software data faults may corrupt either memory (i.e. Random Access Memory (RAM), Flash, EEPROM, hardware registers, Direct Memory Access (DMA)), or initialization data or calibration data (i.e. in pre-compile, link-time, and post-build). They may affect logical data processing and data transmission among SWCs (in inter/ intra Electronic Control Unit (ECU) communication).

Memory data integrity faults affect present multicore processor performance capabilities. The memory interference is illustrated by means of memory faults as:

(1) Non-safe arithmetic pointer performed by lower-ASIL SWC that may cause unintended modification of critical data stored in memory; or

(2) Non-safe concurrent access of unions shared among mixed-ASIL SWCs; or

(3) Non-safe dynamic memory allocation may cause an unintended modification of critical data stored in the heap area operated by lower-ASIL SWCs; or

(4) Stack segment overflow resulted from the worst case consumption of lower-ASIL interrupts/ tasks into the segment/ private data area stack allocated to tasks/ interrupts of higher-ASIL SWC (i.e. it also exceeds the tolerated margin); or

(5) Shared Non-Volatile Memory (NVM) blocks among mixed-ASIL SWCs in which a lower-ASIL SWC may delete its data in a way causes improper erase of NVM area allocated to higher-ASIL SWCs; or

(6) Out-of-bounds array accesses performed by a lower-ASIL SWC that may cause corruption of private data of a higher-ASIL SWC.

Only real-time interferences, due to timing faults, OS faults, and sequence faults, are discussed in the study [11] with possible solutions that mitigate the real-time multicore processor challenges into efficient architectures complied with ISO 26262. Conventional state-of-the-art memory interference protection mechanisms have been introduced in the literature [15]. They examine limited number of memory faults of single/ multicore processors.

This work is unprecedented and sets the basis for future development and discussions. The main contribution of this work is to provide software architecture emphasized with the compliance to ISO 26262 FFI methods and principals to detect and react to all possible means of memory corruption failures for sophisticated and complex multi-layered-cache multicore architectures, including AUTOSAR [16]. This paper also provides new algorithms related to safety design mechanisms implemented and examined on many microcontroller targets (Aurix, Renesas, and NXP). It also proposes all required safety-related configuration for memory protection mechanisms.

The rest of the paper is organized as follows. Section 2 represents means of proposed spatial protection mechanisms as double inverse redundant storage algorithms. They are supportive methods for freedom from memory interference challenges for memory systematic faults. While Section 3 shows the hamming distance for the random faults detection. Moreover, section 4 introduces measurement results represented as a Monte Carlo analysis to show the effectiveness of the proposed mechanisms held on several platforms. Finally, a conclusion is provided in Section 5.

2. PROPOSED DESIGN ALGORITHM AND ANALYSIS FOR SYSTEMATIC MEMORY FAULTS PAGE SETUP

Safety is one of the key issues in the development of road vehicles. Development and integration of automotive functionalities strengthen the need for functional safety and the need to provide evidence that functional safety objectives are satisfied. When considering software driven embedded systems, such as automotive ECUs, interference can happen at various levels. The software of a component can access and manipulate another component by writing faulty data into the memory allocated to the other component (data flow).

The memory interference section of ISO 26262-6 [3] Annex D.2.3 illustrates that safety measures such as memory protection, parity bits, error-correcting code (ECC), cyclic redundancy check (CRC), redundant storage, restricted access to memory, static analysis of memory accessing software and static allocation can be used. However, it lacks of appropriate

verification methods as detailed safety analysis to identify critical memories that protection mechanisms are used for. This arises the aim of introducing enhanced modified detection and reaction mechanisms for such memory faults.

In spatial protection, a non-safety code is forbidden to have authorized access rights on a safety-related data. Such granted permissions to the non-safety code would interpret many faults such as: systematic deterministic faults that obligate memory safety measures to be eliminated, and random faults that reveal due to hardware abnormal conditions, happen according to the hardware probability distribution, which produce intermittent, permanent and transient faulty element. These faults could produce memory content corruption, an overflow/ underflow to the memory stack, unauthorized access rights to another SWC memory and data inconsistency. Various safety mechanisms are presented to detect the systematic faults, based on the system ASIL level.

Figure 1 interprets a dual-core architecture with a high-speed communication path illustrated as Interconnect with a shared L3 cache. The L3 cache is shared with a memory controller via the Memory Protection Unit (MPU), and also with all physical cores and logical cores, if exist for high power and performance efficiencies. Each physical core has an individual L2 cache.

For a faster simultaneous multi-threading OS, each core has its private instruction and data L1 cache, as well as the shared memory controllers placed among all physical cores. Moreover, there is no inherited timing interferences among the cores. Each core has redundant 9 banks of 5 registers (control, status, address and error information registers) linked to hardware safety units. Therefore, the architecture includes a safe hardware error reporting mechanism for: uncorrected errors, uncorrected recoverable errors, and corrected errors.

All cores are interconnected with buses, crossbars, meshes and typical routed communication structures. To have a coherent system, interconnect accesses require arbitration accesses from the other cores due to the utilized architecture memory hierarchy defined as (L1, L2 and L3) caches per each core. Furthermore, additional core communication is required, since the L1 cache data of a core may be old as this data is renewed either in the L1 cache of another core, or in the memory controller.

One of the preferred safety mechanisms for the systematic faults and shall be applicable for systems with ASIL-A or higher is the double inverse redundant storage with majority voting and with error detection codes. The double inverse storage is a modified multiple copies mechanism, which is used to compare XORed data copies of stored in different blocks and fix corruption in critical RAM data of higher-ASIL SWCs caused by lower-ASIL SWCs.

Although the redundant storage mechanism is not new, many new modifications are introduced to have an efficient systematic faults detection and reaction than the conventional mechanism. This is provided by introducing many layers of error detection mechanism in a read/ write data algorithms sufficient to produce an effective diagnostic coverage.

The error detection codes can be used to detect corruption in NVM data or even communication messages higher-ASIL SWCs caused by lower-ASIL SWCs. Cyclic Redundancy Code (CRC), checksum, and parity bits are means of the error detection codes. This code is calculated and compared with the error detection code attached with the message/ stored within NVM block. If they are not identical, this indicates message data is corrupted.

On top of that, the upper-layered mechanism of the double inverse storage is also used to detect and recover critical data corruption that may happen due to unauthorized access write.

Algorithm 1. Enhanced Double Inverse Redundant Storage Mechanism for a Read Operation.

Input. BlkAdd1, BlkAdd2, BlkAdd3, MemRowCRC[], Data[]

Output. ReadDataFlag

```
1:    while (ECU is power-up) do
2:     if (ReadRequest()) then
3:       if ((MemRowCRC[BlkAdd1] is invalid) ||
4:           (MemRowCRC[BlkAdd2] is invalid) ||
5:           (MemRowCRC[BlkAdd3] is invalid)) then
6:           /* Validate the CRC of each row memory blo-
7:           ck contain aimed data address */
8:           NumberOfReadFaultTrials++;  /* RamTst
9:           SWC is priodically validating this variable */
10:          ReadDataFlag = 0x11; /* corresponding
11:          fault with hamming distance*/
12:          SetRecoveryModeOption(ReadDataFlag);
13:          /* recovery mode is based on the read corrup-
14:              tion type */
15:          break(); /* repeat memory read */
16:      else if ((CRC(Data[BlkAdd1]) is invalid) ||
17:                (CRC(Data[BlkAdd2]) is invalid) ||
18:                (CRC(Data[BlkAdd3]) is invalid)) then
19:                /*consistency check of data CRC in each
20:                   memory block*/
21:               NumberOfReadFaultTrials++;
22:               ReadDataFlag = 0xEE; /* not verified */
23:               SetRecoveryModeOption(ReadDataFlag);
24:               break (); /* repeat memory read */
25:      else if ((Data[BlkAdd1] != ~Data[BlkAdd2]) ||
26:                (Data[BlkAdd1] !=~Data[BlkAdd3]) ||
27:               (Data[BlkAdd2] != ~Data[BlkAdd3])) then
28:               /* check whether data in each address eq-
29:               uals to its redundant double inverse in ot-
30:               her memory blocks */
31:               NumberOfReadFaultTrials++;
32:               ReadDataFlag = 0xDD;
33:               SetRecoveryModeOption(ReadDataFlag);
34:               break(); /* repeat memory read */
35:      else () then
36:        Read data from any address in any redundant
37:        block nearest to the program counter;
38:        ReadDataFlag = 0x22; /*verified data read*/
39:      end if
40:    end if
41:   end while
```

It should be restricted to very safety-critical data with following criteria:

(1) A single variable corruption shall lead directly to a violation of safety goals;

(2) No other plausibility checks available to detect such data corruptions, and

(3) A variable is updated in event-based style (i.e. no periodic calculation and update), and

(4) The variable update is very slow (i.e. a not allowed proper recovery mechanism within FTTI).

In contrast, the multiple copies safety mechanism increases the RAM consumption, the Flash consumption and the CPU load. The main reason is that, during the operation, if the redundancy check is performed not directly before read usage, then the increased time in between data read and data usage poses a higher risk of corruption due to a faulty ISR, a task

interruption or a RAM memory corruption. Hence, it cannot be utilized to protect all the data within a software code. Similarly, it is not preferred to protect the stack area, or large buffers. In addition, it shall not be used to detect random hardware faults in RAM.

One or more redundant copies of safety critical variables should be stored separately in physical memory only in ASIL-D systems. The selection of proper storage methods to minimize the probability of damage to all copies is very important for significantly improving the safety of stored data. Since this proposed mechanism is not preferred in lower-ASIL systems due to the high cost of implementation. Thus, it's fine to have the data in the same physical memory as long they are separated logically in the memory with suitable erroneous detection and reaction mechanisms.

An initial solution is illustrated as follows: a multiple copies mechanism shall be performed in the main SWC responsible of invoking the tasks that utilize the aimed data to read/ write.

Besides, a more optimized time solution reveals in checking the redundant data copies in the OS PreTaskHook protection capability. The calling ASIL SWC shall validate multiple copies that belong to each safety-critical task. Then, the redundant storage mechanism shall be executed from PreTaskHook for each safety-critical task. Finally, an update to the multiple copies is held in the OS PostTaskHook protection capability. In order to optimize the execution time of the PreTaskHook and to be able to test large complex, CRC or checksum codes shall be incorporated for critical data verification all at once, instead of checking individual critical data separately.

Algorithm 1 and Algorithm 2 represent the double inverse redundant storage mechanisms for read and write operations, respectively. They perform multi-layered verifications before data read and write accordingly. In this enhanced mechanism, one or more redundant copies of the safety-critical variables are stored in physically separated memory locations to minimize the chance in which all copies are corrupted. Original data and the corresponding copies shall contain the value and its inverse, accordingly to increase hamming distance and to decrease the percentage of failure. All copies of a section that includes critical data, stored in different memory blocks shall be updated together, if the original data get updated. Meanwhile, the address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Thus, data copies shall be checked periodically, to point usage within the software code so as to minimize the change for a false detection. In other words, read-back of NVM blocks can be used by a higher-ASIL SWC to ensure that such blocks are written correctly by lower-ASIL NVM stack, given that the calculation of the error detection codes are verified. In addition to cover hardware/ software faults that may result from lower-ASIL device drivers, commands feedback shall be read and compared to the requested commands. This ensures correct application of critical commands by detecting such deviations of improper commands requested by lower-ASIL SWCs.

Before using the data copies, a check for consistency occurs. The read original data with its CRC code placed according to a specific polynomial equation are compared against the stored corresponding ones. If the stored data or its CRC are corrupted, the redundant data are checked with its redundant memory blocks. In case of inconsistency, a proper recovery action is activated such as performing a majority voting among redundant memory sections.

If more data memory blocks are correct (including critical data and CRC), it will be used to update the minor corrupted stored data and its CRC. If the majority stored data is corrupted another possible recovery mechanism as a microcontroller reset to refresh the corrupted blocks with using default safe values with their correct generated CRC. If an overlay area is used, the tasks using this area shall be mutually exclusive.

This reveals there is a need of:

(1) At least 2 copies if the recovery mechanism is relevant only for reset and restore defaults of a non-safety original value; and

(2) At least 3 copies shall be stored, in case of a recovery of safety-related original value.

Static analysis tools (i.e. Polyspace, KlocWork or QAC), that check software code compliance to MISRA rules, shall be used to detect possible errors (i.e. a null pointer access, control pointer handling, out-of-bound array access, division over zero, variable overflow, wrong bitwise operation, unreachable ASIL code statements) that may lead to corrupt critical variables. For corruption reduction, array indexing shall be the only allowed form of pointer arithmetic. Before the data read-write, the multi-layer verification shall be performed, but this would inevitably lead to the reduction of the read-write speed as an expense of capturing the systematic faults. The fact that introducing safety mechanisms to have a fully compliant design with ISO 26262 affects the road vehicle functionality.

In summary, Algorithm 1 behaves as follows:

(1) It verifies if the CRC of the complete row in RAM, that contains the physical address of the aimed data read of the original block and the corresponding redundant blocks, is valid;

(2) It verifies if the contained CRC field included in the aimed data read is valid for all redundant memory blocks; and

(3) It verifies if the redundant data in other memory blocks stored as double inverse of their original containing memory block.

If all verifications are performed, then the read operation shall be performed successfully. If any of those verifications fail, a corresponding flag status is raised and the recovery mechanism will be immediately fired, accordingly.

Meanwhile, Algorithm 2 behaves as follows:

(1) It verifies if the CRC of the complete row in RAM, that contains the physical address of the aimed data read of the original block and the corresponding redundant blocks, is valid;

(2) It performs the write operation from the DataBuffer for each redundant memory block;

(3) It verifies whether written data and their containing CRC equal to the requested data (including CRC) in the DataBuffer for each block;

(4) It performs the write consistency checks among redundant blocks after the write operation as it verifies if the contained CRC field included in the aimed data read is valid for all redundant memory blocks; and

(5) It verifies if the redundant data in other memory blocks stored as double inverse of their original containing memory block.

If all verifications are performed, then the write operation shall be performed successfully. If any of those verifications fail, a corresponding flag status is raised and the recovery mechanism will be immediately fired, accordingly.

One or more redundant copies of safety or security critical variables should be stored separately in physical memory. The selection of proper storage methods to minimize the probability of damage to all copies is very important for significantly improving the safety or the security of stored data.

Algorithm 2. Enhanced Double Inverse Redundant Storage Mechanism for a Write Operation.

Input. BlkAdd1, BlkAdd2, BlkAdd3, MemRowCRC[], Data[], DataBuffer[]

Output. WriteDataFlag

```
1:  while (ECU is power-up) do
2:      if (WriteRequest(DataBuffer[])) then
3:        if ((MemRowCRC[BlkAdd1] is invalid) ||
4:            (MemRowCRC[BlkAdd2] is invalid) ||
5:            (MemRowCRC[BlkAdd3] is invalid)) then
6:            /* Validate the CRC of each row memory blo-
7:            ck contain aimed data address */
8:            NumberOfWriteFaultTrials++;  /* RamTst
9:            SWC is priodically validating this variable */
10:           WriteDataFlag = 0x11; /* corresponding
11:           fault with hamming distance*/
12:           SetRecoveryModeOption(WriteDataFlag);
13:           /* recovery mode is based on the write corru-
14:               ption type */
15:           break(); /* repeat memory write */
16:       end if
17:       Data[BlkAdd1] = DataBuffer[BlkAdd1]);
18:       /* write variable on its redundant storage */
19:       Data[BlkAdd2] = DataBuffer[BlkAdd2]);
20:       Data[BlkAdd3] = DataBuffer[BlkAdd3]);
21:       /*verify data read after write to verify back*/
22:       if ((Data[BlkAdd1] != DataBuffer[BlkAdd1]) ||
23:          (Data[BlkAdd2] != DataBuffer[BlkAdd2]) ||
24:          (Data[BlkAdd3] != DataBuffer[BlkAdd3]) then
25:            NumberOfWriteFaultTrials++;
26:            WriteDataFlag = 0x77;
27:            SetRecoveryModeOption(WriteDataFlag);
28:            break(); /* repeat memory write */
29:       else if ((CRC(Data[BlkAdd1]) !=
30:                    CRC(DataBuffer[BlkAdd1]) ||
31:                  (CRC(Data[BlkAdd2]) !=
32:                   CRC(DataBuffer[BlkAdd2]) ||
33:                  (CRC(Data[BlkAdd3]) !=
34:                   CRC(DataBuffer[BlkAdd3])) then
35:                  NumberOfWriteFaultTrials++;
36:                 WriteDataFlag = 0x88;
37:                 SetRecoveryModeOption(WriteDataFlag);
38:                 break(); /* repeat memory write */
39:       else () then
40:       Free(DataBuffer[]); /*verified write operation*/
41:       end if
42:       /* Do as same as Algorithm 1 row#16 to verify
43:       write consistency over the memory blocks */
44:       if ((CRC(Data[BlkAdd1]) is invalid) ||
45:           (CRC(Data[BlkAdd2]) is invalid) ||
25:           (CRC(Data[BlkAdd3]) is invalid)) then
26:           /*CRC consistency check after write*/
28:           NumberOfWriteFaultTrials++;
29:           WriteDataFlag = 0xEE; /* not verified */
30:           SetRecoveryModeOption(WriteDataFlag);
31:           break (); /* perform memory consistency */
32:       else if ((Data[BlkAdd1] != ~Data[BlkAdd2]) ||
33:                  (Data[BlkAdd1] != ~Data[BlkAdd3]) ||
34:                 (Data[BlkAdd2] != ~Data[BlkAdd3]) then
35:                 /* check whether data in each address eq-
36:                 uals to its redundant double inverse in ot-
37:                 her memory blocks */
38:                 NumberOfWriteFaultTrials++;
39:                 WriteDataFlag = 0xDD;
40:                 SetRecoveryModeOption(WriteDataFlag);
41:                 break(); /*repeat memory consistency*/
42:       else () then
19:         WriteDataFlag = 0x22; /*verified data Write*/
20:       end if
21:     end if
44:   end while
```

3. PROPOSED DESIGN ALGORITHM AND ANALYSIS FOR RANDOM FAULTS IN RAM

A key issue in designing any error correcting code is making sure that any two valid code words are sufficiently dissimilar so that corruption of a single bit (or possibly a small number of bits) does not turn one valid code word into another. To measure the distance between two code words, we just count the number of bits that differ between them. If we are doing this in hardware or software, we can just XOR the two code words and count the number of 1 bits in the result. This count is called the Hamming distance.

Implementing the hamming distance as a safety mechanism addresses random hardware faults in RAM, undetectable error, bi-symmetric inversion bit (bit flipping) and burst errors in systems with ASIL-A criticality or higher. High level of hamming distance shall be applied to critical data, critical state-machine, coding with true/ false patterns, enumerator type declarations, memory blocks CRC, validity flags and defined constants. In this mechanism, the number of complementary bits between 2 adjacent states (true, or false) should be greater than 4. Thus, it is preferred to have the 2 complementary states of a critical variable as: A5, 5A or 69, 96 accordingly. Software tools may be used to detect the required hamming value if a variable includes higher states.

Although Algorithm 1 and Algorithm 2 target systematic faults in RAM, random faults might occur in such local variables in those algorithms. Thus, hamming distance mechanism is a cost-reduction safety implementation needed for the variables ReadDataFlag and WriteDataFlag.

In Algorithm 1, ReadDataFlag has 4 states (represent failure modes), as shown in Table 1, as follows: {0x11, 0xEE, 0xDD, 0x22}. The highest hamming distance is among {0x11, 0xEE}, {0x22, 0xDD} by 8-bit differences. On the other hand, the lowest hamming distance is among {0x11, 0xDD}, {0x01, 0x22}, {0xEE, 0x22}, {0xEE, 0xDD} by 4-bit differences. In Algorithm 2, WriteDataFlag has 6 states (represent failure modes), as shown in Table 2, as follows: {0x11, 0xEE, 0xDD, 0x22, 0x77, 0x88}. The highest hamming distance is among {0x11, 0xEE}, {0x22, 0xDD}, {0x77, 0x88} by 8-bit differences. On the other hand, the lowest hamming distance is among {0x11, 0xDD}, {0x01, 0x22}, {0xEE, 0x22}, {0xEE, 0xDD}, {0x11, 0x77}, {0x11, 0x88}, {0xEE, 0x77}, {0xEE, 0x88}, {0xDD, 0x77}, {0xDD, 0x88} , {0x22, 0x77}, {0x22, 0x88} by 4-bit differences.

Table 1. Algorithm 1 ReadDataFlag hamming distance states

State	Failure mode	Definition
(00010001)b	A	A fault pre-read operation due to initial invalid memory blocks CRC
(11101110)b	B	A fault read-verification operation due to inconsistent memory blocks CRC
(11011101)b	C	A fault read-verification operation due to inconsistent memory blocks data
(00100010)b	D	A verified data read operation

Table 2. Algorithm 2 WriteDataFlag hamming distance states

State	Failure mode	Definition
(00010001)b	A	A fault pre-write operation due to initial invalid memory blocks CRC
(11101110)b	B	A fault write-verification operation due to inconsistent memory blocks CRC
(11011101)b	C	A fault write-verification operation due to inconsistent memory blocks data
(00100010)b	D	A verified data write operation
(01110111)b	E	A fault write operation due to inconsistent memory block data after-write and data buffer
(10001000)b	F	A fault write operation due to inconsistent memory block CRC after-write and data buffer CRC

4. MEASUREMENT RESULTS

The proposed safety mechanisms have been evaluated and verified for Aurix Tri-core and Renesas RH850 targets to assess to have a fully compliant architecture with principles and methods of ISO 26262.

Failure Mode and Effect and Diagnostic Analysis (FMEDA) and the Fault Tree Analysis (FTA) are recommended by ISO 26262 Part 5 [3] as they are the most widely used quantitative safety analysis techniques in the automotive industry. In any quantitative analysis, "Diagnostic Coverage (DC)" of the safety mechanisms is a crucial parameter that affects the final safety metrics. The diagnostic coverage is a measure of effectiveness of the diagnostics implemented in the system. Mathematically, it is the ratio of the failures detected and/or controlled by a safety mechanism to the total failures) in the element. Failure modes ranged from A to F as represented in Table 1 and Table 2.

Determining the diagnostic coverage in practice is not trivial. To simplify the process, ISO 26262 provides a "starting point" for estimating the DC values of a safety mechanism based on their applicability to a system. They are classified as Low (60%), Medium (90%) and High (99%) diagnostic coverage. The safety mechanisms are classified to these corresponding levels (low, medium, high) depending on factors varying from:

(1) Variations in the source of the fault type detected by the diagnostic.

(2) Specific implementation of a safety mechanism technologies implemented in the system

(3) The execution timing of the safety mechanism with respect to the FTTI.

$$K_{DC} = \sum (X \times K_{FMC,x}) \quad (1)$$

Eq. (1) represents the diagnostic coverage of the proposed safety mechanisms (K_{DC}), where X is the failure mode distribution for a failure mode x. This x equals the values in range of A ~ D, as per Table 1 and in range of A ~ F, as per Table 2. $K_{FMC,x}$ is the failure mode coverage of the failure mode x.

To have an efficient normal distribution, Monte Carlo analysis has been performed for the proposed designed diagnostic coverage that is corresponding to all related failure modes as discussed in Table 1 for the Algorithm 1 and in Table 2 for the Algorithm 2 to assess the proposed design robustness in case of memory systematic faults and random faults in RAM variations. Figure 3 shows that the mean diagnostic coverage which is around > 99% (High) by adding SWC stubs to simulate each failure mode. Table 3 represents the DC for both algorithms for the different failure modes to validate the effectiveness of the proposed mechanisms at 1000 number of samples of measurement runs held on several platform targets.

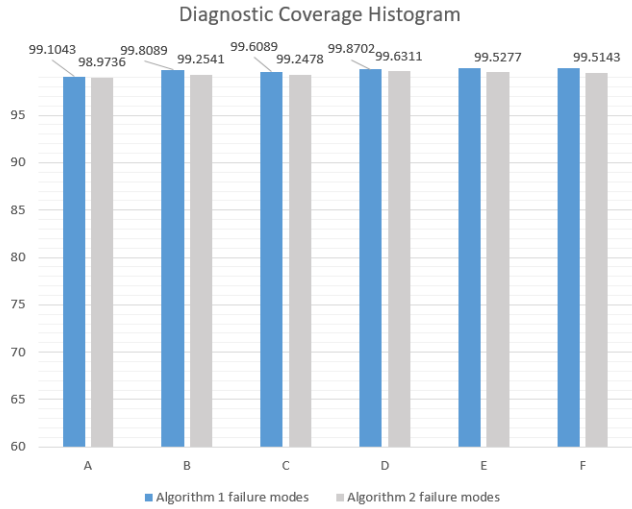

Figure 3. Histogram of Monte Carlo analysis for the proposed safety mechanisms diagnostic coverage for 1000 samples of execution runs

Table 3. Typical diagnostic coverage (DC) including Monte Carlo mismatches for all failure modes affect Algorithm 1 and Algorithm 2

Failure mode	Algorithm 1 DC	Algorithm 2 DC
A	99.1043%	98.9736%
B	99.8089%	99.2541%
C	99.6089%	99.2478%
D	99.8702%	99.6311%
E	100%	99.5277%
F	100%	99.5143%

The worst case DC takes place at the failure mode A in Algorithm 2 by 98.9736%. This means that the Algorithm 2 can detect 989.73 out of possible 1000 potential systematic faults due to interfered QM or lower-ASIL SWCs to higher-ASIL SWC.

5. CONCLUSION

In this paper, memory interference challenges of multicore architectures for autonomous driving applications have been explored, leveraged, analyzed and mitigated. Furthermore, various novel solutions design and their relevant configuration are proposed, to present complex architectures designs to be compliant with the ISO 26262 methods and principals based on the examined system architecture ASIL. The proposed safety mechanisms target memory data integrity faults detection and immediate reaction mechanisms, enough to let the system behave in the safe state before the defined FTTI.

Memory faults are categorized as systematic faults, and random hardware faults that produce intermittent, permanent and transient faulty element. These faults are represented as a result of non-safe accesses (of arithmetic pointer, unions,

dynamic memory allocation, out-of-bound array, shared NVM, blocks, stack segment) to the memory area contained by ASIL SWC. These faults could produce memory content corruption, an overflow/ underflow to the memory stack, unauthorized access rights to another SWC memory and data inconsistency.

Stack-monitoring mechanism is recommended to minimize such critical data failures. Logically related safety data shall be included in a critical section to ensure consistency data so as not to be accessed atomically and to be only used by higher-ASIL SWCs (i.e. senor readings and sensor status shall be read together, otherwise we can't ensure if sensor readings are correct).

Multiple copies, and double inverse redundant storage design mechanisms are presented with new modified algorithms to mitigate the CPU load and memory consumption. They are introduced as a means of faults detection, with new enhanced design configurations, placed in Aurix Tri-core, NXP Freescale MPC and Renesas RH850 platforms, to mitigate systematic memory data integrity faults. As they are considered software configuration to hardware features, another mechanism is proposed to overcome such limitation. Diagnostic coverage has been measured with the support of the failure modes occurrence variations as a result of Monte Carlo analysis which shows that the DC of the proposed design mechanisms exceeds 99% as a worst case scenario.

Another safety mechanism, hamming distance, is proposed to contain random hardware faults in RAM such as. Various methods of fault reaction are introduced to fulfill the presented safety detection mechanisms strategy. Refreshment to the NVM memory blocks if there is a memory corruption. OS features as the OS protection hooks shall be used to enhance the reaction efficiency.

From the perspective of vehicle safety, this proposes a new optimized dual inverse redundant storage algorithm to alleviate system safety storage data failures, and introduces other safety mechanisms to deal with random safety failures in the memory. This has been verified in the practical cases, fully in line with the ISO26262 safety standard.

ACKNOWLEDGMENT

The work presented here has been partially carried out for the framework of autonomous driving applications architectures, which are supported by the Research and Development Center of Valeo Egypt.

REFERENCES

[1] El-Bayoumi, A., Salem, M., Khalil, A., El-Emam, E. (2015). A new Checkout-and-Testing Equiment (CTE) for a satellite telemetry using LabVIEW. 36th IEEE Aerospace Conference, Big Sky, MT, pp. 1-9. https://doi.org/10.1109/AERO.2015.7119305

[2] Casarsa, M., Harutyunyan, G. (2019). Case study and advanced functional safety solution for automotive SoCs. 2018 IEEE International Test Conference (ITC), Phoenix, AZ, USA, pp. 1-8. https://doi.org/10.1109/TEST.2018.8624740

[3] ISO 26262:2018- Road Vehicles - Functional Safety – Part 1–12. https://www.iso.org/standards.html, accessed on Dec., 2018.

[4] Wei, Y., Le, Y., Xie, G., Zhang, L. (2019). Development cost optimization for multi-functional mixed-criticality embedded systems. IEEE Access, 7: 88949-88959. https://doi.org/10.1109/ACCESS.2019.2925048

[5] El-Bayoumi, A., Mostafa, H., Soliman, A. (2016). A novel MIM-capacitor based 1-GS/s 14-bit variation-tolerant fully-differential VTC circuit. World Scientific Journal of Circuits, Systems and Computers, 26(5): 1-35. https://doi.org/10.1142/S0218126617500736

[6] Su, F., Goteti, P., Zhang, M. (2019). On freedom from interference in mixed-criticality systems: A causal learning approach. 2019 IEEE International Test Conference (ITC), Washington, DC, USA, pp. 1-10. https://doi.org/10.1109/ITC44170.2019.9000160

[7] Goebel, A., Mader, R., Tripon, O. (2017). Performance and freedom from interference - a contradiction in embedded automotive multi-core applications? 2017 IEEE 30th International Conference on Architecture of Computing Systems (ARCS), pp. 1-9.

[8] Zimmer, B., Dropmann, C., Hänger, J.U. (2014). A systematic approach for software interference analysis. 2014 IEEE 25th International Symposium on Software Reliability Engineering, Naples, pp. 78-87. https://doi.org/10.1109/ISSRE.2014.12

[9] Xie, G., Zeng, G., Li, R. (2020). Safety enhancement for real-time parallel applications in distributed automotive embedded systems: A stable stopping approach. IEEE Transactions on Parallel and Distributed Systems, 31(9): 2067-2080. https://doi.org/10.1109/TPDS.2020.2984719

[10] Schliecker, S., Negrean, M., Ernst, R. (2009). Response time analysis on multicore ECUs with shared resources. IEEE Transactions on Industrial Informatics, 5(4): 402-413. https://doi.org/10.1109/TII.2009.2032068

[11] Mutlu, O., Moscibroda, T. (2009). Parallelism-aware batch scheduling: Enabling high-performance and fair shared memory controllers. IEEE Micro, 29(1): 22-32. https://doi.org/10.1109/MM.2009.12

[12] Iturbe, X., Venu, B., Jagst, J., Ozer, E., Harrod, P., Turner, C. (2018). Addressing functional safety challenges in autonomous vehicles with the arm TCL S architecture. IEEE Design & Test Magazine, 35(3): 7-14. https://doi.org/10.1109/MDAT.2018.2799799

[13] Martin, H., Winkler, B., Grubmüller, S., Watzenig, D. (2019). Identification of performance limitations of sensing technologies for automated driving. 2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE), Graz, Austria, pp. 1-6. https://doi.org/10.1109/ICCVE45908.2019.8965181

[14] Sini, J., Violante, M., Dodde, V., Gnaniah, R., Pecorella L. (2019). A novel simulation-based approach for ISO 26262 hazard analysis and risk assessment. 2019 25th IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS), Rhodes, Greece, pp. 253-254. https://doi.org/10.1109/IOLTS.2019.8854385

[15] Koser, E., Berthold, K., Pujari, R.K., Stechele, W. (2016). A Chip-level Redundant Threading (CRT) scheme for shared-memory protection. 2016 International Conference on High Performance Computing & Simulation (HPCS), Innsbruck, pp. 116-124. https://doi.org/10.1109/HPCSim.2016.7568324

[16] AUTOSAR Layered Architecture. http://www.autosar.org/standards/classic-platform/release-40/software-architecture/general/, accessed on Jul. 2017.

NOMENCLATURE

ASIL	automotive safety integrity level
OS	operating system
SWC	software component
QM	quality management
FTTI	fault time tolerant interval
FFI	freedom from interference
RAM	random access memory
DMA	direct memory access
ECU	electronic control unit
NVM	non-volatile memory
MPU	memory protection unit
CRC	cyclic redundancy code
DC	diagnostic coverage
FTA	fault tree analysis
FMEDA	failure mode and effect and diagnostic analysis