SafeDM: a Hardware Diversity Monitor for Redundant Execution on Non-lockstepped Cores

Francisco Bas[†,‡], Pedro Benedicte[†], Sergi Alcaide[†], Guillem Cabo[†], Fabio Mazzocchetti[†], Jaume Abella[†]
[†]Barcelona Supercomputing Center (BSC)
[‡]Universitat Politècnica de Catalunya (UPC)

***Abstract*—Computing systems in the safety domain, such as those in avionics or space, require specific safety measures related to the criticality of the deployment. A problem these systems face is that of transient failures in hardware. A solution commonly used to tackle potential failures is to introduce redundancy in these systems, for example 2 cores that execute the same program at the same time. However, redundancy does not solve all potential failures, such as Common Cause Failures (CCF), where a single fault affects both cores identically (e.g. a voltage droop). If both redundant cores have identical state when the fault occurs, then there may be a CCF since the fault can affect both cores in the same way. To avoid CCF it is critical to know that there is diversity in the execution amongst the redundant cores.**

In this paper we introduce SafeDM, a hardware Diversity Monitor that quantifies the diversity of each redundant processor to guarantee that CCF will not go unnoticed, and without needing to deploy lockstepped cores. SafeDM computes data and instruction diversity separately, using different techniques appropriate for each case. We integrate SafeDM in a RISC-V FPGA space MPSoC from Cobham Gaisler where SafeDM is proven effective with a large benchmark suite, incurring low area and power overheads. Overall, SafeDM is an effective hardware solution to quantify diversity in cores performing redundant execution.

I. Introduction

Safety-related systems undergo strict development processes in accordance with domain-specific safety regulations to prove that their safety goals are satisfied. For instance, in the case of automotive systems, the main functional safety standard is ISO26262 [14], which defines several Automotive Safety Integrity Levels (ASIL), from A to D, being ASIL-D the most stringent one, as well as Quality Managed (QM) systems and components, which correspond to those systems and components without safety requirements.

In the case of ASIL-D systems and components, strong guarantees are needed on the fact that a single fault cannot lead the whole system to an undetected failure. In the case of computing cores, this is generally achieved with Dual Core LockStep (DCLS), as indicated in ISO26262. DCLS builds upon two identical cores running the same software, being one of them a shadow core not visible at the user level, and executing the redundant task with some staggering so that both cores never hold the same state and any fault affecting both of them can only produce distinct errors, which, therefore, can easily be detected. However, such an approach is expensive because for each computing core, there is a shadow core that can only be used for lockstepped execution, not to deliver performance.

Commercial-Off-The-Shelf (COTS) processors for embedded systems rarely implement DCLS, at least for moderate or high-performance cores due to cost constraints (i.e., not wasting half of the performance power of the SoC) and, therefore, are not amenable to run ASIL-C/D functionalities. To overcome this limitation, some works have recently proposed solutions to enforce diversity either by software means [3] or by hardware means [4] with an appropriate monitor that stalls the trail core when its staggering w.r.t. the head core is regarded as too short. Hence, there is some risk of the trail core catching up with the head one, hence losing diversity. Unfortunately, those solutions impose constraints such as having to run redundant tasks with identical control flow at the software level, which may not occur, for instance, if they read input data at different time instants, if they run parallel applications and thread synchronization occurs differently, or simply if non-deterministic conditions are used for loops (e.g., conditions partially depending on a random value).

To address the constraints of diverse redundancy based on staggering, this paper presents SafeDM, a Safe Diversity Monitor, which quantifies the diversity existing across the state of two cores, but without interfering with execution. SafeDM builds upon the observation that, naturally, software-redundant execution tends to be diverse due to the serialization of the access to some hardware shared resources and real-time operating system (RTOS) services, as well as due to the use of different address ranges. Hence, rather than enforcing some staggering to enforce diversity, SafeDM monitors whether diversity effectively holds transparently to the execution notifying whether such diversity is lost and how often. The existence of SafeDM allows developing a safety concept for ASIL-D functionalities where redundant threads are let to run without imposing any type of staggering or process stall and, instead, the SafeDM provides evidence that diversity exists and only notifies the RTOS about diversity loss through interrupts to prevent the loss from independence cause any side effect in the form of increased vulnerability.

In particular, our contributions are as follows:

1) We present SafeDM, a diversity monitor to enable ASIL-D compliance on regular (non-lockstepped) cores.
2) We integrate SafeDM in a 4-core multicore by Cobham Gaisler for the space domain.
3) We evaluate SafeDM as part of Gaisler's SoC triggering scenarios where diversity is naturally tiny, as well as scenarios with abundant diversity. Our results show that SafeDM accurately detects the loss of diversity at the expense of occasional false positives at most.

The rest of the paper is organized as follows. Section II provides some background on functional safety and diversity. Section III presents SafeDM. Section IV introduces the target platform and how SafeDM has been integrated. SafeDM is evaluated in Section V. Section VI describes some related work. Finally, Section VII draws the main conclusions of this work.

II. Background

The development process of safety-related electronics systems is similar for several domains, including industrial electronics [12], automotive [14] and railway [7] among others. Without loss of generality, in this paper, we focus on the automotive domain and its main functional safety standard, ISO26262.

978-3-9819263-6-1/DATE22/©2022 EDAA

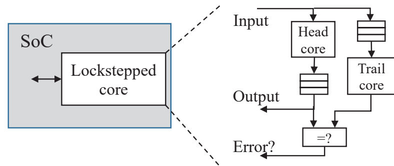


Fig. 1: High-level schematic of a lockstepped core.

The safety development process follows the typical 'V' model. It starts with the definition of the safety goals of the system, which are propagated into different safety requirements. When specifying the architecture of the system, those safety requirements are mapped into system components applying ASIL decomposition as needed so that each item has the lowest ASIL possible while still fulfilling the overall safety requirements. The safety requirements mapped to each item (e.g., a CPU) determine the type of means needed for the design, verification and validation of such item.

In the case of ASIL-D hardware items, they must be capable of, at least, detecting any single fault. Redundancy provides some protection, but it is not enough to avoid the so-called *Common Cause Failures (CCF)*. CCF correspond to those faults that can cause identical errors in redundant components so that errors go unnoticed and become failures at some scope. To avoid CCF, safety standards impose the use of *diverse redundancy*, so that if a single fault affects all redundant components, at least the errors experienced will differ and will be detected. In the case of storage and interconnects, the typical solutions resort to Error Correction Codes (ECC) and Cyclic Redundancy Checking (CRC). In the case of computing components (e.g., a processor core), full redundancy is needed, normally implemented in the form of lockstepped cores where two identical cores are tied together replicating their inputs, executing with some cycles of staggering, and comparing their outputs for error detection as illustrated in Figure 1. This scheme is used, for instance, by the Infineon AURIX [13] and STmicroelectronics SPC570Sx [31] microcontroller families.

However, many moderate and high-performance processors considered for the automotive domain, either because of their own performance or because they come along with specific accelerators needed for other functionalities (e.g., GPUs), do not implement lockstepping. Hence, alternative means are needed to achieve diverse redundancy. It can be achieved with diverse implementations of the software, but this solution may double design, verification and validation costs for such software. Diverse redundancy can also be achieved by running equivalent software on diverse cores. However, this implies that the SoC has at least two types of cores, and performance is limited by that of the slowest core, hence wasting the benefits provided by high-performance cores. Instead, solutions based on software (identical) replication with some form of staggering can provide advantages compared to those of lockstepping, either with software staggering [3] or with an ad-hoc hardware module [4]. However, those solutions are intrusive with the execution.

Recently, it has been shown that diversity naturally exists due to the increasing complexity of the platforms [22], which would remove the need for enforcing any staggering at all. However, the safety concept of the system needs evidence of that. Hence, we propose SafeDM to tackle this challenge.

III. SafeDM Design

This section introduces SafeDM, a diversity monitor providing evidence of existing diversity, and hence supporting the safety concept of ASIL-D systems. In particular, we first introduce the rationale behind SafeDM, and later on its design.

A. Rationale behind SafeDM

As explained before, diversity is expected to emanate from system complexity [22], and this trend is expected to hold given that both hardware and software components become increasingly complex over time. However, in order to build a safety concept, either diversity needs to be enforced (e.g. with hardware or software mechanisms [3], [4]) or evidence of its existence needs to be retrieved. The former guarantees diversity by construction, but existing mechanisms are necessarily intrusive with tasks execution by, preventively, stalling the trail process when it is regarded as not sufficiently staggered w.r.t. the head process. The latter allows building a safety measure that, upon the detection of lack of diversity, takes an action. Such action can either be the same action it would be taken upon the detection of an error, or a softer action since lack of diversity indicates lack of protection, but generally not an error.

We note that, as explained later, SafeDM can only raise false positives (i.e. notifying lack of diversity when, in fact, there is diversity), but not false negatives. Moreover, those false positives – if any – occur seldom in practice. Hence, applying the same safety measure as if an error had occurred is a viable and simple strategy. Note that, in the context of automotive systems, including ASIL-D systems, errors can be managed by simply dropping the task. For instance, ASIL-D systems such as braking and steering are executed at high frequency (e.g. every 50ms) and a hazard can occur if, for instance, errors are not detected within a larger period (e.g. 200ms), which is the Fault Tolerant Time Interval (FTTI). Hence, if a job of the braking task is dropped, hence preserving the decision taken 50ms ago during 50 additional ms, the system still remains safe as long as new job drops do not occur consecutively.

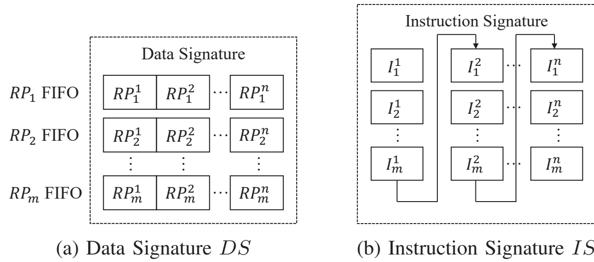
B. Design

We note that lack of diversity occurs when two identical computing components (e.g. two cores) perform *identical* activity simultaneously, so that voltage levels and current flows are virtually identical in the same core locations, being just subject to unavoidable, and typically low, variations (e.g. due to random process variations). Hence, if the cores are executing different instructions in a given pipeline stage, or operating different data, voltages and currents differ across cores and hence, upon a fault affecting both cores analogously, the effects – and hence the errors if any – will differ.

Therefore, to know whether there is diversity in the program execution of two different cores at a given cycle, we need to consider both the instructions and data that are in the pipeline at that given moment. If both cores are executing the same instructions in all the stages of the pipeline and each stage is using the same data, then there is no diversity between cores unless it comes from other sources[1].

Therefore, we need a method that allows summarizing the core state in terms of both instructions and data, for comparison to detect whether diversity exists. This is the Diversity Monitor (DM), which is composed of two different signatures: the Data Signature (DS) and the Instruction Signature (IS), encapsulating the state of data and instructions each, respectively. Next,

[1]Note that, if other sources, that we neglect, bring diversity, SafeDM could notify lack of diversity unnecessarily, hence causing a false positive, which nevertheless does not challenge safety.



(a) Data Signature DS (b) Instruction Signature IS

Fig. 2: Data and instruction signatures

we introduce these two signatures, and later we develop on the concept of the Diversity Monitor.

1) Data Signature (DS): To quantify the diversity in the data being used in each core at a given time instant, the data being read/written for the last n cycles on each of the register ports is used. The reason for it is that all data being used in the pipeline stages is read and/or stored from/to the register file, so using the data read/written for a number of cycles guarantees that all the data being used in the processor is captured. Note that, while some data may be read through bypasses, such data is also sent to the register file for writing, and hence, observed at the register file ports. The size of n depends on the depth of the processor pipeline and is implementation specific.

For each of the register ports m, a FIFO queue is used to store the last n values read or written, depending on whether the port is a read or write port. When a cycle passes, the oldest data entry is removed from the head of the queue, and the new data is inserted in the tail of the FIFO queue. By recording the data at the register ports every cycle rather than every time the register file is read or written, we avoid situations where registers are read/written in the same order but with different timing (e.g. with some staggering), hence with diversity, but no diversity would be detected if we restricted ourselves only to check the last values read or written disregarding their timing.

The Data Signature (DS) is computed by concatenating all the values of all the FIFOs.

For example, for the data in m register ports RP_1 to RP_m, considering the data of the last n cycles and denoting the data of register port x at cycle y as RP_x^y, DS consists of the concatenation of $n{*}m$ register port entries (also see Figure 2a):

$$DS = RP_1^1 RP_1^2 ... RP_1^n ... RP_2^1 ... RP_2^n ... RP_m^1 ... RP_m^n$$

To compare the DS of two redundant cores, c_0 and c_1, we use their Data Signatures DS_0 and DS_1, respectively. If the difference is 0, it means that the data being read/written for the last number of captured cycles is the same, so potentially there is no diversity. Otherwise, if DS is different from 0, it means that there is diversity since some of the data in the pipeline is different across cores. Formally stated:

$$DataDiversity = (DS_0 \neq DS_1)$$

2) Instruction Signature (IS): To know if there is diversity between the instructions being executed in two cores at a given time, we need to consider all the instructions that are in execution (in flight) in the pipeline at that cycle, and the specific pipeline stage each of them is. To account for this, a sequence of the last o executed instructions in each of the cores must be stored in a FIFO queue. Note that a FIFO queue is a suitable block to store instructions since they are fetched and retired in order. Instructions will be enqueued as they are fetched and dequeued as they are retired (aka committed). The Instruction Signature (IS) is computed by concatenating all of the values of the FIFO.

Note that such an approach will indicate that diversity is only lost whenever two cores process the same subset of instructions in the same order, regardless of whether they are being processed at different stages, which could lead to false positives. However, cores for safety-related systems are often relatively simple implementing superscalar, yet in-order, pipelines. Hence, we analyzed the actual core where we instantiate SafeDM (Cobham Gaisler's NOEL-V core), which we describe later, and realized that it fetches up to p instructions per cycle (2 in NOEL-V), and the instructions in one stage (either 1 or 2) move to the following stage as a group (either all or none). Hence, rather than just keeping the list of instructions in the pipeline, we keep the list of instructions in each pipeline stage so that, if two cores are processing the same instruction but any of them is in a different pipeline stage across cores, their IS will differ.

For a core whose pipeline width is p instructions, and has o pipeline stages, we note as I_x^y the instruction in position x in the yth stage of the pipeline (also see Figure 2b). Note that I_x^y can be an empty slot if no instruction was fetched in that slot, or if the instructions in that pipeline stage were promoted to the next stage and those in previous stages did not reach this stage yet. Overall, IS is as follows:

$$IS = I_1^1 I_1^2 ... I_1^p ... I_2^1 ... I_2^p ... I_o^1 ... I_o^p$$

Note that if the assumptions made for NOEL-V cores did not apply to the target core, then IS would restrict to the list of instructions fetched but not yet retired.

Analogously to the DS, we consider that two cores lack diversity if their Instruction Signatures are equal:

$$InstructionDiversity = (IS_0 \neq IS_1)$$

3) Diversity Monitor (DM): Overall, if either DS or IS differ across cores, there is some diversity and hence, lack of diversity is reported only whenever both signatures match across cores. Note that, whenever there is no diversity there is risk of experiencing a CCF, but such CCF does not have to occur necessarily. Hence, SafeDM has been devised so that lack of diversity can be managed in three different ways: (1) reporting it as soon as it occurs once raising an interrupt; (2) reporting it as soon as it occurs a given user-defined number of times with an interrupt; or (3) not raising any interrupt and letting the operating system polling the count of cycles without diversity whenever needed.

What approach to follow out of those three is allowed by SafeDM programming it accordingly.

4) Advantages of SafeDM w.r.t. staggering enforcing methods: As explained before, some solutions enforce some staggering across cores to guarantee diversity [3], [4]. Those approaches impose the constraint that both cores must execute *exactly* the same instruction stream so that enforcing some staggering guarantees diversity. Note that, if both cores executed different instruction streams (e.g. one of them executes some additional iterations of a loop) then they could lack diversity despite having different instruction counts. This is problematic for parallel programs, where synchronization effects may make a given thread execute different instruction streams, or for programs with non-deterministic conditions (e.g. if-conditions dependent on random values).

SafeDM removes those constraints altogether. SafeDM only considers the real state of the cores regardless of how they

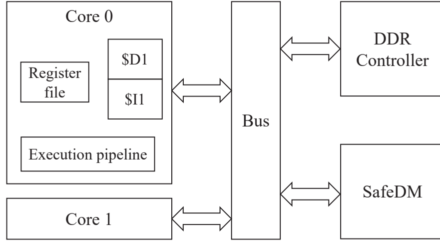


Fig. 3: MPSoC schematic with SafeDM

reached that state. Hence, SafeDM puts no constraints on the software run in each core and it could even be used to support diverse software implementations of the same function, which could potentially lack diversity if they use, for instance, functions from the same library or the same operating system services.

IV. SafeDM Integration

A. MPSoC Platform

SafeDM has been implemented and evaluated in a commercial MultiProcessor System on Chip (MPSoC) used in the space industry, and developed by Cobham Gaisler [34], which builds upon NOEL-V cores. The NOEL-V based MPSoC consists of reusable VHDL IP cores connected via on-chip buses, based on the standard ARM AMBA 2.0. We have developed SafeDM in VHDL and integrated it into the NOEL-V based platform.

1) MultiProcessor System on Chip: The MPSoC consists of 2 RISC-V 64-bit dual-issue 7-stage pipeline NOEL-V cores 3. The main bus for communication is a 128 bit-wide Advanced High-performance Bus (AHB). SafeDM is connected via an Advanced Peripheral Bus (APB) interface to the main bus. Each core has a private L1 Data and Instruction cache. The data cache is write-through and with a write-no-allocate policy. The L2 cache is shared amongst the cores and connected to the main bus, together with the memory controller.

B. Integration

We have implemented SafeDM as an APB slave that interfaces with the rest of the MPSoC through the bus using the APB interface. With this, the SafeDM is also portable to other systems implementing the APB interface.

The main modules implemented can be seen in Figure 4. In this integration, in addition to the modules required to implement SafeDM, we have also added two extra modules for testing the module (Instruction diff) and gathering of results (History module).

1) Signature generator: This is the main module that stores the register and instruction values in specific FIFOs. For the data values, it requires the enable signals and the value being read or written. Each register port has its own FIFO, in this case, a total of 4. To compute DS, all the FIFOs for each core are concatenated. Then, these are compared in a separate comparator, and the data is stored in the History module.

Similarly, for the instructions, the instruction encoding plus the valid bit are used, but in this case, just one FIFO is used per core. Both FIFOs are compared, and the result of the comparison is also sent to the History module.

In both cases, the hold signal is used to not overwrite any values in the FIFOs if the pipeline is stalled.

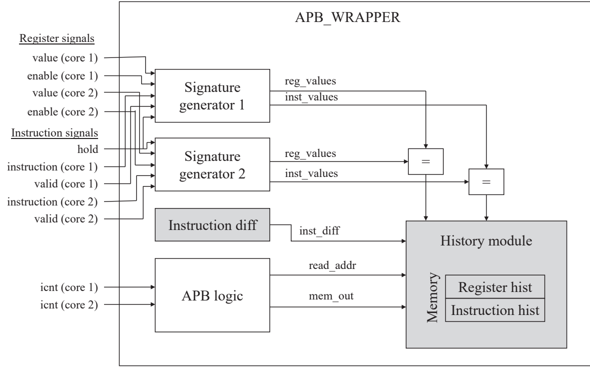


Fig. 4: SafeDM internal block diagram

TABLE I: Taclebench results with different initial staggering.

Staggering	0 nops		100 nops		1000 nops		10000 nops	
Benchmark	Zero stag	No div	Zero stag	No div	Zero stag	No div	Zero stag	No div
binarysearch	0	0	0	0	0	0	0	0
bitcount	212	3	0	0	0	0	0	0
bitonic	0	0	0	0	24	0	0	0
bsort	0	0	83	0	0	0	0	0
complex_upd.	50	0	0	0	0	0	0	0
cosf	0	0	0	0	0	0	0	0
countnegative	0	0	0	0	0	0	0	0
cubic	7268	757	6797	805	0	0	0	0
deg2rad	23	0	0	0	0	0	0	0
fac	11	0	0	0	0	0	0	0
fft	5341	0	0	0	0	0	0	0
filterbank	2136	539	0	0	0	0	0	0
fir2dim	0	0	0	0	0	0	0	0
iir	1903	651	0	0	0	0	0	0
insertsort	611	0	0	0	0	0	0	0
isqrt	0	0	0	0	0	0	0	0
jfdctint	2	0	0	0	0	0	0	0
lms	0	0	0	0	0	0	0	0
ludcmp	2840	574	0	0	0	0	0	0
matrix1	0	0	0	0	0	0	0	0
md5	3140	2050	0	0	0	0	0	0
minver	758	100	0	0	0	0	0	0
pm	67528	655	4884	0	417763	0	0	0
prime	26	0	0	0	0	0	0	0
quicksort	19289	1118	7387	891	1560	0	0	0
rad2deg	0	0	0	0	0	0	0	0
recursion	0	0	0	0	0	0	0	0
sha	1215	0	0	0	0	0	0	0
st	471	26	0	0	0	0	0	0

2) APB logic: This module is designed to communicate with the bus using the APB protocol. The rest of the implementation is agnostic of the bus so that replacing this specific logic would allow for SafeDM to be adapted to another bus protocol easily.

3) Instruction diff: In our experiments, in order to know how far apart the programs being executed are (staggering), we use this module that increases or decreases the count each time core 0 or 1, respectively, commits an instruction. This module is particularly useful to measure the staggering when instruction streams across cores are identical, as it is the case in our evaluation.

4) History module: To collect results on how often there is lack of diversity, we have implemented a history module that stores this information for both instructions and data. In particular, it stores the results in a histogram fashion, where the bin sizes can be configured.

V. Evaluation

A. Evaluation Framework

The SafeDM has been evaluated by synthesizing the MPSoC described in Section IV into a Xilinx Kintex UltraScale KCU105 evaluation kit.

To evaluate SafeDM, we use the TACLe Benchmarks [8], a set of benchmarks of varying types and sizes, specifically designed for the evaluation of critical real-time embedded systems. Those benchmarks are self-contained, meaning that they do not need to read any data from files or peripherals, hence easing their deployment on bare-metal setups.

We assess SafeDM on bare-metal, without any operating system, so that we can exercise control over the experiments and, whenever needed, analyze the timing behavior with tools such as Modelsim, to see in a cycle-by-cycle basis what occurs in the pipeline of the cores and in SafeDM.

B. Experiments

In a real platform deployment, programs will be started by the real-time operating system (RTOS) that, depending on its features, will be able to start them simultaneously in different cores, or with some unintended staggering if, for instance, programs are made to run by a single process that sets them to run sequentially. To account for all these potential scenarios, we have designed several sets of experiments:

- Without staggering: in this case both cores start the (redundant) program at the same time. We load the program in both cores and synchronize them so that they start exactly at the same cycle.
- With staggering: we introduce a controlled delay between one core starting the program and the other one. We use the same synchronization mechanism, but one of the cores will first execute a number of nop (no-operation) instructions before it runs the actual program. We have considered 3 scenarios here depending on the number of nop instructions between cores: 100, 1,000 and 10,000.

SafeDM is agnostic to whether there is some staggering or its magnitude. SafeDM simply assesses whether the indicated cores exhibit diverse behavior, leaving up to the RTOS or the user deciding whether some staggering needs being introduced.

The results of the experiment have been summarized in Table I. We have run each benchmark in a several times as follows. For the experiments starting without staggering, we have run them 4 times. For the experiments starting with staggering, we have run them twice for each of the three staggering values: one with one core starting first, and another one with the other core starting first.

For all experiments, we have reported the number of cycles when the distance between cores (staggering) is 0, as well as the number of cycles when SafeDM reports that there is no diversity. Across the multiple runs, we selected the highest values found, although variations across runs are low. We have considered that to report no diversity, both Data and Instruction signatures need to be equal.

C. Results Analysis

We observe that, as expected, staggering is 0 infrequently, and lack of diversity occurs even less frequently. In particular, programs execute more than 56 millions of instructions on average, and lack of staggering occurs up to 14,500 cycles on average for one setup. Lack of diversity only occurs up to 224 cycles on average for one setup. Hence, even if non considered sources of diversity existed in those cases, meaning that they are false positives, their frequency would still be negligible.

In the experiment with no staggering, most of the benchmarks maintain this difference of 0 instructions for some cycles, but then quickly diverge due to the serialization in the access to the different shared resources of the MPSoC. For example, when both of them get to a load instruction for the first time, they miss in L1 caches, and request access to the bus to access L2/memory. One core is granted access first and get its load served whereas the other is delayed, thus breaking the staggering of 0 cycles. However, even if staggering is zero sporadically, in most of the cases there is some diversity because either instructions are made some different progress in the pipeline, or simply because some of the values operated differ given that redundant threads are created by software, and hence have different address spaces. Therefore, whenever an address is read and/or operated, even if the same operation is performed simultaneously in the other core, the actual address read and/or operated differs, hence bringing some diversity.

Note also that, with the help of Modelsim, we have visually inspected the contents of the pipelines of the cores in multiple cases with staggering and diversity, without staggering but with diversity, and with neither staggering or diversity, to validate that SafeDM behaved as specified.

While SafeDM is agnostic to whether some staggering is imposed or not at software level, we have evaluated scenarios with increasing initial staggering to observe the impact in the results. As shown, generally, when increasing initial staggering, the cycles with zero staggering and no diversity quickly decrease and tend to vanish. There is only one exception: `pm` (pattern matching) benchmark with 1,000 nops staggering. This program experiences a so-called timing anomaly where, by delaying the start of the program, it ends up executing faster. In particular, the core ahead starts sending its store misses to L2 cache. Eventually, the delayed core attempts to do the same but its store operations are kept in its core-local store buffer awaiting for the bus to become idle. However, this allows that multiple stores to the same cache line that would otherwise be sent in multiple transactions, are grouped into a single transaction in the store buffer, hence reducing the latency to write all data. Eventually, and due to the lack of system level effects such as interrupts, for instance, or other tasks scheduled before that could alter the initial state of branch predictors and caches, both `pm` benchmarks, each one in one core, get unluckily synchronized for a while, and only accessing core-local resources (e.g. L1 caches), hence with zero staggering. However, since addresses accessed differ, and pipelines are not fully synchronized (they may hold the same instructions but many times in different stages), there is diversity despite the null staggering.

Overall, we see that SafeDM is an effective module to detect lack of diversity in a non-intrusive manner with program execution, and can be used to build a safety concept on top where, upon lack of diversity – which SafeDM detects as needed – corrective actions are taken.

D. SafeDM Overheads

We have measured the cost of SafeDM in the FPGA. In terms of area, SafeDM, as it would be implemented in a deployment scenario (what is described in Section IV without accounting for the History module that is just added for results gathering), requires 4,000 LUTs, hence a 3.4% overhead only for such a simple MPSoC. Note that such cost would decrease in relative

TABLE II: Classification of non-lockstepped redundant execution techniques for CPUs.

Diversity unaware	Diversity enforced (intrusive)	Diversity monitored (non-intrusive)
[9], [9]–[11], [17], [19], [20] [23], [24], [26]–[30]	[3], [4]	Our work

terms for larger MPSoCs including, for instance, accelerators, additional cache levels, etc. The average power used by the module is even smaller, adding less than 1% of extra power consumption (0.019W on top of over 2W that the baseline MPSoC consumes.)

VI. Related Work

Apart from DCLS, already discussed in Section I, redundancy has been the subject of investigation of many works, mostly considering transient faults in the context of redundant multi-threading [23], [26], or considering both, transient and permanent faults with cross-core redundancy [10], [17], [19]. In some cases, even partial redundancy has been considered [9], [18]. However, none of those works mitigates CCF since no diversity is guaranteed (e.g. due to using the same components in a core, or due to lack of staggering across cores). Diversity is effectively enforced with the solution presented in [4]. Software-only counterparts have also been investigated [11], [20], [24], [27]–[30], yet without guaranteeing diversity. Only a recent work enforces diversity via software [3]. Both, hardware and software-only solutions enforcing diversity are intrusive with the execution of software enforcing some specific staggering. However, as shown in [22], diversity is expected to exist anyway. Hence, SafeDM, our diversity monitor, suffices to provide evidence of existing diversity without being intrusive with execution, as summarized in Table II.

Other works have focused on providing redundancy (without diversity) for GPUs, either with [16], [21], [32], [33] or without [6], [15], [33] hardware support. GPU diversity has also been achieved with [1] and without [2] hardware support. However, GPU solutions cannot be directly ported to CPUs.

Finally, some authors [25] have focused on the recovery opportunities for dual diverse redundant designs rather than on how to guarantee the existence of diversity.

VII. Conclusions

The safety concept for high-integrity systems requires redundant execution with diversity for the most critical tasks. However, lockstepped processors require coupled cores so that half of them are not visible (and usable) at user level. Hence, designs with independent cores that can be used for lockstepped execution opportunistically only when needed are needed for efficiency reasons. Some solutions have been proposed based on software redundancy, and enforcing execution staggering either by software or hardware means. However, those solutions are intrusive with program execution, and pose a number of constraints to the execution since redundant programs must execute identical instruction streams, which is challenging, for instance, for parallel applications where redundant threads may behave differently due to synchronization effects.

To tackle these challenges, this paper presents SafeDM, a hardware Diversity Monitor. SafeDM detects lack of diversity in any cycle so that the safety concept can build on top of SafeDM to deploy corrective actions if diversity is lost. We have integrated SafeDM in a commercial FPGA-based space MPSoC, and shown that SafeDM effectively detects lack of diversity. To allow interested users getting access to SafeDM, it is available as an open-source component in [5].

Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 871467 and by the Spanish Ministry of Science and Innovation under grant PID2019-107255GB-C21/AEI/10.13039/501100011033.

References

[1] S. Alcaide et al. High-integrity gpu designs for critical real-time automotive systems. In *DATE*, 2019.
[2] S. Alcaide et al. Software-only Diverse Redundancy on GPUs for Autonomous Driving Platforms. In *IOLTS*, 2019.
[3] S. Alcaide et al. Software-only based diverse redundancy for asil-d automotive applications on embedded hpc platforms. In *DFT*, 2020.
[4] F. Bas et al. SafeDE: a flexible diversity enforcement hardware module for light-lockstepping. In *IOLTS*, 2021.
[5] BSC - CAOS. SafeDM. https://bsccaos.github.io.
[6] M. Dimitrov et al. Understanding software approaches for gpgpu reliability. 2009.
[7] European Committee for Electrotechnical Standardization. *EN 50126-1. Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*, 2017.
[8] H. Falk et al. TACLeBench: A benchmark collection to support worst-case execution time research. In *WCET Workshop*, 2016.
[9] J. Fu et al. On-demand thread-level fault detection in a concurrent programming environment. In *SAMOS*, 2013.
[10] M Gomaa et al. Transient-fault recovery for chip multiprocessors. In *ISCA*, 2003.
[11] F. Haas et al. Fault-tolerant execution on cots multi-core processors with hardware transactional memory support. In *ARCS*, 2017.
[12] IEC. *IEC 61508. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*, 2010.
[13] Infineon. AURIX Multicore 32-bit Microcontroller Family to Meet Safety and Powertrain Requirements of Upcoming Vehicle Generations, 2012.
[14] International Standards Organization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
[15] S. Jain et al. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *RTAS*, 2019.
[16] H. Jeon et al. Warped-DMR: Light-weight error detection for GPGPU. In *MICRO*, 2012.
[17] C. LaFrieda et al. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *DSN*, 2007.
[18] B. H. Meyer et al. Cost-effective safety and fault localization using distributed temporal redundancy. In *CASES*, 2011.
[19] S. S. Mukherjee et al. Detailed design and evaluation of redundant multithreading alternatives. In *ISCA*, 2002.
[20] H. Mushtaq et al. Efficient software-based fault tolerance approach on multicore platforms. In *DATE*, 2013.
[21] R. Nathan et al. Argus-G: Comprehensive, low-cost error detection for GPGPU cores. *IEEE Computer Architecture Letters*, 2015.
[22] P. Okech et al. Inherent diversity in replicated architectures. *CoRR*, abs/1510.02086, 2015.
[23] S. K. Reinhardt et al. Transient fault detection via simultaneous multithreading. In *ISCA*, 2000.
[24] G. A. Reis et al. SWIFT: Software implemented fault tolerance. In *CGO*, 2005.
[25] P. Reviriego et al. Diverse double modular redundancy: A new direction for soft-error detection and correction. *IEEE Design Test*, 2013.
[26] E. Rotenberg. AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. *FTC*, 1999.
[27] J. D. Scales et al. The design of a practical system for fault-tolerant virtual machines. *Operating Systems Review (ACM)*, 2010.
[28] A Shye et al. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *DSN*, 2007.
[29] A. Shye et al. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 2009.
[30] H. So et al. Expert: Effective and flexible error protection by redundant multithreading. *DATE*, 2018.
[31] STMicroelectronics. SPC570Sx - 32-bit Power Architecture MCU for Automotive Chassis and Safety Applications, 2018.
[32] M. B. Sullivan et al. Swapcodes: Error codes for hardware-software cooperative gpu pipeline error detection. In *MICRO*, 2018.
[33] J. Wadden et al. Real-world design and evaluation of compiler-managed gpu redundant multithreading. In *ISCA*, 2014.
[34] G. Wessman et al. De-RISC: the first RISC-V space-grade platform for safety-critical systems. In *Space Computing Conference (SCC)*, 2021.