SafeDE: A Low-Cost Hardware Solution to Enforce Diverse Redundancy in Multicores

Francisco Bas*†, Sergi Alcaide*†, Guillem Cabo*, Pedro Benedicte*, Jaume Abella*
*Barcelona Supercomputing Center (BSC)
{francisco.basjalon, sergi.alcaide, guillem.cabo, pedro.benedicteillescas, jaume.abella}@bsc.es
† Universitat Politècnica de Catalunya (UPC)

***Abstract*—Failure risk must be tiny in high-integrity systems, such as those in cars, satellites and aircraft. Hence, safety measures must be deployed to avoid a single fault leading to a failure. Redundancy has been often used to address this concern, but it has been proven insufficient if a single fault can cause the same error in all redundant elements, which defeats the purpose of redundancy for error detection. Hence, to avoid this scenario, diversity is implemented along with redundancy, being lockstep execution the most popular diverse redundancy solution for computing cores. However, classic lockstep solutions have non-negligible limitations if implemented in hardware (e.g., half of the cores can only be used for redundant execution and are not even visible at user level), or in software (e.g., the software loop to enforce staggering is long and costs performance).**

This paper tackles the limitations of classic lockstep solutions by providing an extended analysis and evaluation of SafeDE, a Diversity Enforcement hardware module combining the short loop to enforce diversity of hardware solutions, and the non-intrusiveness of software solutions. Hence, cores can operate in lockstep mode efficiently or run independent tasks. In this paper, we present SafeDE and its rationale, its application to N-modular systems, its hardware and software integration, and an evaluation showing its performance and area efficiency, and its behavior in the presence of faults.

***Index Terms*—functional safety, redundancy, diversity, faults**

I. Introduction

High-performance multicores are needed to deliver the increasing performance demands of highly-automated and autonomous systems in automotive, avionics and space, among other domains. However, performance demands come along with safety requirements in those domains, as dictated in corresponding functional safety standards (e.g., ISO26262 in automotive [16]). One such requirement for high-integrity systems consists of avoiding the unacceptable risk (aka unreasonable risk in ISO26262 terminology) of *common cause failures* (CCFs), where a CCF stands for a failure caused by a single fault affecting all redundant components analogously. Note that CCFs relate to the effect, i.e. the failure, not to the source, i.e. the fault. Hence, CCFs could be caused, for instance, by a soft error affecting clock logic of two cores, or by a defect causing voltage sporadic droops.

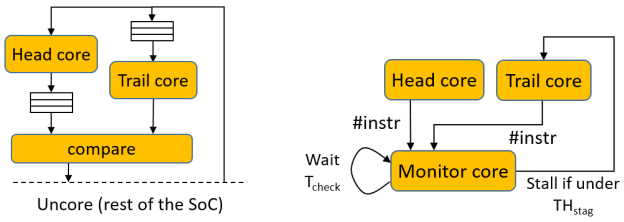
CCFs are generally mitigated by using diverse redundancy [5], so that, even if a fault affects all redundant components (e.g., a voltage droop), since they have different state, potential errors differ and can be detected. Error correction codes (ECC) for storage, Cyclic Redundancy Coding (CRC) for communications, and lockstepping for computation are the most common solutions [2]. The latter – lockstep execution [15], [17], [18], [36] – is the focus of this work.

Lockstep execution (hardware-only) builds upon identical redundant cores running the same software with some staggering – i.e. the head core runs N cycles ahead of the trail one – being one of them the one effectively sending and receiving external signals (e.g., load/store data, interrupts, etc.). The outputs of the other core are just used for comparison for error detection reasons. However, in such a scheme only one core is visible at user level, thus not allowing to use both cores to run independent tasks if lockstep execution is not needed.

A light-weight lockstep execution scheme has been recently proposed to overcome the limitations of regular hardware-only lockstepping [3]. Such solution builds on software redundancy (i.e., the task is run redundantly from the software layers), and on a software monitor enforcing sufficient staggering between redundant processes. However, due to the slow software monitoring loop, staggering is significant (e.g., 100μs - 1ms), which imposes a performance loss as significant as such staggering (as opposed to the hardware-only solution whose staggering is few cycles), and requires an additional core to run the monitor periodically.

This paper presents extensive details and evaluation of SafeDE, a Diversity Enforcement hardware module overcoming the limitations of the previous two schemes. In particular, SafeDE, which we first introduced in [6] and extend in this work, implements the light-weight lockstep execution scheme, but with a hardware monitor (SafeDE module) rather than a software module, hence allowing for few-cycles staggering and not needing any additional core to run any monitor software. Moreover, the integration of SafeDE does not require modifying the cores being monitored, hence being a lowly intrusive solution. The main contributions of this work are as follows:

- We present SafeDE, the monitoring module to enable lowly-intrusive diverse redundancy with a flexible scheme that can be enabled or disabled at convenience. We further detail how to extend it to arbitrary redundancy (e.g., with 3 or more cores instead of only 2).
- We implement it at VHDL in a SoC for the space domain based on CAES Gaisler's NOEL-V cores [11], and provide details on its bare metal and Linux integrations.



(a) Hardware-only (b) Software-only

Fig. 1. Schematic of the existing lockstep schemes.

- We assess SafeDE's performance and area overheads, as well as effectiveness through a fault injection campaign.

The rest of the paper is organized as follows. Background is presented in Section II. SafeDE and its hardware and software integrations are introduced in Section III. Section IV evaluates SafeDE. Related work is presented in Section V. Section VI summarizes this work.

II. Background

This section reviews the main concepts and approaches related to lockstep execution relevant for this work.

A. Redundancy, Diversity and Sphere of Replication

The design, as well as the verification and validation (V&V) stages for safety-related systems arguably remove unreasonable risk of any kind of systematic fault, either software or hardware related. However, random hardware faults are unavoidable in nature, so they must be managed with suitable safety measures, being diverse redundancy a mandatory safety measure for the highest Safety Integrity Levels (SIL for short).

Diverse redundancy can be realized by using diverse hardware and/or software, but these approaches may double part of the design, and V&V costs. Alternatively, diverse redundancy can also be realized by executing the same software in identical hardware, but with some staggering, hence guaranteeing that replicas hold different state at any time instant so that any common fault will lead to diverse errors (if any). This approach, if realized with two cores, is referred to as Dual Core LockStepping (DCLS) and is used in several commercial processors [15], [36], [38].

The sphere of replication determines what outputs of the replicas are compared to detect errors. In the case of hardware-based lockstep, such sphere includes only a core, so any off-core activity (data fetch or store beyond in-core caches, I/O activity, interrupts, etc.) is compared across cores for error detection. In the case of light-weight lockstep execution, it is limited to programs or code regions without I/O, and detects errors by comparing data outputs at the end of the execution.

B. Lockstep Schemes

Tight hardware-based lockstep execution. This approach, implemented in processors such as, for instance, the Infineon AURIX family [15], uses two physical cores (head and trail cores) out of which only one (e.g., head core) is visible at user level and the other one (e.g., trail core) is a shadow core. They perform *exactly* the same activity but shifted by N cycles, so that the state of the trail core matches that of the head core N cycles before. External requests (data load/store, interrupts, etc.) are compared before being exposed externally, hence needing some buffering to store head requests during N cycles. Analogously, responses are delivered immediately to the head core when they arrive, but queued during N cycles before being delivered to the trail core, which also needs some buffering. Such scheme is depicted in Figure 1(a). Note that staggering is typically low (e.g., 2 or 3 cycles) to keep buffering overheads low.

Light-weight software-based lockstep execution. In this approach, redundancy is created at software level by running a given program (task) twice on different cores [3]. Those task replicas run along with a monitor thread, which is deployed in a third core, to enforce some staggering across replicas so that one of them becomes the head and the other one the trail task (and core). Note that the monitor itself is unprotected, so this approach requires the execution of the monitor to occur on a core with hardware-based lockstepping, either in the same or another chip. In detail, the operation of this scheme is as follows (see Figure 1(b)): the monitor schedules redundant processes (replicas) in two different cores, but only allows the head core to make progress. The monitor collects the number of instructions executed by each core ($\#instr$ in the figure), and only allows the trail core to execute its task if $\#instr_{head} - \#instr_{trail}$, namely the staggering in terms of number of instructions, exceeds a given threshold TH_{stag}. Such condition is checked by the monitor every T_{check} cycles to decide whether the trail core is allowed to proceed during the following interval. Only when the head core finishes its execution, the trail core is allowed to run unrestrictedly. Results from both executions are compared when both cores complete their execution. Note that between two consecutive checks of the staggering, the trail core could execute up to $T_{check} \cdot CommitWidth$ instructions, where $CommitWidth$ stands for the maximum number of instructions that can be retired per cycle. TH_{stag} must be strictly higher than $T_{check} \cdot CommitWidth$. As shown in [3], due to the software overheads to collect $\#instr_{head}$ and $\#instr_{trail}$, and to stall a process – if needed, TH_{stag} must be a number of instructions taking at least around 100μs to run.

III. SafeDE: a Diversity Enforcement hardware module

This section presents the architecture of SafeDE, its features and limitations, its extension towards N-modular redundancy, and its implementation and integration (both hardware and software) details.

A. SafeDE Architecture

SafeDE builds on the light-weight lockstepping concept, which has only been implemented in software so far [3], with the aim of preserving its advantages and removing its main limitations, i.e., the need for an extra core to run the monitor and a long feedback loop imposing a large staggering. SafeDE is architected as a tiny component connected to the two

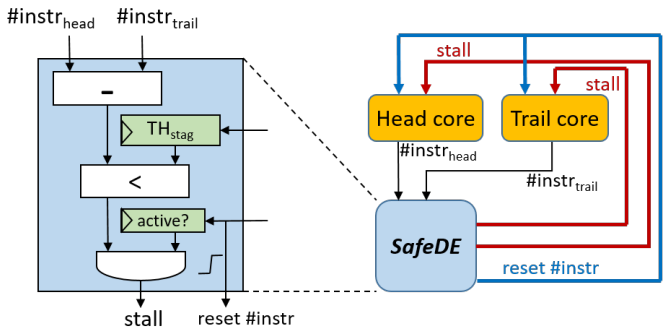


Fig. 2. SafeDE architecture.

monitored cores, as shown in Figure 2. SafeDE collects the instruction counts from the two cores, namely $\#instr_{head}$ and $\#instr_{trail}$, and generates the stall signal for the trail core. As for the software-only solution, SafeDE checks whether the head core is at least TH_{stag} instructions ahead of the trail one. If this is not the case, the stall signal for the trail core is raised, which stalls its pipeline by stalling one or several of its stages (e.g., stalling the commit stage).

SafeDE parameters. The configuration registers of SafeDE are TH_{stag}, $active$, $CritSec1$ and $CritSec2$, and they operate as follows:

- TH_{stag} corresponds to the minimum staggering (in terms of number of instructions) to be enforced between the head and trail cores. Typically, it has a very low value (e.g., 10 instructions), hence implementing a staggering distance much smaller than that of software-only solutions, and comparable to that of tight hardware-based lockstep execution.
- $active$ determines whether SafeDE is active. If this signal is reset, SafeDE is completely neutral since it can never stall that trail core.
- $CritSec1$ ($CritSec2$) is set by the head (trail) core when it enters the code region needing lockstep, and reset when leaving it. Hence, lockstepping must be enforced when $CritSec1$ and $CritSec2$ are both set, as this indicates that both cores are executing the code region needing lockstep execution.

SafeDE operation. While $active = 0$, SafeDE is inactive. Eventually, TH_{stag} is programmed and $active$ is set, hence activating SafeDE. Activating SafeDE automatically resets $CritSec1$ and $CritSec2$ keeping SafeDE ready but innocuous until $CritSec2$ is activated. Eventually, one of the two cores activates its $CritSec$ register, becomes the head core, and its instruction counter ($\#instr_{head}$) is reset and starts counting. Whenever the other core sets its $CritSec$ register, it becomes the trail core, and its instruction counter ($\#instr_{trail}$) is also reset. If the head core is not ahead TH_{stag} instructions of the trail core, SafeDE sets the stall signal for the trail core. Note that, since any of the two cores could be the trail core depending on which one sets its $CritSec$ first, the $stall$ signal exists for both cores. Note that, if the staggering is too low when the trail core sets its $CritSec$ ($\#instr_{head} - \#instr_{trail} < TH_{stag}$), the $stall$ signal for the trail core will be raised immediately. Whenever the staggering is enough, the trail core is allowed to resume its execution. Note that SafeDE checks every cycle whether the staggering is enough. This allows using tiny staggering (TH_{stag}) values, in contrast with the large staggering needed by the software-only solution. Moreover, SafeDE controls this condition, hence not needing any additional core to run any monitor software. Also note that by performing such check every cycle, negligible switching power is induced since $\#instr_{head}$ and $\#instr_{trail}$ barely change. Eventually, the head core reaches the end of its protected code region and resets its $CritSec$ register. At that point, SafeDE becomes innocuous again not raising any stall signal, hence letting the trail core reaching also the end of its protected region.

Software process. At software level, end users need to configure and set SafeDE active with the corresponding driver, and typically, use an API to schedule both redundant processes to the corresponding cores managing $CritSec$ registers accordingly. Those software components are described later in this section in the context of bare metal and Linux integrations.

B. Features and Limitations Analysis

This section presents the key features and limitations of SafeDE, and how they compare against the software-only solution [3].

1) SafeDE features:

- **Low cost**. SafeDE is a tiny hardware module implementing light-weight lockstep execution that avoids the need for an extra core to run a monitor at specific (tight) time intervals, as opposed to the software-only solution.
- **Low staggering**. By controlling the feedback loop at hardware level, it is checked every cycle and hence, staggering can be kept to a minimum (e.g., 10 or 20 instructions). Note that the software-only solution needs a staggering value of many thousands of instructions to reach a staggering above 100μs, as discussed before.
- **Flexibility**. SafeDE can be easily enabled and disabled. Hence, the main overheads relate to the creation of the redundant processes at software level, as needed in the context of light-weight lockstep execution, but not to the actual implementation of SafeDE, which does not impose further limitations.
- **Low intrusiveness**. SafeDE needs some signals to be exported from cores, such as those to read and reset instruction counts, and the pipeline stalling signal. These modifications are much lighter than those needed in the case of tight lockstep execution. While the software-only solution does not require any hardware change, as opposed to SafeDE, it may need modifying the operating system to enable the management of the instruction counts remotely from other cores. SafeDE does not need any such operating system modification.

2) SafeDE limitations:

- **Non-null intrusiveness**. While hardware modifications needed by SafeDE are light, it needs hardware support, and hence, cannot be used on COTS multicores, as opposed to the software-only solution.

- **Limited applicability**. Light-weight lockstepping relies on the redundant processes executing identical instruction streams to guarantee the effectiveness of the approach. While this is usually the case, it precludes the use of this scheme for programs whose control path is non-deterministic (e.g., based on random choices independent across redundant processes). Also, SafeDE may not be used for parallel programs if the number of instructions of any thread may vary depending on the order in which they get a specific lock, since this could make redundant threads execute a different number of instructions. Also, since light-weight lockstepping exposes all activity redundantly, it should not be used along with I/O operations that may change the functionality of the system if repeated. In any case, note that those limitations relate to the light-weight lockstepping scheme rather than to SafeDE, and hence, affect also the software-only solution.
- **Limited diversity**. By using two cores with staggered execution, SafeDE, as well as the software-only solution, provide physical and time diversity. However, if CCFs can be induced by the core design (e.g., physically weak gates identical in both cores), other types of diversity, such as layout diversity, are needed, and such diversity cannot be reached without appropriate (and intrusive) hardware modifications.
- **SafeDE hardening**. SafeDE must be hardened to mitigate the risk of a single fault in SafeDE leading to a failure. Alternatively, SafeDE can also be implemented with physical diverse redundancy, as for tight lockstepping replicating the scheme in Figure 1(a), but applied to SafeDE instead of to the cores.

3) Scope of applicability: As discussed before, light-weight lockstepping has limited applicability, hence affecting both SafeDE and the software-only solution. Therefore, SafeDE can only be used for some code regions rather than for the full program. Code regions exercising light-weight lockstepping limitations must be run on cores implementing tight lockstepping. Hence, at least two cores must implement tight hardware-based lockstepping. However, compute intensive parts of the code can be managed with SafeDE. This opens the door to using deployments with two tightly lockstepped cores (e.g., to run I/O-related code regions) and the remaining cores building on SafeDE. For instance, an 8-core multicore would offer 7 user-visible cores by shadowing only one of the cores for tight lockstepping. Hence, one could run a varying number of lockstepped and non-lockstepped tasks, ranging from 4 lockstepped to 1 lockstepped and 6 non-lockstepped. Instead, if tight lockstepping is used for all cores, then only 4 cores are visible at user level regardless of whether tasks need such support.

C. Towards N-modular Redundancy

Note that, while we implement and assess SafeDE in the context of dual-modular redundancy (DMR), it can be easily extended to N-modular redundancy, which may be needed for $N > 2$ in some domains such as, for instance, avionics or medical, where 3-modular or even 5-modular redundancy may be needed.

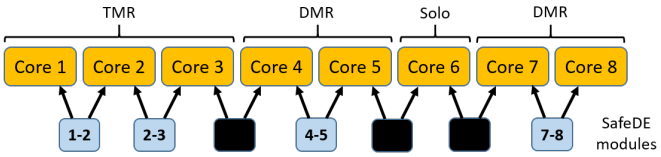


Fig. 3. N-modular redundancy scheme example with SafeDE.

Regardless of the value of N, SafeDE must manage the N cores so that one of them is the head core, one of them is the trail core, and the remaining ones inherit both, head and trail behavior. For instance, in the case of triple-modular redundancy (TMR), core 1 is the head core w.r.t. core 2, core 2 is the trail core w.r.t. core 1 and the head core w.r.t. core 3, and core 3 is the trail core w.r.t. core 2. Therefore, given N cores, SafeDE must activate the $stall_i$ signal for $core_i$ if the following condition holds: $\#instr_{i-1} - \#instr_i < TH_{stag}$, where $\#instr_{i-1}$ and $\#instr_i$ correspond to the instruction counts of $core_{i-1}$ and $core_i$ respectively, and $1 \leq i < N$.

In the context of N-modular redundancy, note that the operation with $CritSec$ is analogous to DMR, hence cores take the role of $core_i$ when they are the i^{th} core setting their corresponding $CritSec$ register. A core $core_i$ cannot be further stalled when $core_{i-1}$ leaves its critical region by resetting its $CritSec$ register. However, the remaining cores (from $core_{i+1}$ to $core_N$) can still be stalled if their staggering becomes too low w.r.t. their respective head cores.

One potential realization of the overall concept with flexible N-modular redundancy could impose that the cores in the multicore are physically paired for SafeDE operation so that $core_1$ is the head of $core_2$, $core_2$ of $core_3$, and so on and so forth. Then, $N-1$ SafeDE modules are deployed connecting each pair of consecutive cores. Finally, by activating appropriate SafeDE modules, one could have any combination of N-modular redundancy at will. For instance, Figure 3 illustrates a case where an 8-core multicore uses 3 cores with TMR (cores 1-3), two pairs of 2 cores with DMR (cores 4-5 and 7-8), and 1 core running independently (core 6). This is achieved by enabling specific SafeDE modules (light-colored ones) and keeping others inactive (black ones).

D. Implementation and Integration

As a proof of concept, we have integrated SafeDE in an industrial space MultiProcessor System on Chip (MPSoC) based on CAES Gaisler RISC-V NOEL-V cores [11]. This platform consists of a consistent set of reusable VHDL IP cores by Gaisler whose main interface is a set of common on-chip buses. Those buses implement the standard AMBA 2.0, and SafeDE has been implemented in VHDL as another IP core compatible with such bus interface.

1) System on Chip: SafeDE is integrated and evaluated in a specific MPSoC instance including 2 Gaisler's NOEL-V 64-bit cores. Those cores are dual-issue, implement the RISC-V Instruction Set Architecture (ISA), include 7 pipeline stages, and local L1 data and instruction caches. Cores are connected among them, and to a shared L2 cache and the memory subsystem through a 128-bit AMBA Advanced High-performance

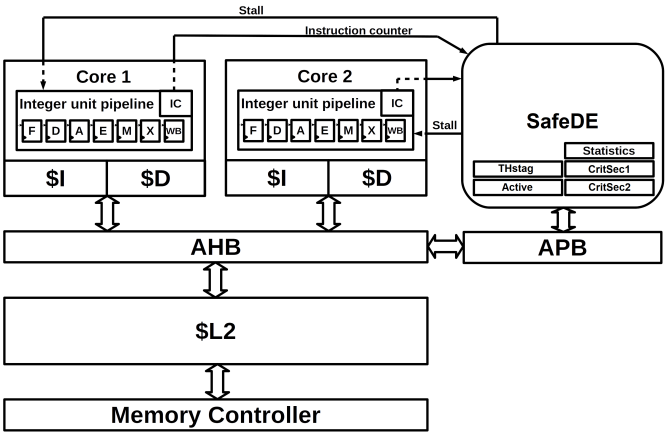


Fig. 4. High-level representation of SafeDE integrated into the system.

Bus (AHB). Components requiring low bandwidth, such as for instance, SafeDE, are connected instead through an AMBA Advanced Peripheral Bus (APB).

2) Hardware integration: SafeDE interface builds on the standard APB interface to make it highly portable. In particular, SafeDE is an APB slave in the SoC. SafeDE is also directly connected to the cores to collect their instruction counters, which are mapped to SafeDE as inputs, to stall the trail core whenever needed. The instruction counters determine the number of instructions executed by each core and used to compute the real staggering across head and trail cores. Regarding the stall signal produced by SafeDE, it is ORed with an internal core signal in charge of freezing the pipeline by not allowing register values to be updated. Overall, the only modifications needed in the cores include exporting the instruction counter[1] to SafeDE, and placing an OR gate appropriately to stall the pipeline whenever needed. The SoC including SafeDE is depicted in Figure 4 for completeness.

3) Configuration and operation: SafeDE configuration registers, namely TH_{stag}, $active$, $CritSec1$ and $CritSec2$, are mapped to specific memory addresses. Their operation is detailed in Section III-A. Apart from those functional registers, SafeDE also includes several statistics registers collecting information such as the maximum and minimum staggering observed, number of stall signal activations (i.e., how many times the $stall$ signal is raised), number of stall cycles for the trail core, executed instructions of each core, etc. Since SafeDE has an APB interface and memory-mapped registers, all its registers can be read and written with regular load and store operations.

E. Software Integration

We have developed the software interface to control the SafeDE IP. We considered two scenarios where SafeDE can be deployed: bare-metal systems, and systems with an operating system (Linux in our case). Hence, we implemented two software integrations. A C library, which is enough for a bare metal setup, and a driver for a Linux setup.

[1]Note that the instruction counter, along with the cycle counter, are the main counters in any processor and are generally implemented in any SoC.

1) Bare Metal Integration: To integrate SafeDE software in a bare metal setup we have created an API that consists of a C library to configure the internal SafeDE registers. The API contains functions to enable and reset SafeDE, configure the staggering, indicate when the critical section starts and finishes for each core, and gather the execution results (statistics). At run time, SafeDE must be initialized first. Such initialization includes configuring the staggering, and setting the reset and enable bits using the API. Whenever a core starts or finishes the critical region, a function from the API has to be executed to notify the SafeDE module. Once the task has finished, the statistics from the safe execution can be retrieved, again, using a function provided in the API library.

2) Linux Integration: In Linux, it is not possible to access directly the memory positions mapped to SafeDE registers from the user space. To allow the user to access SafeDE registers, we need to access the kernel space using a driver. We have programmed a Linux driver and a C library (API) that allow the user to communicate with the SafeDE module.

For the Linux API, we use a set of functions analogous to those for the bare metal integration, but this time managed through the driver. The API calls communicate with the driver by writing the commands into a Linux device file, a special file in the Linux file system created during the driver initialization process. Later, the driver reads the commands from the device file and modifies the SafeDE registers accordingly.

As stated before, one of the SafeDE limitations is that both cores have to execute the exact same instructions. However, in a preemptive operating system, a process in the critical region may be preempted. Since the critical region is active, instructions executed in that core would be counted as part of the critical region, hence de-synchronizing staggering in an arbitrary manner. Deactivating the corresponding $CritSec$ register would not be a better solution since it would not occur immediately, hence altering the instruction count anyway, and would lead to the virtual finalization of the lockstep execution. Some guidelines to avoid this situation are as follows:

1) In Linux, each process has a mask indicating in which subset of the cores that process can be executed (a.k.a. process affinity). The particular case in which this subset is only one core, is named binding. In our strategy, we bind the two critical processes into two cores and modify all the other processes (including kernel processes) affinity to avoid preemption of the critical processes.
2) In order to give the highest priority to the two redundant critical tasks so that they start immediately and unnecessary stalls do not occur, we must enqueue those tasks into the highest priority queue, which is SCHED_FIFO queue. We perform that employing the Linux system call *sched_setscheduler()*.

Note that, if a real-time operating system (RTOS) for safety-related systems was used instead of Linux (e.g., fentISS' XtratuM, SYSGO's PikeOS, RTEMS), such problem would not exist and simpler mechanisms related to task scheduling and priority assignment would suffice to guarantee the non-preemptive execution of light-weight lockstepped processes.

IV. Evaluation

This section assesses SafeDE in the context of the aforementioned MPSoC. For that purpose, we synthesized the MPSoC including SafeDE into a Xilinx Kintex UltraScale KCU105 evaluation kit.

A. Functional Validation

SafeDE implementation in the FPGA is validated with usual VHDL testbenches. To validate its correct operation with representative programs, we have used a bare metal setup and have ported the TACLeBench benchmark suite [8]. TACLeBench suite is a set of self-contained and open source benchmarks for real-time systems. These benchmarks have their inputs hardcoded along with the source code, thus easing their porting onto a bare metal setup. Their execution times are typically between few hundreds and few millions of cycles, hence easing debugging and the analysis of unexpected results during simulation or emulation. In our evaluation, we use a staggering of 20 cycles ($TH_{stag} = 20$), and recorded the lowest staggering observed during the execution building on the SafeDE statistics register recording this value. Our experiments confirm that the real staggering experienced across all executions of all benchmarks has been never below 20 cycles, therefore confirming the efficacy of SafeDE to preserve a sufficient staggering.

B. Fault injection

We have performed a simulation-based fault injection campaign to evaluate the CCF detection capabilities of the aforementioned platform integrating SafeDE. We have added non-synthesizable logic into the VHDL files of the CAES Gaisler NOEL-V cores to inject faults in four different locations of the pipeline:

- **ALU:** The fault is injected in a random bit, of one of the inputs of one of the two ALUs randomly selected (both the input and the ALU).
- **Late ALU:** Analogous to the previous one, but for late ALUs in the pipeline instead of regular ones.
- **Memory:** In the memory stage, the fault is injected in a random bit of the data to be written in the cache.
- **Write back:** In the write back stage, the fault is injected in a random bit in a random write port of the register file.

In our fault injection campaign, three fault models have been considered: stuck-at-0, stuck-at-1, and bit flip. They set the value of the bit selected for injection to 0, 1 and its logical complementary value respectively. We have enforced the fault during 1 cycle only, but since most of the faults became quickly masked, we have repeated the experiments making faults last 10 cycles instead. In the case of bit flip, we flip the bit and keep such value for 10 cycles regardless of whether the bit is modified.

Since simulations are slow, we have performed our fault injection campaign in one specific benchmark: `FAC`. In principle, no CCFs should escape SafeDE itself, and we expected to find only CCFs that cannot be detected by light-weight and/or tight lockstepping, regardless of how this is implemented. As shown next, all those insights are already observed with this benchmark. `FAC` computes the factorial of several numbers and accumulates their results. The benchmark code is replicated in memory and executed in both cores simultaneously with SafeDE active and imposing a minimum staggering of 20 instructions. Faults are injected in a cycle selected randomly in the period where both redundant tasks are active, i.e. since the trail core starts until the head core finishes. The fault location, namely ALU, late ALU, memory or write back, is randomly selected. The fault is injected in the same location and cycle in both cores to emulate a CCF[2].

TABLE I
Fault injection results classified by fault model.

	Fault Model	Timeout	Crash	SW detected	I. Mem SDC	DUE	Masked	Undetected
1 cycle	Stuck-at-0	5	0	15	0	0	3980	0
	Stuck-at-1	32	4	32	0	0	3932	0
	Bit flip	37	1	38	0	0	3924	0
10 cycles	Stuck-at-0	34	0	45	0	0	3921	0
	Stuck-at-1	342	31	89	0	0	3537	2
	Bit flip	345	45	117	0	0	3489	4

We have analyzed the results of each simulation considering the comparison of results with the golden run outputs, memory dump comparison, and monitoring the AHB transmission during the simulation (to check for invalid memory accesses).

The possible outcomes considered are based on the categories in [21], which we extend conveniently to consider additional categories relevant for CCFs:

- **Timeout:** The simulation exceeds 90 seconds. Note that the fault-free simulation takes 35 seconds.
- **Crash:** The simulation process is terminated abnormally.
- **Software detected:** Error detected by software comparison between the results of both cores, including differences across data in memory.
- **Identical memory SDC:** Both executions produce the same memory corruptions (SDC). They are identified in the post-analysis using the memory dumps.
- **DUE:** One of the cores wrote outside its memory limits causing a non recoverable error. It is detected with the AHB transaction monitoring.
- **Masked:** Outputs were identical to the golden run outputs, including memory dumps and AHB transmissions.
- **Undetected error:** The software comparison did not raise any error but the result is different from the golden run.

Note that, only *undetected error* and *identical memory SDC* categories correspond to CCFs.

Table I summarizes the fault injection results. For each fault model and fault duration (either 1 or 10 cycles), we performed 4,000 simulations. As shown, out of the 24,000 simulations performed, some simulations led to *timeout* or *crash*, hence

[2]Note, however, that, by having different core states, some faults affecting both cores (e.g., voltage droops) could easily lead to different faults in the cores (e.g., different effects in different locations), hence reducing the probability of a CCF.

making the error easily detectable. The number of such cases is higher for a larger fault duration. Some other faults led to errors detected by comparing the outputs, including data in memory (*SW detected*). Some simulations produced errors in memory, but none of those cases had the two memory dumps equal. Hence, *identical memory SDC* – one of the two CCF categories – is always zero. During the injection campaign, no DUE appeared. Still, some injections that were classified in the timeout, crash or error detected categories also wrote out of their memory limits. Most of the faults injected were masked. This is particularly true if the fault duration is short (1 cycle). Finally, only in six of the simulations the error produced could not be detected by the software comparison. In all occasions, the faults were injected in the write back stage. Since these six experiments would correspond to a CCF despite using lockstepping, we analyze them in detail.

The benchmark `FAC` calculates and accumulates the factorials of the numbers from 12 to 0 using recursion. As shown in Figure 5, the assembly code obtained after compilation has three main sections: initialization, an outer loop to accumulate the factorials and an inner loop to derive the factorials. The fault injected (a bit flip) in the head core is applied while the factorial of 8 is being processed in the inner loop. The least significant bit of the second write port of the register file becomes 1 instead of 0, and the value store is erroneous for up to 10 cycles. However, those values are read through a bypass in the inner loop instead of from the register file, so no impact is observed in the inner loop. In the last iteration of the inner loop, the result of the factorial is stored in register `a4` (line 8) with an erroneous value (due to the duration of the fault). Later, the register `a4` is read as an operand for `addw` instruction in line 17, which accumulates all the calculated factorials. Therefore, this error is propagated to the output. Particularly, the erroneous result contains one bit flip in the least significant bit with respect to the correct result.

In the trail core, the fault is injected while the core is executing the outer loop. The fault affects several instructions, but all the errors except one are masked because they are either overwritten or not read since they are forwarded like in the head core. The error not masked is produced in line 17 when the result of the addition is written back to the `a6` register, which stores the accumulation of the factorials. Again, the final value contains one bit flip with respect to the correct output in the least significant bit. Therefore, even though the fault affects both cores differently, in the end, it produces the same bit flip in the accumulated register (`a6`), which stores the output value. Thus, the software is not capable of detecting the error, not because of a malfunctioning of SafeDE, but because of the semantics of the benchmark `FAC`. In fact, even if we used tight lockstepping, external core activity would be identical for both cores and no error would be detected.

We content that, despite we could produce this apparent CCF in our fault injection campaign, such effect would be very unlikely to occur in practice since both cores have different states when the fault occurs, and this should lead to heterogeneous electric impact, hence causing heterogeneous errors (e.g., affecting only one core, or affecting different bits or locations of both cores).

```
00000000400010c0 <fac_main>:
   ...
   Initialization code
   ...
1  li      a3,0
2  addw    a6,a6,a4
3  blt     a5,a2,40001150 <fac_main+0x90>
-------------------------------------------------------
4  mv      a5,a2
5  li      a4,1
-------------------------------------------------------
6  mv      a3,a5
7  addiw   a5,a5,-1
8  mulw    a4,a3,a4
9  bnez    a5,4000111c <fac_main+0x5c>
-------------------------------------------------------
10 lw      a5,0(a7)
11 addiw   a1,a1,2
12 addw    a1,a1,a0
13 mv      a0,a2
14 addiw   a2,a0,1
15 sext.w  a5,a5
16 li      a3,1
17 addw    a6,a6,a4
18 bge     a5,a2,40001114 <fac_main+0x54>
-------------------------------------------------------
19 sw      a6,0(t3)
20 beqz    a3,40001160 <fac_main+0xa0>
21 sw      a1,0(t1)
22 ret
23 ret
24 ret
```

Inner loop

Outer loop

Fig. 5. Excerpt of the assembly code of the benchmark `FAC`.

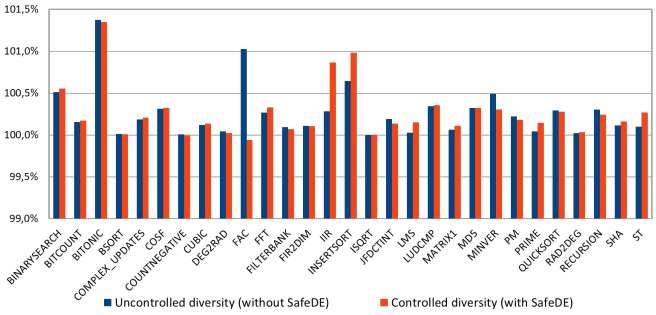


Fig. 6. Average execution time of different TACLeBench benchmarks normalized w.r.t. their execution time in isolation.

C. Execution time overhead

We have measured the performance of the TACLeBench in three different scenarios: (1) *Isolation*, where only one core runs the benchmark; (2) *Redundancy without diversity*, where two cores run the benchmark without enforcing any staggering; and (3) *Redundancy with diversity* enforced by SafeDE with $TH_{stag} = 20$. Such TH_{stag} is set large enough so that both cores cannot hold any common instruction in their pipelines to avoid a case where such instruction causes all the activity and hence, a CCF is possible. Since the pipeline has 7 stages, and each stage can hold up to 2 instructions, any value above 14 suffices to guarantee that instructions executed across cores at any time differ.

To discount the effect of effects such as DRAM refreshes and other minor performance variations, we run each benchmark 1,000 times and report average cycle counts. In any case, variations observed are up to few tens of cycles across runs.

As shown in Figure 6, performance degrades only by 0.3%

TABLE II
CLASSIFICATION OF REDUNDANT EXECUTION TECHNIQUES.

Strategy	Target	Diversity	Approaches
HW	CPU	Yes (tight)	[18], [24], [36]
		Yes (light)	**Our approach (low staggering)**
		No	[9], [10], [22], [25], [28], [31]
	GPU	Partially	[1]
		No	[20], [23], [27], [37]
SW-Only	CPU	Yes (light)	[3] (high staggering)
		No	[12], [29], [32], [33], [4], [26], [34], [35]
	GPU	Partially	[2]
		No	[7], [19], [39], [40]

on average (up to 1.3% for `BITONIC`) w.r.t. isolation runs, and 0.003%, so ≈0% (up to 0.6% for `IIR`) w.r.t. redundancy without diversity. Performance variations across runs, and even marginal performance gains with SafeDE relate to the initial state of the branch predictor, and to memory alignment of the binaries impacting the instruction cache behavior, since in the case of SafeDE we execute additional instructions to configure SafeDE. The relative effect of those variations can be larger for short programs, as it is the case for `FAC`, with variations in the range of 1%. In fact, those variations have a higher impact than the tiny performance degradation of SafeDE w.r.t. redundancy without diversity.

Overall, performance overheads are tiny due the very low staggering threshold needed by SafeDE (20 cycles in our case), which is far lower than that for the software-only solution (e.g., 100μs or more) [3].

D. Hardware costs

We have used the Vivado 2018.1 Toolchain to synthesize our MPSoC for the Xilinx UltraScale KCU105 FPGA. SafeDE required 261 LUTs and 417 registers, whereas the whole SoC required approximately 114,000 LUTs and 74,000 registers, and each core individually 38,000 LUTs and 17,000 registers. Hence, SafeDE has negligible hardware costs (0.23% of the LUTs and 0.56% of the registers of the entire SoC). Those results could be further improved by dropping the statistics additions of SafeDE.

V. RELATED WORK

Some works investigate redundancy for CPUs, yet without diversity including Redundant Multi-Threading in a core [28], [31], across different cores [10], [22], [25], and providing only partial redundancy [9], [24]. Those works lack diversity by reexecuting on the same hardware, or using different cores without staggering. Software-only solutions for CPUs build on the compiler to enforce redundancy creating monitoring threads or resorting to transactional memory [12], [26], [29], [33]–[35]. Unfortunately, at least CCFs affecting redundant computing units are not covered by those solutions.

Some works targeting GPUs provide hardware support for redundancy [20], [27], [37], [40] or software-only support [7], [19], [40], but they fail to guarantee diversity. Diverse redundancy on GPUs has been proven doable with [1] and without hardware support [2]. Note that those solutions are GPU-specific, so cannot be applied to CPUs.

As discussed before, some processors implement tight lockstep in the form of DMR (Infineon AURIX processor family [15], ST Microelectronics SPC56XL70 [36]), or TMR (Arm Cortex-R5 based designs [17], [18]). Other works focus on how to expose latent errors to shorten detection time [14], and how to enhance error recovery [13]. Finally, Reviriego et al. [30] focus on how to perform recovery efficiently for DCLS (diverse DMR) designs. However, as detailed before, tight lockstepping halves (for DMR) the number of user-visible cores, which impacts flexibility and, ultimately, performance.

Light-weight lockstepping for CPUs has been addressed with software-only solutions so far [3], but staggering is significant and an additional core is needed to run the monitor. Our solution, SafeDE, drastically reduces staggering and removes the need for an additional core at the expense of introducing a lowly intrusive hardware module. Table II summarizes related work and puts SafeDE in context.

VI. CONCLUSIONS

Safety-related systems must implement diverse redundancy for the highest integrity functionalities to avoid CCFs. Tight lockstepping is the *de facto* solution for CPUs, but it makes half of the cores not visible at user level, so significant performance is lost when none or few high-integrity tasks are run. Light-weight lockstepping has been proposed recently to overcome such limitation and gain flexibility. However, existing solutions build on slow software feedback loops that impose large staggering and require an additional core to run the monitoring process.

This paper introduces SafeDE, a tiny module implementing light-weight lockstepping with a very short feedback loop (e.g., 20 cycles), hence causing negligible performance impact, and not needing any additional core since SafeDE itself controls the feedback loop. Our results show that SafeDE incurs both negligible performance degradation (0.5% on average) and hardware overheads (≈0.5% extra SoC area) w.r.t. to a non-redundant industrial SoC, and effectively captures all CCFs that would also be captured by tight lockstep execution.

ACKNOWLEDGEMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 871467. This work has also been partially supported 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*, pages 1–4, 2020.
[4] M. S. Alhakeem et al. A Framework for Adaptive Software-Based Reliability in COTS Many-Core Processors. In *ARCS*, 2015.
[5] A. Avizienis and J.P.J. Kelly. Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80, 1984.

[6] F. Bas et al. SafeDE: a flexible diversity enforcement hardware module for light-lockstepping. In *IOLTS*, 2021.

[7] M. Dimitrov et al. Understanding software approaches for gpgpu reliability. 2009.

[8] H. Falk et al. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *WCET workshop*.

[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] H2020 SELENE consortium. SELENE RISC-V open source hardware platform. https://gitlab.com/selene-riscv-platform, 2021.

[12] F. Haas et al. Fault-tolerant execution on cots multi-core processors with hardware transactional memory support. In *ARCS*, 2017.

[13] C. Hernandez and J. Abella. Low-cost checkpointing in automotive safety-relevant systems. In *DATE*, 2015.

[14] C. Hernandez and J. Abella. Timely Error Detection for Effective Recovery in Light-Lockstep Automotive Systems. *IEEE TCAD*, 2015.

[15] Infineon. AURIX Multicore 32-bit Microcontroller Family to Meet Safety and Powertrain Requirements of Upcoming Vehicle Generations, 2012.

[16] International Standards Organization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[17] X. Iturbe et al. Addressing Functional Safety Challenges in Autonomous Vehicles with the Arm Triple Core Lock-Step (TCLS) Architecture. *IEEE Design and Test*, 2018.

[18] X. Iturbe et al. The Arm triple core lock-step (TCLS) processor. *ACM Transactions on Computer Systems*, 2019.

[19] S. Jain et al. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *RTAS*, 2019.

[20] H. Jeon et al. Warped-DMR: Light-weight error detection for GPGPU. In *MICRO*, 2012.

[21] M. Kaliorakis et al. Merlin: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment. *SIGARCH Comput. Archit. News*, 45(2):241–254, 2017.

[22] C. LaFrieda et al. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *DSN*, 2007.

[23] A. Mahmoud et al. Optimizing software-directed instruction replication for gpu error detection. In *SC*, 2018.

[24] B. H. Meyer et al. Cost-effective safety and fault localization using distributed temporal redundancy. In *CASES*, 2011.

[25] S. S. Mukherjee et al. Detailed design and evaluation of redundant multithreading alternatives. In *ISCA*, 2002.

[26] H. Mushtaq et al. Efficient software-based fault tolerance approach on multicore platforms. In *DATE*, 2013.

[27] R. Nathan and D.J. Sorin. Argus-G: Comprehensive, low-cost error detection for GPGPU cores. *IEEE Computer Architecture Letters*, 2015.

[28] S. K. Reinhardt et al. Transient fault detection via simultaneous multithreading. In *ISCA*, 2000.

[29] G. A. Reis et al. SWIFT: Software implemented fault tolerance. In *CGO*, 2005.

[30] P. Reviriego et al. Diverse double modular redundancy: A new direction for soft-error detection and correction. *IEEE Design Test*, 30(2):87–95, 2013.

[31] E. Rotenberg. AR-SMT: a microarchitectural approach to fault tolerance in microprocessors. *FTC*, 1999.

[32] J. D. Scales et al. The design of a practical system for fault-tolerant virtual machines. *Operating Systems Review (ACM)*, 2010.

[33] A Shye et al. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *DSN*, 2007.

[34] A. Shye et al. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 2009.

[35] H. So et al. Expert: Effective and flexible error protection by redundant multithreading. *DATE*, 2018.

[36] STMicroelectronics. 32-bit Power Architecture microcontroller for automotive SIL3/ASILD chassis and safety applications, 2014.

[37] M. B. Sullivan et al. Swapcodes: Error codes for hardware-software cooperative gpu pipeline error detection. In *MICRO*, 2018.

[38] Synopsys, Inc. Synopsys Announces Industry's First ASIL D Ready Dual-Core Lockstep Processor IP with Integrated Safety Monitor. https://www.eejournal.com/industry_news/synopsys-simplifies-automotive-soc-development-with-new-arc-\\functional-safety-processor-ip, 2017.

[39] V. Vargas et al. NMR-MPar: A fault-tolerance approach for multi-core and many-core processors. *Applied Sciences (Switzerland)*, 2018.

[40] J. Wadden et al. Real-world design and evaluation of compiler-managed gpu redundant multithreading. In *ISCA*, 2014.