Software-only based Diverse Redundancy for ASIL-D Automotive Applications on Embedded HPC Platforms

Sergi Alcaide^{†,‡}, Leonidas Kosmidis[†], Carles Hernandez^{*,†}, Jaume Abella[†]

[†] Barcelona Supercomputing Center (BSC)

[‡] Universitat Politècnica de Catalunya (UPC) ^{*} Universitat Politècnica de Valencia (UPV)

Abstract—High-Performance Computing (HPC) platforms become a must in automotive systems to enable autonomous driving. However, automotive platforms must avoid Common Cause Failures (CCFs), as indicated by the ISO26262 automotive safety standard. CCFs can be avoided enforcing diverse redundancy. Unfortunately, HPC platforms fail to provide such support. This paper proposes a flexible and efficient software-based scheme to implement diverse redundancy on HPC platforms. A software implementation on a Commercial Off-The-Shelf ARM multicore proves the effectiveness of this scheme to guarantee diverse redundancy with negligible performance degradation. Our solution is the first step towards an automotive-compliant HPC platform.

I. INTRODUCTION

Autonomous driving (AD) frameworks require high performance, e.g. to process images from cameras. Embedded High-Performance Computing (eHPC) platforms delivering such performance must deliver Automotive Safety Integrity Level (ASIL) D compliance, as dictated by ISO26262 functional safety standard [1], since AD systems must remain operational despite faults. Such ASIL-D compliance imposes the use of diverse redundancy (e.g Dual-Core LockStep (DCLS) execution) [2]. Existing eHPC platforms (e.g. RENESAS R-Car H3 and the NVIDIA Xavier platforms) usually lack explicit hardware support for enforcing some form of diverse redundancy and software strategies are, therefore, needed [3].

This paper tackles this challenge by proposing a *software-only* flexible and efficient approach to achieve diverse redundancy by construction for eHPC multicores in general, and for ARM-based multicores in particular. Our solution is based on a monitor process able to orchestrate the execution of redundant applications guaranteeing the computing resources used are physically diverse, and ensuring that the same dynamic instruction in both redundant applications never executes simultaneously, thus providing time diversity. Hence, by providing both time and space diversity, CCFs for relevant fault models are avoided. In particular, our contributions are as follows:

- 1) A flexible and efficient software-based approach to enforce diverse redundancy on high-performance multicores, with requirements met by most popular processor families.
- 2) A tailoring of the approach for a Commercial Off-The-Shelf (COTS) ARM-based multicore, as an illustrative example, proving the feasibility of the approach.
- 3) Quantitative evidence of the approach to achieve diverse redundancy with tiny execution time cost. In particular, our results show performance degradation around 4% on average and up to 10% in one case for a variety of automotive benchmarks.

Strategy	Target	Diversity (CCF considered)	Approaches
HW	CPU	Yes	[4], [5], [6]
		No	[7], [8], [9], [10], [11], [12]
	GPU	Partially	[13]
SW-Only	CPU	No	[14], [15], [16], [17]
		Yes	Our Approach
		No	[18], [19], [20], [21], [22], [23], [24], [25]
	GPU	Partially	[3]
		No	[26], [27], [28], [29]

TABLE I: Classification of HW and SW-only fault-tolerant techniques

II. RELATED WORK

ASIL D compliant ST Microelectronics SPC56XL70 [4] and Infineon AURIX processor family [30] implement DCLS, whereas some Arm Cortex-R5 designs implement Triple-Core Lockstep [5], but fail to provide enough performance for AD systems [31]. Some improvements shorten time-to-detection for errors [32] or enhance recovery processes [33], but do not improve performance.

Another family of solutions for high-performance CPUs builds upon thread redundancy inside a single core [8], [7], or across cores [9], [10], [11], even with only partial redundancy [6], [12]. However, those solutions require hardware support for thread synchronization, and differently to DCLS, do not guarantee diversity. SW-only solutions also exist for CPUs, introducing redundancy at compiler levels [21], [22], building on transactional memory [18] or creating a monitoring process to check for errors [19], [24], among other solutions [25]. However, none of them guarantees staggering execution for redundant threads/processes, so CCFs may cause the same error in both threads/processes, which may lead to a failure. For example, authors in [24] consider the process granularity when applying redundancy, but they do not impose control on the pace of the redundant execution, which made them vulnerable to CCFs since they cores may have exactly the same state when the fault affecting both cores identically occurs.

Analogous solutions for accelerators (e.g. GPUs or the Kalray MPPA family) have been proposed, either with hardware support [27], [15], [16], [17] or with software-only support [26], [27], [28], but none of them guarantees diversity to protect against CCFs. Some preliminary solutions guarantee diversity to some extent for GPUs either with [13] or without hardware support [3].

However, to the best of our knowledge, no software-only solution guarantees diversity for CPUs, which is the challenge we address in this work, as summarized in Table I.

Fig. 1: Schematic of the diverse redundancy execution strategy.

III. SOFTWARE-BASED DIVERSE REDUNDANCY APPROACH

We present our technique for an abstract multicore first, and then we specify the realization of our technique on a specific multicore (ARM Cortex-A57 based) for illustration purposes.

A. Diverse Redundancy across the entire multicore

The overall strategy targets only creating redundant computations and ensuring that they are performed diversely in time (i.e. at different time points) and in space (i.e. in different cores). This prevents CCFs for very relevant fault models since faults affecting multiple components simultaneously (e.g. voltage droops) do not affect redundant computations identically by performing redundant computations at different time instants. Analogously, by using different cores for redundant computations, faults affecting specific hardware components (e.g. a faulty core) do not affect both computations identically.

Data transmission and storage are regarded as intrinsically diverse and redundant in the space domain since data sent/received from/to the cores is, in general, ECC-protected (Error Correcting Codes), thus meaning that any fault corrupting data would be detected whenever read by checking the ECC. In particular, data stored in local cache memories in the cores is normally ECC protected, and such codes are propagated all the way to memory, where they are checked upon read operations, thus providing fault detection capabilities at least for single-bit upsets.

B. Specification of the Execution Strategy

The approach we propose to achieve diverse redundancy consists of the following main steps, illustrated in Figure 1:

- 1) The MCU (Microcontroller Unit) replicates input and output data and buffers for the task to be executed on the eHPC multicore (HPCM for short). We refer to redundant processes as head and trail processes.
- 2) The MCU offloads both processes onto different cores in the HPCM.
- 3) The MCU accesses the instruction count (IC) for the head and trail processes (IC_{head} and IC_{trail}) at a given time frequency, T_{check} .
- 4) The trailing process is not allowed to make any progress unless the head process is a given number of instructions $I_{stagger}$ ahead in execution.
 - a) If $(IC_{head} - IC_{trail}) < I_{stagger}$, then the trail process progressed too fast during the last time interval and there is some risk of catching the head process during the next interval. Hence, the trail process is stalled during the following time interval (T_{check} time).

Fig. 2: Strategy used in the 4-core Jetson TX2

- b) Instead, if $(IC_{head} - IC_{trail}) \geq I_{stagger}$, then both processes are allowed to progress during T_{check} .

Overall, the approach monitors the execution of the redundant processes on the HPCM, ensuring that the staggering is large enough so that the trail process cannot catch up with the head one. Particular care must be taken setting $I_{stagger}$ and T_{check} to ensure that $I_{stagger}$ is large enough so that a process cannot execute more than $I_{stagger}$ instructions in a single time interval, T_{check} , plus the time needed to check the instruction counts and send a stall signal to the trail process. Also, T_{check} must be high enough so that monitoring overheads are kept low in relative terms, but low enough so that redundant processes are not overly staggered, which would lead to increased performance penalization to complete redundant execution. A specific analysis of those parameters will occur during the implementation phase.

Note that our approach is intended to be used in code regions without I/O activity. I/O latencies are normally large and hence, code regions with I/O operations do not require high CPU performance. Instead, they can be executed in the MCU with native DCLS. Thus, only compute intensive kernels are intended to be replicated and offloaded onto the eHPC for their diverse and redundant execution.

C. Realization on an ARM-based Multicore

We integrate our strategy on a 4-core ARM Cortex-A57, part of the NVIDIA Jetson TX2 platform. The reason for this choice is using commercially available cores with low-power operation, as it is the target of automotive platforms. While many multicores with different Instruction Set Architectures (ISAs) could fit in this description, we chose the aforementioned ARM multicore due to its availability in our lab, although we do not foresee any limitation to realize our strategy on different multicores.

The actual platform used for implementing our strategy lacks some features, such as an MCU processor mastering the HPCM and AUTOSAR support¹. Therefore, we run the monitor process in a core in the HPCM, and integrate our approach on top of Linux, using Message Passing Interface (MPI) for process communication across the monitor and the two redundant task processes. Part of our future work consists of integrating our approach on platforms with MCU and HPCM, with AUTOSAR support, and further enabling the integration for parallel applications, for which a preliminary assessment already shows the viability to pair corresponding threads across redundant processes.

Note that, since we integrate our strategy on Linux, performance variability is expected to be higher than in AUTOSAR-based platforms. In order to mitigate variability to some extent,

¹AUTOSAR (AUTomotive Open System ARchitecture) is the standard software architecture for automotive systems.

we have concentrated as many Linux services as possible in a specific core used neither by the monitor nor by the redundant application processes (core D in Figure 2).

Based on the analysis of the platform (e.g. cycle time, Maximum IPC) and specific latency measurements, we have set the monitor interval short enough to ensure that staggering is sufficient so that the trail thread cannot catch up during this interval.

D. Scope of the proposal and Fault Model

Safety-critical systems need both (1) a fault detection mechanism and (2) a recovery mechanism. This work proposes a fault detection mechanism, and we rely on the MCU (see Figure 1) for the recovery actions (e.g. reset and restart the HPCM or part of it) whenever a fault is detected.

The proposed solution provides fault detection by means of software-only diverse redundancy for any type of transient fault (e.g. voltage droops or radiation effects), and permanent faults affecting core components randomly. Other sources of CCFs, such as those caused by systematic fabrication effects (e.g. untested layout defects) can only be mitigated with physical diversity in the design (e.g. with different fabrication masks). Hence, if those fault types are regarded as relevant, additional support is needed to complement our solution. The fault detection will occur at the end of the execution when comparing the results of both processes in the MCU. Additionally, our monitor acts also as watchdog by monitoring the progress of both processes. If eventually one of them does not make any progress during a predefined time (i.e. its instruction count does not vary), a fault is detected. Such behavior can occur if one process gets stalled due to a fault, or executes a different number of instructions due to a fault. In the latter case, one of the processes will reach the end of the execution with fewer instructions than the other one, and the monitor will detect such behavior.

IV. EVALUATION

A. Framework

As a representative platform, we build upon the 4-core ARM Cortex-A57 complex of an NVIDIA Jetson TX2 board. As representative real-time automotive benchmarks, we use the EEMBC Autobench suite [34], which reflects some functionalities relevant for automotive critical real-time embedded systems. Additionally, we include a matrix multiplication benchmark (referred to as `matmul` in the rest of the section), since AD frameworks build on neural networks which, ultimately, build upon matrix multiplications [35]. To capture potential performance variations, each benchmark is run 500 times for each setup considered.

B. Overheads Assessment

Our first set of experiments focuses on the cost of our approach and the sensitivity to the monitoring frequency (i.e. T_{check} interval). For that purpose, we have considered 4 setups, which we name as follows:

- **BASELINE**: Original non-redundant version.
- **No monitor (NM)**: Redundant version without any monitoring in place, thus reflecting the impact in execution time due to sharing multicore hardware resources.
- **Passive (P)**: This one includes the monitor, which performs all actions needed except stalling the trail process.

Fig. 3: Execution times, in the form of a boxplot, for the different setups and T_{check} values (including EEMBCs and matrix multiplication).

- **Safe (S)**: Complete implementation of our approach where the trail process is stalled whenever it could catch up the head process, thus preserving diversity.

For the passive and safe setups, we consider different T_{check} values, ranging between 0.0001 and 0.01 seconds, so between $100\mu s$ and $10ms$. Note that the higher T_{check} , the lower the overhead of monitoring, but the higher the potential impact in execution time due to stalling the trail process in the safe setup.

Results of the different setups and T_{check} values for EEMBCs and `matmul` are shown in Figure 3. The figure summarizes the normalized results for the 8,500 runs (500 runs for each of the 17 benchmarks). As shown, multicore contention (NM setup) and monitor overheads (P setups) cause negligible impact in execution time. The safe implementation (S in the plot) incurs some overhead due to stalling the trail process to preserve timing diversity. As shown, such overhead increases quite proportionally to the duration of T_{check} , thus revealing that the impact in execution time of delaying the trail process is far higher than the impact of frequent monitoring. On the other hand, very frequent monitoring is unwanted since this activity needs being scheduled in the MCU, where computation resources are scarce and shared across a number of safety-critical activities. Thus, we select $T_{check} = 0.001$ (1 *ms*) as the best tradeoff for this particular platform, and this is the configuration we use in the rest of this section.

To validate that diversity is attained, we check whether the head process has been always ahead of the trail process in terms of executed instructions. If in a given run, this is not the case at least once, we regard such execution as unsafe due to the potential lack of diversity. Our results confirm, as expected, that our mechanism preserves diversity for all monitoring actions in all runs of all benchmarks. However, if the execution is allowed to proceed without any control, despite the initial staggering (P version), it is quite common having the trail process executing ahead of the head one at least during part of the execution, so that only around 23% of the runs turned out to be safe, at least at the time of monitoring.

C. Performance Assessment

In Figure 4 we show boxplots for all benchmarks with $T_{check} = 0.001s$ (1 *ms*) setup, normalized w.r.t. their re-

Fig. 4: Boxplot of the execution times of our approach for $T_{check} = 0.001s$ (1 ms)

spective (non-redundant) baseline cases. Average execution times increase only by few percents, 5.4% on average across benchmarks and up to 12.4% (bitmnp01). Overall, this indicates that performance degradation due to our mechanism is low and, moreover, as shown before, part of the overheads corresponds to the contention experienced due to running processes redundantly. Note that there are typically 2 or 3 outliers per benchmark, which in practice occur in the order of once every 20 seconds (2-3 occurrences in 500 runs of around 100ms each). This behavior relates to some periodic system activities due to the use of a regular Linux version. Those effects will be avoided when an appropriate AUTOSAR-compliant operating system is deployed on the HPCM.

Absolute average execution times range from 100ms to 120ms for EEMBCs benchmarks, while for matmul it is 282ms, so absolute overheads are typically below 10ms. If such overhead is still regarded as too high, it can be reduced at the expense of increasing the frequency of execution of the monitoring process, as shown in Figure 3.

Overly high execution times in Figure 4 are some tens of ms above their median, and such large variation can only be attributed to the uncontrolled resource sharing. This indicates the convenience for setting appropriate setups in the HPCM to limit the impact of worst-case contention in shared resources.

V. CONCLUSIONS

This paper provides a flexible and efficient strategy to achieve diverse redundancy on COTS multicores and accelerators. Our solution imposes low requirements on the platform, such as having instruction counters per core and the ability to read them remotely – subject to having appropriate permissions. Those requirements are met by most existing HPC COTS platforms, thus facilitating the integration of our strategy. Our evaluation confirms that, as expected, diversity is achieved by construction when our mechanism is in place, execution time overheads are low, and execution overheads in the HPCM and MCU can be traded off easily by increasing/decreasing the monitoring frequency.

ACKNOWLEDGEMENTS

This work has been supported by the Spanish Ministry of Science and Innovation under grant PID2019- 107255GB and the European Union’s Horizon 2020 Research and Innovation programme under Grant Agreement No 826647 (EPI).

REFERENCES

- [1] International Standards Organization, *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
- [2] Synopsys, Inc., “Synopsys Announces Industry’s First ASIL D Ready Dual-Core Lockstep Processor IP with Integrated Safety Monitor,” https://www.eeworld.com/industry_news/synopsys-simplifies-automotive-soc-development-with-new-arc-functional-safety-processor-ip, 2017.
- [3] S. Alcaide et al., “Software-only Diverse Redundancy on GPUs for Autonomous Driving Platforms,” in *IOLTS*, 2019.
- [4] STMicroelectronics, “32-bit Power Architecture microcontroller for automotive SIL3/ASILD chassis and safety applications,” 2014.
- [5] X. Iturbe et al., “The Arm triple core lock-step (TCLS) processor,” *ACM Transactions on Computer Systems*, 2019.
- [6] B. H. Meyer et al., “Cost-effective safety and fault localization using distributed temporal redundancy,” in *CASES*, 2011.
- [7] E. Rotenberg, “AR-SMT: a microarchitectural approach to fault tolerance in microprocessors,” *FTC*, 1999.
- [8] S. K. Reinhardt et al., “Transient fault detection via simultaneous multithreading,” in *ISCA*, 2000.
- [9] S. S. Mukherjee et al., “Detailed design and evaluation of redundant multithreading alternatives,” in *ISCA*, 2002.
- [10] M. Goma et al., “Transient-fault recovery for chip multiprocessors,” in *ISCA*, 2003.
- [11] C. LaFrieda et al., “Utilizing dynamically coupled cores to form a resilient chip multiprocessor,” in *DSN*, 2007.
- [12] J. Fu et al., “On-demand thread-level fault detection in a concurrent programming environment,” in *SAMOS*, 2013.
- [13] S. Alcaide et al., “High-integrity gpu designs for critical real-time automotive systems,” in *DATE*, 2019.
- [14] A. Mahmoud et al., “Optimizing software-directed instruction replication for gpu error detection,” in *SC*, 2018.
- [15] H. Jeon et al., “Warped-DMR: Light-weight error detection for GPGPU,” in *MICRO*, 2012.
- [16] M. B. Sullivan et al., “Swapcodes: Error codes for hardware-software cooperative gpu pipeline error detection,” in *MICRO*, 2018.
- [17] R. Nathan and D. J. Sorin, “Argus-G: Comprehensive, low-cost error detection for GPGPU cores,” *IEEE Computer Architecture Letters*, 2015.
- [18] F. Haas et al., “Fault-tolerant execution on cots multi-core processors with hardware transactional memory support,” in *ARCS*, 2017.
- [19] A. Shye et al., “Using process-level redundancy to exploit multiple cores for transient fault tolerance,” in *DSN*, 2007.
- [20] J. D. Scales et al., “The design of a practical system for fault-tolerant virtual machines,” *Operating Systems Review (ACM)*, 2010.
- [21] G. A. Reis et al., “SWIFT: Software implemented fault tolerance,” in *CGO*, 2005.
- [22] H. So et al., “Expert: Effective and flexible error protection by redundant multithreading,” *DATE*, 2018.
- [23] M. S. Alhakeem et al., “A Framework for Adaptive Software-Based Reliability in COTS Many-Core Processors,” in *ARCS*, 2015.
- [24] A. Shye et al., “PLR: A software approach to transient fault tolerance for multicore architectures,” *IEEE Transactions on Dependable and Secure Computing*, 2009.
- [25] H. Mushtaq et al., “Efficient software-based fault tolerance approach on multicore platforms,” in *DATE*, 2013.
- [26] M. Dimitrov et al., “Understanding software approaches for gpgpu reliability,” 2009.
- [27] J. Wadden et al., “Real-world design and evaluation of compiler-managed gpu redundant multithreading,” in *ISCA*, 2014.
- [28] S. Jain et al., “Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs,” in *RTAS*, 2019.
- [29] V. Vargas et al., “NMR-MPar: A fault-tolerance approach for multi-core and many-core processors,” *Applied Sciences (Switzerland)*, 2018.
- [30] Infineon, “AURIX Multicore 32-bit Microcontroller Family to Meet Safety and Powertrain Requirements of Upcoming Vehicle Generations,” 2012.
- [31] 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.
- [32] C. Hernandez and J. Abella, “Timely Error Detection for Effective Recovery in Light-Lockstep Automotive Systems,” *IEEE TCAD*, 2015.
- [33] C. Hernandez and J. Abella, “Low-cost checkpointing in automotive safety-relevant systems,” in *DATE*, 2015.
- [34] J. Poovey et al., “A benchmark characterization of the EEMBC benchmark suite,” *IEEE Micro*, 2009.
- [35] H. Tabani et al., “A cross-layer review of deep learning frameworks to ease their optimization and reuse,” in *ISORC*, 2020.