© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI 10.1109/IOLTS52814.2021.9486715

SafeDE: a flexible Diversity Enforcement hardware module for light-lockstepping

Francisco Bas[†,‡], Sergi Alcaide[†], Ruben Lorenzo[†], Guillem Cabo[†], Guillermo Gil[†], Oriol Sala[†,‡], Fabio Mazzocchetti[†], David Trilla[†], Jaume Abella[†]
[†] Barcelona Supercomputing Center (BSC). Barcelona, Spain
[‡] Universitat Politècnica de Catalunya (UPC), Barcelona, Spain

***Abstract*—Safety-related systems, such as those in automotive, avionics and space, impose the existence of appropriate safety measures to meet the safety requirements of the system. In the case of the highest integrity level functionalities (e.g. ASIL-D in automotive), diverse redundancy must be deployed to avoid unreasonable risk of a single fault leading the system to a failure (e.g. using lockstepped cores). However, existing lockstep solutions are either (1) highly intrusive and inflexible coupling two cores with hardware means, or (2) costly in terms of execution time and monitoring if a software monitor thread checks that cores running redundantly preserve sufficient staggering.**

This paper presents SafeDE, a Diversity Enforcement hardware module providing light-lockstep support by means of a non-intrusive and flexible hardware module that preserves staggering across cores running redundant threads, thus bringing time diversity. SafeDE reconciles the lightness and flexibility of software-only solutions, even allowing using the cores without any lockstepping, as well as the tighter staggering of hardware-only solutions that allow using staggering values of few cycles, instead of hundreds of microseconds, as for software-only solutions. Our integration of SafeDE in a RISC-V FPGA-based space multicore from Cobham Gaisler shows that staggering is effectively preserved, and SafeDE overheads are negligible in terms of area and performance due to staggering.

I. Introduction

Increased autonomy levels and improved features in cars, satellites and planes lead to increasing performance demands for those systems. Existing multicores and accelerators deliver the level of performance needed. However, safety-related systems such as those in automotive, space and avionics need to meet specific safety requirements for their adoption. Those requirements are dictated by the specific safety integrity level of the functionality at hand, and are particularly demanding for the highest levels (e.g. Automotive Safety Integrity Level, ASIL, D in automotive [16]).

A key safety requirement for the highest integrity level systems is the capability of not causing a failure due to a single fault. Those faults are normally mitigated by means of diverse redundancy [5], so that redundancy exists but it is sufficiently diverse so that a single fault affecting all replicas would cause different effects, thus allowing, at least, detecting the fault.

Solutions for diverse redundancy often relate to Error Correction Codes (ECC) for storage and Cyclic Redundancy Coding (CRC) for communications [2]. Computing elements (e.g. cores), instead, usually require full replication and thus, resort to dual (DMR) [10], [21], [24] or triple modular redundancy (TMR) [18]. However, such redundancy is not enough and diversity is also needed to avoid that a single fault (e.g. a voltage droop or a permanent fault) affects redundant instances identically. Such diverse redundancy is achieved with lockstepping, where two (or more) identical cores execute the same software redundantly, but with some staggering among them, so that the state of the cores differs at any point in time, and thus, a fault cannot produce the same error in all redundant copies, which could go unnoticed otherwise.

Lockstepping, in the form of Dual Core LockStepping (DCLS) [15], [35], [37], is generally implemented at hardware level tying two cores together operating with few cycles of staggering (i.e. the head core executes the same software N cycles ahead of the trail core). External requests are generated by one of the cores (e.g. head one), but not sent until compared to those of the other core (e.g. trail one). Upon a match, loads, stores, interrupts and any other type of request is sent, but just once (not redundantly). Responses are duplicated and delivered to both cores preserving the staggering (i.e. delivering them N cycles later to the trail core). This ensures consistent states across lockstepped cores, but with some staggering to preserve diversity. Such a solution, however, makes only one of the cores be visible at software level, and precludes the user from using those cores independently.

Light-weight software-only lockstepping has been proposed to reduce the cost of full lockstepping [3]. Such light-weight lockstepping resorts to software redundancy, and to the existence of a software monitor enforcing staggering across redundant threads. While such a solution has been proven effective, and compatible with Commercial Off-The-Shelf (COTS) processors, it requires native lockstepping (hardware-based) for the core running the monitor, and imposes some non-negligible staggering (e.g. $100\mu s$) to allow the monitor to collect information of the progress of redundant threads without causing too high relative interference.

Overall, both hardware-based and light software-based lockstepping pose a number of limitations to achieve diverse redundancy. This paper, addresses this challenge by proposing a different tradeoff achieving most of the benefits of both approaches with low cost.

This paper presents *SafeDE*[1], a Diversity Enforcement hardware module providing light-lockstep support by means of a non-intrusive and flexible (programmable) hardware module that preserves staggering across cores running redundant threads, thus bringing time diversity. In particular, SafeDE is a tiny hardware module performing the same monitoring tasks as the software-only solution, but with a much lower staggering (just few cycles instead of $100\mu s$), and without requiring native lockstepped cores. Compared to native hardware lockstepping, SafeDE can be integrated without modifying IP cores, thus with limited intrusiveness, and allows using cores indepen-

[1]Available as an open-source component in https://bsccaos.github.io [6].

dently instead of always in lockstep mode. In particular, the contributions of this paper are as follows:

- We present SafeDE, a new hardware/software scheme for efficient, flexible and lowly-intrusive light lockstepping to achieve diverse redundancy.
- We implement and verify SafeDE in VHDL.
- We successfully integrate SafeDE in a space SoC based on Cobham Gaisler's NOEL-V cores, implemented in a FPGA, which is already a commercial setup for this platform reaching commercial readiness by early 2022 [11].

The rest of the paper is organized as follows. Section II provides some background. SafeDE is presented in Section III and evaluated in Section IV. Section V reviews related work. Section VI concludes this paper.

II. Background

This section provides some background on the need for some form of lockstepped execution in safety-related systems, and on the existing solutions to achieve it.

A. Redundancy, Diversity and Sphere of Replication

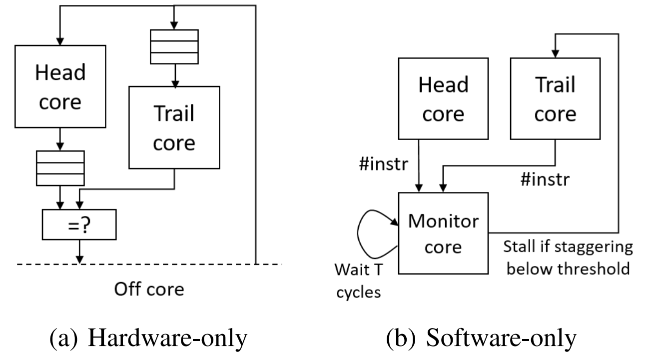
Safety-related systems are designed so that unreasonable risk due to software faults of any kind and systematic hardware faults is avoided by design, verification and validation (V&V). However, random hardware faults cannot be avoided and appropriate safety measures need to be deployed, such as for instance, diverse redundancy for the highest safety integrity levels (SIL for short).

There are two main approaches to achieve diverse redundancy: using diverse hardware and/or software so that replicas (e.g. redundant threads) execution is diverse in nature, or using identical hardware and software and enforcing diversity by making identical replicas run on identical hardware (but not the same hardware unit) with some staggering so that hardware state is different at any point in time. Generally, the latter is preferred since it reduces design as well as V&V costs because only one software unit and hardware unit needs to be designed, verified and validated, rather than having to do so for multiple units. This is, for instance, the case of DCLS in Infineon AURIX multicores [15].

Lockstep operation is usually implemented comparing only off-core activities such as load and store requests, as well as interrupts and exceptions. Thus, cores execute software redundantly with some staggering and include some buffering capabilities to store off-core requests of the head thread until they are compared with the trail thread ones, as well as to store off-core responses for the trail thread, which are delivered to the corresponding core with some staggering w.r.t. the head thread. Other off-core resources, such as storage elements (e.g. caches, memories) and communication elements (e.g. buses, crossbars) build upon ECC and CRC to reach diverse redundancy.

B. Lockstep Strands

Hardware-based tight lockstepping builds upon two tightly coupled cores operating with identical state but with N cycles of staggering, thus meaning that the head core is exactly N cycles ahead in the exection of the trail core. Therefore, during fault-free operation, the trail core delivers exactly the same external signals as the head core but N cycles later. This is managed with appropriate queues, as illustrated in Figure 1(a). In particular, output activity of the head core needs to be stored during N cycles until the trail core produces the same outputs. Then, they are compared and, if they match, the corresponding outputs are made visible out of the lockstepped cores complex, e.g. sending a load request to memory, storing data or signalling an interrupt. Externally, *lockstepped cores are perceived as just one core* since only the activity generated by one of them is sent. Responses for those requests of a single core arrive just once, so they need to be replicated and delivered to both cores. In order to preserve the staggering, buffering is needed in the trail core side to keep data during N cycles before delivering it to the core. Such a scheme introduces a delay of N cycles in any external access either for the outgoing requests (head core) or for the incoming responses (trail core). Nevertheless, N is typically 2 or 3 cycles, so the impact of staggering is limited.



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

Fig. 1. Schematic of hardware and software-based lockstep schemes.

Software-based light lockstepping builds upon two cores able to operate independently executing different programs. Therefore, redundant threads need being created by software and scheduled accordingly into the head and trail cores which, in practice, are identical among them and inherit head or trail behavior only due to software management of their progress. The (simplified) operation, which we illustrate in Figure 1(b), requires of a supporting monitor thread, which runs in another core, either in the same chip or another. For instance, this scheme has been devised to run the monitor in a processor with native hardware lockstepping, thus guaranteeing safety for the monitor, so that the monitor manages redundant threads of multiple software components running in a (likely powerful) multicore without lockstepping support. In particular, the monitor thread creates the two redundant processes and keeps the trail core stalled. Periodically, the monitor collects the number of instructions executed by the head and trail cores ($\#instr$ in the figure), and compares them. If the difference, $\#instr_{head} - \#instr_{trail}$, is above a given threshold TH_{stag}, then the trail core is allowed to make progress. Else, the trail core is stalled. Every T_{check} cycles the monitor repeats the process. Note that TH_{stag} needs to be carefully set to guarantee that, even if the head core got stalled and the trail one made progress at its maximum speed during T_{check} cycles, the head core would still have a number of instructions executed higher than the trail one. If this is the case, both cores can be allowed to make progress unsupervisedly during T_{check} cycles until the next monitoring check. There is an obvious relationship between the monitoring frequency and the staggering time: the looser the monitoring (i.e. the higher T_{check}) so that monitoring overhead decreases, the higher the

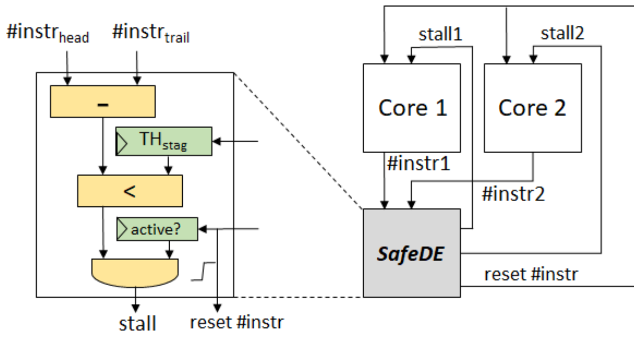


Fig. 2. SafeDE architecture.

staggering needed (i.e. the higher TH_{stag}). As shown in [3], TH_{stag} needs to be typically at least the maximum number of instructions that could be executed in 100μs. Those $100\mu s$ are roughly the execution time increase that lockstepped execution will cause to let the trail core finish the execution of the trail thread after the head one does so with the head thread.

III. SafeDE: a Diversity Enforcement hardware module

This section presents SafeDE. First, we introduce the architecture of SafeDE. Then, we analyze its pros and cons in comparison with the software-only solution. Finally, we provide implementation and integration details about its deployment in a commercial space multicore.

A. *SafeDE Architecture*

SafeDE is architected to be the hardware counterpart of software-based light lockstepping. Its objective is keeping as much as possible the advantages of the software-only solution, while mitigating its limitations, which we discuss in next subsection. For that purpose, SafeDE is devised as a tiny module coupled to each pair of cores potentially needing to operate lockstepped. This is illustrated in Figure 2. As shown, SafeDE requires the instruction count values of the lockstepped cores, as well as an interface signal to stall the trail core. SafeDE subtracts the number of instructions of the trail core from those of the head core, $\#instr_{head} - \#instr_{trail}$, as in the case of the software-only solution, and compares the difference against a staggering threshold, TH_{stag}. Only when the head core is not sufficiently ahead of the trail core, and SafeDE is active, a stall signal is sent to the trail core, which must be used to stall the core whenever set. Such stall can be achieved, for instance, stalling all stages (e.g. blocking pipeline latches), stalling only the commit stage, or stalling only the fetch stage, to name few alternatives.

SafeDE parameters. SafeDE has four configuration registers: TH_{stag}, $active$, $CritSec1$ and $CritSec2$.

- TH_{stag} stands for the minimum number of instructions that the head core must be ahead of the trail core. Typical values are few instructions, very much in line with hardware-based tight lockstepped cores. Note that this value is several orders of magnitude lower than that for software-only solutions.
- $active$ signal indicates whether SafeDE is active or not. If reset, SafeDE produces no effect. Else, SafeDE operates normally. Whenever the parameter is set to 1 (input signal raised), lockstep operation is possible, which is practically controlled by the other registers.
- $CritSec1$ and $CritSec2$ registers indicate whether core 1 (head) and core 2 (trail) have entered the "critical" code region to be executed in lockstep mode.

SafeDE operation. SafeDE is, at some point, inactive ($active = 0$). No action is performed by SafeDE until $active$ is set, regardless of the values of the other registers. Eventually, SafeDE is programmed setting TH_{stag} as needed, and then $active$ is set. Once $active = 1$, $CritSec1$ and $CritSec2$ are reset, and SafeDE awaits for the corresponding activation of $CritSec1$ and $CritSec2$. $CritSec1$ and $CritSec2$ are set by core1 and core2 respectively when they reach their critical section. The first core entering the critical section will take the role of the head core. Whenever the head core sets its $CritSec$ register, its instruction count ($\#instr_{head}$) resets and starts counting the committed instructions. If the trail core activates its $CritSec$ register before the staggering is large enough ($\#instr_{head} - \#instr_{trail} < TH_{stag}$), the stall signal is sent to the trail core. As soon as the staggering is enough, the stall signal is reset and the trail core starts execution. Note that SafeDE performs the subtraction $\#instr_{head} - \#instr_{trail}$ and the comparison against TH_{stag} every cycle. This allows using very low values for TH_{stag} as opposed to the software-only solution. Despite performing such action every cycle, note that $\#instr_{head}$ and $\#instr_{trail}$ barely change every cycle (e.g. they do not change or just are incremented by up to very few instructions). Therefore, activity due to the subtraction and comparison is tiny since most of input signals remain constant, and hence, signal switching occurs seldom. Whenever the head core reaches the end of its critical section, it resets $CritSec1$. From that point onwards, SafeDE allows the trail core execute without any stall until it also finishes its critical region ($CritSec2$ is also reset).

Software process. To use SafeDE, end users, either by themselves or with the support from an appropriate API, need to create the two redundant processes, schedule them to the head and trail cores, and keep them stopped (e.g. with SIG_STOP signals). Then, SafeDE needs being configured and set to active state. Finally, both redundant processes need to be set to active (e.g. with SIG_CONT signals).

B. *Features and Limitations Analysis*

SafeDE offers a different tradeoff to that of the software-only solution. Next, we detail its main features and limitations along with whether they are common with the software-only solution in [3] or not.

1) *SafeDE features:*

- **Low cost**. As shown before, SafeDE is a tiny module offering support to enforce staggering to achieve diverse redundancy. SafeDE releases the system from having to allocate a task with strict timing requirements (e.g. executing at a very specific frequency), as needed in the case of the software-only solution.
- **Low staggering**. By being implemented in hardware and monitoring instruction counts from the head and trail cores constantly, SafeDE can guarantee diversity even if staggering is low (e.g. TH_{stag} set to few instructions). This is also an advantage w.r.t. the software-only solution, which requires heavier activities by being conducted by

software means, reading remote registers, issuing interrupts to stall/resume execution in a core, etc., so needing a much higher staggering than SafeDE.

- **Flexibility**. SafeDE can be enabled/disabled at will, thus being usable at very fine granularity. Still, the granularity is dictated by the ability of the software to create the redundant processes and stop/resume them at start up. Hence, flexibility is high, but the granularity at which SafeDE can be used is similar to that of the software-only solution.
- **Low intrusiveness**. SafeDE needs cores to export few signals for its integration, thus being much less intrusive than hardware-based tight lockstepping. In particular, SafeDE needs cores to expose their instruction count register for monitoring purposes, an appropriate stall signal to stall the trail core whenever needed, and the reset signal for the instruction count register. Differently, the software-only solution does not require any hardware change, although it may require modifications into the operating system to allow reading the instruction count register from remote cores. SafeDE, instead, does not need any such operating system change.

2) SafeDE limitations:

- **Non-null intrusiveness**. Hardware changes required by SafeDE are minor, but they are not null, so differently to the software-only solution, SafeDE cannot be deployed on COTS ASIC multicores.
- **Limited applicability**. SafeDE, as well as the software-only solution, needs that redundant processes execute exactly the same instructions so that software does not diverge. Otherwise, SafeDE (and the software solution) would become ineffective. For instance, this implies that SafeDE should not be used for functions whose execution path depends on random choices or physical address bits, which could follow different paths across redundant processes. For the former case, SafeDE can be used if identical random values are enforced across processes (e.g. providing the same random number stream with a software-implemented pseudo-random number generator initialized identically for both processes). Similarly, SafeDE cannot be used for parallel applications if the execution path and the instruction count may vary across redundant threads due to synchronization (e.g. different order to access sequential code regions). In any case, such limitation is analogous to that of the software-only solution. Also, SafeDE (and software solutions) cannot be applied for processes with some form of I/O accesses, since those accesses need to be performed only once generally, but redundant processes would perform those accesses twice.
- **Limited diversity**. Physical diversity is achieved by running redundant processes in different cores. SafeDE, as well as the software only solution, provides also time diversity. However, other sources of *common cause failures* (i.e. identical failures in both cores due to a single fault), such as for instance those related to physical degradation of specific gates of the processor need other types of diversity (e.g. layout diversity) that cannot be achieved by any external monitor, either hardware (SafeDE) or software.

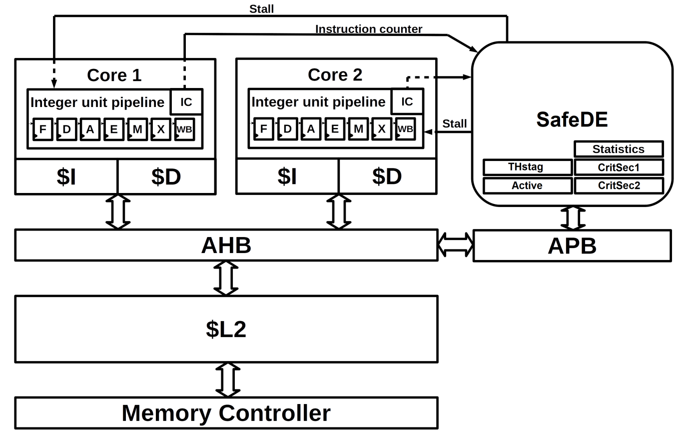


Fig. 3. High-level representation of SafeDE integrated into the system.

- **SafeDE hardening**. Since faults could also affect SafeDE, it must be hardened to meet sufficiently low failure rates or, simply, deployed with physical diverse redundancy, as for tight lockstepped cores (e.g. using the scheme in Figure 1(a) but for SafeDE instead of the cores).

3) Scope of applicability: Due to the limited applicability of SafeDE, as in the case of software-only solutions, SafeDE cannot be applied to all software but to code regions. For instance, if data is read from a sensor, processed and sent to an actuator, SafeDE and software only solutions can be applied to the processing part only. If full (end-to-end) diverse redundancy is needed, native (hardware-based) tight lockstepping support is needed, as explained in [3]. In particular, at least two cores need to be paired with hardware-based tight lockstepping to execute sensor and actuator interactions. However, the most performance hungry part of the execution (i.e. data processing) can be managed with SafeDE. Thus, deployments where only two cores provide hardware-based tight lockstepping, and the rest support light lockstepping with SafeDE would be highly efficient. For instance, an 8-core multicore could be deployed with (a) 4 pairs of hardware-based tight lockstepped cores, thus offering only 4 user-visible cores; (b) 1 pair of hardware-based tight lockstepped cores and 6 non-lockstepped cores, thus offering 7 user-visible cores, but only up to 1 concurrent task running with diverse redundancy; or (c) 1 pair of hardware-based tight lockstepped cores and 6 cores paired with SafeDE, thus offering 7 user-visible cores, and supporting up to 4 concurrent tasks running with diverse redundancy.

C. Implementation and Integration

To prove its feasibility, SafeDE has been integrated and evaluated in an industrial space product, namely the RISC-V based Cobham Gaisler NOEL-V MultiProcessor System on Chip (MPSoC). In this platform, Cobham Gaisler provides an integrated set of reusable VHDL IP cores centered around common on-chip buses. The buses of the selected MPSoC are based on the standard AMBA 2.0. SafeDE is designed in VHDL as one of the reusable Gaisler IP cores.

1) System on Chip: The SoC where SafeDE is tested comprises 2 RISC-V based 64-bit dual-issue 7-stages pipeline NOEL-V cores. Apart from the main 128 bits Advanced High-performance Bus (AHB), another AHB is used for debugging

purposes. For low bandwidth peripherals as SafeDE, an Advanced Peripheral Bus (APB) is employed.

Each core includes private L1 Data and Instruction caches. Data L1 caches are write-through with a write-no-allocate policy. A shared L2 cache is connected to the main shared AMBA AHB, and to the memory controller.

2) Integration: SafeDE is integrated as an APB slave connected to the system through a standard APB interface. Thus, SafeDE is highly portable and can be easily embedded into any system implementing an APB interface.

Apart from the APB standard signals, SafeDE needs a few interconnections with the cores. The instruction counter of each core has to be mapped as an input. Instruction counters are employed to calculate the total number of instructions commited by each core and compute its difference. SafeDE has one output to stall the trail core. That SafeDE output is ORed with an internal pipeline signal in charge of holding the pipeline (i.e. keeping constant the pipeline registers values). Therefore, when SafeDE needs to stall the trail core, SafeDE can stall that core just asserting the respective output. Therefore, the only modifications needed in the cores correspond to exporting the instruction counter[2] value to make it visible to SafeDE, and placing the OR gate needed let SafeDE set the pipeline stall signal whenever needed. Figure 3 shows a high-level representation of SafeDE integrated into the system.

3) Configuration and operation: SafeDE is controlled and configured by means of four internal registers. Each register is mapped to a specific SoC memory position. The first one, TH_{stag}, is used to configure the minimum staggering. The second one, $active$, is used to enable/disable SafeDE. Each of the two remaining registers, $CritSec1$ and $CritSec2$, is coupled with one core and set to 1 when the respective core starts the critical section (i.e. with a store instruction to this register in the application), and set to 0 when it finishes its execution. This procedure allows SafeDE to synchronize both cores at the beginning of the critical section, even if they do not start simultaneously, as explained before. Neither of the cores assumes the role of trail or head core until its critical section starts.

In addition to these four registers, SafeDE has also several registers to gather some statistics such as maximum staggering, minimum staggering, times that the trail core has been stalled, how many cycles the trail core has been stalled, committed instructions by each core, etc. The connection of SafeDE to the MPSoC allows SafeDE registers to be written and read through usual load and store operations.

IV. Evaluation

We evaluate SafeDE by synthesizing the RISC-V multicore SoC into a Xilinx Kintex UltraScale KCU105 evaluation kit.

A. Validation

In order to validate the correct functioning of SafeDE once implemented in the FPGA, we have added a register recording the lowest staggering observed between the head and tail cores. We have used the TACLeBench benchmark suite [8], which is a set of open-source self-contained benchmarks intended to evaluate basic functionalities in real-time systems. They have been chosen because, since their source files already include inputs hardcoded, they can be easily compiled and run on a baremetal setup without any support to read data from files. Moreover, since some of the benchmarks are quite simple (i.e. execution times range between some hundreds and some millions of cycles), they ease debugging and validation on a simulated environment. Therefore, we have set the staggering to 10 cycles, $TH_{stag} = 20$, and recorded the lowest staggering observed across all benchmarks. Our experiments confirm that the actual staggering has never been below this number of cycles, hence providing evidence that SafeDE works as expected.

B. Execution time overhead

To elucidate the impact of SafeDE in terms of computational overhead, we have run the TACLeBench benchmarks in three different scenarios:

- *Isolation*: only one core executes the benchmark and the other core remains idle.
- *Redundancy without diversity*: two different cores execute the same benchmark without any control mechanism.
- *Redundancy with diversity* enforced by SafeDE: SafeDE guarantees that the minimum staggering, TH_{stag}, is never exceeded.

In our evaluation, we have set $TH_{stag} = 20$ for illustration purposes. Note that the lowest value that must be used for the staggering relates to the pipeline depth of the core (7 stages in the specific platform used) given that the instructions difference is obtained using committed instructions. Hence, using a pipelined core, it could occur that, by the time staggering is about to fall below the threshold and the trail core stalled (i.e. its commit stage is stalled), the pipeline of the trail core could be executing some common instructions to those of the head core in some of the stages if the staggering is too low. Thus, we have set the threshold to be high enough so that this cannot happen in a pipeline with 7 stages and a pipeline width of 2 instructions.

Note that, by using a light-lockstep approach, redundant processes are generated loading binaries twice (one for each core) in different memory segments. We execute each benchmark 1,000 times and use the average cycle count for each one for our evaluation to discount the effect of small variations due to, for instance, delays caused by DRAM refreshes. In any case, absolute variations observed are always in the order of few tens of cycles.

Results are shown in Figure 4. As shown, in all the cases, the execution time overhead with SafeDE w.r.t. the execution time in isolation and in two cores without SafeDE is negligible. In particular, SafeDE causes an execution time degradation in most of the cases below 0.5%, and up to 1.3% in one case (`BITONIC` benchmark) w.r.t. the execution time in isolation. If we compare it against the redundant execution without enforcing diversity, execution time degradation is generally below 0.1% (in some cases performance even marginally improves), and up to 0.6% for `IIR` benchmark.

In some cases, minor performance variations between the isolation and redundant versions of the programs are observed, being those differences neither caused by interference between redundant threads, nor by SafeDE operation itself. Instead, those variations are caused by the initial core state (e.g. branch predictor state), or changes in instruction cache behavior due to changes in the memory alignment of the binaries with and

[2]Note that virtually any processor implements instruction and cycle performance monitoring counters.

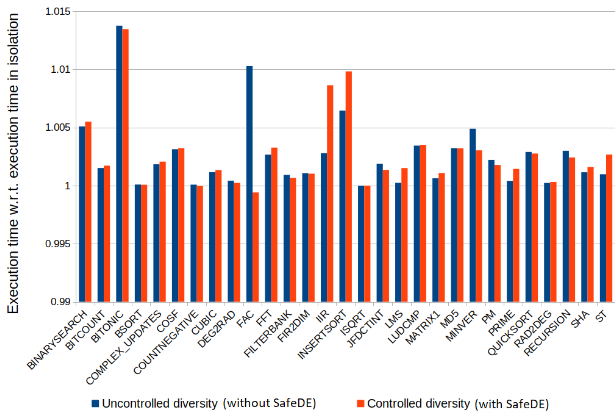


Fig. 4. Execution time of different TACLeBench benchmakrs normalized w.r.t. their execution time in isolation. Each benchmark is executed 1,000 times.

TABLE I
CLASSIFICATION OF REDUNDANT EXECUTION TECHNIQUES.

Strategy	Target	Diversity	Approaches
HW	CPU	Yes (tight)	[18], [23], [35]
		Yes (light)	**Our approach (low staggering)**
		No	[9], [10], [21], [24], [27], [30]
	GPU	Partially	[1]
		No	[20], [22], [26], [36]
SW-Only	CPU	Yes (light)	[3] (high staggering)
		No	[12], [28], [31], [32], [4], [25], [33], [34]
	GPU	Partially	[2]
		No	[7], [19], [38], [39]

without thread redundancy. In the case of `FAC` benchmark, since it is a small benchmark (around 700 instructions only), these tiny effects have a visible impact in relative terms (e.g. 1% execution time increase without SafeDE and 0.1% decrease with SafeDE).

Overall, as expected, execution time increase can be regarded as negligible since the staggering threshold can be set very low (20 cycles in our evaluation), thus far below the 100μs, which correspond to many thousands of cycles, imposed by the software-only solution [3].

C. Hardware costs

We synthesized our RISC-V design using the Vivado 2018 Toolchain and target the FPGA present in the Xilinx UltraScale KCU105. The overall cost of SafeDE implementation is 261 LUTs and 417 registers, whereas the entire SoC uses approximately 114,000 LUTs and 74,000 registers. Each core uses around 38,000 LUTs and 17,000 registers. Hence, SafeDE is a low-cost component representing just 0.23% of the entire SoC LUTs and 0.56% of entire SoC registers. SafeDE uses just 0.35% of the LUTs of the pair of cores it manages, and 1.23% of their registers. These numbers can be further improved by removing all the logic devoted to gather statistics.

V. RELATED WORK

Some solutions to implement redundancy relate to Redundant Multi-Threading in a multi-threaded core [27], [30], as well as across different cores [10], [21], [24], and even building on partial redundancy [9], [23]. Unfortunately, none of those solutions guarantees diversity, either by reusing the same hardware inside the core, or by failing to impose any staggering at all. Some software-only solutions for CPUs enforce redundancy by compiler means, building on transactional memory, or creating monitoring threads [12], [25], [28], [32]–[34]. However, none of those solutions is effective to capture single faults affecting all redundant units (a.k.a. Common Cause Faults, CCFs).

The particular case of GPUs has been addressed to provide redundancy with hardware support [20], [26], [36], [39] or relying only on software solutions [7], [19], [39], but not providing diversity. Both redundancy and diversity have been achieved in GPUs with [1] and without hardware support [2]. Unfortunately, those solutions are GPU specific and cannot be extrapolated to CPUs.

Some designs provide native tight lockstep, such as ST Microelectronics SPC56XL70 [35] and Infineon AURIX processor family [15], which provide DCLS. Analogously, some Arm Cortex-R5 designs extend lockstep to triple-core implementations [17], [18]. Since errors are only detected when erroneous data is exposed beyond the core sphere, some authors have proposed extensions to shorten error detection time [14], and to enhance recovery [13]. In any case, as explained before, tight lockstep makes half of the cores non-visible to the user, thus diminishing flexibility drastically.

Reviriego et al. [29] show that, for some dual diverse redundant designs, it is possible perform recovery if the erroneous output can be identified without needing to compare it against a fault-free reference, which allows to tell what of the two diverse redundant designs delivers the correct output.

Flexible diverse redundancy on CPUs has been implemented so far with software-only solutions [3] but, as explained before, staggering imposed is large, which leads to non-negligible performance degradation if tasks duration is short (e.g. between 100μs and 1ms). Our work aims at leveraging very limited hardware cost to mitigate such limitation of software-only solutions, thus complementing existing solutions (see Table I).

VI. CONCLUSIONS

Diverse redundancy is a mandatory safety measure for safety-related systems to avoid that a single fault leads the system to a failure. Diverse redundancy is generally implemented by means of tight lockstep computing cores. However, the cost of such an approach is huge due to making half of the cores non-visible for the user, so that they cannot be used for other purposes different to redundancy. Some recent work advocates for the use of light-lockstep approaches to complement tight-lockstep ones, therefore gaining flexibility. However, those solutions impose a large staggering between redundant threads, which may have a very large impact in performance for tasks with low duration.

In this paper we present SafeDE, a tiny module supporting light-lockstep in a much more effective manner than software only solutions. Our design has been shown to cause very low performance degradation (typically below 0.5%) w.r.t. non-redundant execution, and increase hardware costs by far less than 1% for a commercial space SoC.

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.

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] BSC - CAOS. SafeTI. https://bsccaos.github.io.

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

[8] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[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] P. Gomez et al. De-RISC Dependable Real-time Infrastructure for Safety-critical Computer Systems. *Ada User Journal*, June 2020.

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

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

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

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

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

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

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

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

[29] Pedro Reviriego, Chris J. Bleakley, and Juan Antonio Maestro. Diverse double modular redundancy: A new direction for soft-error detection and correction. *IEEE Design Test*, 30(2):87–95, 2013.

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

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

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

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

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

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

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

[37] 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.

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

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