Hardware/Software solutions to enable the use of high-performance processors in the most stringent safety-critical systems

Sergi Alcaide Portet

Universitat Politècnica de Catalunya

Computer Architecture Department

PhD thesis

Doctoral programme on Computer Architecture

11th of April, 2023

Hardware/Software solutions to enable the use of high-performance processors in the most stringent safety-critical systems

Sergi Alcaide Portet
April 2023

Universitat Politècnica de Catalunya
Computer Architecture Department

A thesis submitted in fulfillment of
the requirements for the degree of
Doctor of Philosophy in Computer Architecture

Advisor: Leonidas Kosmidis, PhD, Barcelona Supercomputing Center
CoAdvisor: Jaume Abella, PhD, Barcelona Supercomputing Center

Per la meva Mimi i pares, pel seu infinit suport i per la iaia Mela, que vas poder veure com començava la tesis i espero que puguis veure com l'acabo des d'allà on siguis

Acknowledgements

Now that the Ph.D. endeavor is reaching its end, I can look back and reflect on what lifted a lot of the weight of this enterprise from my shoulders. I want to devote this page to the people that helped me in easing the hardships of reaching the highest step in the educational stair, my Ph.D.

I want to start by thanking my advisors, Leonidas, Jaume and Carles and Fran for all the opportunities and knowledge they have given me and all the trust they have deposited in me. Also to my CAOS colleagues during the journey, Jordi, Pedro, David, Xavi, Miguel, Jeremy, Mateo, Ivan and many more who came and went through the group either in Nexus II or in Til·lers. To my university colleagues Enric, Eric, Carla, Julián, Cristian, Alberto, Axel, Killian and Raül, guilty of most of the procrastination but also the moments of relief. I also want to thank my hard-core infancy friends from my hometown, Vic which whom I share not only moments but with some of them a home too: Jan, Arnau, Pol, Guillem R., Guillem O., Sergi and Pere. To my other friends from Vic and Vilanova de Sau and Girona whom I also shared a home with them too: Pol, Victor P., Judit, Victor S, Christian, Marc, Edu and Carles. Most of the fun in my life involved at least one of them. To my current roommates, Laura and Fran and to Matteo, thanks for those afternoons in "Sandwichez" that allowed me to advance in the writing of this Thesis. I also want to dedicate some words to the multiple basketball friends I met along the way who helped me to de-stress my mind: In *Sara Sevilla* games: Natàlia, Adrián, Aitor, Anna, Isra, Helena, Radek, Lorenzo, Matteo and many more. From Vic, Sant Julià and Manlleu: Pau, Martí, Marc, Manel, Miki among ohters. Colleagues from BSC such as Max, Ruben, Victor, Sergi among many others.

I am also thankful for my grandparents. My grandmothers and grandfathers Iaia Mela, Caterina, Antonio, Jaume for their kindness and benevolence And finally, but most importantly, I want to thank my parents M. Carme, Toni, who raised, taught and bestowed on me an attitude that allows me to pursue higher standards and overcome failures and hardships, and my sister who always had my back. I want to thank them for their sacrifices and for giving me everything despite getting much less in return. Thanks to you I can thrive and be content with my life.
Moltes gràcies a tots, aquesta tesi també és per a vosaltres.
Muchas gracias a todos, esta tesis también es para vosotros.
Thank you all, this thesis is also for you.

Abstract

Future Safety-Critical Systems require a boost in guaranteed performance in order to satisfy the increasing performance demands of the state-of-the-art complex software features. An approach to achieve these performance requirements is the usage of High Performance Computing (HPC) components which can deliver more computation power than current safety-critical components. However, the dependability support of these HPC components are not the same as that for the safety-critical components, so HPC components can jeopardize the functional safety of the entire system, especially since some of the highest-criticality functionalities may be executed entirely on top of these components (e.g., neural networks in a Graphical Processing Unit (GPU)). Based on the safety requirements of performance-hungry critical applications, such as those for an autonomous operation, these HPC components must comply with the highest criticality levels, hence including the required dependability support.

The overarching goal of this thesis is to present techniques to achieve that in different HPC components. In particular, we focus on GPUs and multicores. The techniques presented aim at providing *diverse redundant* execution, as needed to avoid Common Cause Failure (CCF)s, which are those defeating safety measures (e.g., pure redundancy) as a consequence of a single-point fault (e.g., a fault affecting both redundant instances identically). Such a solution is comparable to the lockstep execution employed on safety-critical processors such as [1, 2].

The first set of contributions of this thesis focuses on enabling *diverse redundant* execution on a single GPU. We propose two different solutions: (1) a slight hardware modification affecting the internal scheduler of the GPU and (2) a software-only approach that requires knowledge of the hardware resources of the GPU. In these contributions, we also analyze the staggering created due to the CPU-GPU inherent interaction.

Finally, the last contribution relates to multicore systems. Similarly to the previous contributions, we focused on enabling *diverse redundant* execution on this component. However, executing a workload twice in two different cores is something relatively simple with modern programming models (e.g., OpenMP, OpenMPI). The real challenge is in using the limited observability and controllability channels to maintain and guarantee the (time) diversity between these two redundant executions like the lockstep approach. Note that lockstep is an expensive approach that hijacks

half of the cores, which are non-visible to the user and cannot be used for non-critical applications. Instead, if a flexible software-only solution for COTS multicores existed, all cores could be used by non-critical applications when not needed for the safety-critical ones. Thus, maximizing their utilization. To tackle this challenge we proposed a software-only solution with small requirements that can be met by most existing COTS multicore.

These contributions allow pushing the usage of HPC parts even for applications with the highest integrity levels by providing solutions to realize diverse redundancy, a crucial safety requirement for those applications.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AD Autonomous Driving. 1, 3, 4, 6, 8, 18, 19, 59–63, 66–68, 71, 82, 85, 109, 111

ADAS Advanced Driver Assistance Systems. 1, 3, 4, 15, 59, 61

ALU Arithmetic Logic Unit. 46

ASIC Application-Specific Integrated Circuit. 82

ASIL Automotive Safety Integrity Level. xv, 4, 8, 25, 26, 61–63, 67, 71–74, 82, 83, 85–88, 94, 95, 108, 109, 111

AUTOSAR AUTomotive Open System ARchitecture. 115

CCF Common Cause Failure. viii, 18, 21, 22, 36, 73, 74, 77, 86, 95, 100, 109, 111, 112, 117

COTS Commercial Off-The-Shelf. ix, 3–5, 8, 9, 27, 37, 71–73, 79, 82, 86, 93, 95, 100, 105, 106, 109, 110, 112, 114, 122–124

CPU Central Processing Unit. viii, 3, 9, 31, 34–36, 38, 45, 49, 53, 54, 56, 73, 74, 76, 79, 82, 86, 90, 94, 99, 101–103, 105, 109, 115

CRC Cyclic Redundancy Check. 6, 23, 73, 74, 87, 95

CRTES Critical Real-Time Embedded Systems. 59, 61, 66, 68

DCLS Dual-Core Lockstep. 71, 74, 88, 115

DMA Direct Memory Access. 88

DMR Dual Modular Redundancy. 100, 103, 104

DNN Deep Neural Networks. 59

DRAM Dynamic Random Access Memory. 72, 96, 97

DUE Detected Uncorrectable Errors. 107

ECC Error Correction Codes. 4, 6, 23, 73, 74, 87, 88, 93, 95, 112, 114

eHPC Embedded High-Performance Computing. 111, 115

eHPCM Embedded High-Performance Computing Multicore. 112–116, 119, 120, 122

EPI European Processor Initiative. 111

ESA European Space Agency. 7, 124

FIT Failures In Time. 43

FP Floating Point. 90

FPGA Field-Programmable Gate Array. 1, 38, 82, 109

FTTI Fault Tolerant Time Interval. 74, 94, 108

GPU Graphical Processing Unit. viii, 1–4, 6–10, 27, 31, 33–37, 41, 45, 46, 48–50, 52–55, 60–64, 66–68, 71–75, 77–83, 85–88, 90, 93–106, 109, 110, 123–126

GPUs Graphical Processing Units. 1, 3, 7, 15, 38, 45, 60, 61, 63, 64, 67, 69

HPC High Performance Computing. viii, ix, 2–7, 9, 85, 86, 111, 122–124

ISA Instruction Set Architecture. 115

MCU MicroController Unit. 27, 112–117, 122

ML Machine Learning. 59, 60, 66, 67

MPI Message Passing Interface. 115, 116

NRE Non-Recurring Expenses. 71

OS Operative System. 114

PMC Performance Monitoring Counter. 114, 116, 124

PMCs Performance Monitoring Counters. 114–116

QM Quality Management. 26

RAM Random Access Memory. 23

SDC Silent Data Corruption. 107, 110

SECDED Single Error Correction, Double Error Detection. 73

SM Streaming Multiprocessor. 8, 46, 48, 52, 72–74, 77–83, 86, 93–95, 97–100, 104, 123

SoC System on Chip. 3, 5, 41, 61, 83, 124

SOR Sphere of Replication. 20, 88

TDP Thermal Design Power. 37

TMR Triple Modular Redundancy. 9, 86, 93, 100, 103, 104, 109, 123

WCET Worst-Case Execution Time. 41, 61, 66, 68, 69

Chapter 1

Introduction

1.1 Trends on Safety-Critical Systems

In the last decade, the increasing use of Artificial Intelligence (AI) in general, and Machine Learning techniques in particular, such as Convolutional Neural Networks, has increased performance demands and, in most cases, influenced the development of new hardware (ASIC, accelerators) or the diversification of markets for others [8] (e.g., Graphical Processing Units (GPUs), Field-Programmable Gate Array (FPGA)). GPUs, for example, time ago were generally used for graphics rendering and gaming purposes. Nowadays, they can be seen in multiple domains, used to mine cryptocurrencies [9], in the industry 4.0 [10], or even in research to accelerate genome sequencing methods [11], which validates their viability as a computing element for parallel algorithms.

Following the same trend, Automotive Safety-Critical Systems have experienced an increased demand for performance, mostly due to Advanced Driver Assistance Systems (ADAS) and Autonomous Driving (AD). For instance, ARM predicted in 2015 that vehicle computing performance (i.e., ADAS) would increase by 100x in ten years (2015-2025) [12]. This has led, in general, to higher complexity for the platforms used, for example, with the spread of multicores in this type of system. Still, the usage of different cores introduces a new categorization of the platforms based on the types of multicores used. *Safety platforms* includes multicores designed for the automotive domain such as the Infineon AURIX family [1] among others [13, 14, 15]; *Generic platforms* which use only generic multicores mostly used for non-safety functionalities such as multimedia and *Hybrid platforms* which combines generic multicores with a certifiable "safety island" (e.g., Zynq UltraScale+ [16]), which has attracted some interesting researches regarding how these platforms deal with QoS [17]. However, differently from other domains, safety-critical real-time systems have multiple constraints that are as relevant as performance or even more important. These are the constraints related to real-time and safety, since, as the name indicates, these systems need to be *safe* to use. Nevertheless, the increased the complexity of the systems backfires in terms of safety since it makes them even more challenging to get certified as a safe system, in other words, to prove that they are safe to use.

Functional safety, which can be seen as the avoidance of unreasonable risk of failures and malfunction for electronic systems, is an essential aspect of safety-critical systems since these systems may be in charge of human lives, nuclear plants, trains, etc. Malfunctions of those systems may result in death, environmental harm or loss, or severe damage to equipment. To ensure a safe execution, safety-critical systems need to undergo an exhaustive Verification and Validation (V&V) process, which guarantees that the safety goals specified in an early stage of the design are met. These V&V processes and the guidelines to follow at the design, validation and verification stages are specified in the Functional Safety Standards of each subdomain, as we will see later on the Background section 2.

This Thesis focuses on the automotive domain mainly, which is a particular subdomain in safety-critical systems, but most discussions and contributions can also be applied to other safety-critical subdomains with similarities such as space, avionics, or railway.

As discussed, the Automotive domain is entitled to increase its safety-critical systems' performance and provide functional safety alongside them. The adoption of components from other computation domains, mainly from High Performance Computing (HPC), for performance purposes is currently one of the trends being followed by both the industry [18, 19] and the research community [20]. Mainly, GPUs have been one, but not the only (e.g., multicores, FPGA and ASICs) of the high-performance components introduced into traditional safety-critical systems [21, 22]. However, the safety considerations of these systems cannot be easily achieved with the level of complexity of these components and their lack of intrinsic safety mechanisms. This has led to one of the main challenges of the automotive industry as well as a hot topic for the safety-critical research community. Because of the difficulty of the challenge, no system has achieved the maximum Automotive Safety Integrity Level yet, labeled as ASIL-D, following this type of design (HPC Component + Safety-Critical System). Industrial prototype trial failures are an example of how the safety implications in these systems are important [23, 24, 25].

1.2 Challenges in the Automotive Safety-Critical Systems

In the previous section, we have described that the trends in safety-critical systems impose different challenges for coming years. The increased complexity due to the performance requirements, and the need for high safety integrity levels because of the functionalities these systems are in charge of, are aspects tackled by this Thesis. Notably, we focus on both, software techniques and minimal hardware modifications that can be employed on Commercial Off-The-Shelf (COTS) products from the High Performance Computing (HPC) domain to allow them to be used *safely* in the automotive safety-critical systems, avoiding failures due to a variety of fault types.

An extra challenge that comes along with AD is that these systems must be fail-operational. This occurs because these systems have an unprecedented level of control and autonomy and cannot rely on the driver anymore (e.g., there may not even be a driving wheel). A fail-operational system must guarantee the full or degraded operation of the given function even in the presence of a failure. An example of a fail-operational system is the landing system of an aircraft. In the event of a failure, the remaining part of the automatic system must be able to finish the landing by itself. This challenge is new since ADAS or less automated systems were not requested to be fail-operational because they could rely on a *safe state*; for instance, turning on a certain red-light on the car light panel and passing the responsibility to the driver. Therefore, schemes used in those cases, such as employing an external safety monitor in charge of the HPC component/accelerator, do not apply to the problem presented if they do not provide fail-operational capabilities to the system.

1.2.1 COTS GPUs

Graphical Processing Units (GPUs) can deliver a tremendous boost in performance in algorithms that can be parallelized. GPU throughput-oriented nature is based on having a massive number of threads that can manipulate near data in parallel. For instance, if we compare a modern CPU against a modern GPU, we see that an HPC CPU Intel I9 11900KF (11th Generation) has available 16 threads [26] whereas an HPC GPU, the NVIDIA RTX 3080 [27], has 8704 threads available (although they have fewer capabilities).

Traditionally, big COTS GPU manufacturer companies, such as NVIDIA or AMD, were not interested in trying to design GPUs for safety-critical domains. Primarily due to economic reasons since the costs (designing, V&V processes, etc.) were too high concerning the benefits (amount of products they could sell in the automotive market and the price that those markets can afford for GPUs). Nevertheless, both have become more interested in this business in the last few years. By adapting their COTS GPUs into SoC with a focus on the automotive domain (e.g., NVIDIA Xavier [28, 18, 21]), generally combining in the same SoC both a small multicore (up to 8) with a small COTS GPU. Apart from these two, smaller companies with the main business in the automotive domain have designed smaller GPUs to be used on

the car [29]. However, although these GPUs are inside the car, they are used mostly in the multimedia system, which does not have the same safety requirements – if any – as other systems inside the car, since it is not considered a safety-critical system or has not reached high ASIL (e.g., ASIL-B in [22]).

The proliferation of ADAS and AD is responsible for high-performance processing demand in the most safety-critical systems in modern cars. In AD, there are two stages of the pipeline where this huge demand occurs, being the perception stage, the most demanding one. In the perception stage, abundant data from multiple sensors of the car (e.g., cameras, LIDAR) must be processed within a short timeframe to detect and classify objects. In particular, several elements need to be determined, such as the regions where these objects are (e.g., bounding boxes in images), the object type (e.g., a person, a car), and the confidence of the detection. Complex neural networks such as, for instance, convolutional neural networks with tens of layers, are used in this process in the current state-of-the-art in image classification software techniques [30, 31]. These detections are later used across multiple perception iterations to obtain more reliable detections and relevant information about object trajectories and speed. Based on all that information, driving decisions are taken to try to reach the destination while avoiding accidents and adhering to driving rules such as speed limits, signs, and the like.

With this, COTS GPUs are a good fit for the automotive safety-critical systems at least, to address the performance demand since we have seen their parallel-thread brute force matches well with the algorithms used to process abundant data arriving at high frequency. However, as mentioned, safety-critical systems have other constraints apart from performance, which are the main challenges still to be solved. Safety mechanisms to provide functional safety are minimum (ECC in modern GPUs) or not present at all since COTS GPUs were designed for HPC were such a level of safety (and in particular reliability) is not required. For instance, computing the wrong color for a pixel in a frame is not a big issue and probably will go unnoticed. Also, the non-disclosure of the internal structure of the GPUs presents a significant obstacle for the research community, since internal key elements such as the Kernel Scheduler, are only theoretically described, but no information of their implementation is disclosed.

The main challenge of this Thesis for GPUs is to present ideas, either by software or minimal hardware modifications, that enable them to be used in automotive safety-critical systems safely and explain the rationale of why with these techniques/-modifications, those systems will be certifiable for the highest integrity level (ASIL-D in automotive). With this, we want to lay the foundations of how both the automotive industry and the research community may develop safe high-performance computing platforms for AD vehicles in the near future.

1.2.2 COTS Multicores

Another candidate computing component that will remain being used in automotive safety-critical systems is the COTS Multicore. For instance, AD systems generally include control tasks for monitoring and supervision, and lowly parallel processes such

as route planning and object list management, which are particularly suitable for multicore processors. Commercially, multicores appeared in the 2000s as a consequence of mainly three decisive factors:

1. The Memory wall
2. The Instruction Level Parallelism (ILP) wall
3. The Power wall

Firstly, the gap between processor and memory speed led to the appearance of the cache hierarchy, but caches are useful as long as they remain small. If the cache size increases, their speed decreases. Thus, it is faster to have multiple smaller caches operated independently by multiple cores, rather than a single bigger cache for a single core. Secondly, the complexity of finding more parallelism at the instruction level makes it more challenging to keep a high-performance single-core busy. On the other side, in a multicore system, multiple processes can be executed in parallel thanks to scheduling them onto different cores. Finally, the frequency increase to make a single core go faster increased exponentially the amount of power required and heat produced. Still, thanks to Moore's Law [32], the number of transistors available every two years, pushed the system designers to *simply* replicate the cores in a SoC since transistors came *for free*. Although, more complications arose due to consistency and coherency between the private caches of the multiple processors. The trend during the last decades has been the usage of multicore processors rather than single cores.

Lockstep strategy as a way to implement redundancy has been known for a long time [33]. (dual-core) lockstepping employs two redundant cores to work as one. However, since only one core is visible from the user's perspective, it has not been considered a multicore. Multicores' adoption has been delayed with respect to the HPC domain because of the numerous challenges that arose for their usage on the safety-critical domain. The main reasons are:

- The shared resources' timing interference due to concurrent cores running tasks on the same chip. This introduces non-functional dependencies that imply counting for additional delays, which can violate the timing constraints of the tasks.
- COTS HPC multicore products (and HPC components in general) tend to improve the average execution time rather than the worst-case execution time. Generally, this leads to hard-to-determine worst-case execution times and produces a more complex scenario for verification phases such as timing analysis.

Still, multicores adoption in safety-critical systems appeared commercially after the 2010s [15, 34] and has been a hot topic for the research community during all these years [35, 36, 37, 38] and still presents many challenges which need to be addressed such as counting and limiting the interferences in the shared resources or trying to find the worst-case execution time for a task in such a complex system. However, in this

Thesis, we will focus on enabling *safe* execution, by providing functional safety, with software-only techniques. We will address this challenge not only on single-threaded applications, but we will also extend it to multi-threaded applications as well.

1.3 Motivation

The challenges described in Section 1.2 open the door to multiple research lines. Out of those, the main focus of this Thesis is to analyze the design and execution of specific HPC components, mainly GPUs, and multicores, from the reliability/safety perspective. The main target is to offer the chosen HPC components with fault-tolerance capabilities with minimal modification. The type of faults considered is the transient fault, faults that escape the testing phase and require to be dealt with during on-field execution due to their random nature. For this, we commit to the ISO26262 automotive safety standard [39], which imposes the use of some source of independent redundancy to provide detection for these faults. Thus, the main target for each component is to provide and guarantee the *independent redundancy* for the execution of safety-critical functionalities. The *independent* term relates to the diversity required between the two redundant executions enforced to avoid common cause failures due to the same fault affecting both executions similarly. Whereas storage/memory and communication elements are considered protected with the usage of ECC and CRC, computational elements require the usage of lockstep. Still, lockstep can be applied at different granularities. As shown in commercial lockstepped multicores devices [34], the most efficient granularity is on-chip. However, HPC components such as GPU or multicores lack support for lockstep. The goal of this Thesis is to bring such support in the form of software-only solutions compatible with existing HW technology and lightweight HW changes to improve the efficiency of our solution.

With this, specific areas will benefit from our proposals: introducing new techniques/strategies to GPU software designers to achieve a safe execution, improving the safety features in hardware designs inherited from the HPC domain, especially in GPUs, but also in multicores, vector accelerators, etc. Moreover, the automotive industry may benefit by adopting designs/strategies, which can improve the reliability characteristics of their systems (e.g., introducing fail-operational capabilities) with HPC components. In more detail, the expected benefits of this Thesis are as follows:

- **GPU software safe strategies**. GPUs software frameworks such as OpenCL [40], CUDA [41] or even BrookAuto [42] may integrate some of the techniques proposed in this Thesis. This would allow deploying their frameworks in safety-related domains such as Automotive or Space. With this, platforms in these domains would boost in terms of computational power and, thanks to our techniques, still ensure functional safety. All together would make a one step closer to the idea of the Autonomous Driving (AD).

- **GPU hardware designers**. Specific minimal-hardware modifications will be introduced in one of our contributions for GPUs. As explained in the contribution, this technique can be integrated into GPUs from different vendors. Allowing multiple vendors the opportunity which allows them to reuse their products in the safety-critical domain with lower modifications.

- **Other Safety-Critical domains** Traditionally, the safety-critical systems' domain has avoided the usage of HPC in their systems for several reasons explained earlier in the introduction. Nevertheless, embracing the contributions of this Thesis, domains with similar characteristics to the Automotive, such as the Space domain, which is also researching for new technologies to meet the performance requirements for future missions, may also consider employing the solutions proposed in this Thesis. In fact, there are already some ESA projects that are already in this direction [43, 44, 45].

- **Other hardware and software designers** Parts of our safety analysis on GPUs and multicores can also be extrapolated to other hardware components or software programming models. With a particular emphasis, not only on vector-based accelerators, because of their similarities with GPUs designs.

1.4 Contributions

This Thesis advances the safety techniques on HPC components to be used in the most stringent safety-critical systems safely, despite faults, and improve the overall computing performance of these systems thanks to the performance delivered by the HPC components. The main HPC components focused on this Thesis are the GPUs and multicores. However, based on the analysis and the techniques shown, other HPC components could use similar approaches.

The main focus of the work in this Thesis is to achieve a safe execution on the HPC components analyzed which can enable their usage on safety-critical systems. Since HPC components do not have the same safety properties as the rest of the systems, several challenges arise to make sure that these components are safe to use in the safety-critical system's domain. Hence, different solutions have been proposed to tackle these challenges. For instance, for GPUs, a hardware minimal-modification and a software-only solution have been proposed, which have similar capabilities but have different requirements. Hardware designers can employ the hardware solution on newer systems, while software engineers can employ the software-only solution on current systems.

The six contributions of this Thesis are divided into two major topics, based on the HPC component focused. As a summary, Table 1.1 shows the two different Topics, GPUs and Multicores, together with their contributions. For each contribution, we can see the subtopic or strategy used for each one together with the name of the Conference if the publication was published in conference proceedings or the name of the Journal for the journal publications. Below, we list and give more details about each contribution.

Table 1.1: Contributions, the focus of work and publications.

	Topic	Subtopic	Focus	Publications
1	GPUs	Challenges	Introduction to the AD challenges for safety-critical systems	IEEE Micro 2018
2		Hardware	Diverse-redundant execution	DATE 2019
3		Software	Diverse-redundant execution	IOLTS 2019
4		Software	Triple diverse-redundant execution	DSN 2020
5		Hardware & Software	Diverse-redundant execution Triple diverse-redundant execution	IEEE Transactions on emerging topics in computing
6	Multicores	Software	Diverse-redundant execution, Single-thread execution	DFTS 2020

1. To start the Thesis, we begin with a contribution in which we discuss the main challenges in hardware and software design to embrace the usage of the GPUs in AD when satisfying safety regulatory standards. With an emphasis on the ISO 26262 functional safety requirements.

2. The next contribution tackles the COTS GPU from the Hardware perspective. More precisely, it proposes minor modifications of the scheduling policies of GPUs that allow guaranteeing by construction diverse redundancy, thus reaching ASIL-D compliance efficiently without the need of increasing design and/or procurement costs. In particular, it shows how the explicit control of the SMs used for a given kernel, together with the serialization of redundant execution in some cases, allows achieving diverse redundancy with low cost with respect to uncontrolled redundancy.

3. Analogously to the previous contribution, this one addresses the challenge from the software perspective. It analyzes how COTS GPUs can be used to provide diverse redundancy by means of qualitative and quantitative assessments, reaching the following findings:

    - ① GPUs offer the degree of physical redundancy needed to enable some form of loose lockstep execution. In particular, plenty of redundant computation units are in place, and storage and communication means could be protected analogously to those in the microcontroller (e.g., with ECC and CRC).
    - ② Some kernels can be executed redundantly and simultaneously in a staggered manner in a GPU, thus achieving diverse redundancy naturally, whereas others cannot be executed simultaneously due to being either too short or too resource-demanding.

- ③ For those kernels failing to achieve diversity, appropriate software transformations allow them to achieve it, either by reducing the number of resources used simultaneously or by executing them on the serial microcontroller if they are too short.

4. This contribution extends the work of the previous one and analyzes the suitability of COTS GPU to deliver fault tolerance by implementing software-only diverse TMR. It also shows how employing the different subprocedures intrinsic of the CPU-GPU interaction for kernel offloading, a minimum initial staggering ($10\mu s$) between the redundant executions is guaranteed.

5. A Journal extension of the previous contributions was published to conclude the GPU contributions. This work not only contains a summary of the previous contributions but also includes new research material. In particular, the previous software-only solutions required an early inspection of the kernel and its behavior on the desired platform since only *friendly* kernels are guaranteed to be executed in a diverse redundant manner. This contribution presents a protocol and his formal validation to modify a GPU kernel from *heavy* to *friendly*. It also includes the result of a GPU fault-injection campaign to test the fault tolerance capabilities of the software solutions. Finally, it wraps up all the GPU solutions and compares them side by side on a GPU simulator.

6. The last contribution tackles the COTS Multicores and provides a flexible and efficient strategy to achieve diverse redundancy on COTS multicores and accelerators. The solution imposes low requirements on the platform, which are met by most existing HPC COTS platforms, thus facilitating the integration of the proposed strategy. The evaluation confirms that diversity is achieved by construction when the mechanism presented is in place and execution time overheads are low and can be traded off easily by increasing/decreasing the monitoring frequency.

1.5 Thesis Organization

The thesis is organized as follows:

- **Chapter 1 - Introduction:** We start by introducing the main topic, describing the different contributions and listing the publications that have been produced as a result of the work done in this Thesis.

- **Chapter 2 - Background:** Next, we describe the background required to understand the rest of the Thesis. It illustrates the basics of Safety-Critical Systems as well as a brief introduction to the automotive safety standard (ISO26262). Next, follows a short background on the two components addressed in this Thesis: GPUs and Multicores.

- **Chapter 3 - Experimental Setup:** Describes the experimental setup used in this Thesis. We first describe the different hardware setups used in the contributions. Then due to the diverse methodologies employed for each contribution, a dedicated section is used for each contribution.

- **Chapter 4 to 7 - Contributions:** Following, we have the chapters devoted to the contributions: First, we have the GPU contributions, and lastly the Multicore one. GPU contributions start with the contribution that discusses the challenges and opportunities for GPU in the Automotive domain. The next one proposes a hardware solution. Then, the three remaining GPU contributions have been merged in a single chapter. Finally, we finish with the Multicore one. The organization inside each contribution chapter follows the same structure:

  1. Introduces the topic and the opposing challenges.
  2. Faces the main problem, explaining the issues and the proposed solution
  3. The setup and evaluation are detailed
  4. Collects the related work on the topic
  5. Conclusions summarizing the contribution

- **Chapter 8 - Conclusions:** We finish with the chapter that summarizes the work done in this thesis and discusses the potential work that could be inspired in the future.

1.6 List of Publications

As a product of the work done in this thesis, 6 publications have been made (5 in international conferences and 1 in a journal) and 1 more publication has been submitted to an international conference.

1. **Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain** [46]
   Sergi Alcaide, Leonidas Kosmidis, Hamid Tabani, Carles Hernandez, Jaume Abella, Francisco J. Cazorla
   IEEE Micro (Volume: 38, Issue: 6, Nov.-Dec. 1 2018), 2018
   DOI: 10.1109/MM.2018.2873870

2. **High-Integrity GPU Designs for Critical Real-Time Automotive Systems** [47]
   Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, Jaume Abella
   Design, Automation and Test in Europe Conference & Exhibition(DATE), 2019
   DOI: 10.23919/DATE.2019.8715177

3. **Software-only Diverse Redundancy on GPUs for Autonomous Driving Platforms** [48]
   Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, Jaume Abella
   IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS), 2019
   10.1109/IOLTS.2019.8854378

4. **Software-Only Triple Diverse Redundancy on GPUs for Autonomous Driving Platforms** [49]
   Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, Jaume Abella
   50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S), 2020
   10.1109/DSN-S50200.2020.00045

5. **Achieving diverse redundancy for GPU Kernels** [50]
   Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, Jaume Abella
   IEEE TETC (2022) Transactions on Emerging Topics in Computing - Special Section on Defect and Fault Tolerance in Nanoscale Systems for Emerging Computing Paradigms and Applications, April 2022
   10.1109/TETC.2021.3101922

6. **Software-only based Diverse Redundancy for ASIL-D Automotive Applications on Embedded HPC Platforms** [51]
   Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, Jaume Abella
   IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2020
   10.1109/DFT50435.2020.9250750

The authors would like to list some publications that although not part of this Thesis, are a set of contributions the candidate has worked on during the development of the Thesis. Some publications used the work done in this Thesis' contributions as their basis and others are related to the same domain.

7. **GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward** [52]
   Leonidas Kosmidis, Iván Rodriguez, Alvaro Jover-Alvarez, Sergi Alcaide, Jérôme Lachaize, Olivier Notebaert, Antoine Certain and David Steenari
   Design, Automation & Test in Europe Conference & Exhibition(DATE) 2021

8. **GPU4S: Embedded GPUs in Space - Latest project updates** [45]
   Leonidas Kosmidis and Iván Rodriguez and Álvaro Jover and Sergi Alcaide and Jérôme Lachaize and Jaume Abella and Olivier Notebaert and Francisco J. Cazorla and David Steenari
   Microprocessors and Microsystems Volume 77, 2020

9. **A Software-Only Approach to Enable Diverse Redundancy on Intel GPUs for Safety-Related Kernels**(To appear)
   Nikolaos Andriotis, Alejandro Serrano, Sergi Alcaide, Jaume Abella, Francisco J. Cazorla, Yang Peng, Andrea Baldovin, Michael Paulitsch, Vladimir Tsymbal
   SAC 2023 38th ACM/SIGAPP Symposium On Applied Computing, 2023

10. **SafeDE: a flexible Diversity Enforcement hardware module for light-lockstepping** [53]
    Francisco Bas, Sergi Alcaide, Ruben Lorenzo, Guillem Cabo, Guillermo Gil, Oriol Sala, Fabio Mazzocchetti, David Trilla and Jaume Abella
    IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS), 2021

11. **Security, reliability and test aspects of the RISC-V ecosystem** [54]
    Jaume Abella, Sergi Alcaide, Jens Anders, Francisco Bas, Steffen Becker, Elke De Mulder, Nourhan Elhamawy, Frank K. Gürkaynak, Helena Handschuh, Carles Hernandez, Mike Hutter, Leonidas Kosmidis, Ilia Polian, Mathhias Sauer, Stefan Wagner, Francesco Regazzoni
    IEEE European Test Symposium (ETS), 2021

12. **SafeDE: A low-cost hardware solution to enforce diverse redundancy in multicores** [55]
    Francisco Bas, Sergi Alcaide, Guillem Cabo, Pedro Benedicte, Jaume Abella
    IEEE Transactions on Device and Materials Reliability, vol. 22, no. 2, pp. 111-119, June 2022

13. **Assessment of redundant kernel execution on embedded GPU under proton irradiation**(To appear)
    Sergi Alcaide, Alejandro Serrano-Cases, M.A. Romero, Y. Morilla, and Sergio Cuenca-Asensi
    Special issue of the IEEE Transactions on Nuclear Science

14. **SafeX: Open Source Hardware and Software Components for Safety-Critical System** [56]
Sergi Alcaide, Guillem Cabo, Francisco Bas, Pedro Benedicte, Francisco Fuentes, Feng Chang, Ilham Lasfar, Ramon Canal, Jaume Abella
FDL 2022 17th Forum on Specification & Design Languages (invited to the special session on Safety and Security in Cyber-Physical Systems)

15. **SafeDX: Standalone Modules Providing Diverse Redundancy for Safety-Critical Applications** [57]
Ramon Canal, Francisco Bas, Sergi Alcaide, Guillem Cabo, Pedro Benedicte, Francisco Fuentes, Feng Chang, Ilham Lasfar, Jaume Abella
SAMOS 2022 22nd International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation

16. **SafeSoftDR: A Library to Enable Software-based Diverse Redundancy for Safety-Critical Tasks** [58]
Fabio Mazzocchetti, Sergi Alcaide, Francisco Bas, Pedro Benedicte, Guillem Cabo, Feng Chang, Francisco Fuentes, Jaume Abella
FORECAST 2022 Functional Properties and Dependability in Cyber-Physical Systems Workshop (held jointly with HiPEAC 2022 Conference)

17. **Software-only Light Lockstepping for Critical Automotive Multi-threaded Applications on Embedded HPC Platforms**(To appear)
Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, Jaume Abella

Chapter 2

Background

This chapter provides the domain's essential knowledge to understand this Thesis and later, introduces the functional elements used. It is divided into four blocks: First, we introduce safety-critical systems' concepts and taxonomy, as well as relevant mechanisms and safety properties. Secondly, we briefly introduce the certification and automotive safety standards regulating automotive safety-critical systems. Last, we detail the two computing components studied in this Thesis: the Graphical Processing Units (GPUs) and the multicores.

2.1 Basic concepts of Safety-Critical Systems

This section briefly introduces the essential concepts of this Thesis, which include the Safety-Critical Systems domain and its terminology, which is essential to understand the contributions of this Thesis and their value to the Safety-Critical domain.

2.1.1 Safety-Critical Systems and their Taxonomy

The main topic of this Thesis is the Safety-Critical Systems domain in general, and automotive Safety-Critical Systems in particular. Safety-Critical Systems are systems whose malfunction or failure may lead to casualties or severe injury, loss or severe damage to equipment/property, or produce environmental harm. The most crucial goal of those systems is to operate correctly and timely, despite faults. In the Automotive domain, electronic Safety-Critical Systems are all those systems inside vehicles that can produce any of the severe consequences described before upon malfunction (e.g., the braking system or the steering system).

The trend in the last years has been to automate more and more driving functionalities to assist and guide the driver. All these new *intelligent* functionalities have been grouped into what is known as the Advanced Driver Assistance Systems (ADAS). Currently, cars with level three of autonomous driving can already be purchased [59, 60][1]. The expectation is to reach the fourth level by 2030 [61]. However,

[1]Note that J3016 [60] describes 6 automation levels ranging from 0 (no automation at all) to 5 (full automation).

this has increased the computing requirements of the electronic systems inside the car, and will further increase in the foreseeable future. In addition, other systems inside the car that does non-relate to Safety (e.g., the infotainment systems) are also expected to require more computing power and more connectivity outside the car. With this escalation of computing requirements, traditional and more simple systems (e.g., single core) cannot deliver the amount of performance demanded and an increase in the overall complexity of the systems inside a car is expected by using more complex components such as multicore or accelerators.

Next, we introduce some background concepts related to safety-critical systems, these are:

Safety and Functional Safety

Safety is a dependability attribute used to express the absence of unacceptable risk [3], which may lead to the fatal consequences described earlier. In this Thesis, we focus on the Safety of electronic control systems, which is introduced as *Functional Safety* in most safety standards [62, 39].

Faults, errors, and failures

A **fault** is a defect within the system that may (or may not) lead to a logic, timing or electrical error. In the literature, it is described as "*the adjudged or hypothesized cause of an error*" [3]. Faults can be classified according to eight basic viewpoints, as we can see in Figure 2-1. These viewpoints include the dimension of the fault, whether is there a malicious objective etc. The eight viewpoints lead to the *elementary fault* classes [3, 63].

A fault may affect or not affect the internal state of the system. For instance, a cosmic ray may affect the voltage of an internal circuit, but if this modification does not modify the electronic interpretation of the value, (modifying the voltage such as a 0 is now considered a 1 or vice versa), the fault was not activated, and no error was produced, named as a dormant fault in the literature. Instead, if it modifies the *internal state* of the system (e.g., changing a bit in a register), the fault has caused an error and is named an active fault. Then, the definition of an **error** is the modification of the correct internal state of a system due to a fault.

Errors can still be masked, meaning that the error does not affect the system service. For instance, imagine a cosmic ray creates a bitflip, a type of error that changes the internal state in a single element, in a register that used to store a 0 and now stores a 1. The output of this register is then used as the input in an AND gate. If the other AND gate input was already a 0, it would not modify the original/correct output (see Figure 2-2). Other bitflips may affect memory bits that are not currently read, but since eventually they may be used, they modify the internal state of the system. This type of error is classified as Single-Data Corruption (SDC) and is one of the most challenging errors to track down because usually, they remain hidden or are activated a long time after the fault happened. Soft errors are the term used when errors are activated by transient or intermittent faults, whereas hard errors are

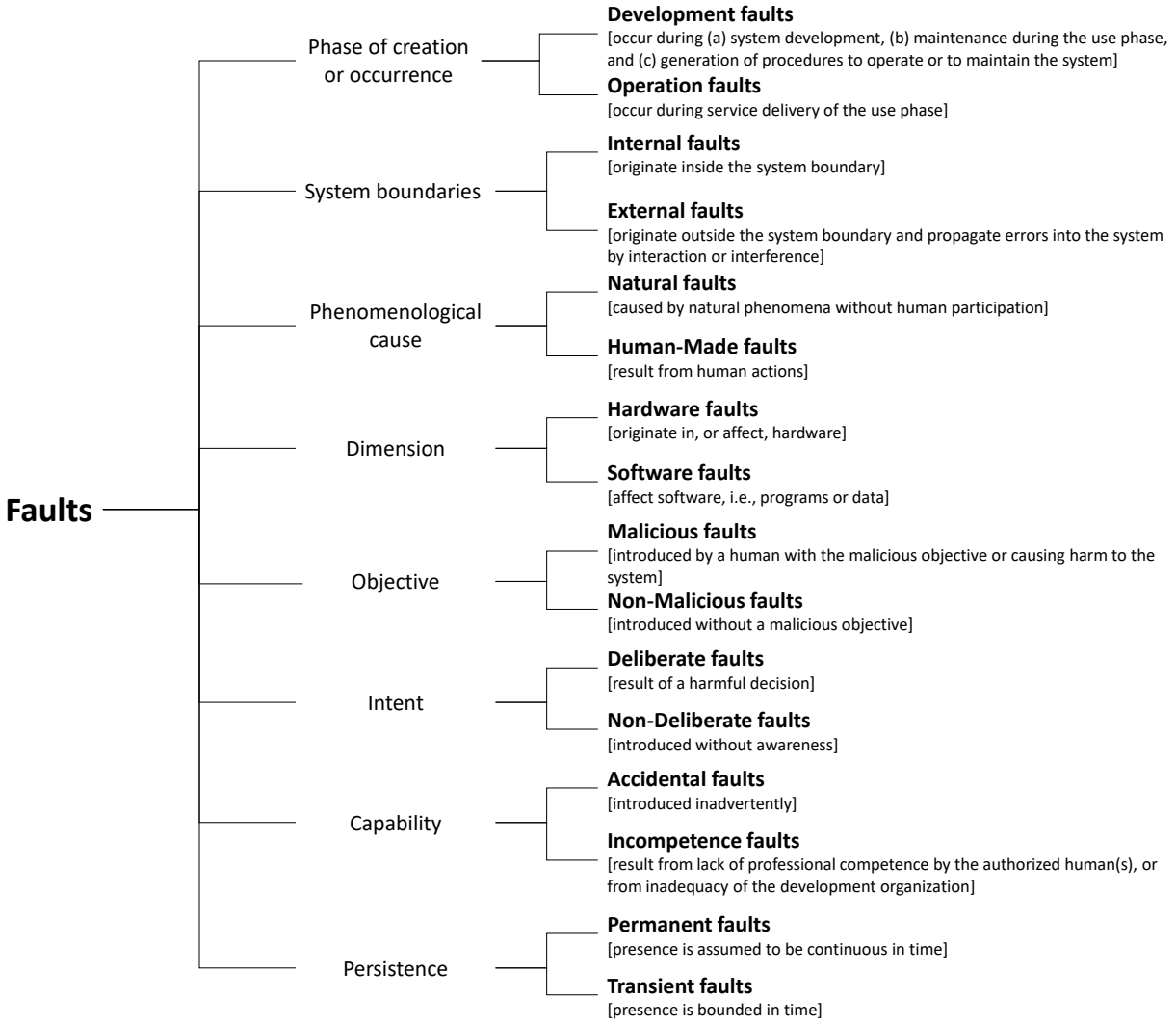


Figure 2-1: Fault classification based on the eight basic viewpoints, extracted from [3].

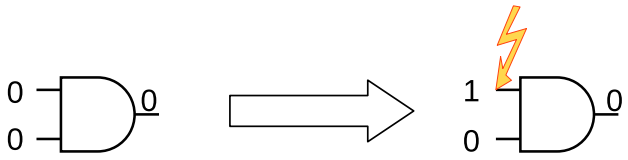


Figure 2-2: Example of a *masked* error. The error modified one input of the gate, but it does not propagate into a failure.

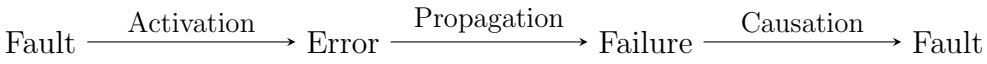


Figure 2-3: Relationship between Fault, Error, and Failure [3].

those errors that are reproducible since their activation is permanent. An error, or modification of the correct internal state of a system, may lead to a service failure, often abbreviated as a failure. A **failure** occurs when the system fails to perform its required operation/service or deviates from the correct service.

To understand the taxonomy of the relationship between faults, errors, and failures, we have included a diagram in Figure 2-3. When a fault modifies the correct behavior of a system, it is defined as activated, and the consequence of this activation is an error. This error can then be masked if it does not produce a system's failure (i.e., bad result, timing). If it does, and the system cannot deliver the correct result inside the timing specified, we say that the error has been propagated and created a failure, as we can see in the arrow in the middle of the Figure. Finally, the relation between the final failure and the original fault is defined as causation, as it is the causality of it.

It is also important to understand situations involving multiple faults and/or failures. Given a system, a single fault is a fault only caused by one harmful human action or one adverse physical event. Multiple faults are two or more concurrent, overlapping, or sequential single faults whose activation, i.e., errors overlap in time. This means that the activations of these faults are concurrently present in the system. Then, classification is made based on the causality of these faults. If the faults are attributed to different causes are referred to as **independent faults**, whereas faults attributed to a common cause are referred to as **related faults**. Related faults generally cause similar errors, while independent faults usually cause distinct errors, meaning that detection mechanisms can identify multiple fault sources. The failures caused by similar errors are named **common-mode failures** or **Common-cause failures** (CCF for short) and are very relevant to this Thesis since they require a peculiar property in the safety mechanism in order to be detected: diversity.

Fail-operational

A fundamental term that needs to be described to understand this Thesis is **fail-operational**. A fail-operational system is a system that must continue to operate correctly and timely despite a failure in the system itself. This is crucial for Autonomous Driving (AD) since the highest levels of autonomy cannot rely on a fail-safe

state. For instance, upon a failure in a critical system, non-autonomous driving cars may detect the failure and turn on a specific light in the driving panel and rely on the driver to notice it and take proper action. Instead, on AD, the systems cannot rely on a driver and need to continue operating despite the failure detected. Therefore, AD systems or components employed on the AD operations must be fail-operational to guarantee, for instance, that the car will not stop driving in the middle of a highway despite the presence of a failure in the decision system.

2.1.2 Redundancy and Diversity

Two mechanisms are required to provide fault tolerance to a system: (1) fault detection and (2) recovery mechanisms. The approach used to protect a system from faults is the following; first, we need a mechanism that can detect faults and avoid their propagation to other systems/components, which is the fault-detection mechanism. Then, a second mechanism is required to correct the error or recover the system's internal state into the correct one, the recovery mechanism. In this Thesis, we will provide proposals and solutions for the first of the two mechanisms, fault detection. Next, we describe two of the strategies used for that purpose, namely **Redundancy** and **Diversity**:

Redundancy

Safety measures often build upon redundancy. Redundant execution has been regarded as a practical approach for error detection, either by means of time or space replication. Mainly, it is well suited to detect soft errors caused by random hardware faults. Since these faults can appear anytime, continuously-active detection mechanisms are required. As its name indicates, the redundancy approach is based on replicating the computation. However, two strategies diverge based on how the redundant computation occurs:

① The first approach can be seen in the Figure 2-4. Since most of the faults are temporary (except permanent faults), the concept is that by re-executing the same operation n times, the majority of these executions will be fault-free (**time redundancy**). Thus, by saving the results of all the executions and performing a comparison of these results, we can detect one of the executions has experienced a fault since the the same result was expected from all the replicated executions and only a fault/error can explain why they differ[2]. Then recovery/restarting actions will need to be considered.

② In contrast, **space redundancy** requires replicating the component which executes the functionality we desire to protect, *Original Module* in 2-5. Similarly, in this case, the fault will likely only affect one or a small subset of the replicas. A comparison of the computed results is required, like in the time redundancy approach.

[2]Note that such assumption holds as long as redundant executions are provided with identical inputs, including the processor state since, otherwise, executions could diverge due to other reasons (e.g., consuming data from a random number generator whose state differs across re-executions).

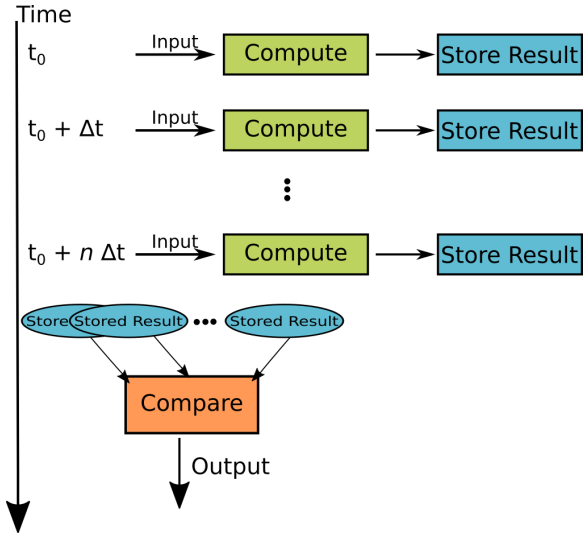


Figure 2-4: n-Time redundancy scheme

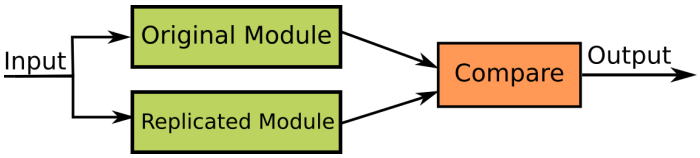


Figure 2-5: Dual space redundancy scheme

Time redundancy (e.g., re-execution in the same core) [64, 65] is particularly suitable to detect soft errors. Still, it requires important hardware modifications for simultaneous re-execution in a simultaneously multi-threaded core. Instead, for a single-thread core, the same program can be executed serially twice, but this will roughly double the execution time. Either way, errors produced due to permanent and intermittent faults (e.g., those caused due to degradation or "telegraph radio noise" [66]) are very likely to repeat in both executions, thus remaining undetected. On the other hand, space redundancy requires the execution of the program in two computing elements (cores), typically simultaneously, which in comparison with the time redundancy scheme is better in terms of execution time, but requires increasing the costs and area because of including a replica of the protected component.

The **Sphere of Replication (SOR)** is a term used to denote the elements that are replicated and thus protected in space redundancy. Since elements inside the SoR will be replicated, the inputs for all these elements must be replicated as well (if signals transporting the inputs cannot be considered safe or are not protected). The replicated elements' output needs to be compared before exiting the SoR to detect any error inside the SoR.

There is a tradeoff in selecting the granularity at which the SoR is defined. Smaller granularities may protect specific components more susceptible to errors and have shorter fault detection latencies. However, they incur higher overheads (timing, power, etc.) since the comparisons are more frequently performed (e.g., at every cycle) and, are also more likely to raise many false positives due to faults that will

not become errors due to fault masking. Furthermore, more intrusiveness, and fine-grained hardware modifications, are required, which increases costs due to three main reasons: 1) large IP modifications are required, 2) lack of flexibility to use the cores in non-lockstep modes (for mixed-criticality systems) and, 3) increases the validation complexity of the circuitry in charge of performing a number of comparisons across cores every cycle [67]. Instead, larger granularities, such as core granularity, do not require those many intrusive hardware modifications. Although comparisons are not as frequent as for smaller granularities, which reduces the timing overheads w.r.t. smaller granularities, it also means that the fault detection latency is increased.

Diversity

In the context of safety-critical systems, diversity is a technique employed to mitigate CCFs. CCFs, as detailed in Section 2.1.1, occur for those faults that can affect redundant components in a similar manner, which defeats the purpose of space redundancy for error detection. To avoid this scenario, diversity is implemented along with redundancy to ensure that faults affecting redundant components will not produce errors in both components or will create different errors. With diversity, the comparison mechanism will detect the fault. While qualitatively diversity is a well-understood concept, quantifying diversity is still an open challenge [68, 69, 70, 71, 72]. Common methods to quantify diversity employed in the industry include fault injection campaigns against CCF [73] and detailed inspection by specialized engineers. Diversity may be achieved in different ways:

- Technology: Two different technologies could be employed to perform the same functionality (e.g., LIDAR vs RADAR)
- Providers: Employ two software companies to develop the same functionality
- ISA: Use two different Instruction Set Architecture (ISA) for the same component (e.g., PowerPC vs x86)
- Design: Design two different chips to perform the same functionality
- Library: Link the software with two different libraries
- ...

The methods mentioned above deliver two diverse components by design. However, these strategies are very costly and increase the complexity of the verification and validation (V&V) processes, thus not practical for many domains. From the electrical perspective, a distinct strategy to achieve diversity is to use different lithographies for two replicated components, which will induce them with different electrical capabilities/behavior in front of faults but also introduces many challenges that difficult its adoption. Instead, one of the most popular techniques to achieve a diverse redundancy solution for computing cores is the lockstep approach, which we will discuss in the next section.

2.1.3 Lockstep

Lockstep execution [33] is a common approach that combines, space redundancy and time diversity. It is used on safety-critical processors [74, 75] to achieve the most stringent certification levels such as ASIL-D in automotive [39] or DAL-A in avionics [76]. Lockstep uses two identical cores as redundant cores. However, only one of them is visible from the user's perspective. The non-visible (shadow) core executes the same software with a forced *delay/staggering* (i.e., one core runs N cycles ahead of the other). Only one core effectively sends and receives external signals (e.g., load/store data, interrupts, etc.) The outputs of the other core are just used for comparison for error detection reasons. A schematic of a lockstep processor is shown in Figure 2-6, where checkpointing and rollback can be implemented either at the hardware level (as shown) or at the software level by raising an interrupt upon a comparison mismatch. With this, Common Cause Failure (CCF) will affect the two executions differently since the *internal state* of the cores will be different (e.g., executing different instructions) at any time. Thus, if two errors are produced due to a common fault, very likely those errors will be different and will be detected by means of comparison.

Similar to *plain* space redundancy, different SoR can also be applied in a lockstep approach. Commercially, HP nonstop servers [77] perform lockstep execution at a very coarse SoR since full boards are replicated. In the opposite direction, some examples exist which used lockstep at instruction or even processor stage SoR [78]. For the latter, different strategies exist to perform the comparison [79]. For instance, either each core can access memory, hence duplicating memory accesses, or a single core accesses memory and addresses and data to be sent/received to/from memory are compared when received or prior to being sent to memory.

Usually, when the two cores are used, like in the schematic (Figure 2-6), the solution is known as Dual-Core Lockstep (DCLS). An alternative is to use three core replicas instead [79, 80, 2], also known as Triple-Core LockStep (TCLS). In this approach, instead of performing just a comparison between the three results, a major vote decision is used. Upon a fault affecting one of the executions, TCLS is able to detect which one of them is the erroneous one since the other two will have the same (correct) output. As said, in DCLS, the system cannot know which one of the executions is the correct one. In front of a mismatch, DCLS requires re-executing the same computation for both cores, whereas, in the TCLS, the execution can continue with the remaining two non-faulty cores executing as DCLS. Eventually, the internal state of a non-faulty core can be transferred to the faulty core without stopping the execution of the non-faulty cores. TCLS is most costly in terms of area and costs as it requires, not just another replica, but more complex logic for comparison. However, it avoids re-execution. TCLS is often employed in systems where re-execution time is higher than the deadline, and re-execution cannot be employed, or where safety regulations already impose higher protection than that offered by DCLS (e.g., avionics).

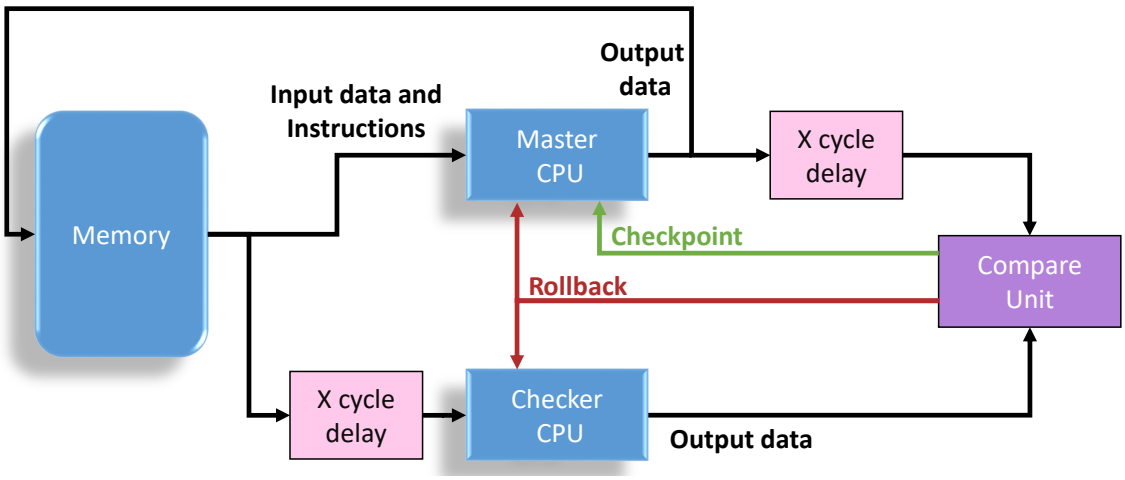


Figure 2-6: An schematic example of a Dual Core Lockstep (DCLS).

In general, lockstep execution is the most cost-effective way to achieve a diverse-redundant execution and certify it, since a single design of the *protected* element needs to be developed easing not only the design but also the Verification and Validation (V&V) effort. Also, since it is a hardware mechanism, it has a smaller fault-detection latency than any software-based approach, which reduces the timing overheads. Furthermore, to obtain higher coverage against CCFs, it is required the use of independent power and clock sources, the electrical isolation of interfaces and may also require physical separation to avoid CCFs. Using two identical cores facilitates these last requirements, since identical – yet independent – redundant power and clock sources can be used.

2.1.4 Memory and data transmission safety mechanisms

Aside from the computing components, memory and data transmissions are also vulnerable to errors. In data transmissions, Crosstalk is one of the fault sources. Crosstalk is a phenomenon by which a signal transmitted creates an undesired effect in another circuit or channel due to undesired capacitive, inductive, or conductive coupling from one circuit to another. These effects can be easily detected by the usage of Cyclic Redundancy Checks (or CRC for short). CRC is an error-detecting code that is computed before the transmission of data and sent together as a *check value*. This value is often computed as a polynomial division of their contents. Later, the receiver may perform the same computations with the data received and check that the result matches the *check value* received.

Random Access Memory (RAM) and memory technologies have historically needed to face reliability issues since their appearance in the 1970s-80s [81]. Thus, error detection and correction solutions have been in place for a long time ago. Generally, hamming codes [82] are employed for Error Detection and/or Correction Codes (EDC/ECC for short). These are a set of algorithms used on the data itself, which produce some bits depending on the data values. A bit-flip can be detected when

performing a check, and the computed codes do not match the stored ones. There is a tradeoff between the number of bits used for encoding, and detection and correction capabilities. The more bits used, the more capabilities can be provided. Of course, this comes at the expense of requiring more space in memories to store the extra bits and the complexity of computing these bits. For instance, a popular coding used is SECDED (Single-Error Correction and Double-Error Detection), which allows detecting two bit-flips and even correcting when only one bit-flip occurs. However, these coding approaches require several bits to be stored for the codes (e.g., 7 code bits used to protect 32-bit data).

In modern RAMs, the ECC and the logic required are managed internally and are non-visible from the core view, commonly known as ECC RAM. Nonetheless, faster memory technologies, such as cache memory, with lower latencies, may not use these codes due to the latency of computing the code and the bit overhead they require since they are often smaller memories. Instead, it is common to use parity bits that allow detecting single bit-flips in exchange for only one-bit of overhead, and the computation is performed in just one cycle. A single bit is computed with the data's parity (if the amount of '1's is even or odd). Modern level-1 cache memories often employ parity, whereas level-2 or level-3 may employ more costly ECC codes.

2.1.5 Nanoscale Level relevance for reliability

The advances in the manufacturing process, electronic design tools, and CMOS downscaling technology enable the production of 7nm and even smaller devices. However, this increased integration exacerbates several threats to reliability such as Process, Voltage, and Temperature (PVT) variability, device aging, transient faults, and permanent faults [83]. This leads to two opposite trends: decreased reliability due to the use of smaller devices and lower supply voltage, thus increasing the susceptibility to particle strikes and small defects, and increased reliability by applying process-level fault mitigation strategies (e.g., improved lithographic process), or deploying fault-tolerant solutions (e.g., hardening, redundancy). With this, the resulting FIT rates, in general, are kept low, but the need for solutions like diverse redundancy increases, particularly for safety-critical systems.

2.2 Certification and Automotive Safety Standards

The release of safety-critical systems into the market needs to adhere to the legal regulations of each country or state, typically set on a per-domain basis. The release depends on a certification process where, depending on the target market, (a) an independent certification authority must approve that the system is suitable and safe enough (acceptable low risks levels) for its intended functionality (e.g., avionics), or (b) the system developer itself develops the product according to the relevant regulations without any certification authority having to approve the process (e.g., automotive). This certification process implies high development costs, where generally, the higher the safety integrity level, the higher the cost of safety certification.

For instance, the highest criticality systems must ensure an extremely low probability of failure, below 10^{-9} [76] per hour of operation (approximately less than once every 114,155 years).

Certification is generally performed against domain-specific standards. In the case of functional safety, most of them derive from the generic international IEC61508 [62] standard for the Functional Safety of Electrical/Electronic/Programmable Electronic safety-related systems, which is suitable for various industrial sectors. Following we can be seen a small list of safety standards from different domains:

- ISO26262 [39] - Automotive
- DO178C [76] - Avionics
- EN50126-2 [84] - Railway
- IEC 62304:2006/AMD 1:2015 [85] - Medical Device (Software)
- IEC62061 [86] - Manufacturing
- IEC61513 [87] - Nuclear

2.2.1 ISO26262 - Automotive Safety standard

ISO26262 is the Automotive Safety standard which is very relevant to this Thesis. Now, we are going to see a brief introduction to the standard, including the requirements imposed and which guidelines are given for developing an automotive safety-critical system. Next, we will introduce the Automotive Safety Integrity Level (ASIL) concept, its different levels, and the ASIL decomposition technique. Finally, we are going to show how the autonomous driving challenge has required the creation of another safety standard, the ISO/PAS 21448 [88] and which relation has with the ISO26262.

ISO26262 is an adaptation of the IEC61508 [62] to comply with the needs specific to the application sector of electrical and electronic systems within road vehicles. In particular, the main difference between both standards is that IEC61508 assumes that safety monitoring is performed aside from the system design, whereas ISO26262 describes a development process where safety monitoring is integrated into the system design itself.

ASIL in ISO26262

ISO26262 uses risk-based integrity levels named as Automotive Safety Integrity Level (ASIL) to classify items. By classifying them in different ASILs, it provides per-level applicable safety requirements to avoid unreasonable residual risk. It also provides requirements for validation and confirmation measures to ensure a sufficient and acceptable safety level.

Severity class	Probability class	Controllability class		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Table 2.1: ASIL determination based on the three classes.

Four different levels are defined being ASIL A, the lowest safety integrity level, and ASIL D the highest one (in order ASIL A, ASIL B, ASIL C, ASIL D). Another level is defined, which denotes no safety requirements to comply with ISO26262, the Quality Management (often abbreviated as QM). These levels are determined based on three classes:

- **Severity**: The potential harm in case of the hazardous event (3 levels)
- **Probability** of exposure: The probability of exposure for each operational situation for each hazardous event (4 levels)
- **Controllability**: Upon the hazardous event, the controllability by the driver or other persons potentially at risk (3 levels)

With the following classes defined, the ASIL level of an element is calculated based on Table 2.1.

ASIL decomposition

Under a given ASIL, some random failure rates are considered acceptable (e.g., $< 10^{-8}h^{-1}$ for ASIL-D, $< 10^{-7}h^{-1}$ for ASIL-C ... [39]), and some specific diagnostic coverage must be achieved, being these levels more stringent for the highest ASIL. Since reaching certain coverage levels and failure rates may impose excessive costs (e.g., requiring expensive safety measures), specific ASIL can be reached with the appropriate combination of lower ASIL components. This solution is often employed since lower ASIL components are cheaper to design and verify than higher ASIL ones. This process is named *ASIL decomposition* and has its own constraints. In order to build a higher ASIL component based on two lower ASIL components, the two lower ASIL components employed must provide *sufficient independence*. This is

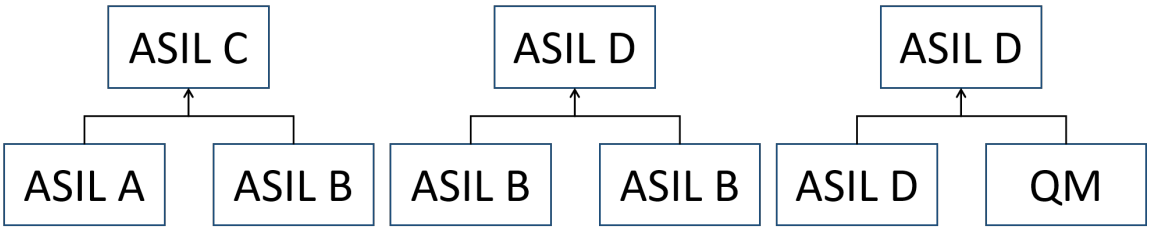


Figure 2-7: Examples of ASIL decomposition.

to prove that both mechanisms are protected against common cause failures (e.g., using two clock signals). A typical example consists of building an ASIL-D MCU by using two ASIL-B cores operating in lockstep. In the case of computing components, diversity is typically achieved using identical cores and software stacks running with some staggering, as we have seen in the lockstep section 2.1.3.

Furthermore, ASIL decomposition is also used for cost reduction trading off availability for fail-safe systems. In particular, a component of a given ASIL (e.g., ASIL D) can be decomposed into, for instance, one ASIL-D component and one or several QM ones, as shown in the right-most example in Figure 2-7. In this case, the ASIL-D component must be able to preserve safety despite the failures of the other components. This could be employed, for systems using High-Performance Computing components, working as QM, as long as an ASIL-D MCU guarantees error management for errors occurring in the HPC component.

When architecting the system and decomposing it into multiple components (e.g., redundant channels), a key characteristic of the system determines the feasibility of some options: whether a safe state exists. If such a state exists (e.g., timely notify the driver and disable the faulty system), then faults can be managed by reaching the safe state within the deadline, thus trading safety for availability. An example of this outcome is, again, the rightmost ASIL decomposition in Figure 2-7, where an ASIL-D system can be built using an COTS GPU regarded as QM and an ASIL-D Micro-Controller Unit (MCU) operating as a watchdog which inherits all the safety requirements, monitoring the GPU execution. The MCU could implement native Dual-Core Lockstep (DCLS), bringing diverse redundancy and means to diagnose faults in the MCU. The MCU, however, must be able to detect faults in the GPU. Appropriate fault detection means are, in general, application-dependent. For instance, software components running on the GPU may run without any type of redundancy as long as the MCU is capable of diagnosing errors in the output coming from software running on the GPU (e.g., output values are compatible with previous output values and with the used input values). Alternatively, software running on the GPU may build upon software-based diverse redundancy orchestrated from the MCU. Such a solution is more generic and usable for any application.

While most systems related to braking and steering resort to some sort of driver intervention to manage potentially hazardous situations, for the highest autonomy levels in AD – levels 3 to 5 as described in J3016 standard [89] – control can only be

transferred to the driver in some circumstances (levels 3 and 4) or simply can never be transferred (level 5). With this, the example of using an ASIL-D monitor with a QM device in charge only of these functionalities is not valid anymore since, upon an error in the QM device, the ASIL-D monitor will be unable to carry out the task timely. Hence, such a system – a fail-operational system – will require fault tolerance.

If the deadline permits it, we may build on the same solutions for fault detection as for fail-safe systems, combined with appropriate safety measures to keep the GPU providing service (e.g., re-run with the same or newer inputs). Note that if the FTTI is too tight to tolerate re-execution, then we may need to build on diverse redundancy with 2oo3 (2 out of 3) redundancy levels on the GPU, being the MCU the one acting as the voter. In both cases, either with fail-safe or fail-operational systems, the MCU itself might not suffice to provide fault tolerance and redundant GPUs may be needed or the accelerator used must be certified to reach ASIL-D on its own and be fail-operational.

ISO26262 and ISO21448 (SOTIF)

Safe road vehicle development was first introduced in the ISO26262 standard [39]. The standard was built on the implicit assumption that system control and managed data could be regarded as separate elements. With this, a control system, despite being aware of the type of data that will be processed (e.g., range values, frequency of arrival, etc.), is designed without needing the data. Thus, with respect to the safety requirements, casuistic control can be implemented and specified without the need for actual data. Instead, data is required for the verification and validation processes as a way to gather evidence on the safety requirements compliance of the design. Some functionalities in the autonomous driving process, such as perception and prediction, require going through a safety life cycle because they inherit safety requirements. These functionalities do not follow the usual specification, design, and implementation process of conventional automotive functionalities. Alternatively, the accurate and efficient implementations of some functionalities (e.g., camera-based object detection) are mainly realized by employing machine learning and, in particular, deep learning using neural networks.

Neural networks in particular, and machine learning in general, come with three challenges not considered in the definition of ISO26262.

First, neural networks cannot separate control and data as opposed to usual automotive functionalities. The data itself greatly specifies the decision algorithms. The data dictates the weights of the different network layers performing processes like object recognition. Therefore, data needs to be used during the training phase, and the weights and the neural network itself form the control algorithm.

Second, the accuracy of a neural network is determined by the scenarios and data used for training. For instance, a neural network trained with highway driving conditions may decide poorly or erroneously in an urban scenario and vice versa. Hence, special care must be taken to select the appropriate training scenarios so that the neural network's functionalities operate correctly in real driving conditions.

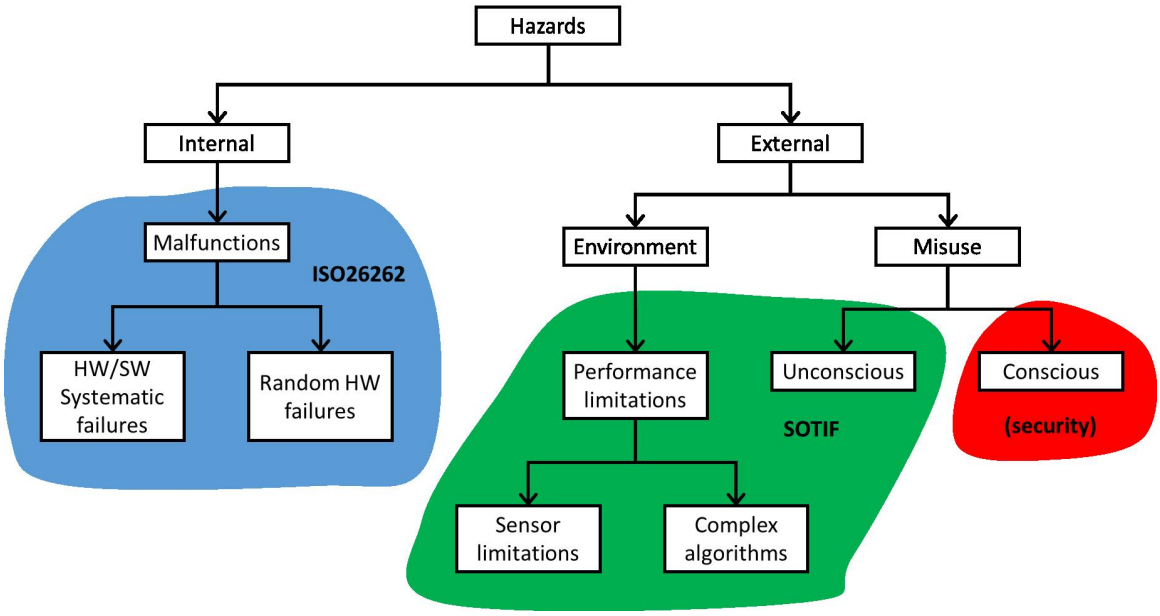


Figure 2-8: Hazard decomposition and scope of ISO26262 and SOTIF.

Usually, a tradeoff exists between generalization and specialization: the broader the set of scenarios required to operate, the more general the training must be, and the less optimal under each specific scenario the performance will be. On the contrary, specialized training for very few scenarios may make the system highly accurate and optimal for these scenarios, but ineffective or dangerous to drive in other scenarios not considered.

Last, neural networks are stochastic in nature. Their outcome comes in the form of multiple outcomes with different confidence levels for each one. Although those confidence levels can be categorized into deterministic outcomes using thresholds (e.g., if confidence is above a certain level, the object is detected as real or otherwise discarded), their current behavior does not conceptually match with the expected behavior of systems described in ISO26262. Instead, in ISO26262, results expected are to be binary, correct, or erroneous, with the latter requiring some sort of safety measures to prevent failures. Still, those characteristics bring opportunities for specific safety measures that are not well-matched to deterministic systems.

Overall, the use of data-defined systems, with training scenarios determining the system's behavior, and with stochastic behavior, do not match well with ISO26262. Therefore, an appropriate safety standard has been released recently with those system characteristics in mind: ISO/PAS 21448 "Safety Of The Intended Functionality", also known as SOTIF [88].

With the release of SOTIF, hazards can now be divided among the two safety standards (and security), depending on their source. We can see a schematic in Figure 2-8. As shown, the scope of ISO26262 includes the hazards caused by malfunctions whose source is, to some extent, internal (i.e., related to the system's control logic). This includes systematic failures, either hardware or software related, which must be avoided by design so that the remaining risk can be regarded as residual. Moreover,

ISO26262 considers random hardware faults, which cannot be avoided and can occur during operation. ISO26262 imposes the deployment of safety measures to guarantee with enough confidence that those faults will not lead to a system failure.

On the other side, the scope of SOTIF spans mainly external factors. This includes input data either from the environment or from driver commands. Particularly, SOTIF is designed so that the system is intended and trained considering expected input data. Differently from ISO26262, SOTIF needs the data for its specification and design. While in ISO26262, data is used for testing only, and limits the coverage of the test campaign of a correct-by-design item, in SOTIF, data determines the system's design. Thus, lacking relevant data may lead to a failure to operate correctly and/or timely. For instance, from time to time, test cases in real scenarios end with accidents, such as the Uber autonomous car, which crashed and killed a pedestrian [90].

2.3 GPU

A Graphical Processing Unit (GPU) is a computing device specialized for computer image processing acceleration and output display. Because the provided high computing capacity enables the acceleration of parallel computationally-demanding algorithms, its usage has expanded in the last decades from its original purpose (e.g., GPUs for image processing and gaming) to other domains such as safety-critical autonomous systems [91, 92], high-performance general-purpose computing (including supercomputers [93]), or even to mine cryptocurrencies. The main philosophy of GPU architectures is to maximize throughput, namely the volume of data processed per time unit. With this goal, GPUs are built with a significant amount of threads able to operate different data and with a memory able to feed all those threads if appropriate data access patterns are employed.

2.3.1 GPU Architecture

Following NVIDIA (and OpenCL) terminology, the GPU fundamental processing units are the Streaming Multiprocessors (or compute units), SMs for short which we can see in orange in the left part of Figure 2-9. Each SM contains several scalar cores (or processing elements) and other resources such as a register file, shared memories, and warp schedulers. A scalar core is a pipelined Arithmetic Logic Unit (ALU) capable of executing integer and floating-point instructions (in yellow, on the right side). With this, SMs can execute multiple, usually 32, threads simultaneously, grouped in *warps* (or wavefront). Other specialized units coexist with the scalar cores in the execution pipeline, such as the Load/Store units used by memory instructions (in green), the special function units (SFU), optimized for certain mathematical functions, or the tensor cores in most modern GPUs, among others (in magenta).

GPUs cannot be the only computing component in a system since they lack some key features (e.g., interrupts, I/O support) to perform all the functions required in a system. Instead, they are mainly used as accelerators and require a host, composed of one or more CPU cores or CPU for short, to send both the data and the commands, and perform all these functionalities that GPUs cannot do. GPU functions are launched asynchronously from the host; functions are known as *kernels*.

A schematic of a typical CUDA workflow can be seen in Figure 2-10. Before launching a *kernel*, the user must prepare the data to be processed. ❶ Allocating the required memory on the GPU and ❷ transferring the data from the CPU (host) memory to the GPU (device) memory. Once the GPU has finished the ❸ kernel execution, the reverse memory transfer operation is required, ❹ allocating space for the result data in the host memory, and transferring the results back from the device to the host. Finally, ❺ deallocation of the data allocated in the GPU memory is also required.

When launching a *kernel*, the user specifies a *grid* configuration, with the number of blocks and threads per block to be executed. Hence, the user selects the number of threads and splits them into blocks. The kernel/global scheduler (in green in the left

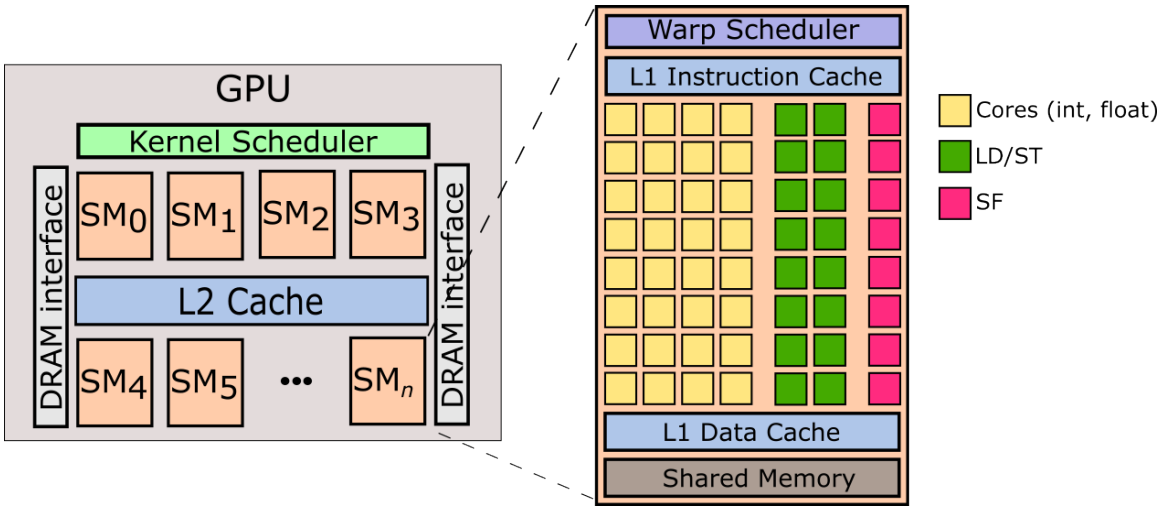


Figure 2-9: Generic GPU schematic. Inside an SM we can observe the three different functional units: (1) Cores which operate with integers and floating point operands, (2) Memory Operations (load and stores) and (3) Special Functions. For space constraints, some components inside the SM have been omitted (e.g., Register File or operand selection)

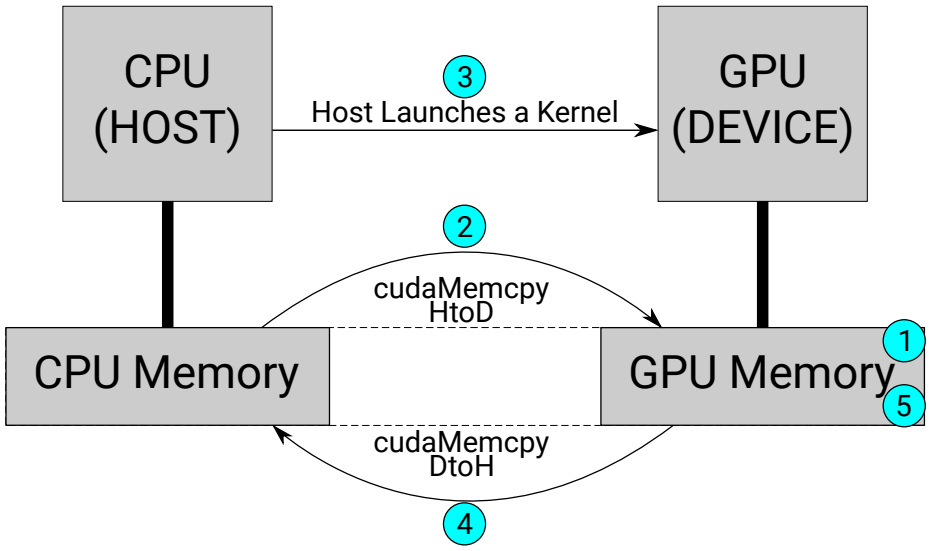


Figure 2-10: Typical CUDA Workflow

part of 2-9) then distributes the blocks into SMs based on the scheduling algorithm, which considers the current occupancy of the SMs. Finally, the warp-scheduler(s) inside the SM, fetches, decodes and issues instructions to the execution pipelines. During the execution, resources like shared memory and the register file within an SM are shared by the threads executed in the same SM.

GPU specialization

Currently, available GPU devices go beyond the basic definition of GPUs since, due to their utilization in different domains, they have been specialized with some differences. For instance, GPUs can be found as part of SoCs (System-on-Chip) in embedded systems, which means that both the host (CPU) and the device (GPU) are literally in the same chip. The particularity of these systems is that typically there is no dedicated memory for the GPU; instead, the memory is shared between the host (CPUs) and the GPU. For that purpose, for example, it is possible to use OpenCL Shared Virtual Memory (SVM) and CUDA Unified Virtual Memory (UVM) features, but taking into consideration the need to manage memory coherency. The NVIDIA Xavier [18] is an example of these systems, which contains 512 scalar cores inside the GPU and 8 CPU Cores (NVIDIA Carmel Armv8.2). This product targets the industrial, automotive, and space domains. Other examples are the GPUs used in scientific computing, where power consumption is not as relevant as throughput and larger GPUs (in terms of SM count) are used despite their higher power consumption.

2.3.2 GPU Software

GPU software programming is commonly based on standardized APIs or associated industry-standard programming languages, such as Compute Unified Device Architecture (CUDA), Open Computing Language (OpenCL), Open Graphics Library (OpenGL) variants, Vulkan, Open Multi-Processing (OpenMP), and Open Accelerators (OpenACC).

Some of them are primarily built for graphics, such as OpenGL variants, while others are for general-purpose computations (e.g., CUDA, OpenCL, OpenMP, OpenACC). Some of them support both graphics and computing, such as Vulkan.

OpenGL SC (Safety-Critical) is the only one of these APIs designed to comply with safety-critical systems' requirements, but it is primarily intended for graphics processing. All the aforementioned general-purpose GPU programming models present challenges for their use in critical systems since they violate design guidelines used in critical systems' software development and certification. In particular, all these programming models require dynamic GPU memory allocation and pointers, whose use is discouraged by several safety-critical standards and language subsets for critical systems. For this reason, Kosmidis et al. [42] proposed the use of a subset of Brook [94], a CUDA-like language that is the predecessor of CUDA and OpenCL. This subset, Brook Auto, is shown to be appropriate for the development of safety-critical GPU applications and eases their certification [95].

2.3.3 GPU in Safety-Critical Systems

GPUs are increasingly considered for the development of safety-critical systems in multiple domains such as transportation (e.g., automotive [96, 97, 98], railway [99], avionics [100, 95], space [101, 44, 91], and industrial machinery [102, 103]). This trend answers the incremental computational requirements for the systems in those domains. Multiple papers exist in the literature discussing safety strategies and potential applications of COTS GPUs in those domains. For instance, Xie et al. [104] summarize recent advances in automotive functional safety design methodologies and discuss ISO26262 [39], SOTIF [88] and the trends in functional safety design methodologies. In [96], Olmedo et al. discuss some of the challenges that system engineers have to face in terms of real-time constraints and functional safety when the GPUs are chosen as accelerators for ADAS. A technical report from NASA [91] was written by E. Wyrwas, providing some insight into GPU architecture and its potential applications in the Space domain. In Europe, Kosmidis et al., [43] presented the project GPU4Space, in which the usage of GPUs in space has been evaluated together with the European Space Agency (ESA).

Dos Santos et al. [105] proposed the concepts of KVF (Kernel Vulnerability Factor) and LVF (Layer Vulnerability Factor), metrics relating reliability and GPU architecture. Later they applied a fault injection campaign using SASSIFI [106], an NVIDIA open-source framework to perform error injection campaigns for NVIDIA GPUs, by inserting instructions. Moreover, a selective hardening of the well-known algorithms YOLO [31] and HOG are proposed and later evaluated. Yang et al. [107] identified common pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems, which later Amert et al. [108] benefit by reverse-engineering the kernel scheduling in the NVIDIA TX2, which is a crucial factor for evaluating the contentions suffered by the GPU applications (kernels).

Following this trend, in the last years, many works have been presented evaluating the safety capabilities of the GPUs. For instance, Oliveira et al. [109] performed a radiation evaluation using a neutron beam. The efficiency and efficacy of the proposed duplication with comparison (DWC) strategies were experimentally evaluated and compared with the chip's ECC mechanism. As demonstrated, ECC is efficient in reducing the SDC rate in modern GPUs but significantly increases the occurrence of functional interruptions (FI). DWC strategies can be more effective than ECC when input data are duplicated. Memory is also evaluated, in [92]; an in-depth analysis of GPU radiation sensitivity both at low-level – by accessing the raw memory structures error rate – and at operative-level – by measuring the silent data corruption and the Functional Interruption rate of a representative set of parallel applications. Moreover, in [110], Cini et al. evaluate the memory interference effects of multicore CPUs and GPUs.

Other fault injection works include [111, 112] and [113], where GUFI is presented as a fault injection tool running on top of the well-known GPU simulator, GPGPU-sim [114, 115]. Furthermore, in order to evaluate the resilience of the GPU in real

scenario conditions, a fault injection campaign was performed using accelerated neutron beams in [116] on a GPU while it was running a deep neural network (DNN) used by object detection algorithms.

Another set of papers proposes to improve GPU's reliability. The most extended approach is by applying a redundant execution, either by utilizing sleeping threads [117] or adding an extra Streaming Processor to execute the redundant work [118] among other techniques [119, 120, 121]. Another approach is to reduce the accuracy of the applications, which improves the reliability against SDC in the register file [122]. In [123], Pilla et al. applied software hardening when executing the Fast Fourier Transform (FFT). Wadden et al. [124] proposed compiler-managed techniques to create redundant threads at different granularities. Finally, Condia et at. [125] evaluated a hardening technique in the register file by performing a fault injection.

2.3.4 Path Towards Certification

Including high-performance devices such as GPUs in safety-relevant products is already an ongoing challenge in several industries, including the automotive one, which has been successfully tackled under very specific circumstances, such as specific GPU devices and runtimes. We refer the interested reader to the survey in [126] for further details on the state of the art. However, the general adoption of those devices in safety-relevant products has still some challenges ahead.

The techniques proposed in this thesis, and those targeting GPUs in particular, are subject to a large subset of those challenges affecting the general adoption of GPUs in safety-relevant products, such as the use of devices fabricated with tiny technology nodes, and using practices uncommon in the fabrication of ASIL-D certified microcontrollers. However, those challenges are not specific to the solutions presented in this thesis and, instead, we assume that they will be eventually solved. This is analogous to the integration and evaluation in automotive systems of GPUs whose power consumption largely exceeds power targets for the domain, expecting power to decrease generation after generation so that it is low enough by the time those GPUs are fully embraced by commercial automotive autonomous driving systems. However, the solutions in this thesis bring some new certification challenges when compared to the current practice to deploy GPU-based safety-critical systems. The current assumption is that GPU devices are fully duplicated for diverse redundancy purposes, bringing independence through the use of separated devices, potentially installed in separated boards and potentially with separated power sources to gain independence. However, GPUs used are identical, hence meaning that solutions exist (or are foreseen) to manage systematic errors that could relate to the GPU design itself. Our assumption is that an ASIL-D compliant CPU orchestrates the redundant execution on the GPUs, and also monitors execution for error detection. Upon an error, such CPU takes corrective actions such as, for instance, resetting those GPUs and re-executing. If the error relates to a physical defect, such defect is highly likely to manifest analogously in both GPUs, and hence, either it is assumed that such an error cannot happen or, if regarded as possible, system-level measures can preserve safety anyway.

In our case, we consider a single GPU to achieve diverse redundancy, which brings important advantages in terms of reliability due to having only one instead of two devices (hence halving the exposure to transient faults). On the other hand, such a solution brings the following risks:

1. Using a single device instead of two.

2. Having some shared component across redundant kernels.

Regarding the former risk, it is analogous to the use of a single ASIL-D compliant CPU where diverse redundancy is implemented internally by using replicated cores, and ECC-protected memories. Therefore, we assume that the same considerations used for such CPU can be used for our single-GPU-based solution. Note that those solutions may include an external watchdog to check aliveness and reset the device upon an error, which in the case of the GPU would be the CPU itself. The implications of an unrecoverable permanent fault in the GPU would be comparable to those of an unrecoverable permanent fault in the CPU in CPU-based systems. Hence, we assume that the same assumptions hold in both cases (e.g., regarding such a scenario as too unlikely, or using alternative methods to preserve safety). Regarding the latter risk, our method imposes the use of different computing resources by construction, hence avoiding CCFs in the computing components. Shared intra-GPU memories and interconnects are expected to be ECC-protected, hence providing intrinsic diverse redundancy against CCFs. However, some parts of the logic may be shared and non-replicated, such as the hardware scheduler, as explained before. For those components, we assume that one of the following two alternatives is needed:

1. Managing faults in such a shared component in the same way as faults affecting other shared parts of the device (e.g., the power and clock networks).

2. Replicating (with some form of diversity) such hardware component to avoid CCFs by design. Note that combinational logic excluding computing parts is tiny in a GPU, so replicating them would incur negligible costs.

Overall, we foresee no extra certification risks for our contributions in the context of GPUs, and all considerations should have a manageable impact.

2.3.5 Power and Reliability Considerations

GPUs offer a tradeoff between specialization and general purpose to realize acceleration. For instance, they are more efficient than CPUs for massively parallel computations (i.e., lower energy per FLOP), but still are sufficiently general-purpose to ease programmability and use across different problems, which typically implies sacrificing power efficiency to keep such malleability. Moreover, the number of GPUs needed in the automotive domain is orders of magnitude lower than those needed for laptops, desktops, servers, and data centers. Hence, while some chip vendors have tailored their designs for the automotive domain (e.g., NVIDIA Jetson GPU family [127]), changes to the architecture have been limited in practice, and automotive

GPUs inherit the main characteristics of their counterparts for the mainstream markets, where performance is maximized within relaxed Thermal Design Power (TDP), and reliability is a less critical concern than in automotive systems since faulty parts can be replaced more easily.

In the context of automotive systems, instead, performance is not the main driver for GPUs, and they must be deployed aiming at minimizing energy consumption and reliability concerns. Hence, while some safety requirements require duplication of the computation, such duplication must be much preferably performed on the chip, as for DCLS (Dual-Core LockStep) CPUs to avoid having 2 GPUs, whose joint energy consumption would be higher than having a single device, and whose reliability would be a concern due to having two parts plus some wiring to interconnect them instead of a single integrated device. Therefore, the target of our work is deploying diverse redundancy on a single chip, even for GPUs.

2.4 Multicores

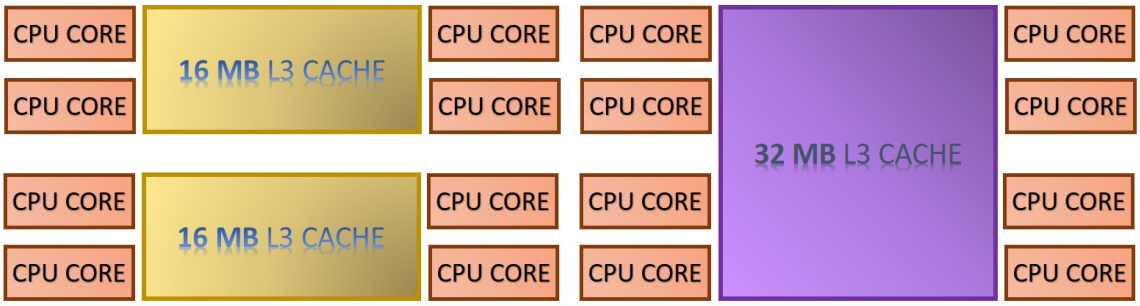
In this section, we will discuss Multicores and their solutions in the safety-critical domain. First, we will start with an introduction of Multicores, later we are going to review their general architecture and finally review their applications in the context of safety-critical systems.

2.4.1 Introduction to Multicores

The usage of single-core devices with higher frequency is not considered competitive in several domains due to multiple reasons: the increased sensibility to Electromagnetic Interference (EMI) [128], lower reliability of thermal dissipation fans [129, 130] and cooling systems volume and weight [131]. Instead, Commercial Off-The-Shelf (COTS) Multicore devices are becoming dominant in silicon manufacturer roadmaps [132, 133, 134, 135, 1, 136, 137] and can provide a cross-domain potential solution, e.g., automotive [138, 139, 140], avionics [132, 141], railway [142, 143], industrial control [144, 145] and medical applications [83]. In the context of this discussion, dual-core devices that operate in lockstep mode are considered single-core devices since, from the user's perspective there is only one core available.

2.4.2 Multicore Architecture

As mentioned in the Introduction (see Section 1.2), Multicores became commercial when the single-core systems development was no longer economically viable, and the transistor count continued scaling following Moore's Law [32]. To quickly provide a new competitive system inside the timing constraints of the market, and due to the number of transistors available, system designers conceived the idea to replicate the cores already used and include them in the same SoC. Using exact replicas was the most efficient strategy to produce Multicores since they avoided designing multiple cores, and integrating them is easier and more efficient. With this, only one core



(a) AMD Zen 2 Layout **(b)** AMD Zen 3 Layout

Figure 2-11: AMD Zen 2 and AMD Zen 3 Multicore Layouts

needed to be developed and, compared to using diverse cores, the verification and validation processes were easier. For instance, resources such as the clock generator or the power system may also be shared or duplicated. However, some parts (e.g., memory systems) of the system became more complex due to having multiple cores having access to them. Memory coherence is an example, as at any point in time, reading a memory position should have the same value no matter which core performs the reading. Therefore, private caches are more problematic than before. Arbiters and schedulers of *shared resources* got a vital role to play in multicores since now they have to manage the accesses to shared resources reasonably.

Generally, Multicore systems group cores in *clusters*. Each core has a certain number of private cache memories (usually L1 data and L1 instructions), other cache memories shared among the cores within the cluster (usually the L2 or parts of it), and the rest is shared among all the system clusters. There are many different strategies when dealing with the *grouping* of the cores and how they access/share memory. We can see a comparison of two different cluster examples in Figure 2-11. On the left side, AMD Zen 2 released in 2019, groups the CPUs in clusters of 4 cores and shares 16 MB of L3. Differently, Zen3, released in 2020, arranges cores in clusters of 8 cores and shares 32 MB of L3.

The term *heterogeneous computing* is employed to define platforms that include different (heterogeneous) computing elements. It appeared as a consequence of the generality of the computation in general cores. Generic cores can execute any computation but because of their generality, some computations may have limited efficiency. On the other hand, accelerators are built to be very computational or power efficient for very specific tasks, thus *accelerating* them. In a system, once a particular task is very often executed and is not very efficiently executed, it may be interesting to design a particular component to perform this task faster and more efficiently. For instance, a long time ago, floating-point units (FPUs) were not inside the pipeline and were considered accelerators. Due to their extended use in many applications, system engineers included them directly in the pipeline. Nowadays, almost all CPUs include FPUs inside the pipeline in the execution stage due to their considerable utilization in most applications. Heterogenous computing includes dedicated accelerators, GPUs, FPGA, vectorial units, but also clusters (groups of cores) with different cores (for

different purposes). These other components may be memory coherent or not with the rest of the system. *Memory coherent* means that any modification made by these components can be seen automatically by any other component of the system, whereas non-coherent does not. The reliability of these components is also an important issue. Firstly, because they can perform computations that belong to a critical task which means that they inherit safety requirements as well. Instead, if not used for a critical task, they can still affect the execution of critical tasks from different angles. For instance, they can create contention on a shared resource used by a core running a critical real-time task and increase its execution beyond its deadline. Alternatively, they could also suffer a fault that can lead to a modification outside their memory space, potentially jeopardizing the entire system. As we will see, these characteristics fall under the description of mixed-criticality systems that we are going to discuss in Section 2.4.3.

Recently, a new trend in HPC has been the usage of the BIG.little architecture (e.g., Apple's new SoC M1 [146], Intel i9 12900K [147]) after several years of homogenous Multicore clusters; this is using multiple clusters of the same cores. Instead, the BIG.little architecture [148] was first created by ARM to be used on SoC-targeting smartphones. Generally, it contains two types of cores divided into two heterogeneous clusters: including big cores with high computing power and high power consumption and smaller cores with modest computational power but more energy-efficient and, therefore, lower power consumption. Ideally, when the system requires high computational power, the BIG cores are powered-up, and the little ones are powered off. Whenever higher computational requirements are not required, a switch is performed by powering off the BIG cores and powering up the small ones. This is particularly useful on smartphones and battery devices to deliver high computing performance when needed and save battery when not needed; an example schematic can be seen in Figure 2-12. This trend is currently emerging in safety-critical systems, for instance, with the adoption of platforms such as the Xilinx Zynq UltraScale+, which includes an Arm Cortex A53 core cluster and an Arm Cortex R5 core cluster. Such a platform is being considered in a number of avionics and railway projects, among others [149].

2.4.3 Multicores in Safety-related systems

The usage of Multicore devices, not counting lockstep as a Multicore device, for safety-critical functionalities, has been a research trend in the last 15 - 20 years [150, 151, 132, 152, 153, 154, 155]. Having multiple cores available in the system allows to execute of more than one process simultaneously or use them for parallel applications [156], hence improving the computing power substantially with respect to a single-core system.

However, Multicore devices also present new challenges, which have delayed their adoption in real-time and safety-critical systems. These challenges are mainly, but not only, due to the real-time guarantees that these systems need to provide as dictated by the safety standards. For instance, some resources are shared among cores, such as a bus or the main memory. These shared resources may be accessed simultaneously

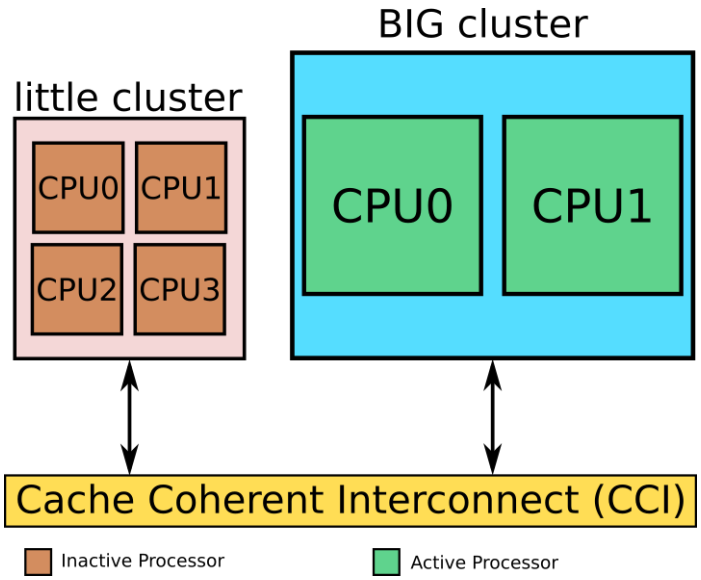


Figure 2-12: Example scheme of a Multicore with Big.little architecture. In this time instant, the BIG cluster is active whereas the little is inactive.

by multiple cores. However, in general, they cannot serve all requests simultaneously and introduce some form of serialization. Therefore, the core(s) granted access creates contention on the other cores waiting to access the resource.

A good example is the use of shared caches across multiple cores. The execution time of an application in a Multicore can lead to worse execution time than in a single core due to competition for the cache space. Core A can evict a line from the cache that, otherwise, would be hit by Core B in the near future. Later, core B, upon missing in the cache due to the evicted line, could produce further evictions also affecting any core (including core B itself), hence with a cascade effect. Moreover, such behavior can repeat pathologically (e.g., due to the regular and iterative nature of programs).

Other challenges include, for instance, reaching the diagnostic coverages imposed by the safety standards, which is a challenging task due to the increased complexity of Multicore systems, where testing some components or states may require specific synchronization across cores, which may be hard – if at all possible – to enforce. Regarding timing, standard computer architecture is driven by the following paradigm: *make the common case fast and the uncommon case correct*. This design approach leads to architectures where computer architects optimize the average-case execution time at the expense of the worst-case execution time (WCET). Modeling the dynamic features of current processors, memories, and interconnects for WCET analysis often result in computationally infeasible problems. The bounds calculated by the analysis are, therefore, overly conservative. Hence, some WCET-amenable designs have been devised [157], yet they are not broadly adopted by end users due to their limited performance and complex programmability (e.g., using software controlled scratchpads instead of cache memories).

In the case of COTS processors, instead, timing analysis can become very demanding since the number of sources that can introduce timing variations to execution makes it more complex to determine the WCET for a given application due to the number of uncertainties. For instance, accessing shared resources is a source of timing uncertainty since this access is not only affected by the software executed in the core under analysis but also by the software being run on the other cores. Thus, engineers need to consider the entire system as an entity and make decisions based on it.

Some solutions such as *time partitioning* or *space partitioning* can help with some of these timing problems, but we will not go further since they are out of the scope of this Thesis.

Since the general solution for single-core systems is to use lockstep as the means to achieve diverse redundancy, the straightforward idea for a Multicore system is to use lockstep on multiple cores. However, this comes at the expense of only using half of the area devoted to the cores for performance reasons since the rest are just executing the same instructions redundantly. Still, while half of the cores are not user-visible, multicores improve the performance capabilities of single-core safety-critical systems.

Mixed-criticality systems

Since now a single device has more computing power and can handle multiple tasks in parallel, mixed-criticality systems are on the order of the day. Mixed-criticality systems are safety-critical systems that simultaneously execute multiple criticality-level tasks. These systems need to prioritize the higher criticality applications in the scheduling of the shared resources while applying their best effort to reduce the contention or slowdown suffered by other less critical (or non-critical at all) tasks.

In this scenario, safety-critical system developers of Multicore device-based mixed-criticality systems must deal with two conflicting or contradictory constraints. These are, on the one hand, conservative functional safety standards that are based on the historical best safety industrial practices of the last decades, which have very limited consideration of Multicore devices such as Chapter 11 in ISO26262 [39]. On the other hand, a fast-evolving and highly innovative semiconductor industry that is continuously evolving its shrinking technologies and capable of integrating more and more cores in its systems [158]. Further details on the challenges brought by those conflicting constraints can be found in here [159].

Hetereogenous computing in safety-critical domains

We can also find *heterogenous computing* in safety-related platforms as well. For instance, this is the case of the Xilinx Zync Ultrascale+ [16], popular in the avionics domain [149], which includes two different Multicore clusters. These are the Application Core Unit (APU), a quad-core ARM Cortex-A53 and the Real-Time Processing Unit (RPU), a dual-core with ARM Cortex R5. Moreover, inside the SoC we can also find a small GPU and programmable logic that can be used to implement another accelerator. Another example, this one targeting the automotive domain, is the

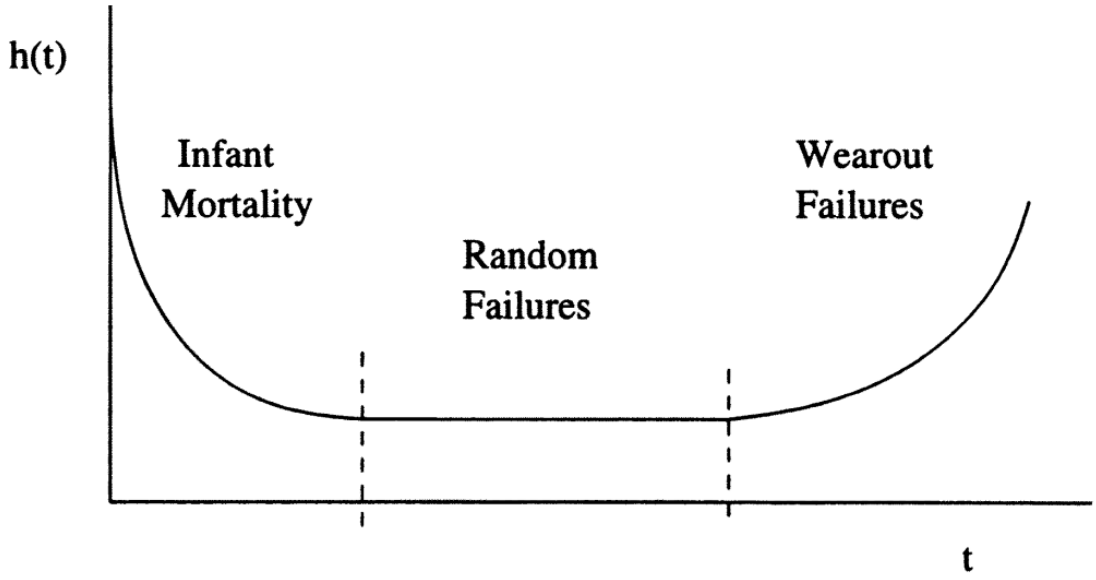


Figure 2-13: Bathtub curve with the three distinct regions with the hazard function of time to failure in the y-axis and the time in the x-axis. Extracted from [4].

Jetson AGX Xavier Industrial, which contains an 8-core Nvidia Carmel cluster and a dual-core with ARM Cortex R5, operating as lockstep. Both platforms execute the safety functionalities in the R5 cores, while non-safety functionalities can be executed on the other cluster.

Semiconductor Industry Trend

As presented by Corradi [134], while evolving, the semiconductor industry has reached a mature level that faces a technological and economic challenge. At the industry level, fewer companies (chip makers) can afford the investments required to produce chips with smaller transistors. This has led to a reduction in the number of leading semiconductor companies and a general trend toward the mass-production of Multicore SoC optimized for maximum average performance that targets multiple domains with a single SoC (e.g., servers, desktop, gaming) [134]. Only the automotive domain, from the safety-related semiconductor, has a niche market big enough to provide economically competitive safety-specific solutions compliant with previous safety certification standards. Therefore, it is expected that Multicore devices used in the development of safety-related systems will have a higher dependency on fewer manufacturers producing chips for non-safety-related markets.

Functional safety techniques in Multicore devices in Safety-critical systems

Safety techniques are in general divided into four main categories based on the challenge they address:

- **Reliability**: Described as the probability that the device will operate correctly during a period of time. The most relevant metric is the failure rate of the device (λ), which is expressed in Failures In Time (FIT), the number of failures expected in a billion (10^9) cumulative hours of the product's operation. To briefly introduce the concept and illustrate the variables that affect Reliability, we will use the curve (*bathtub shape* [4]) to characterize the operation of a population of devices. In the curve (Figure 2-13), we can easily see the 3 distinct periods. Next, we describe them very briefly:

  1. **Infant Mortality:** Models the earlier failures due to design or manufacturing defects that have not been completely removed after testing, this subpopulation is often referred to as the "weak siblings".
  2. **Random Failures:** The second period models the interval in which the "weak siblings" have disappeared and the only source of hazards (failures) are the random faults.
  3. **Wearout Failures:** The last period models the stage in which failures due to aging start to appear in concurrence with the random faults. The increasing failures due to *aging* are responsible for the increment of the function.

- **Diagnostic Coverage**: *DC* for short, denotes the effectiveness of diagnosis techniques to detect dangerous errors, expressed in coverage percentage with respect to all possible dangerous errors. DC is classified as low (60% to 90%), medium (90% to 99%) and high (>= 99%) [62]. DC is calculated as the ratio between the dangerous detected failure rate (λdd) and dangerous failure rate (λd), which includes both dangerous detected (λdd) and dangerous undetected failure rates (λdu). The diagnostic coverage of digital circuits can be calculated and measured using methods such as Failure Mode and Effects Analysis (FMEA) and fault injection [160]. Dangerous detected errors usually require the definition of an associated reaction (e.g., activate safe state, error correction).

- **Temporal independence**: Is defined as the characteristic to ensure that "one element shall not cause another element to function incorrectly by taking too high a share of the available processor execution time, or by blocking the execution of the other element by locking a shared resource" [62] so "that elements will not adversely interfere with each other's execution behavior such that a dangerous failure would occur" in the time domain [62].

- **Spatial independence**: Defined as the attribute which allows that "data used by one element shall not be changed by another element" so "that elements will not adversely interfere with each other's execution behavior such that a dangerous failure would occur" [62].

Safety techniques can be applied at different levels as well. Therefore, another categorization is based on the *device granularity* at which they are applied. Next, we show a small description as well as examples of the safety techniques based on this categorization:

- **Nanoscale level**: This scope includes circuit-level techniques as well as techniques implemented during the product fabrication to prevent specific physical effects from happening, such as aging, or include thermal and voltage management. For instance, run-time diagnosis of soft and aging errors is used to improve diagnostic coverage (e.g., monitoring circuits, self-tests).

- **Component level**: To improve reliability in this scope, common techniques for all types of components include *hardening*, or redundancy. Whereas redundancy usually provides the highest reliability increase, it also has a higher hardware overhead (e.g., 300% for triplication) than hardening (e.g., 15% for the LEON3FT space processor [161]). Particularly, to improve the diagnostic coverage at the core level, the most relevant technique for this Thesis, since the contributions presented are built on top of it, is the *Lockstep redundant execution*. This technique provides a safety standard compliant (e.g., ISO 26262, IEC 61508) medium to high diagnostic coverage with a high hardware overhead cost (e.g., >= 200%) and minimum performance overhead [162, 163]. Lockstep redundant execution is a cost-competitive and common technique used in COTS safety devices. However, dual-core devices that operate in lockstep to increase the core Diagnostic Coverage are considered single-core devices, as explained before. Other techniques include software validation of specific execution invariants (e.g., Argus [164]) and other generic software techniques such as control flow checking [62, 163] or anomaly detection techniques that use dedicated or modified circuits to detect errors (e.g., [165]).

- **Device level**: Device-level reliability is composed by the addition of safe and dangerous failure rates of all components or at least the components that take part in the execution of safety functions. Thus, device-level reliability is defined by the device architecture and building components' reliability. Diagnostic coverage is generally addressed from three main perspectives (1) temporal diagnosis, (2) spatial diagnosis, and (3) safe start-up/shut-down. Examples for (1) include time constraint monitoring, trapping unexpected interrupts, watchdogs [102, 145, 130] and run-time monitoring diagnosis [166, 167]. (2) Spatial diagnosis is required to ensure that data of a given function is not modified by another element or function. This is generally performed by dedicated built-in components (e.g., Snoop Control Unit [168]) in Multicore devices. Cache and memory coherency diagnosis can be performed by built-in hardware diagnosis [169], or generic software techniques (e.g., [170, 168]).

Chapter 3

Experimental Setup

This Chapter covers the experimental setup and methodology that have been used in the development of this Thesis. The structure of the sections of this chapter is the following:

The contributions in this thesis tackle a similar challenge (enable a *diverse redundant* execution) from two angles: whether to use a GPU or a CPU, and whether to provide software-only or hardware/software solutions.

As such, we use heterogeneous frameworks across chapters such as a GPU simulator and GPU-relevant benchmarks for hardware solutions on GPU. COTS GPUs and GPU-relevant benchmarks are employed for the software-only solutions on GPU. Lastly, COTS CPUs and CPU-relevant benchmarks are used by the software-only solutions on CPU.

This chapter is divided into three main blocks. Firstly, we describe the different hardware setups that we have used in the contributions, including details and block diagrams of the platforms. Second, we describe the two different benchmark suites employed, one for the GPU contributions and the other one for the CPU. The third block consists of the methodology. Due to the variance of the setups for each contribution, we have divided the methodology block into small sections, one for each contribution.

3.1 Hardware Setup

This section details the different hardware setups used for the contributions of this Thesis. Firstly, in subsection 3.1.1, we describe the Server CAOS17, which has connected the two GPUs used in the GPU-related contributions. The GTX 1050 Ti and GTX 1080 Ti are both described in their subsections 3.1.2, and 3.1.3, respectively. Lastly, we detail the platform employed for the last contribution *Software-only based Diverse Redundancy for ASIL-D Automotive Applications on Embedded HPC Platforms*, the NVIDIA Jetson Tx2 3.1.4.

Note that, in some of our experiments, we have used high-end GPUs that may not be suited for an automotive platform due to their power usage. However, both, the availability of these GPUs in our lab, along with the need for higher performance

GPUs for the workloads at hand (Rodinia benchmarks are, in many cases, larger versions of automotive workloads), led to the use of these GPUs for some of our works.

3.1.1 Server CAOS17

The server of our research group, named CAOS17, has the two GPUs employed connected through PCI Express, the specifications of the server are the following:

- CPU: AMD Ryzen 7 1800X Eight-Core Processor
- Memory: 64 GB at 2133 MHz
- OS: Ubuntu 18.04.6 LTS (Kernel version 4.15.0-189-generic)
- GPU_1: NVIDIA GTX 1050 Ti
- GPU_2: NVIDIA GTX 1080 Ti

Executions on the real hardware have been executed multiple times and programmed to launch at night to avoid interference and slowdowns due to other users' tasks running at the same time. The results shown are the average of those executions. Instead, since simulation results are not affected by the system load, simulations did not have this limitation and were able to be executed at any time.

3.1.2 NVIDIA GTX1050 Ti

The GPU selected for most of our experiments and modeled in GPGPUSim, is the NVIDIA GTX1050 Ti [171]. We chose this GPU due to multiple factors. First, it is a relatively large GPU with 6 Streaming Multiprocessor (SM), which allowed us to design and test our scheduling proposals. Second, due to its availability in our lab.

The NVIDIA GTX1050 Ti is a Pascal GPU, Pascal is the codename of a GPU microarchitecture developed by NVIDIA and first introduced in April 2016 as the successor of the Maxwell architecture, and it is the same GPU micro-architecture used in the NVIDIA PX2 AutoChauffer product, found in modern high-end cars, which is only available to affiliated NVIDIA automotive partners. The chipset used for the GTX 1050 Ti is named GP107. It was manufactured using Samsung's 14 nm finFET process. This GPU contains 768 CUDA cores divided into 6 SM and 4 GB GDDR5 memory with a 128-bit interface, we can see that in the block diagram in 3-1.

In each of the six SMs of the 1050Ti we can see structures shared by all the threads executing simultaneously, such as the instruction buffer, the warp scheduler or the register file, and units used individually by each thread, basically the replicated functional units: the LD/ST (load/store) units, the NVIDIA Cores (similar to an ALU), or the Special Function Units (SFU). All of them can be seen in Figure 3-2.

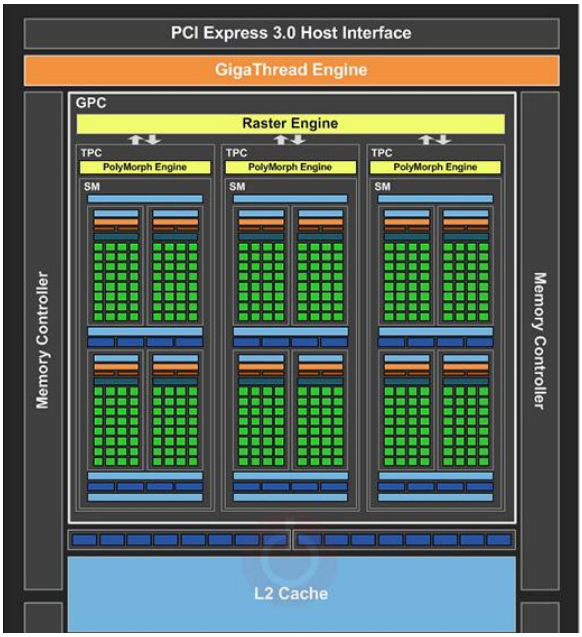


Figure 3-1: Diagram of the GTX 1050 Ti (edited from https://www.techpowerup.com/gpu-specs/nvidia-gp107.g801)

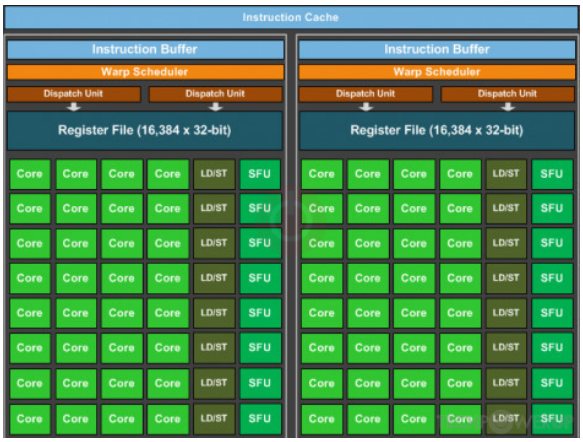


Figure 3-2: SM diagram of the PASCAL microarchitecture belonging to the GTX 1050 Ti (edited from https://www.techpowerup.com/gpu-specs/nvidia-gp107.g801)

3.1.3 NVIDIA GTX1080 Ti

In the Triple Modular Redundancy experiments, we used a different GPU since the GTX 1050 Ti was, in some cases, too small to contain three instances of a given kernel at the same time. Instead, we chose the NVIDIA GTX 1080 Ti [172]. The 1080 Ti is an NVIDIA GPU of the same microarchitecture as the 1050 Ti, the Pascal. The chipset used is different; its codename is GP102 and has a different manufacturing process than the 1050 Ti, the TSMC 16 nm process. The internals of the SMs are the same on both GPUs since they belong to the same microarchitecture. However, 1080 has a total of 28 SMs in comparison to the 6 in 1050. This means a total of 3584 CUDA cores versus the 768 in 1050. To support the increased memory load, the amount of memory is also larger with 11GB and a bus interface of 352 bits.

3.1.4 NVIDIA Jetson Tegra TX2 SoC

In the proposal *Software-only based Diverse Redundancy for ASIL-D Automotive Applications on Embedded HPC Platforms* we have used the NVIDIA Jetson TX2 SoC [127] as an evaluation platform. The Jetson TX2 SoC has the following characteristics:

- Three CPU clusters: dual-core NVIDIA Denver 64-bit, quad-core ARM Cortex A57 [5] and a small Cortex-R5
- 256-core NVIDIA Pascal GPU
- 8 GB 128-bit LPDDR4 Memory

We can observe a schematic of the Tegra TX2 in Figure 3-3, where the different CPU clusters are highlighted using colored squares. We can see that the Cortex-R5 is shown multiple times due to their multiple tasks. However, the Cortex-R5 cluster(red color) is not visible from the user's perspective and thus not-relevant for the contributions to this Thesis. Similarly, the Pascal GPU in this SoC is non-relevant for the corresponding contribution since it only focuses on the CPU clusters, and it is too small (i.e., only two SMs) for the GPU contributions.

3.2 Software Setup

To evaluate our contributions we have mainly used two different benchmarks. For the GPU contributions we selected the Rodinia benchmark suite whereas for the multicore contribution, we selected the EEMBCs Autobench suite.

3.2.1 Rodinia

The Rodinia Benchmark Suite [7, 173] is a widely used benchmark suite in the GPGPU domain. Rodinia is a heterogeneous-computing, very recognized benchmark suite, and its original scientific paper, published in 2009, already has more than 3000

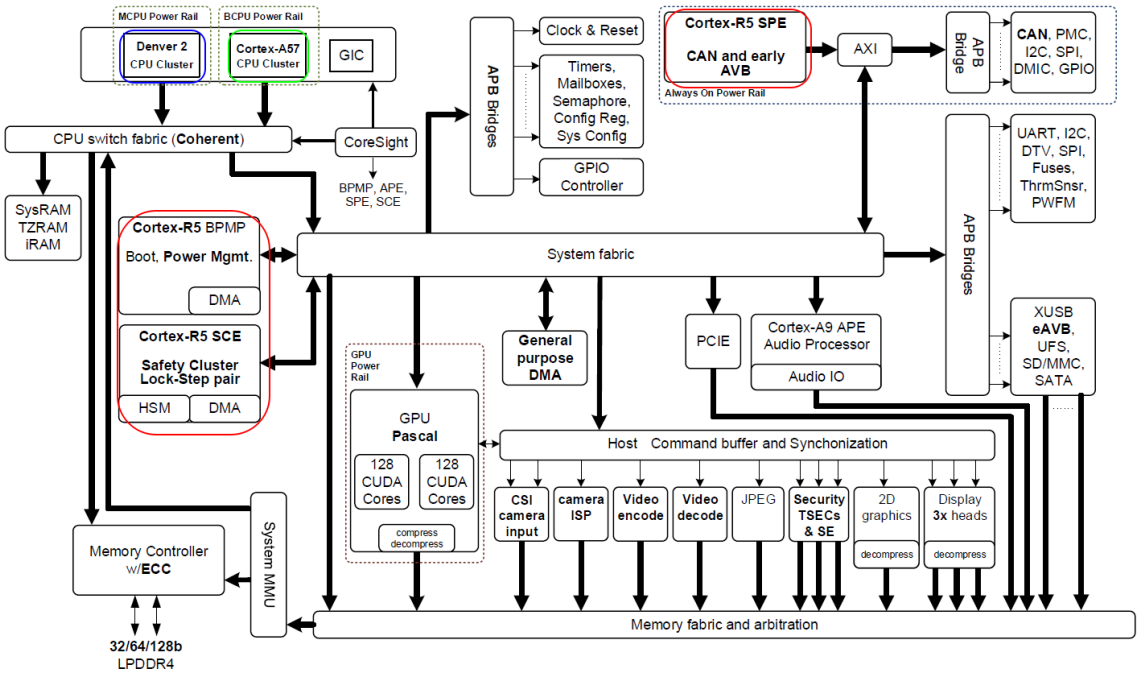


Figure 3-3: NVIDIA Tegra TX2 schematic, with three CPU clusters: the dual-core NVIDIA Denver 64-bit (blue square), the quad-core ARM Cortex A57 [5] (green square), and a small Cortex-R5 in charge of (critical tasks, transparent to the user), edited from [6]

citations. It is a collection of parallel scientific applications which target heterogeneous computing platforms with both multicore CPUs and GPUs. All applications have support for CUDA [41], OpenCL [40] and OpenMP (API for shared-memory parallel programming in C/C++) depending on the compilation flags used.

However, the baseline applications do not support redundant kernel execution. Since this is a requirement of our proposal, we have manually modified the code to add it. We added preprocessor directives such as *#ifdef REDUNDANT* so that we could compile and obtain different versions of our code with the same source codes. Thanks to these preprocessor directives, we can easily compile different versions by defining these variables when running the compiling command using the *-D* flag (i.e., -DREDUNDANT). The versions used are the baseline application (BASELINE), the application with redundant kernel execution with comparison (REDUNDANT), and the redundant kernel execution serialized with the comparison that we use later (REDUNDANT_SERIALIZE).

An example of the modifications can be seen in the Annex in Listing 1, where all the modifications are prefixed by the *#ifdef* clauses commented above. These clauses are omitted on purpose for the sake of clarity in the code listing shown in the proposal section (Section 6.2.2) of the contribution chapter (see Listing 6.2). The original code shown is extracted from one of the applications (myocyte) from Rodinia, and the modifications are the following:

- Input replication and allocation (if needed), not shown in the listing

- Output replication and allocation are also not shown in the listing
- Creation of CUDA streams to allow concurrent redundant kernel execution (lines 5-7)
- Kernels calls:
    1. *BASELINE*: Original kernel launch (line 18)
    2. *REDUNDANT*: Add the two kernel launches using different CUDA Streams and different arguments (lines 12 & 16)
    3. *REDUNDANT_SERIALIZE*: Add the call to *cudaStreamSynchronize*, which waits until all tasks assigned to the specified queue have finished (line 13-15)
- Getting results back from the GPU memory (BASELINE: lines 28-29, REDUNDANT and REDUNDANT_SERIALIZE: lines 28-32)
- The result comparison which only happens in the REDUNDANT and REDUNDANT_SERIALIZE versions (lines 38-46)

We also employ Rodinia in the contribution *GPU Software-only diverse redundant execution*. Similarly, we add the *#ifdef TRIPLE* for the Triple Modular Redundancy Kernel execution.

3.2.2 EEMBC AutoBench v1.1

The EEMBC automotive suite [174, 175] v1.1 is developed by the Embedded Microprocessor Benchmark Consortium with the objective of measuring the performance of Embedded Systems with the use of programs used commonly in automotive systems. In Table 3.1 we can see the list of benchmarks included in this suite. The structure of these benchmarks is the following: each program has a main loop, executed a number of iterations (configurable by the user) with some calls in the loop body. The number of iterations we use is the default one provided for each benchmark. The input data for each benchmark is already included in the benchmark suite.

3.3 Methodology

The rest of the chapter is divided into subsections, one for each contribution, in which we detail the methodology followed in the evaluation step of each contribution. Detailed information on the tested applications is also described in each subsection or referenced if already described.

Table 3.1: EEMBC Automotive benchmarks

Name	Description
a2time	Angle to Time Conversion
basefp	Basic Integer and Floating Point
bitmnp	Bit Manipulation
cacheb	Cache "Buster"
canrdr	CAN Remote Data Request
aifft	Fast Fourier Transform (FFT)
aifirf	Finite Impulse Response (FIR) Filter
aiifft	Inverse Fast Fourier Transform (iFFT)
aiirflt	Infinite Impulse Response (IIR) Filter
matric	Matrix Arithmetic
pntrch	Pointer Chasing
puwmod	Pulse Width Modulation (PWM)
rspeed	Road Speed Calculation
tblook	Table Lookup and Interpolation
ttsprk	Tooth to Spark

3.4 Methodolgy and Setup used in: *An Analysis of the Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain*

The first contribution, presented in Chapter 4, is substantially theoretical. Thus, few tools have been required for this contribution. In particular, to distinguish between *statement* and *branch* code sections, we use the tool RapiCover [176] in one of the state-of-the-art real-time object detection systems, the *You only look once* or Yolo [31] (for short) version 3.

RapiCover is a tool from RAPITA systems used to collect code coverage for different programming languages, which is used to simplify verification and produce evidence for DO-178 and ISO26262, safety certification for avionics and automotive, respectively. RapiCover works by "*injecting instrumentation code into source code and executing the native build system so that the coverage results are collected during program execution*" [176].

As said, we apply RapiCover to Yolo version 3 [31]. Yolo is an open-source object detection system developed by Josep C. Redmon under the supervision of Ali Farhadi from the University of Washington. Yolov3 is an upgraded version of the original Yolo, including some design changes to improve efficiency.

3.5 Methodolgy and Setup used in: *GPU scheduling policies*

To evaluate the proposals of this contribution, we are required to modify the hardware of a GPU. Due to the unviably of doing it in a real GPU, we implement our proposals on a GPU simulator. On top of the simulator, we executed a set of applications from a known heterogeneous benchmark suite (Rodinia). Furthermore, we also ran some of the modified applications on top of a real GPU available in our department to mimic the timing behavior of one of our proposals.

Next, we describe and discuss these tools and the modifications we made to adapt them to our work.

3.5.1 GPGPUSim

The scheduling policies proposed in this contribution are integrated and evaluated inside GPGPUSim [114, 115] (version 3.2.2). GPGPUSim was created by Tor Aamodt's research group at the University of British Columbia. It is a Graphical Processing Unit (GPU) cycle-level simulator which provides a detailed simulation model of contemporary GPUs. More precisely, it models the features of a modern graphic processor that are relevant to non-graphics applications. It can run GPU computing workloads written in both CUDA [41] or OpenCL [40]. However, this particular version of the simulator only supports up to CUDA 8 [177] (so 8 and below).

With the simulator, we model the NVIDIA 1050Ti [171] (see section 3.1.2). To do this, we use the configuration files for the NVIDIA PASCAL architecture [178], and we adapt them to the target GPU by modifying the amount of SMs and the amount of memory, which are commonly the main differences among different GPU models of the same architecture. Once modeled, we need to make some modifications to the simulator to integrate our proposed scheduling policies.

To implement the two proposed scheduling policies, we modify the default scheduling policy according to the requirements that each proposed policy imposes on the way the available SMs have to be assigned. For one, we use the default scheduling policy implemented in the GPGPUSim and restrict each kernel execution to 3 dedicated SMs (half of the available ones), whereas for the other one, it requires limiting the number of concurrent kernels to 1. To evaluate the proposals, we compare the performance of the proposed policies against the default GPGPUSim scheduler that can allocate all GPU SMs (6) to the kernels without any constraint. More details on the implementation are explained in Section 5.4.1.

To verify that the modifications work correctly, GPGPUSim has a special log file that indicates internal information about the simulation execution. In our case, we are interested in a particular message: A message appears when a thread block is scheduled to an SM, including both the information related to the thread block (i.e., thread block identification number and kernel identification) and the SM assigned to. We validate in the executions that these messages were correct according to the scheduler policy implemented.

3.5.2 CUDA Version and Compiler

In general, in our methodology, we attempt to select the latest stable release of a compiler since it may include fixes of old bugs, features of the latest architectures, and better performance. However, in this case, GPGPUSim only works with particular versions for both the GCC compiler and the CUDA version. Instead, for the software executed directly on the real GPU, we employ the latest versions of both the GCC compiler and CUDA.

3.6 Methodology and Setup used in: *GPU software-only diverse redundant execution*

Some of the tools and benchmarks introduced in the previous contribution have also been employed for this one, as we explain in this section. There are some others that, instead, are used in this contribution exclusively. As for the software we use, we have employed the same benchmarks, the Rodinia benchmark suite [7, 173] (see section 3.2.1). Since some of the experiments have required ad-hoc environments, we review the methodology and setup for each one of them separately.

3.6.1 Slack Measurements

When a GPU kernel is launched in CUDA (and similarly in OpenCL), the NVIDIA Runtime API has to execute a set of functions (*ConfigureCall, KernelSetupArguments and CudaLaunch*) which pass the information of the kernel to the GPU as explained in section 6.3.1. In order to measure the timing of these functions, we use the NVIDIA profiler (known as *nvprof*), from the NVIDIA toolkit in our CAOS17 server setup (see Section 3.1.1). To be able to see these functions, we need to compile the code using the debugger option *-g* [179] which exposes debugging symbols to be captured for debugger programs such as GDB or the nvprof. Then, instead of executing the binary in a regular form, we use the nvprof tool. An example together with a description of the nvprof options description can be seen in the Appendix in Chapter 8.3. We execute 100 times each experiment, and then we extract the execution times of the NVIDIA runtime API function from the trace files produced by nvprof.

3.6.2 COTS GPU Results for diverse DMR

Similarly to the previous section, we use the same setup, in the server CAOS17 (see 3.1.1) and launch the kernels in th same GPU, the GTX 1050 Ti (see 3.1.2). However, an extra modification is required in the code to measure the CUDA calls shown in the plot 6-9. This occurs because the CUDA calls (i.e., memory transfers and kernel calls) are asynchronous by default. This means that, when the CPU performs these calls, it assigns tasks to the memory controller and the GPU, but it does not wait for them to finish. Instead, continues executing the next instructions.

Therefore, in order to measure the execution time of these calls, we need to add some synchronization mechanism so that in the CPU side we can know the instant when they finish.

We added a synchronization method available in CUDA named *CudaDeviceSynchonize()*, which stalls the CPU until all operations in the GPU (including memory transfers) have finished. Since this can make the execution time higher than in the regular execution, we enclosed the synchronization method between *#ifdef* calls (like we did in 3.2.1), in a new binary setup named *TIMING*, where we only gathered the timing of these calls. In the comparison, synchronization is not required since the comparison of the results is directly made in the CPU.

We also need a method to measure time. For this, we use the function *gettimeofday()* from *sys/time.h*, which captures the time in seconds and μseconds. We place two calls to this function before and after the call that we wanted to measure, and later we subtract the times from both calls to obtain the elapsed time between them.

3.6.3 COTS GPU Results for diverse TMR

For the results shown in this contribution, we employ a different setup with respect to the other experiments. In particular, we use the NVIDIA GTX 1080 Ti (see Section 3.1.3) in the CAOS17 server. The Rodinia benchmarks also require a modification; we add a new binary setup named *TRIPLE* which includes a third kernel call, with everything required (e.g., CUDA stream, memory transfer, comparison), enclosed in the *#ifdef*. To ensure that the executions are employing the correct GPU, we started all callings with *CUDA_VISIBLE_DEVICES=0*, which makes only the GPU GTX 1080 Ti visible to the execution.

3.6.4 Fault-detection capabilities evaluation using fault injection

The fault-injector tool we use as the baseline is the NVBitFI [180, 181]. NVBitFI is an automated framework to perform error injection campaigns for GPU applications resilience evaluation. NVBitFI is built on top of another NVIDIA open-source tool, the NVIDIA Binary Instrumentation Tool (NVBit) [182], which is a research prototype of a dynamic binary instrumentation library for NVIDIA GPUs. Information on the fault models used and the target registers for injection are explained in the evaluation section of the proposal in Section 6.3.5. However, here we are going to explain the methodology required by the tool in order to perform a proper fault injection campaign. NVBitFI has a 4-step process to perform the fault injection campaign:

1. **Step 1**: Initially, it is required to merge the directory of both tools, the NVBitFI and the NVBit, and prepare the setting files to target the application where we want to perform the fault injection (*backprop* in our case) and perform an error-free run to create the outputs without error, what is known as a *Golden Run*.

2. **Step 2**: Launch the target application again using the profiler tool. This run profiles the target application and generates the injection list. The injection list is the list of possible injecting points and their characteristics (e.g., such as which type of destination register is). Then, based on the configurations set, creates a subset of the list with the injection points selected and the type of error that will be injected.

3. **Step 3**: Later, the tool receives the injection list created in the previous step and performs all the executions, injecting the corresponding error for each one.

4. **Step 4**: Finally, a parsing of the results is done and a classification of the outcome for each error injection is performed in 4 different categories based on NVBitFI classification.

    - Masked: Error was masked and did not appear in the output of the application.
    - Silent Data Corruption (SDC) undetected: A mismatch in the output was found but was not detected by the detection mechanism.
    - SDC detected (or error detected)[1]: An error was found by the detection mechanism and reflected in the output of the application.
    - DUE: A detected error prevented finishing the execution. Note that these errors are detected regardless of whether the specific error detection mechanism under consideration (i.e., redundancy) is integrated.

For our executions, we make some modifications to the original NVBitFI code. Since our application already has a comparison of the kernel results inside, which could identify some errors previously to the comparison system of the injection tool. We add the category **SDC detected** as an outcome of the error injection, which identifies the errors (or SDCs) that are already identified by the internal comparison system without requiring comparison with the Golden Run output.

Others fault injection tools were considered, such as SASSIFI and GPU-Qin. GPU-Qin [183], from 2014, used an old CUDA debugger (*cuda-gdb*) since it is a debugger-based injector to inject the errors and was discarded due to potential issues of compatibility with using more modern GPUs. SASSIFI [106] was the predecessor of NVBitFI and is now deprecated because it uses the older SASSI interface (the predecessor of NVBit) to perform the injection instead of the NVBit interface.

3.6.5 HW and SW-only solutions side by side on the simulator

As a final part of the evaluation of this contribution, we compare all the different proposals from the execution time perspective. Due to the impossibility to test the hardware proposals in the real platform, GPGPUSim [114, 115] was used since software

[1]We created this category to classify as a different outcome whenever our mechanism is able to detect the fault

proposals could be included. GPGPUSim has already been described in Section 3.5.1 and we use the same configuration employed before, modeling the NVIDIA GTX 1050 Ti, described in Section 3.1.2. The configurations of the benchmarks selected are the *friendly* ones obtained using the protocol proposed.

3.7 Methodolgy and Setup used in: *Software-only based Diverse Redundancy for ASIL-D Automotive Applications on Embedded HPC Platforms*

Before performing the experiments on the NVIDIA Tegra Tx2 SoC, we made some modifications to the platform setup:

Firstly, in order to avoid that one of the redundant applications being executed in one of the DENVER cores instead of the Cortex A57, we deactivate the Denver cores. To do it, we issue the following command in *sudo-mode*, see Listing 4 in Appendix.

Then, to maintain the same frequency in all cores and deactivate the frequency scaling, that could challenge the mechanism to maintain the diversity in the execution (i.e., by speeding up the trail core), we execute the following commands, again in "sudo" mode, see Listing 5 in Appendix.

As explained in the contribution chapter, in order to remove the noise created by other processes we migrate all the Linux processes to the non-used (4th) core. For this, we listed all the processes active, and we changed their CPU affinity mask, so they could only be executed in that particular core.

3.7.1 CPU diverse redundancy execution

To enable the diverse redundancy execution in a single cluster, we have created a single application that uses Open MPI [184] v4.0.1 to orchestrate all the execution.

MPI is used to both communicate and synchronize the CPU monitoring process (Monitor) and the ones that are going to execute the target application (Workers). In order to set up the execution, the applications have been modified to include a set of calls to MPI to synchronize and that use the *MPI_TAG* entry to identify the different messages. These messages are crucial to allow the Monitor to identify which Worker is the Trail and which one is the Head. Moreover, some messages are exchanged before the execution of the application to set up the performance monitoring counters and to make the Workers wait for a specific message in order to start, which allows easily the Monitor to create the initial staggering.

As in the previous contribution (See 3.6.2), we use the call to *gettimeofday* to measure the different timings of the execution.

The monitoring application uses the target applications as an object file in its compilation process to include all the code. In order to ease the compilation, the applications to be protected must be compiled statically since otherwise, the monitoring application would require to know the libraries used by these applications to

compile. The applications used for evaluation were a matrix multiplication as a proof of concept and, later we decided to include the EEMBC AutoBench Performance Benchmark Suite [185] v1.1, which we present in the next Section (see Section 3.2.2).

3.7.2 Executions and versions

To create the different versions of *Baseline, No Monitor, Passive* and *Safe* we have employed the same technique used in Section 3.2.1, where the different versions include more or less code depending on the flags used at compilation time. However, this time we used the flags twice since, as explained above, we compile first the application statically and then the monitoring application.

Executions are done in full isolation since no other user was allowed in the system. The numbers shown in the plots are the results of 500 executions of each benchmark and version.

Chapter 4

An Analysis of the Safety-Related Challenges and Opportunities for GPUs in the Automotive Domain

4.1 Introduction

High-performance-embedded systems are increasingly used in critical domains such as transportation (road vehicles, airplanes, and trains), industrial machinery, health devices, and satellites. Engineers of Critical Real-Time Embedded Systems (CRTES) develop products following commonly accepted best practices, and, in the case of safety-critical systems, showing compliance with legal directives is mandatory before they are allowed to operate. This requires going through a certification process as defined by applicable safety standards, e.g., ISO26262 [39] for road vehicles (see section 2.2). The increasing use of "smart" software functionalities as the main competitive factor in CRTES is relentless. In automotive, the main software functionality relates to driving automation, whose potential benefits range from the reduction of accidents and the CO_2 footprint to increasing people's quality of life by reducing the time they spend driving. These benefits have boosted the trend toward full automation with most mid- and high-end cars already featuring some Advanced Driver Assistance Systems (ADAS) and the first Autonomous Driving (AD) level 3 car already in mass production (the Audi A8). Note that there are five AD levels from 0, or no automation, to level 5: fully automation in which driving is automatically handled in (all) scenarios as complex as those human drivers can encounter on the roads (see section 2.1.1).

AD functionality can be broadly classified into the perception of the environment surrounding the vehicle, localization to estimate the vehicle's position, planning the vehicle trajectory, and controlling vehicle actuators. Perception, the most compute-intensive module, builds on object detection and tracking. In the past few years, impressive improvements in Machine Learning (ML) techniques, e.g., Deep Neural Networks (DNN), have dramatically changed state-of-the-art algorithms for perception by achieving significantly higher accuracy. This has made ML-based perception

techniques the preferred solution in the industry. The other side of the coin is that ML techniques carry unprecedented performance demands in automotive. The performance requirements of AD alone are projected to increase by 100x from 2016 to 2024 [12].

Initial research studies [186] and performance data from chip vendors show the effectiveness of Graphical Processing Units (GPUs) to accelerate ML-based libraries for AD. This has attracted the attention of car manufacturers who have started analyzing GPU's potential to cover AD's performance requirements. Since high-end GPUs targeting AD systems build on designs for the mainstream market, they may find some difficulties to adapt to automotive's specific requirements. In this chapter, we expand some of the contents pointed out in the introduction and background chapters, and analyze some of the main challenges that GPUs, and the software running on them, will face in providing safety assurance in accordance with the ISO26262 functional safety standard, given that a large fraction of the work in this thesis builds upon GPUs. We also cover other relevant challenges such as time predictability. Specifically, the main contributions of this work are:

1. At the hardware level, harsh operation conditions (e.g., extreme temperatures) make GPUs more vulnerable to random hardware faults. Resorting to ISO26262 standard solutions such as diversity and redundancy has to be done cautiously in GPUs. For instance, while GPUs naturally offer redundancy sources, they must be carefully exploited to preserve high performance and prevent a single fault from becoming a common cause of failure in all redundant instances, which could lead the system to a hazardous situation.

2. At the software level, we identify these challenges:

    - The AD builds on generic, i.e., nonautomotive specific, ML libraries. This allows car makers to enjoy the improved functionality (e.g., higher object detection accuracy) of the latest available generic ML libraries, which see a new release every few months. However, their generality increases the probability that the libraries implement hard-to-validate features, increasing the effort to assess libraries' adherence to ISO26262 guidelines on software coding and development. As an illustrative example, our results with YOLO v3, a state-of-the-art object detector system, shows that the code coverage achieved, a basic software unit structural coverage metric is well below the 100% needed in ISO26262.
    - For performance-improving reasons, ML libraries build on low-level GPU-optimized libraries such as cuDNN [187]. The black box, i.e., closed-source, nature of these libraries, however, challenges assessing their adherence to ISO26262 guidelines for software. This requires library owners to undergo the certification process or the use of opensource libraries that provide similar performance to their closed-source counterparts, so that end users take care of its certification. In either case, changes to simplify validation and verification can reduce efficiency and result in the creation of ISO26262-specific branches of the libraries.

- Languages to program GPUs make use of features that hamper software (code) verification activities that already amount to most of the total development effort for the highest safety levels (ASIL D). As illustrative examples, we discuss two well-known features: the use of pointers (that must be prevented) and the use of defensive programming to prevent systematic software faults (that must be favored). We show that ISO26262-aware programming languages prevent (favor) some of those undesired (desired) features while maintaining GPU's performance benefits.

3. Time predictability [188] a fundamental requirement of CRTES, is another relevant challenge in GPUs since it is hard to achieve on (complex) high-performance hardware like GPUs. To address this challenge, we advocate for hardware support to increase observability as a necessary element to obtain evidence of the correctness of the derived Worst-Case Execution Time (WCET) estimates. Furthermore, it is required to analyze those elements in DNN codes that negatively affect predictability, which has only been superficially explored so far.

Overall, adherence to ISO26262, already achieved for ADAS, is challenged by AD. Under ADAS, the computing system acts as a fail-safe system, and in the case of misbehavior, it returns the control to the driver (see section 2.1.1). However, AD makes some systems fail-operational preventing the control to be returned to the driver. This has onerous consequences on the safety solutions adopted to guarantee that the system remains operational upon a fault (also known as fault-tolerant). Recently, the NVIDIA Xavier has been announced as an ISO26262-capable GPU-based SoC for AD. While detailed technical specifications on how this SoC achieves fail-operational capabilities are not yet available, achieving ISO26262 compliance requires appropriate redundancy and diversity strategies (i.e., lockstep, see section 2.1.3). To our understanding, in the NVIDIA platform, this is achieved with a non-negligible amount of replication of functionalities (e.g., using GPUs and deep learning accelerators), which may significantly increase Verification and Validation (V&V) costs due to the use of two different software and hardware implementations or may lead to inefficient solutions since execution time will be dominated by the slowest implementation. In general, it remains unclear to what extent AD systems can be efficiently deployed and validated in a cost-effective manner on an AD-capable SoC. In this proposal, we analyze some of the most relevant challenges related to this matter.

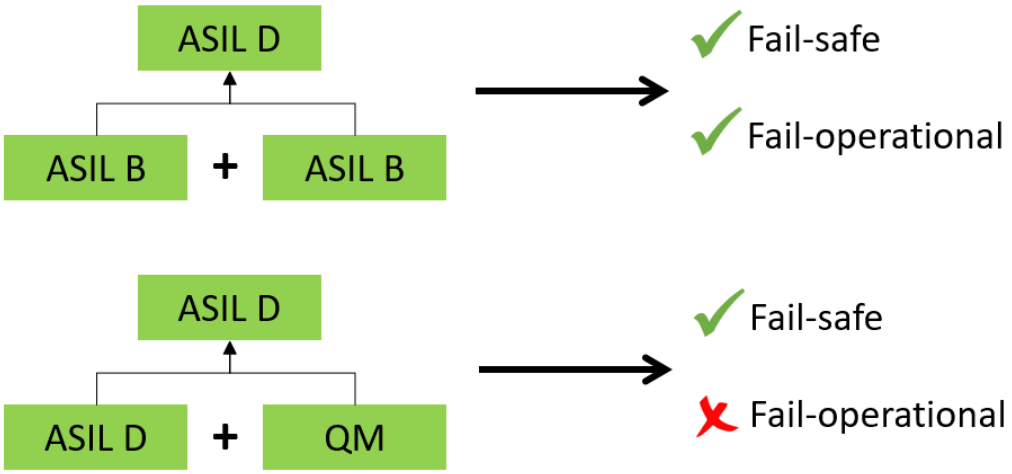


Figure 4-1: Examples of ASIL decomposition and appropriateness for fail-safe and fail-operational systems.

4.2 Safety Assurance: Impact on Hardware

Next, we are going to discuss, in more detail, the impact that safety has on the hardware. Starting with the system design due to the ASIL Decomposition. Then how redundancy and diversity can be applied to GPUs and finally how the harsh environment may affect the technology employed to build the GPUs.

the impact on the hardware that safety requires.

4.2.1 ASIL Decomposition

Traditionally, automotive systems have been considered fail-safe, which requires simpler measures to guarantee adherence to ISO26262, such as returning the control to the driver. However, the transition to fail-operational systems driven by level-5 AD significantly complicates achieving functional safety and hampers some forms of ASIL decomposition that were traditionally employed to save development costs. ASIL decomposition (see section 2.2.1) is used (i) to implement high-ASIL components with redundant and sufficiently independent lower-ASIL components (see top example in Figure 4-1 in which two ASIL B components can be used to reach ASIL D) and (ii) to allow a subset of the components preserve safety, thus remaining at the corresponding ASIL level, whereas others are regarded as QM, since, on a failure, ASIL components will detect it and keep the system safe. Such decomposition is often used to keep monitoring functionalities at the corresponding ASIL level (e.g., ASIL D in the bottom example in Figure 4-1), whereas computation components are regarded as QM. On a failure of the QM component, the monitoring one detects it and moves the system to a safe state, thus impacting availability but not safety. Whether faults occur often is, therefore, a matter of availability and so, business, but functional safety is preserved. In AD, since some ASIL C/D functionalities are fail-operational (e.g., those related to braking and steering), the components implementing such functionality

must achieve the corresponding ASIL, and ASIL decomposition cannot be applied to keep those items as QM since a safe state may not exist at all. Hence, some form of fault tolerance must be incorporated to keep the system operational despite faults.

4.2.2 Redundancy and Diversity

As we have seen redundancy and diversity are two essential attributes when it comes to safety, now we are going to see how these two terms can be applied to GPUs.

GPUs naturally offer lots of hardware redundancy that can be exploited to implement diversity solutions. However, for IP confidentiality reasons, some GPU's internal behaviors (e.g., resource allocation) are managed automatically by hardware, i.e., in a black box manner. Unfortunately, this practice clashes with guaranteeing diversity, since low-level management of the resources from software may be needed. Yet, it is our view that those issues are not roadblocks for the use of GPUs for AD in the automotive domain. That is, the type of homogeneous hardware redundancy offered by GPUs can be made compatible with automotive needs, similar to the case for homogeneous cores operating in a lockstep mode. For instance, identical cores, despite being homogeneous in terms of front-end design, provide diversity by several means like operating with some time shift so that activities carried out at any given time differ across cores, and hence, upon a fault affecting both cores, the effect is necessarily different, and thus, errors can be detected timely. Another diversity technique usually employed in lockstep cores is the use of layout diversity. Similar approaches can be enabled on the top of GPUs as long as common cause failures are avoided by construction by, for instance, using similar concepts as for cores (e.g., allocating separate sets of resources to each redundant thread and operating with some time shift). Also, GPU architectures may evolve and match ISO26262 requirements in the future since modifications will likely have a roughly negligible impact on cost and performance, and GPU vendors like NVIDIA already acknowledge the need for ISO26262 compliance [189].

Summary: GPUs massive parallelism allows supporting NooM (where N < M) redundancy. However, two key open challenges remain. First, guaranteeing diversity by avoiding common cause failures. And second, excessive use of 1oo2 (or 2oo3) may result in unaffordable procurement and energy costs, which calls for providing efficient NooM redundancy solutions.

4.2.3 Ability to Operate in Harsh Environment

Hardware qualified for automotive use needs to have an operating temperature that ranges between –40°C and 150°C for the highest criticality (grade 0 automotive electronics) and increased reliability requirements for soft errors. While those operation conditions are much more challenging than those for server or office electronic equipment, they are affordable for GPUs by employing appropriate circuit designs such as larger transistors and wider wires. However, those design practices may cause some performance degradation due to the use of slower circuits, which may also consume more power. Still, in our view, suitable tradeoffs can be found.

4.3 Safety Assurance: Impact on Software

Safety also affects the Software run on the automotive platforms. We are going to discuss some of the adjustments that are required in the current status of GPU software.

4.3.1 Coding Standard and Architectural Design

Critical software across all sectors needs to comply with coding and development guidelines, in order to facilitate its validation and certification against the standards of the particular domain. In automotive, ISO26262, for instance, recommends the limited use of certain features that complicate the certification of software applications such as pointers and dynamic memory allocation. It also encourages the use of safe language subsets to limit the use of error-prone language features. For instance, MISRA C is a subset of the C language that defines a set of rules that can be statically checked by commercial tools and therefore enforce their use. Besides coding standards, ISO26262 defines (i) requirements on the architecture of critical software that must exhibit properties such as modularity, encapsulation, and minimal complexity; (ii) verification methods of the safety requirements including source code review (walk-through and inspection) and source code analysis (control flow, data flow, static code); and testing methods—used to verify software and ascertain its quality. For instance, at the unit testing level, ISO26262 requires structural code coverage such as statement and branch coverage.

GPU software is based on low-end C-like APIs like CUDA [41] and OpenCL [40], which bring some challenges to show adherence to ISO26262. These challenges include the following.

Use of pointers: CUDA and OpenCL programs use pointers as an indispensable feature of their programming model since the programmer has to explicitly allocate and maintain two separate sets of pointers, one for the host memory and one for the device memory. Moreover, the programmer also has the responsibility to perform memory transfers between these two memory spaces. Note that recent versions of both CUDA (6 and later) and OpenCL (2.0) provide two equivalent features called unified virtual memory and shared virtual memory, respectively. These features simplify the programmability and enhance productivity by taking care of the transfers implicitly, providing the user with the abstraction of a single address space. However, they might incur a performance penalty while introducing another black box in the timing and functional behavior, which complicates the certification of the system as we discuss in the following section. Moreover, even with these features, pointers are still present in the programming model. Brook is a stream-programming language targeting GPUs. In the same way, MISRA C [190] constraints C, Brook Auto [42] defines a subset of Brook rules that are certification friendly, without limiting the expressiveness of the language. For instance, Brook Auto does not expose pointers to the programmer and takes care of those tasks automatically, reducing the possibility of human errors

```
1
2  #define MAX_ITERS 10000
3
4  kernel void foo(float a<>, float b[], out c<>){
5    float acc=0.0;
6    for(int i<0; i < a || i < MAX_ITERS; i++){
7      acc += b[indexof(c).x];
8    }
9
10   c = a + acc;
11 }
12
13 int main(void){
14   float a_h[100], b_d[100], c_h[100];
15   float a_d<100>, b_d<100>, c_d<100>;
16
17   streamRead (a_d, a_h);
18   streamRead (b_d, b_h);
19   foo (a_d, b_d, c_d);
20   streamWrite (c_d, c_h);
21 }
22
```

(a) Brook Auto code

```
1  __global__ void foo(float * a, float * b, float * c){
2    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
3    float acc=0.0;
4    for(int i < 0; i < a[tid]; i++)
5      acc += b[tid];
6
7    c[tid] = a[tid] + acc;
8  }
9
10 int main(void){
11   float a_h[100], b_d[100], c_h[100];
12   float * a_d, * b_d, * c_d;
13
14   cudaMalloc(&a_d, 100*sizeof(float));
15   cudaMalloc(&b_d, 100*sizeof(float));
16   cudaMalloc(&c_d, 100*sizeof(float));
17
18   cudaMemcpy(a_d, a_h, 100*sizeof(float), cudaMemcpyHostToDevice);
19   cudaMemcpy(b_d, b_h, 100*sizeof(float), cudaMemcpyHostToDevice);
20   foo<<<1,100>>>(a_d, b_d, c_d);
21   cudaMemcpy(c_h, c_d, 100*sizeof(float), cudaMemcpyDeviceToHost);
22 }
```

(b) CUDA code

Figure 4-2: Example equivalent code programmed with Brook Auto (top) and CUDA (bottom).

Example: Figure 4-2 (top) shows an illustrative example of Brook Auto that highlights some of its benefits. The sample program launches a GPU kernel that operates on two input data vectors (*streams* in Brook terminology) and generates its result in a third data vector. In the program that calls the kernel, there are two versions of each vector required, one for the host (suffixed "h") and one for the device (suffixed "d"), shown in lines 14 and 15, respectively. The same code is written in CUDA, see Figure 4-2 (bottom), shows that pointers are required both in the GPU, for passing data in the kernel (line 1), as well as on the host side for allocating memory (lines 14–16) and managing the transfers between host and GPU buffers (lines 18–19 and 21). Note that the OpenCL version of the code has the same characteristics as CUDA, but it is more verbose. Therefore, it is omitted for clarity. Brook uses statically defined streams that prevent explicit memory allocations (*cudaMalloc*) and low-level memory management, which could result in programming mistakes due to wrongly supplied size parameters or memory exhaustion. Streams cannot be directly accessed (e.g., indexed) from the host side, since this would result in a compilation error, and they can only be accessed using certain API calls (*streamRead* and *streamWrite*) to copy data from and to host buffers. Stream size is integrated within these calls, preventing out-of-bounds accesses from the host side.

From the kernel side, streams can be accessed in two ways. Regular streams in which each GPU thread accesses its corresponding element in the array, declared as "<>" and *gather* streams, which are declared with "[ ]". In the former case, Brook Auto takes care of accessing the correct element, while in the latter it suppresses potentially illegal out-of-bound accesses ensuring fault isolation.

Other dynamic features. Brook Auto also restricts dynamic language features that can lead to deadlocks or complicate the WCET analysis of the software. For example, notice that the kernel in the Brook Auto example contains an extra defensive-programming condition in the loop (line 6). This condition restricts the number of iterations of the kernel to a statically defined upper bound limit, although the main loop condition is input dependent. The absence of such a statically computed condition would result in a compilation failure, thus enforcing this rule. On the contrary, the CUDA version is unprotected from this type of programming risk, which complicates GPU software certification with ISO26262.

4.3.2 Generic ML and Black-Box CUDA Libraries

Generic ML libraries. The dramatic increase in ML usage in a variety of domains makes the leading artificial intelligence companies to provide several widely used frameworks and highly optimized libraries to facilitate and make better use of available platforms and architectures [191]. Like in many other areas, state-of-the-art AD systems strongly rely on these libraries and use them extensively. However, not only the algorithms but also these frameworks and libraries are quite generic, designed based on totally different objectives to those of CRTES in general and AD particular, which challenges providing evidence that software achieves its safety requirements. In fact, to our knowledge, no study has been carried out on the adherence of those libraries to ISO 26262 software requirements.

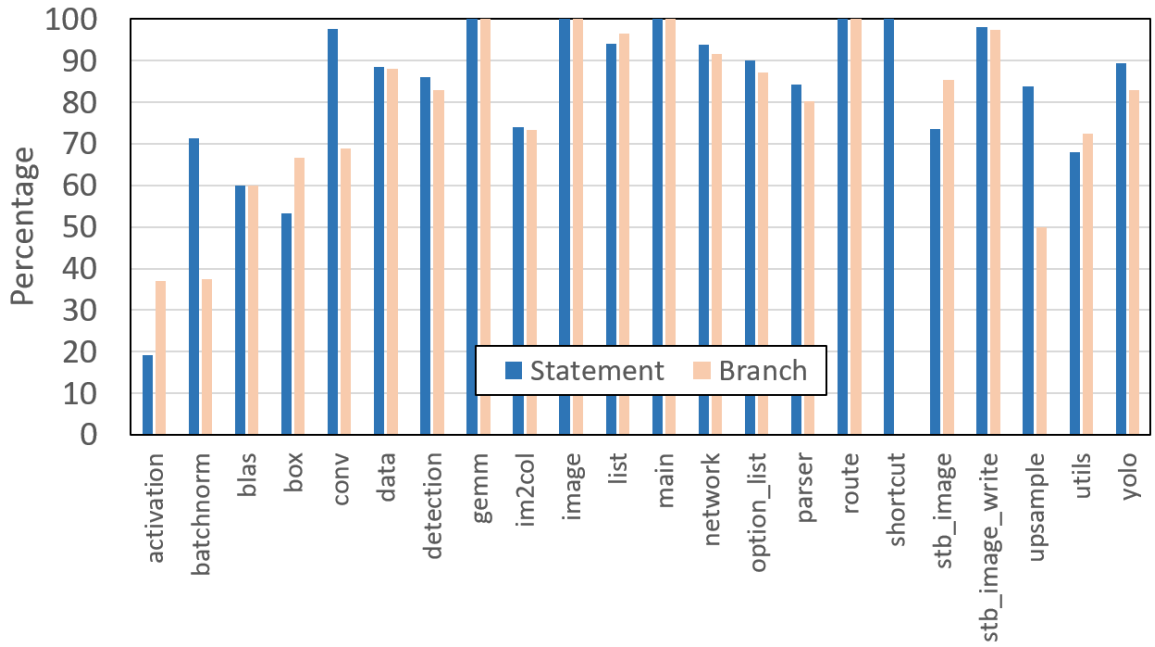


Figure 4-3: Code coverage for YOLO v3.

Example. As an illustrative example, we focus on statement coverage, a basic structural coverage metric at the software unit level. In particular, we run the YOLO v3, which is a state-of-the-art object detector widely used in real AD systems comprising more than 20 functions. We run several real scenario tests and measure simple statement and branch coverage using a RapiCover low-overhead coverage analysis tool [176]. The former captures the fraction of static instructions (those in the binary) executed in the tests, and the latter is the fraction of program branches or conditional states triggered during the tests. Obtained results are shown in Figure 4-3.

Each column represents all the functions in each file. Note that, despite excluding all YOLO functions that were not called, both branch and statement coverage are very low. Average coverage is 83% and 79% for statement and branch, respectively, and as low as 19% and 37%, respectively, for individual files. While ISO26262 does not specify coverage targets, its parent standard, IEC61508 (Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems), recommends 100% coverage for all metrics. Hence, the coverage levels observed for YOLO are not acceptable for any ASIL since either branch or code statement is *highly recommended* (++) for all ASIL levels. It is also worth noting that the concept of code coverage has not even been defined for GPU code. The fact that GPU instructions are single instruction, multiple thread and warp divergence (predicated execution) complicates simply extending CPU code coverage to GPUs.

Low-level CUDA libraries. Libraries used for artificial intelligence and ML in AD—as the majority of widely used operations in GPUs—rely on highly optimized closed-source libraries (e.g., cuBLAS [192] and cuDNN [187]). From a functional safety point of view, these are black boxes without detailed information on their

implementation, code, and algorithm. This might prevent their safety analysis by end users, e.g., source code analysis and code coverage, a mandatory requirement of ISO26262. In our view, overcoming this limitation requires one of the following:

- The use of open-source libraries, which must provide competitive performance. For instance, results show that CUTLASS [193], NVIDIA's open-source collection of CUDA C++ templates and abstractions for implementing high-performance GEMM computations provide very close performance in comparison with cuBLAS.
- Closed-source libraries owners go through the certification process and adapt their libraries to fit ISO26262 requirements.

Code changes to achieve ISO26262 adherence can, however, cause performance loss with respect to the original performance-improving centric code, which is not acceptable in other non-critical domains in which these libraries are used. This can result in the creation of branches of the code specific to the automotive domain, with increased development and maintainability costs.

4.3.3 Domain-Specific Optimizations

Numerous schemes have been proposed to optimize deep learning models (e.g., layer removal and fusion). From those, calibrating the neural network models for lower precision (also known as quantization) is one of the commonly used schemes. In general, these optimizations aim at delivering lower latency and higher throughput for deep learning inference applications and/or reducing the energy profile. However, some of these optimizations can come at the expense of increasing the probability of producing a wrong result by the application. Hence, these approaches directly affect the accuracy of the application by a considerable and wide margin depending on the input data. For a critical domain such as automotive, these schemes decrease the decisiveness of the application. Hence, such optimizations must be used with caution in AD factoring their impact on overall's application accuracy

4.3.4 Time Predictability

In CRTES, functionalities need to be completed within certain timing bounds, called deadlines. Hardware and software architectures in CRTES require time predictable timing behavior that allows deriving tight and reliable WCET estimates [194]. WCET analysis of GPU software is still in a very early stage [195]. Static timing analysis has been performed under very limited scenarios, such as assuming that the kernel is executed on a single streaming multiprocessor, while measurement-based analysis on the other side has been performed without providing enough evidence that the worst-case scenarios have been exercised. Both solutions are negatively affected by the existence of many undocumented features in GPU architecture and software, contributing to analyzing their real-time properties analysis hard, compared to the CPU architectures used traditionally. In our view, a way to alleviate this problem is by increasing GPU

observability. In particular, a more powerful set of WCET-aware monitors (performance monitoring counters) helps to provide insightful information on application worst-case behavior when run on the target hardware as an instrumental element to build a safety argument [196]. On the timing analysis side, the use of statistical-based approaches is on the rise as it fits the increasing execution-time variability applications suffer when running on complex processors such as GPUs [186, 188].

4.4 Conclusions

As the software component to implement safety-related functionality continues to increase in cars, so do its performance requirements and the guarantees required for its correct behavior. The former is covered in a cost-effective manner by deploying software- and hardware-accelerator techniques originally designed for other high-performance, i.e., noncritical, domains. As we have discussed in this chapter, the sustainability of this approach builds on developing well-designed adaptations to address key challenges when satisfying safety regulatory standards. The overall ISO26262 philosophy builds on defining a set of requirements and a set of tests, which emanate from the requirements, that are used to assess whether a particular software implementation is correct. Whether this approach can be directly applicable to ML-based code is still an open question due to the difficulties in defining whether, for instance, object detection software works properly, and defining the tests to assess so. Overall, new interpretations and analyses of how to certify software with respect to that in place by ISO26262 might be necessary.

Chapter 5

GPU scheduling policies

5.1 Introduction

Autonomous Driving imposes the use of high-performance hardware, such as GPUs, to perform object recognition and tracking in real-time. However, differently to the consumer electronics market, critical real-time AD functionalities require a high degree of resilience against faults, in line with the automotive ISO26262 functional safety standard requirements. ISO26262 imposes the use of some source of independent redundancy for the most critical functionalities so that a single fault cannot lead to a failure, being Dual-Core Lockstep (DCLS) with diversity the preferred choice for computing devices. Unfortunately, COTS GPUs do not support diverse DCLS by construction, thus failing to meet ISO26262 requirements efficiently.

In this chapter, we propose lightweight modifications in the internal GPU scheduler to enable diverse DCLS for critical real-time applications without diminishing their performance for non-critical applications. Our solution is applied on COTS GPUs in general, and in NVIDIA GPUs in particular. We show how enabling specific mechanisms for software-controlled kernel scheduling in the GPU, allows guaranteeing that redundant kernels can be executed in different resources so that a single fault cannot lead to a failure, as imposed by ISO26262. Our results on a GPU simulator and an NVIDIA GPU prove the viability of the approach and its effectiveness on high-performance GPU designs needed for AD systems.

More in detail, we make the following contributions:

- We identify the main requirements to enable ASIL-D compliance for Commercial Off-The-Shelf (COTS) GPUs, assessing to what extent they have the potential to meet ASIL-D requirements.

- We provide a set of lowly-intrusive modifications that allow adhering to ASIL-D requirements without diminishing their performance for non-safety-related functionalities. These modifications allow GPU vendors to reuse their designs avoiding a significant increase of their Non-Recurring Expenses (NRE).

- We perform a detailed analysis on an NVIDIA COTS GPU, implement modifications on a GPU simulator [114] where we can assess both performance and ASIL-D compliance, and evaluate what the performance impact would be on the COTS GPU.

5.2 GPU Design and Operation

This section introduces some key concepts related to GPU design and operation, and how those relate to the execution of kernels, and how diverse redundancy could be achieved on top of COTS GPUs. Since different components have different names across GPU vendors, we adhere to NVIDIA nomenclature for the sake of simplicity (and because NVIDIA is already targeting the automotive domain [28]), but concepts apply to virtually any COTS high-performance GPU, for an extended description, please refer to the *GPU Architecure* section (2.3.1) in the Chapter 2 (Background).

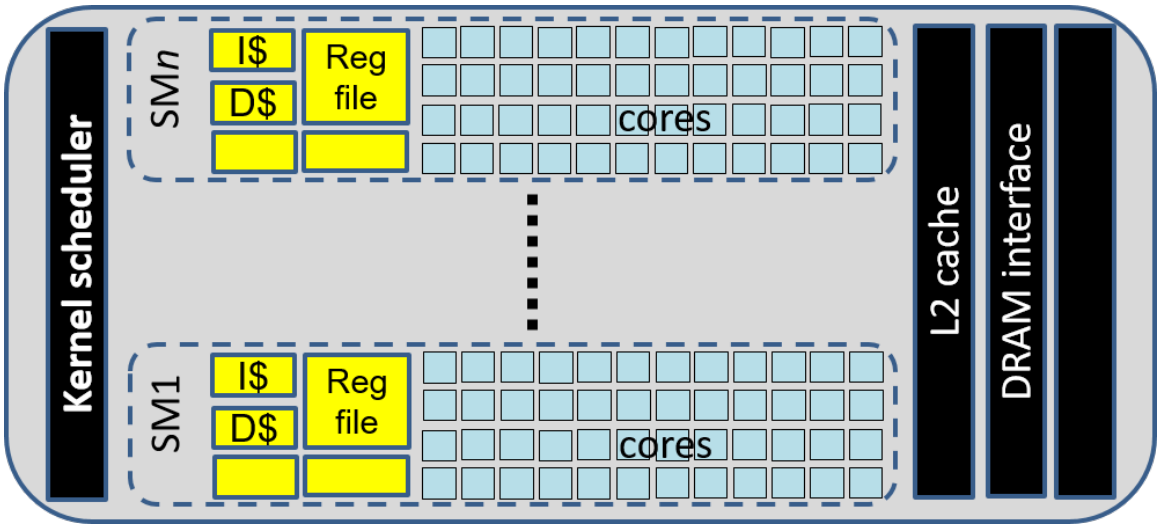


Figure 5-1: GPU generic schematic.

Figure 5-1 shows a schematic of the main GPU components relevant to this discussion. First, the GPU has a number of Streaming Multiprocessor (SM), which we indicate as SM 1 to SM n in the Figure. Each one consists of several execution elements, including CUDA cores, or simply cores, which can execute arithmetic instructions, load/store units, and complex cores for special instructions. We group all of them within the concept of cores for the sake of this discussion. SMs also include a number of internal resources shared across cores, such as instruction and data caches, on-chip shared memory, a register file, and an internal scheduler, warp dispatcher, among others. The GPU also includes a number of resources shared across SMs, such as a second level (L2) cache, DRAM and other interfaces and, a kernel scheduler.

The kernel scheduler dispatches the *thread blocks* (or group of threads) of the kernels to SMs. In particular, the CPU sents kernels to the GPU, each kernel consists of a number of thread blocks, and each thread block is bound to an SM for its entire execution without the possibility to migrate. However, different thread blocks from the same kernel can coexist on an SM provided that there are enough resources.

For instance, if kernels $k1$ and $k2$ are dispatched to the GPU, where $k1$ has 3 thread blocks (tb_1^{k1}, tb_2^{k1}, tb_3^{k1}), $k2$ has 4 thread blocks (tb_1^{k2}, tb_2^{k2}, tb_3^{k2}, tb_4^{k2}), and our GPU has 2 SMs (SM_1 and SM_2), SM_1 may execute tb_1^{k1}, tb_2^{k1}, tb_2^{k2}, tb_4^{k2} in a time-multiplexed manner but not necessarily completed with this order, and SM_2 may therefore execute tb_1^{k2}, tb_3^{k1}, tb_3^{k2} also time multiplexed. Note that newer GPU architectures targeting the high-performance domain may have fewer limitations about executing different kernels in a single SM but, in general, how thread blocks are scheduled to SMs is an undisclosed feature, which, as discussed later, has prominent importance in our work.

5.2.1 Redundancy and Diversity Elements

Storage and communication components can be properly protected from CCFs by using ECC and/or CRC. In fact, some of those components are explicitly protected with those means in NVIDIA GPUs since the Fermi generation [197] back in 2010, including register files, SM cache memories, and shared L2 cache, which employs Single Error Correction, Double Error Detection (SECDED) codes.

Regarding cores, no explicit protection has been reported. However, we consider GPUs that have been shown compatible with ASIL-B ISO26262 requirements [198], and thus, the failure rates and coverage of the cores and the corresponding safety mechanisms are in concordance with the requirements imposed by the certification standard. Additionally, GPUs are intrinsically redundant within an SM and across SMs. Therefore, it is possible executing the same computation twice in different cores at different times so that CCFs are avoided. In particular, CCFs related to defects of a hardware component can be avoided by executing the same computation redundantly in different cores. Transient CCFs related to faults affecting multiple components simultaneously (e.g., a voltage droop) can be avoided by performing redundant execution at different time instances.

Unfortunately, NVIDIA GPUs, as well as other manufacturers, do not provide means to control how thread blocks are scheduled across SMs or a thread block is scheduled within a SM. Even worse, scheduler policies are not even publicly described, which further defeats any attempt to exercise direct control on the execution in the GPU, thus challenging the ability to enforce diverse redundancy on GPUs. Finally, to the best of our knowledge, the global kernel scheduler does not include any form of redundancy for fault detection.

The aim of this contribution is to propose the smallest modifications possible to COTS GPUs to enable diverse redundancy to prevent CCFs.

5.3 Scheduling Strategy for Diverse and Redundant GPU Execution

Execution on the cores (computing and load/store units) needs some form of strategy to reach diverse redundancy, and the global kernel scheduler also needs means to avoid CCFs. In this section, we introduce first our software approach to achieve redundancy, we analyze to what extent diversity can be achieved, and then propose low-cost modifications on the GPU design to achieve fully diverse redundancy.

5.3.1 Kernel Redundancy

In our approach – in line with the existing AD platforms – we consider a system in which ASIL-D capable microcontrollers (e.g., DCLS) offload intensive computations to the GPU. Our strategy consists of executing kernels twice on the GPU and comparing their outcomes in the DCLS cores of the CPU. In particular, a DCLS core (1) allocates memory on the GPU memory space for both redundant kernels, (2) transfers data physically (if needed), (3) launch the two redundant kernels, (4) collects results from both kernels back to the CPU, and (5) compares their outcomes in the DCLS cores. In this scheme, all actions performed on the DCLS cores are naturally protected against CCFs, as well as data communication and storage, which occur on ECC or CRC protected components[1].

We consider identical redundant kernels. In general, one could create different kernel grids, so that thread blocks across redundant kernels differ, in order to introduce some form of diversity. However, the lack of control on the global kernel scheduler and SM internal schedulers prevents from guaranteeing specific diversity levels in the execution in the general case. Therefore, in this contribution, we do not study diverse kernel generation, which is part of our future work but remains beyond the scope of this Thesis.

The process to dispatch kernels to the GPU is intrinsically serial, so redundant kernels arrive at different time instants at the GPU, which might bring some form of diversity. However, this does not guarantee that two redundant thread blocks (from redundant kernels) cannot arrive at different SMs at the same time and, therefore, execute the same operations simultaneously, thus being subject to some CCFs. Concerning permanent CCFs, ensuring diversity would require – as already done for functionally identical core replicas in ASIL-D DCLS processors – implementing some form of physical diversity at the layout and/or floorplan levels. However, even when having this physical level diversity, redundant thread blocks across redundant kernels may end up executing on the same SM at different time instants, thus also being subject to some permanent CCFs. Overall, redundancy can be easily achieved, but further means are required to enforce diversity.

[1]In this contribution, to keep the focus on the GPU design we consider dual modular redundancy suffices to provide fail-operational capabilities (i.e., errors can be recovered within the FTTI by, for instance, re-executing upon error detection. However, our approach could be seamlessly extended to other redundancy levels (e.g., triple modular redundancy)).

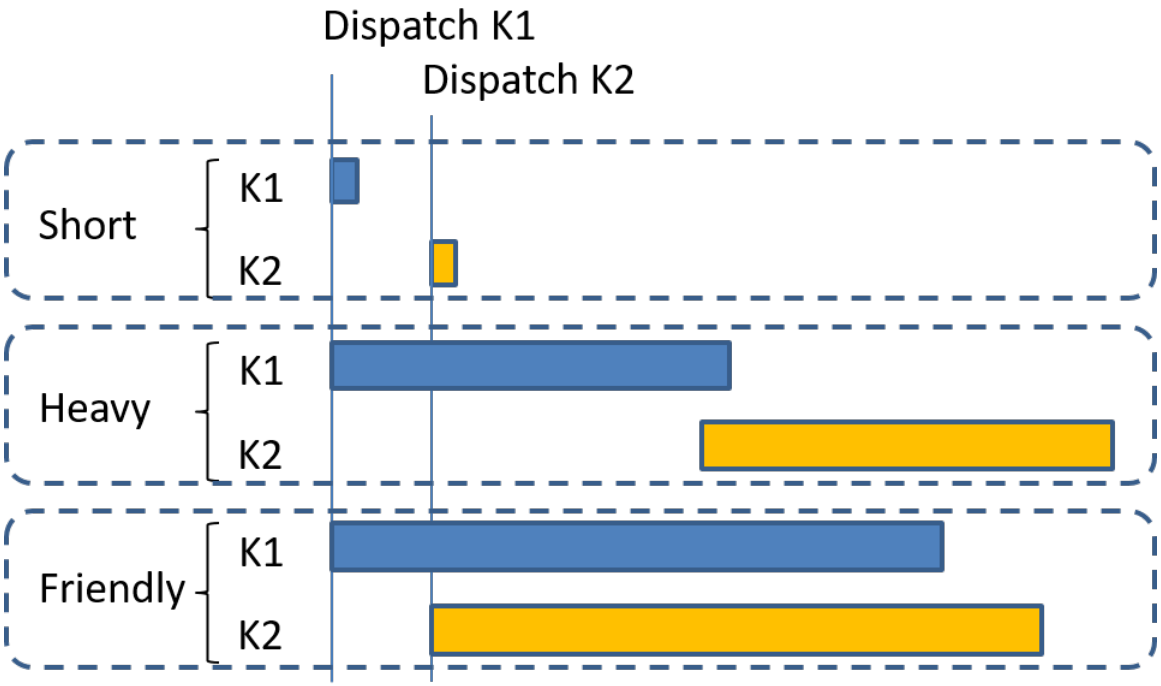


Figure 5-2: Kernel categories based on their overlapping.

5.3.2 Redundant Kernel Execution Patterns

A relevant characteristic for this proposal but also to the following GPU contributions is the ability of a given kernel is to be able to execute two redundant copies concurrently in a single GPU. However, this classification is not only kernel-dependent but also hardware dependent as well. Thus, we start by categorizing kernels based on two criteria: whether they can potentially overlap their execution and whether they use too many resources to prevent overlapping. This leads to three categories, also shown in Figure 5-2 for clarity:

- **Short kernels**. Those kernels execute too fast to overlap practically. In particular, by the time the second kernel is dispatched to the GPU, the first kernel has already finished its execution.
- **Heavy kernels**. Those kernels coexist in the GPU, but a single kernel uses too many resources to allow the other to start their execution. This makes that no overlapping occurs at all, or it is little, just at the end of the execution of one kernel when it starts releasing resources so that the other can effectively start its execution.
- **Friendly kernels**. Those kernels coexist in the GPU and use limited resources so that both kernels can make progress concurrently.

As shown, different kernel types may have different degrees of overlapping (little, none or high). We propose two specific kernel scheduling policies in the GPU that allow achieving diverse redundancy in all cases: *SRRS* and *HALF* policies.

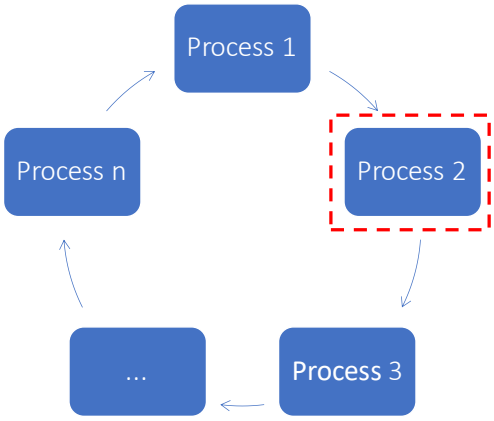


Figure 5-3: Round Robin example. Multiple processes share the CPU and each one is given the same *quantum* time to execute. In this particular instant, Process 2 is using the CPU (marked with the red square).

5.3.3 SRRS (Start, Round-Robin, and Serial) policy

Our first scheduling policy contribution is SRRS, which stands for Start, Round-Robin, and Serial. Before discussing it, let us describe the Round-Robin policy, which our contribution builds upon.

Round-Robin Policy

The Round-Robin scheduling algorithm is one of the most common scheduling algorithms due to its simplicity and fairness. The term dates from the 17th-century French *Rond ruban (round ribbon)*, where the king was not very fond of peasants' petitions and killed the first three signatories of a petition. The peasants brilliantly devised the plan to start signing the petition circularly instead of vertically so that there was not a first signatory in the list [199]. Later, the Round-Robin the principle was adopted in many fields (e.g., telecommunications, storytelling), where each person or actor takes an equal share of something in turn [200].

An example is shown in Figure 5-3, where a number of processes (n) compete to have CPU time. Each process is given a time slot or *quantum*, equal to all processes. Then, processes are scheduled circularly where each one will execute at maximum his *quantum* time. Once the quantum of a process has been consumed, the scheduler will select the following process in the line. In this way, all the processes are given the same fair portion of the CPU time. Note that the term round-robin is also used in contexts where access to the shared resource is given following the order as described, but allowed time may not be pre-defined, so usage time may vary. For instance, processes could be allowed to use an I/O interface for data transmission so that each process is allowed to perform a transmission during its turn, but without imposing any limit on the amount of data to be transmitted by each process, which could differ across processes.

Notice that there are small differences between the example and our scenario. Instead of having one resource (a single core) that needs to be shared among different consumers (processes), in our case, we have multiple resources (SMs) that needs to be shared among two different multiple consumers (thread blocks) of two different groups (kernels). Also, to guarantee diversity, we need to ensure that the same thread blocks from the redundant kernels will not execute in the same functional unit. To achieve this, redundant threads need to be scheduled in different SMs.

SRRS Policy

Once introduced the default Round-Robin algorithm, we detail how we tailor it to our needs. ① We do not start the kernel execution until the GPU is idle; ② we can select the SM where the first thread block will be dispatched; ③ SMs are allocated following a round-robin policy; ④ kernel execution is fully serialized, thus delaying the start of the second (redundant) kernel until the first kernel has finished its execution; ⑤ no further kernel can be executed in the GPU until the second one also finishes.

By using *SRRS* with different starting SMs for both kernels, diversity is achieved naturally. The first kernel finds the GPU idle, starts in a particular SM (e.g., SM_i) and allocates SMs in round-robin order starting from SM_i until the kernel completes its execution, without any interference from any other kernel. The second kernel also finds the GPU idle, so in the same state as the first kernel, and starts its execution in SM_j, where $i \neq j$. SMs are also allocated round-robin, but since the starting SM differs for both kernels and no interference occurs, each single thread block executes in different SMs across redundant kernels. Therefore, any single computation occurs in different kernels at different time instants, thus avoiding CCFs in the cores.

5.3.4 HALF policy

Different from the case of SRRS, HALF does not build on any specific scheduler algorithm. Instead, *HALF* policy builds upon allocating half of the SMs to one kernel and the other half to the other kernel. This division of the resources naturally imposes the use of different SMs for each kernel. On the other hand, the fact that their starting times differ due to the serial dispatch of kernels to the GPU also enforces that any given redundant computation occurs at different time instants. Note that kernels could interfere with the use of shared resources delaying each other. However, their requests can never occur at the same time because (i) either shared resources can process them in parallel, so no interference occurs and so no timing impact, or (ii) requests are serialized (at least partially) for a given shared resource so that a given request arrives before the first kernel than for the second, and the second can neither start nor finish simultaneously with the first one, thus preserving some slack across kernels. Therefore, any single computation occurs both in different SMs and at different time instants, thus avoiding CCFs during the execution inside the SMs.

5.3.5 Diverse Redundancy in the Kernel Scheduler

Both policies, *SRRS* and *HALF*, schedule any given thread block from both kernels at different time instants and to different SMs. Therefore, any fault causing an improper execution of the kernels may have several consequences: (1) execution occurs functionally correctly in different SMs to the ones intended, but still redundantly and with diversity. In this case, no failure occurs. (2) execution occurs functionally correctly in different SMs to the one intended but fails to achieve diversity (e.g., the same computation occurs redundantly on the same SM). In this case, let us recall that ISO26262 requirements relate to the ability to avoid a single fault from causing a failure. Hence, upon a fault in the scheduler, we must assume that the remaining components are fault-free and hence, even if their execution is not diverse, no further fault is expected. (3) execution does not terminate or terminates with errors for at least one kernel (e.g., skipping a thread block). In this latter case, the different behavior of both redundant kernels (each thread block is executed at different times in different SMs) makes that even if there is a physical fault in the scheduler, its behavior will differ across kernels, so evidence on diversity is enough to meet ISO26262 requirements.

A final noteworthy remark in the context of ISO26262 is the fact that we can assume that multiple faults cannot occur simultaneously as long as faults are timely detected. In particular, this means that faults of type (2), so with no functional impact but decreasing diversity, must be detected if related to a physical fault since, otherwise, a future fault on a core in an SM could lead to an undetected error, and thus to a failure. In order to avoid this behavior, the global kernel scheduler must undergo periodic tests so that physical faults do not become latent.

5.3.6 Appropriateness of the Scheduling Policies

Both scheduling policies, *SRRS* and *HALF* achieve diverse redundancy for all kernel types. However, each kernel type is particularly suitable for one of them.

- Short kernels may potentially use many GPU resources during their (short) execution. Since their execution does not overlap at all, *SRRS* is expected to cause no performance degradation at all. Instead, *HALF* could increase kernel execution time, mainly due to contention on the shared resources, thus impacting performance.

- Heavy kernels need many GPU resources and have little or no overlap at all. Hence, *SRRS* may only slightly increase their execution time if they overlap a bit. Instead, *HALF* could easily increase the execution time of a given kernel noticeably while not allowing the other one to start due to a lack of resources.

- Friendly kernels can run concurrently, so using *SRRS* could cause a significant execution time increase due to their serialization if a given kernel is unable to exploit all SMs efficiently. Instead, *HALF* grants each kernel half of the SMs, which is the number of resources they would use if run concurrently without explicit control for the sake of diversity.

Overall, *SRRS* is the most convenient policy for short and heavy kernels, whereas *HALF* fits friendly kernels. Since kernel classification is performed during the analysis phase of the system, the particular policy to use for each one can be decided before system deployment to execute each kernel with the most convenient policy during operation. Note that this implies that specific means are required to select the global kernel scheduler policy during operation, which we foresee as feasible since it is not different from other reconfigurations applied on high-performance components such as enabling/disabling prefetchers, changing fetch policies, and the like.

5.4 Evaluation

This section evaluates the two proposed scheduler algorithms. Firstly, we show an evaluation of the two proposals against the GPGPUSim [114, 115] kernel scheduler described in Section 3.5.1. Secondly, we evaluate the impact on the execution time of *SRRS* against the scheduler of a COTS GPU.

We implemnt the proposed scheduling policies *SRRS* and *HALF* in the simulator (again, described in Section 3.5.1). The GPU model with this simulator is based on the NVIDIA Pascal GPU (GeForce GTX 1050Ti [171]) and consists of 6 SMs. The applications selected to be used on the evaluation are applications from the Rodinia benchmark suite [7, 173] described in Section 3.2.1.

5.4.1 Implementation in GPGPUSim

In order to test our techniques, it has been required to perform software modifications on the applications selected. We manually insert compiler directives to produce multiple binaries from a single application. For each application, we have three different binaries; the first, named *baseline* which, as the name indicates, is the initial version of the application, without any modification. Secondly, we have create a version which duplicates all the kernel executions and uses for each one a unique user-defined queue stream to allow parallelization of operations and disable implicit synchronization. We also replicate their inputs (if they are modified inside the kernel) and their outputs. Finally, after all, kernels are executed, the results are transferred back to the CPU and are compared. We name this version as *redundant* since it creates redundant kernels for each original kernel. Finally, a third version is used, which is a variation of the *redundant* version. In this case, redundant kernels have been forced to be serialized. This version, as we discuss later, allows mimicking the execution time of SRRS on a COTS GPU.

We use different queue streams to dispatch kernels to enable their parallel execution based on the following. If not specified in the call, kernels in CUDA are queued in the default execution queue named *NULL stream*, authors in [107, 201] discovered that the *NULL stream* has an implicit serialization which means:

1. If one operation is executing from the NULL stream, implicit synchronization and blocking occur which means that no other operations can happen simultaneously.

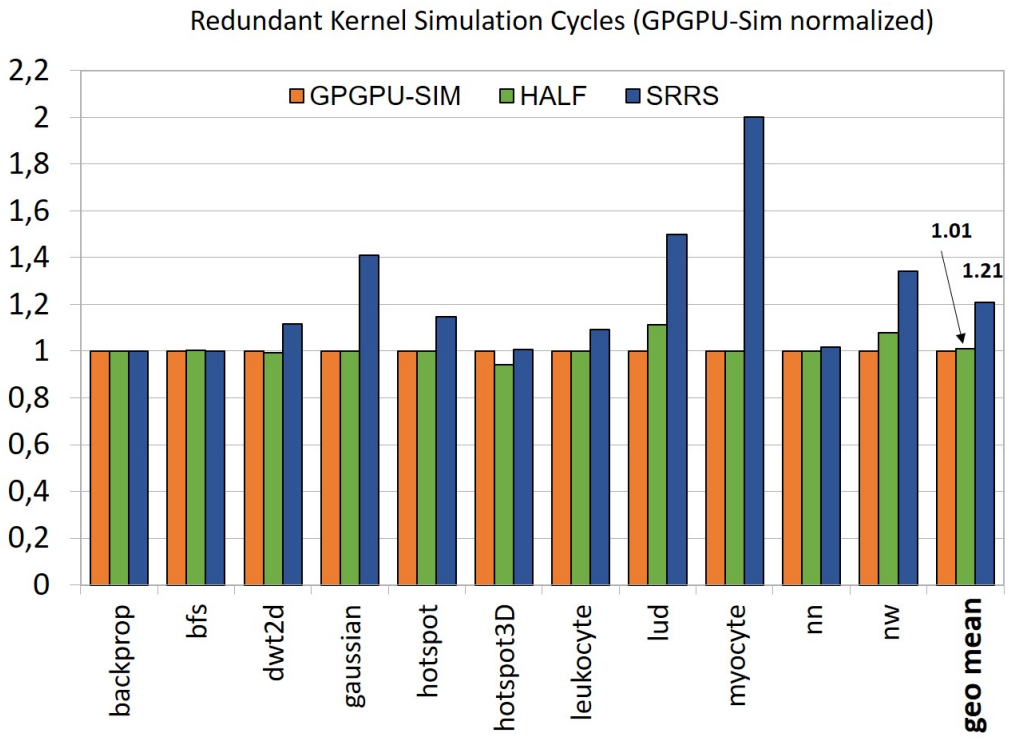


Figure 5-4: Scheduler simulated cycles using GPGPUSim normalized to the default simulator scheduler.

2. To parallelize two operations, these two operations must be executed from different user-defined queue streams (*CUDA streams* in NVIDIA nomenclature) and nothing must be executed from the NULL stream (1).

To evaluate the proposals, we modify the scheduling policy in the simulator according to the requirements that each proposal imposes on the way the available SMs have to be assigned. For *HALF*, we use the default scheduling policy, already implemented in GPGPUSim and restrict each kernel execution to 3 dedicated SMs (half of the available). For *SRRS* we ensure that once a new kernel needs to be scheduled, all SMs are empty. We compare the performance of *SRRS* and *HALF* against the one obtained with the default GPGPUSim scheduler that can allocate all GPU SMs (6) to the kernels without any constraint.

5.4.2 Simulation Results

We execute the *redundant* version of the Rodinia benchmark suite applications on the GPGPUSim using the three kernel schedulers, our two proposals, and the default scheduler of GPGPUSim, as explained earlier. Results comparing the simulated time of *only the kernel execution cycles* for each version of the Kernel scheduler are shown in Figure 5-4.

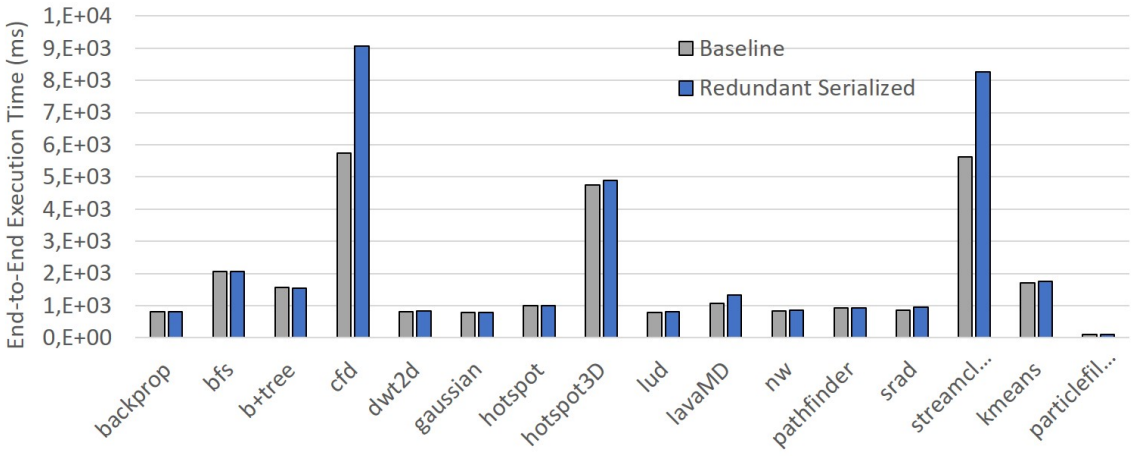


Figure 5-5: SRRS implementation by serializing redundant kernels

For each application, we have three bars; each one represents the simulated cycles of the kernel's execution (so only time spent in the GPU execution) for each scheduler, normalized to the default GPGPU-Sim scheduler. On the left of each application, the white bar represents the timing of the default scheduler. In the center and grey is the execution cycles for the HALF scheduler and in the right and black for the SRRS. We also include the last triplet of bars, which is the result of applying the geometric mean among all the applications' timings.

Due to the costly executions on the simulator, we evaluate a subset of the benchmarks. In particular, we inspected them and identified that most of them include friendly kernels for our modeled GPU. Thus, running additional experiments does not provide further insights. In general, the performance overheads of the proposed scheduler policies are not very high in comparison with the default scheduler (except for *myocyte*). In particular, *HALF* policy performance overheads are negligible for 9 out of the 11 benchmarks analyzed and only 10% in the worst-case (*lud*). The results for the *SRRS* policy are slightly worse due to the extra overheads that this policy incurs to perform the serialization. For *SRRS*, performance overheads can be up to 99%. In general, kernels are friendly or short, and, if they are short, they also require at most half of the SMs. Hence, by restricting them from using half of the SMs with *HALF*, the performance penalty is, in general, very low. Instead, serialization imposed by *SRRS* increases their execution time. The only exception are *bfs* and *backprop*, which have very short kernels requiring more than half of the resources. Hence, serialization imposed by *SRRS* is innocuous, whereas limiting the number of SMs with *HALF* increases execution time. However, since kernel execution time is much lower than the execution time of the CUDA commands to launch the kernels, the relative impact of such an increase is tiny.

5.4.3 COTS GPU Results

Finally, to assess the suitability of the proposed redundant execution in a real environment and understand the impact of redundant execution w.r.t. non-redundant execution, we mimic the implementation of *SRRS* on a COTS GPU and execute it in the NVIDIA GeForce GTX 1050Ti [171] described in section 3.1.2. In this experiment, we include all the timings of the execution from the start to the end: the CPU execution timing, the data transfers between CPU and GPU, as well as the GPU execution time. We serialize the redundant kernel's execution using the CUDA call *cudaDeviceSynchronize()* that prevents the execution of further operations until all previous operations on the GPU have been completed. While such a solution does not enforce diversity due to the lack of control of the particular SMs used, it causes the same timing behavior. Note that mimicking *HALF* is not possible on the COTS GPU, since CUDA doesn't provide control over the SMs used by a kernel.

Figure 5-5 compares the execution time of end-to-end executions of the benchmarks with the redundant serialized and no redundant kernels of the Rodinia benchmarks. By running on a real platform, we could afford to run all benchmarks timely.

The bars in the plot show the result of averaging out 100 executions in ms. As shown in the plot, the redundant execution of the kernels does not incur significant performance degradation for the workloads analyzed. In fact, for all the benchmarks but two (*cfd* and *streamcluster*) the impact of redundant execution is negligible. The main reasons for such behavior are as follows: (1) the impact of *SRRS* is, in general, low, as shown in Figure 5-4; (2) the contribution of the kernel execution to the total execution time of the benchmark is relatively low in general; and (3) the cost of sending input and output data twice and comparing the outputs of the kernels in the CPU is also very low in relative terms for these applications. In the case of *cfd* and *streamcluster*, the two only notable exceptions to this behavior, we note that serialization imposed by *SRRS* has a relatively significant impact on the execution of the kernels and execution time of the benchmarks is dominated mainly by the kernel execution. The latter also makes the relative contribution of duplicating input data, transferring back output data to the CPU twice, and comparing outputs is non-negligible, thus contributing to the execution time increase w.r.t. the non-redundant version of the benchmark.

5.5 Related Work

Some authors assess the effectiveness of FPGA, ASIC, and GPU designs for AD applications [186]. The suitability of GPU utilization in the context of safety-critical applications from the point of view of real-time performance has been assessed in several works [107, 108].

Redesigning GPUs, based on the reliability required for ASIL-D certification, is regarded as too costly. Therefore, commercial platforms such as RENESAS R-Car H3 [22] and NVIDIA Xavier [28] targeting the automotive domain, including a general-purpose high-integrity microcontroller together with a COTS GPU. Thus,

in order to achieve ASIL-D fail-operational capabilities, these platforms rely on diverse software implementations of complex algorithms or fully redundant SoCs, which comes at the expense of drastically increasing the design and V&V costs in the former case, and the hardware cost and reliability concerns in the latter case.

Previous works use spatial partitioning to improve multitasking performance on a single GPU [202, 203] by exploring the scheduling per SM. Instead, Wu et al. [204] use a method that enables program-level spatial scheduling on the GPU by using SM-centric program transformations, which allow executing kernels in the desired SMs. Pai et al. [205] focus on enabling better multi-application concurrency by modifying the GPU runtime to avoid serialization of memory transfers and kernel executions. They also develop the idea of *elastic kernels* by modifying the logical threads to avoid underutilization of the GPU resources and improving the concurrency of multiple kernels. Jain et al. [206] use a software-only technique to partition the GPU in order to execute multiple kernels without interference. Although redundant kernels are not evaluated, computing and also memory partitioning, by using memory coloring, could be employed in our work if we would like to replicate the input data of the redundant kernels. However, neither of those solutions provides redundancy per se nor any means to guarantee diversity since those solutions do not target critical real-time systems.

Some works have been performed in the high-performance domain, targeting reliability by creating RMT (Redundant Multi-Threading) in a GPU [119] or using automatic compiler transformations to transform GPU kernels into redundantly threaded versions [124]. However, none of those solutions guarantees diversity, as needed for ASIL-D automotive systems. Overall, our work is the first attempt to deliver diverse redundancy on GPUs, as needed to reach ASIL-D requirements in automotive systems.

5.6 Conclusions

The use of GPUs for highly-critical autonomous driving (AD) software poses several functional safety requirements for the design and utilization of GPUs. While existing AD-specific GPUs already meet some of those requirements, redundant diversity – needed for ASIL-D software – is not reached efficiently and can only be reached by deploying heterogeneous software implementations and/or computing platforms, which jeopardizes cost and efficiency.

This contribution proposes minor modifications to the scheduling policies of GPUs that allow guaranteeing by construction, diverse redundancy, thus reaching ASIL-D compliance efficiently without the need of increasing design and/or procurement costs. In particular, we show how the explicit control of the SMs used for a given kernel, together with the serialization of redundant execution in some cases, allows achieving diverse redundancy at a low cost with respect to uncontrolled redundancy.

Chapter 6

GPU Software-only diverse redundant execution

6.1 Introduction

Autonomous Driving (AD) systems require integrating a number of High Performance Computing (HPC) platforms in the car. While the performance provided by many existing HPC platforms, often incorporating GPUs suffices to meet the computing requirements of AD systems – as confirmed by existing AD systems demonstrations [207] – it is unclear how these computing systems may meet the highest Automotive Safety Integrity Levels (ASIL) as dictated by ISO26262 [39].

As explained in Section 1.2, safe states may not exist any longer in the context of AD since it may be unacceptable to transfer the control to a hypothetic driver. Instead, the system must keep operating correctly upon a failure, which makes that computing components in ASIL-C/D functionalities must also reach ASIL-C/D. Hence, it becomes critically important to enable some form of lockstep in the GPU part of AD platforms to reach ASIL-C/D to avoid using fully-redundant functionalities. Moreover, such lockstep operation must occur on-chip, as in the case of general-purpose cores (e.g., Infineon AURIX processors [1]), for efficiency and cost reasons, since setting up two GPUs increases hardware costs and reliability concerns.

In this contribution, we tackle this challenge by enabling diverse redundancy on a single GPU with software-only means. In particular, the contributions of this work are as follows:

- A thorough analysis of the features of GPUs, with a focus on an NVIDIA representative, and how they enable or limit diverse redundancy without any hardware modification.
- An analysis of some compute-intensive applications on the GPU identifying different kernel categories depending on whether software redundancy also achieves diversity and, if not, the cause impeding to achieve diversity.
- Software strategies to achieve diversity on those kernels failing to achieve it so that diverse redundancy is achieved for any kernel size.

- An extension of the previous item to implement TMR inside a single COTS GPU.

- Quantitative evidence of how the staggering between redundant execution is created in a single GPU.

- Some discussion on whether those CCFs are still present in the case of diverse TMR on COTS GPUs.

Overall, our approach enables ASIL-C/D compliance on GPUs without needing fully-redundant systems, thus containing design and V&V costs.

The structure of this contribution is the following: First, we introduce the target platform of our proposed solution, the description of the offloading process together with the required software modifications follows, in which we detail the minimal software modifications required to enable the solution, after describing how the interaction of CPU-GPU usually occurs. Next, we describe characteristics that are key to enabling a *diverse redundant* execution to occur safely on a COTS GPU building on software-only means. We start discussing the kernel execution patterns we observe, and how the initial staggering is created in our proposal. Next, we discuss how SMs are shared when multiple kernels are executing concurrently. Following, we detail our solution and how the previously described internals of the GPU affect it. In particular, we show that an initial diversity can be guaranteed if redundant kernels run concurrently. We conclude the proposal by presenting a protocol that modifies any kernel that precludes the concurrent execution of its replica (*heavy* kernel) to enable such concurrency (*friendly* kernel), and an extension that enables TMR execution in a single GPU.

In the evaluation section, we perform different experiments in which we show evidence of the theoretical proposals exposed earlier, such as the staggering creation. We evaluate our solution, especially from the time perspective in different experiments, and show that using the proposed transformation protocol, we can obtain a concurrent kernel execution on a previous *heavy* kernel. Then, we show the results of a fault injection campaign in both the baseline application and our solution, and we compare and discuss the software-only solution with the previous hardware approaches.

Lastly, we include a related work section and the conclusions of the chapter.

6.2 Enabling ASIL-D GPU Operation

6.2.1 Target Platform

High Performance Computing (HPC) ASIL-D capable platforms typically combine a low-performance microcontroller amenable for the automotive domain (i.e., ASIL-D capable) and an HPC accelerator delivering high computation throughput, but whose adherence to ISO26262 requirements is unknown, so its appropriate use for ASIL-C/D systems needs to be investigated.

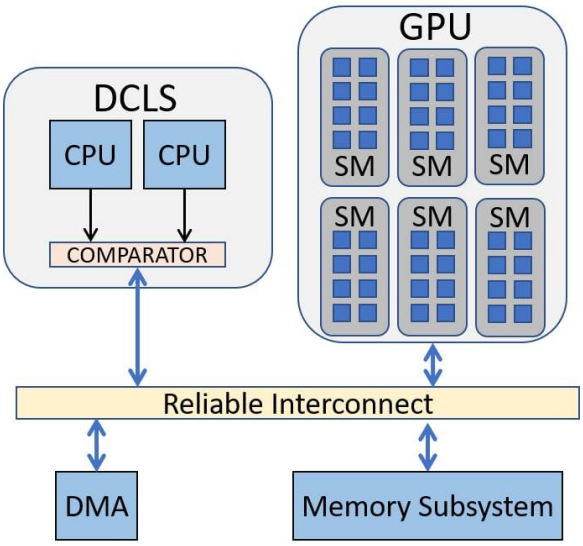


Figure 6-1: Proposed Computing Platform architecture

Without loss of generality, we consider an NVIDIA GPU accelerator, thus analogous to those in NVIDIA Drive and Xavier families for the automotive domain. However, the findings in this contribution can easily be extrapolated to other products.

In this platform, the sequential (*control*) code is executed in the microcontroller in lockstep mode to achieve diverse redundancy, as needed for ASIL-D compliance. Instead, complex and parallel algorithms required for the continuous rendering of the surrounding environment (e.g., object detection and tracking) among other functionalities are offloaded to the GPU accelerator. Figure 6-1 shows a schematic of the proposed hardware platform.

Memory data and on-chip communication during the execution phase of the GPU occur on the same resources (shared) as those used by the ASIL-D MCU and hence, they are naturally protected by specific Error Correcting Codes (ECCs). Additionally, communications between the memory subsystem, the microcontroller and the GPU must be ECC/CRC (Cyclic Redundant Check) protected to guarantee diverse redundancy also on the communication side.

6.2.2 Offloading Process and Software modifications

The following steps are taken to offload computation onto the GPU:

- Offloading preparation process (① allocate memory and ② transfer data from host to the GPU device in Figure 6-2).
- ③ Kernel launching.
- ④ Collection of the results produced and ⑤ deallocating memory.

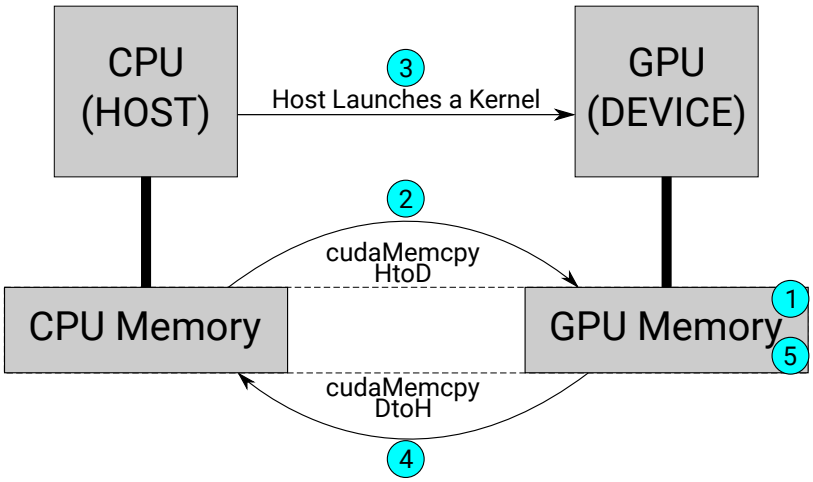


Figure 6-2: Common CUDA Workflow

The computation offloading process from the microcontroller to the GPU has 3 steps: a) preparation of the offloading (① memory allocation and code/data transfer ② in Figure 6-2), b) kernel launching (③) and c) retrieval of the generated results (④ and ⑤ memory deallocation). The preparation step requires sending the code that has to be executed in the GPU and transferring the data from the microcontroller memory to the GPU memory. Although typically the microcontroller and the GPU share the same physical memory, each device retains its own mappings and separate address spaces. Hence, even though the data are not physically transferred, there is still a certain amount of bookkeeping and some data transferring required, e.g., due to cache flushing for consistency reasons, which makes this process not immediate.

The offloading process goes beyond the SOR of the microcontroller (e.g., the DCLS) and involves the memory and/or DMA controllers (see Figure 6-1). Memory data and on-chip communication during the execution phase occur on the same resources as those used by the ASIL-D microcontroller and hence, are protected by specific ECCs, as indicated before. Thus, data movements during the preparation of the offloading process, kernel launching and retrieval are already protected by appropriate safety measures to comply with ASIL-D requirements.

However, computation inside the GPU lacks appropriate support for safety compliance by default. Therefore, some sort of safety measures needs to be deployed for the execution on the GPU, being those measures comparable to the ones of the microcontroller's DCLS cores. Appropriate safety levels can be achieved as follows (see Figure 6-3).

In lines 1-6 from listing 6.2 we can observe how the input data (float pointer *d_A*) and the output data (float pointer *d_C*) are allocated alongside their copies (named as *_redundant*). Next, in lines 8-11, we create a dedicated *cudaStream* per each kernel to avoid serialization and allow them to execute in parallel if they are able. Lines 13-15 show the memory copies of both the original input and his replica. In this case, since is not specified, the *NULL stream* is used, which means that the memory copies are serialized. Using the dedicated streams will benefit in GPUs that have more than one

```
1  //Input and Output data allocation on GPU
2  float *d_A, *d_C;
3  cudaMalloc(d_A, N*sizeof(float));
4
5  cudaMalloc(d_C, N*sizeof(float));
6
7
8  //Stream creation
9  cudaStream_t Streams[1];
10 cudaStreamCreate(&Streams[0]);
11
12
13 //Input data transfer to the GPU
14 cudaMemcpy(d_A, A, N*sizeof(float),
       cudaMemcpyHostToDevice);
15
16
17
18 //Kernel launch
19 kernel<<<NumBlocks, ThreadsPerBlock, 0,
       stream[0]>>>(d_A, d_C, N);
20
21
22
23 //Results transfer to the CPU
24 cudaMemcpy(C, d_C, N*sizeof(float),
       cudaMemcpyDeviceToHost);
25
26
27 //No comparison
```

Listing 6.1: Original CUDA code

```
1  //Input and Output data allocation on GPU
2  float *d_A, *d_C, *d_A_redundant, *d_C_redundant;
3  cudaMalloc(d_A, N*sizeof(float));
4  cudaMalloc(d_A_redundant, N*sizeof(float));
5  cudaMalloc(d_C, N*sizeof(float));
6  cudaMalloc(d_C_redundant, N*sizeof(float));
7
8  //Stream creation
9  cudaStream_t Streams[2];
10 cudaStreamCreate(&Streams[0]);
11 cudaStreamCreate(&Streams[1]);
12
13 //Input and Replicated input data transfer to the GPU
14 cudaMemcpy(d_A, A, N*sizeof(float),
       cudaMemcpyHostToDevice);
15 cudaMemcpy(d_A_redundant, A, N*sizeof(float),
       cudaMemcpyHostToDevice);
16
17 //Redundant Kernel launch
18 kernel<<<NumBlocks, ThreadsPerBlock, 0, stream[0]>>>(
       d_A, d_C);
19 kernel<<<NumBlocks, ThreadsPerBlock, 0, stream[1]>>>(
       d_A_redundant, d_C_redundant);
20
21 //Results and Redundant result transfer to the CPU
22 cudaMemcpy(C, d_C, N*sizeof(float),
       cudaMemcpyDeviceToHost);
23 cudaMemcpy(C_redundant, d_C_redundant, N*sizeof(float),
       cudaMemcpyDeviceToHost);
24 //Comparison of C and C_redundant
```

Listing 6.2: Applying Redundant Kernel Execution

Figure 6-3: Original CUDA code and Redundant Kernel execution, side by side

```
boolean epsilon_diff(float a, float b)
{
  return fabs(a - b) <= FLT_EPSILON;
}
```

Figure 6-4: Example of floating point comparison with a tolerance of FLT_EPSILON

copy engine which was not our case, although worth mentioning. With this, we arrive at the key point of the code, lines 17-19 correspond to the replicated kernel calls. Here we can observe that most of the parameters are the same, except for the dedicated streams (*stream*[0] and *stream*[1]), and the different pointers for input and output (*d_A* and *d_C*), respectively. Kernel calls in CUDA is non-blocking which means that the CPU or microcontroller can execute code after calling a kernel while the GPU is executing. On the other hand, the memory copies used are serialized since we are using again the *NULL stream*. Results will be copied back to the microcontroller in lines 21-23. Later on, the comparison of the results (not shown) will be performed to detect any difference in the execution of the redundant kernels.

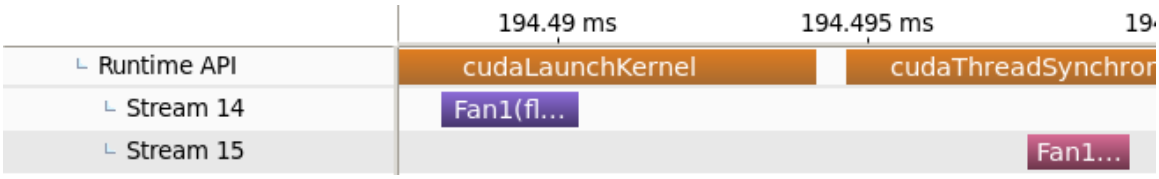
Note that, it would be possible avoiding the replication of read-only input data. However, then such data would not be replicated and some form of protection would be needed in the GPU (e.g., ECC) as a way to mitigate CCFs caused by errors in the non-replicated input data. Therefore, in this work, we assume that all input data for redundant kernels is replicated.

Note that, while we describe our approach for DMR (i.e., with two redundant kernels), it can be applied analogously to create fault-tolerant diverse redundant versions with TMR so that, in case of a mismatch of the results, the right result can be delivered by means of voting.

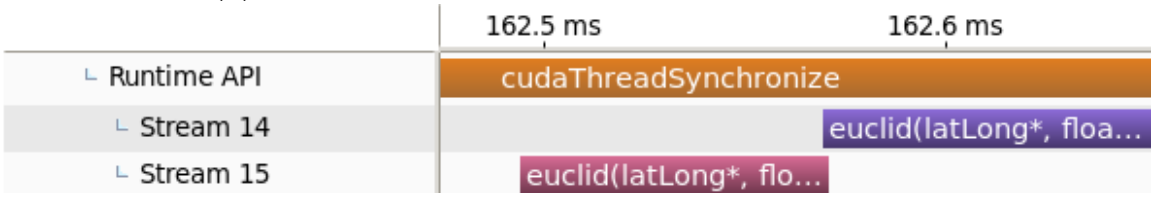
Result Comparison

In order to guarantee that kernel execution on the GPU is correct, a comparison must be made between the two results in the lockstep CPU. The comparison could be parallelized and performed (redundantly) on the GPU. However, our results, as we will see in the evaluation section, show that comparison time is below 1% for most cases, thus not making it worth the effort of porting the comparison to the GPU for most of them.

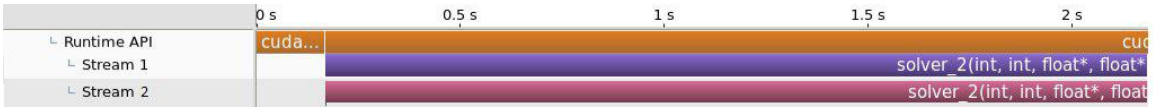
Differently to CPUs, which typically implement the IEEE 754 Floating Point (FP) standard [208], GPUs may not fully adhere to specific standards or may simply schedule work so that FP operations of the redundant kernels occur in a different order [209]. This may lead to different rounding choices, which, ultimately, cause fault-free results to be (slightly) different in practice. Hence, when implementing the result comparison in the CPU, we had to provide some flexibility to tolerate minor deviations with the code shown in Figure 6-4. Note that such code can be further used to tolerate actual errors whose practical impact can be deemed irrelevant.



(a) Short kernel, no overlapping at all (Gaussian application)



(b) Heavy kernel, small overlapping (NN application)



(c) Friendly kernel, large overlapping (Myocyte application)

Figure 6-5: Timelines of redundant executions of Rodinia benchmarks [7] extracted using the NVIDIA Visual Profiler.

6.2.3 Redundant Kernel Execution Patterns

While our solutions work for any kernel, some kernel characteristics make a given solution more appealing than others or even require methods to enable diverse redundant execution as we will see later. For the sake of the discussion, we refresh here the same classification shown in the previous contribution:

- **Short kernels**. Short kernels last too little to overlap because the offloading process of the second kernel takes longer than the execution of the first one, so the first kernel completes its execution before the second starts.

- **Heavy kernels**. While heavy kernels run long enough to overlap, any such kernel needs so many GPU resources that preclude the other kernel from starting its execution until the first one finishes (or is close to finishing). Therefore, the overlap is tiny – if any – and occurs when the first kernel is about to finish and starts releasing GPU resources.

- **Friendly kernels**. Friendly kernels run concurrently in the GPU since their duration is long enough and the demanded GPU resources low enough.

However, in this case, they can be observed on COTS GPUs with the NVIDIA Visual Profiler shown in Figure 6-5 for different Rodinia benchmarks [7]. The two bottom bars of each graph show the execution timespan for the redundant kernels, one in light red and the other in purple. Note that the x-axis scale changes across graphs. In particular, it is 2.2μs for the first graph (short kernel), 0.1 ms for the second graph (heavy kernel), and 2,200ms for the third graph (friendly kernel).

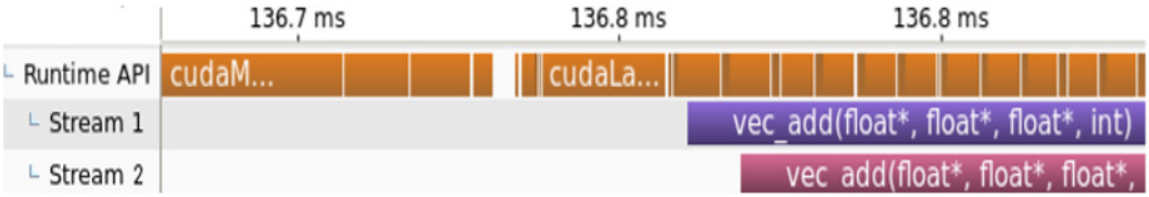


Figure 6-6: Staggered kernel execution of vector addition, obtained using NVIDIA's Visual Profiler

Note that the classification of kernels depends on platform-specific characteristics, such as for instance, the amount of GPU resources, which may make a kernel heavy or friendly depending on whether those resources suffice to execute both redundant kernels concurrently. In the case of automotive applications, they typically have fixed input data sizes since input data always comes from the same sensor (e.g., images from a camera). This allows kernel classification to be performed a priori statically.

6.2.4 Staggering creation

The kernel invocation performs the offloading of the application. Data sets transfer, explicitly initiated by the programmer, must be performed for both kernels before starting any of them to minimize the risk of having short kernels. In other words, the slack between the initiation of both kernels is kept as low as possible. Kernels have small constant and implicit parameters for each kernel launch. Those are set by a configuration call, which performs the arguments passing, followed by a `CUDALaunch` operation performed by the CUDA Runtime for each kernel called. The CUDA Runtime performs all those operations serially, since it is executed on the CPU, thus serializing the launch of the two kernels with some delay (*slack time*) between the two concurrent executions, as illustrated in Figure 6-6. The figure, generated with the NVIDIA Visual Profiler shows the serialization of the CUDA calls (top yellow bar) and how the kernel copies start with some slack in between (bottom purple and light red bars). Therefore, a staggered execution start is guaranteed by construction. Note that, although identical kernels are expected to progress almost identically – thus preserving the staggering, this cannot be guaranteed in general due to the lack of controllability in COTS GPUs.

6.2.5 SM sharing among Kernels

Our software-only solution assumes that SMs cannot be shared across thread blocks from different kernels simultaneously. Instead, we assume that when a kernel starts running, it uses a number of SMs without interruptions (i.e., SMs are only released once they are not needed anymore), and during such a period no other kernel can use those SMs. However, as shown in [210], the scheduling policy for some NVIDIA GPUs may allow, in some situations, sharing SMs across kernels if intra-SM resources allow it. Solutions avoiding this behavior rely on some hardware support (e.g., modifying the kernel scheduler) exist in the literature [211], and the ones presented in this paper

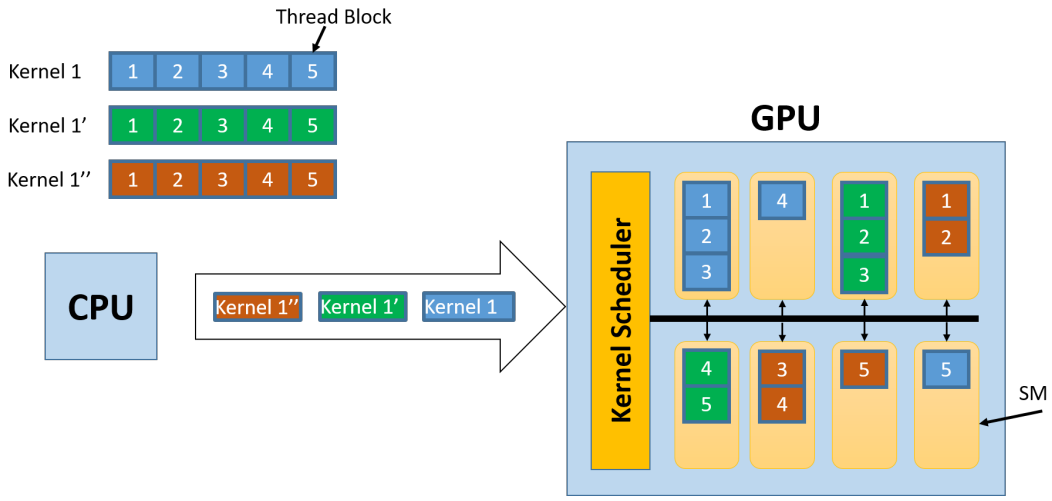


Figure 6-7: Spatial and time redundancy in a GPU execution. Three redundant kernels that contain 5 thread blocks each are scheduled in an 8 SM GPU.

would also solve this issue. However, our target in this section is to achieve diverse redundancy by software-only means. SM sharing across threads can be effectively avoided by building on *persistent threads* [212, 213, 214], where each SM can only be used exclusively by one kernel. Persistent threads bring some lack of flexibility since they impose a behavior similar to that of our HALF hardware solution, plus some overheads for the allocation and management of those threads (e.g., due to polling the GPU on the CPU side to detect the end of the kernel execution). Since, in general, SMs are rarely shared in practice, we avoid using persistent threads in our work and hence, we apply the same software architecture as for the hardware solutions, which includes: data replication, redundant kernel launching with a different stream, and result in comparison at the CPU side. A schematic of how a diverse TMR execution would map into the SMs of the GPU when applying our solution can be seen in Figure 6-7.

6.2.6 Achieving Diverse Redundancy

In the context of COTS GPUs, diverse (independent) redundancy can be achieved for two or more kernels if the following requirements are met:

Req1: Replicated computations do not use the same functional unit block (FUB) to execute the same code on the same data.

Req2: Functionally identical computation units (e.g., CUDA cores) produce different error manifestations in the presence of a single fault affecting several of those computation units.

Req3: Unique resources (e.g., non-replicated buses or interfaces) implement intrinsic diverse redundancy (e.g., ECCs).

Req1 guarantees that permanent faults escaping testing or due to aging cannot cause the same error in the replicated executions. Achieving *Req1* requires having a kernel scheduling policy controlling which FUBs (SMs in NVIDIA GPUs) each kernel will use. This is currently done by the kernel scheduler whose policy is often unknown, as we discussed in the previous chapter, since some GPU vendors, such as NVIDIA, do not release this information [107]. In our evaluation, we show empirically that *Req1* is naturally achieved for those kernels that can run simultaneously.

Note that kernel classification is platform and data-size dependent, so, for instance, a heavy kernel on a particular GPU could be a friendly kernel on another GPU with more resources. Instead, most automotive applications have a fixed data size since the input data always comes from the same sensor (e.g., images from a camera). This reduces the data size variability for most of the kernels. Short and heavy kernels challenge the achievement of diversity.

Solution for Short kernels. Short kernels may be executed directly in the ASIL-D (lockstep) microcontroller, since they do not demand huge computation power. Executing them in the ASIL-D microcontroller guarantees *Req1*, although it must be assessed whether their likely larger execution time in the microcontroller still adheres to the corresponding FTTI. In general, this is the case since short kernels need to take at most a few μs of execution in the GPU not to overlap, and even if they run 100x slower in the CPU, they will stay typically below 1 ms only, which is a very low latency for functions executing typically every few tens of or hundreds of ms at most.

Solution for Heavy kernels. Redundant copies for heavy kernels are executed sequentially due to a lack of resources to run them concurrently, potentially using the same resources for the same computations of both copies, thus defeating diversity. A simple solution for heavy kernels could be relegating them to execute on the ASIL-D microcontroller, as for short kernels. However, these kernels' execution time (heavy) can be arbitrarily large (e.g., tens or hundreds of milliseconds). Thus, slowdowns of 1 or 2 orders of magnitude would easily violate safety requirements for those systems.

Since heavy kernels run a number of threads in parallel, by reorganizing computations, some parallel threads can be serialized (e.g., splitting the kernel into multiple kernels or simply rearranging threads) so that the amount of resources is reduced sufficiently to allow two redundant copies to run concurrently, thus becoming the heavy kernel one or several friendly kernels. Such a solution is always feasible due to the nature of thread execution on GPUs because coherence across parallel threads is not controlled, so any sequential order of the operations across threads is semantically correct. Hence, by serializing parallel threads in any way semantics are preserved. In the next section, we introduce a new method to transform heavy kernels into friendly ones, thus, enabling them to be executed safely and timely, with diverse redundancy, on the GPU.

Solution for Friendly kernels: In the case of friendly kernels, since they can execute concurrently, our software modifications are enough to execute them in different functional units (SMs) by launching them using different CUDA Streams. Since SMs can only execute threads from the same CUDA Stream, the two redundant executions will use different SMs (FUB), thus guaranteeing *Req1*.

As explained before, due to the kernel launches' serializations, an initial staggering between the redundant executions is achieved naturally (see Section 6.2.4). Thus, obtaining a diverse redundant execution in a COTS GPU. Later, in the evaluation section, we will show evidence of this initial staggering. Note that, while such staggering is normally preserved due to the regular and highly deterministic execution of kernels on a GPU, it cannot be guaranteed a priori, and we can only assess it a posteriori to some extent.

Req2 guarantees that a single fault does not lead to a CCF despite redundant computations occurring on different resources. Hence, *Req2* imposes the use of some additional form of diversity. For instance, a usual technique to achieve diversity in functionally identical computational units is using staggered lockstep execution, so that redundant computations, apart from being performed on physically different resources, are also performed at different times. Interestingly, staggered execution, which we discuss in detail before (see Section 6.2.4), can be implicitly achieved with COTS GPUs, and this provides them with protection against some of the most relevant sources of CCFs such as high-voltage pulses or voltage droops. Providing independent redundancy beyond the aforementioned CCFs could also be achieved using some of the techniques (e.g., layout diversity) already employed to achieve diversity in front of the most relevant CCFs in ASIL-D lockstep processors [1, 74]. However, this is not yet provided in COTS GPUs.

Req3 guarantees that diverse redundancy is achieved in unique resources. In general, those resources include interconnects and interfaces, where data and control signals transmitted can be properly protected with ECCs or CRCs. In the case of storage, either it is also ECC-protected or contents are stored redundantly. Finally, combinational logic is usually hard to protect with any form of ECC. For instance, this is the case with the thread scheduler. However, by making redundant kernels run simultaneously in different SMs, each kernel uses different thread schedulers. Moreover, the Kernel scheduler, used to send thread blocks to SMs, also operates with some degree of diversity since the same thread block in redundant copies is dispatched to different SMs. However, diverse redundancy requires physical replication in general. As detailed before, this is practically in place for computation resources in a GPU, but other components, such as the kernel scheduler, may lack such support. In general, whether unique resources adhere to specific ASIL requirements cannot be assessed by industrial users due to the lack of detailed documentation, and observability or controllability means. Thus, whether the failure probability of those components can be deemed as residual risk cannot be assessed directly with software-only approaches. Still, our work shows that most GPU resources can be leveraged by software means to ensure diverse redundancy.

6.2.7 Heavy-to-Friendly Kernel Reshaping Protocol

Next, we present our protocol to transform *heavy* kernels into *friendly* ones systematically. Our approach has been tailored to work with kernels, not using shared memory for inter-thread communication since this is the type of *heavy* kernels found in our evaluation. If shared memory is used for inter-thread communication, this would need to be managed manually. Extending our protocol to these scenarios is part of our future work but outside this Thesis's scope. We first describe the operators on which our protocol builds, then the protocol itself, formal validation of its effectiveness, and finally, discuss its complexity. In simple words, this protocol modifies each redundant kernel to fit in half of the SMs of a given GPU by reducing its parallelism and serializing some threads by combining them.

Thread Coarsening and Block Division Operators

Each redundant kernel uses a certain amount of resources (e.g., registers, shared memory). Whenever the requirements of the combined kernels in terms of resources exceed those available in the GPU (*heavy* kernels), the scheduler prevents them from achieving concurrent kernel execution between the head and shadow kernels. This proposal is based on modifying these kernels' resource requirements, which may increase their execution time, but allows them to execute concurrently. Our protocol uses two techniques: *Thread Coarsening* and *Block Division*, which we introduce next.

Thread coarsening: Thread coarsening is the process of increasing the amount of work performed by each thread. This technique can cause some potential performance improvements: (1) Higher instruction-level parallelism (ILP) [215] by increasing the number of instructions per thread; (2) More efficient DRAM memory bandwidth utilization by reducing the total number of memory-access instructions [216], for those data that more than one thread would otherwise fetch; and (3) Reduction in the number of computing instructions due to redundant computations across threads [217]. This technique, however, can also have several negative effects: (1) Reduction of the total amount of parallelism by reducing the number of threads, which can reduce the performance if there is not enough amount of work (threads) to keep the rest of the GPU busy; (2) Increase of the number of registers per thread; and (3) Worse memory access patterns since neighboring threads may end up accessing non-contiguous memory as stated in [218], which has detrimental effects on cache behavior.

Authors in [219] applied automatic thread Coarsening at compile time to GPU kernels to achieve a 1.3x speedup in a subset of Rodinia benchmarks [7, 173]. However, as mentioned before, thread coarsening may lead to increased cache pressure, thus resulting in performance degradation. Moreover, in the extreme degenerate case, thread coarsening would lead to sequential execution by a single thread, defeating the purpose of using parallel hardware such as GPUs. Therefore, while thread coarsening may produce some performance gains, it is generally not used when the only concern is performance. However, our primary goal is not to increase performance but safety by allowing redundant heavy threads to run concurrently (i.e., becoming friendly).

Block division: Block Division consists of splitting thread blocks into smaller ones, i.e., using fewer threads, while the total number of threads remains the same. This technique can be used when the requirements per thread block exceed SM's resources, preventing the entire kernel from being executed.

This technique is particularly useful to reduce cache pressure and register requirements per thread block. Obviously, this is the remedy technique to use when thread coarsening leads to over-using some resources, mainly registers over-use, since the lack of registers is the only limitation preventing the execution of a thread block. Other shared resources, such as cache space, can lead to lower performance if over-used but would not prevent a thread block's execution. Another limiting resource is the shared memory, a scratchpad memory used to allow threads in the same block to communicate and reduce the DRAM bandwidth. We choose not to include it as part of our protocol since not using it will not impede the execution of a thread block and can slow down execution. As said before, performance is not our primary goal.

Protocol Step by Step

Figure 6-8 details the protocol followed to transform heavy kernels into friendly ones. It builds upon applying thread coarsening and block division iteratively until the resulting thread block does not exceed the number of registers available in an SM, and each kernel uses at most half of the resources available in the GPU. In particular, starting from a kernel whose thread blocks do not exceed the total number of registers available in an SM (so it is schedulable), but uses more than half of the registers of the entire GPU (so it prevents its shadow kernel from running concurrently), the protocol does the following: (1) merges threads so that repeated data fetches and computations can be removed, thus reducing the total number of registers required, although the number of registers per thread increases. (2) Divides thread blocks to decrease the total number of registers per block. Note that, as we discuss later, this process necessarily decreases the number of registers per thread block in each iteration so that friendliness is achieved eventually.

The protocol first initializes the platform and kernel-dependent variables and checks if the redundant kernels can already be executed concurrently or not (lines 1-5). If not, we compute the Thread Coarsening Factor (TCF) by dividing the current number of SMs used (= the number of thread blocks) by the target number of SMs we want to use (up to half of those available in the GPU), as shown in line 6. Next, we apply thread Coarsening to reduce the number of threads by the factor computed (TCF), see line 7. We update the number of SMs used (line 8), which now will be less or equal to half of the SMs, and the number of registers required per thread block (line 9). At this point (line 10), if the number of registers per thread block is lower or equal to half of the SM's registers, kernels can execute concurrently (lines 14-15).

If the concurrent execution is not possible yet, then we require more registers per SM than allowed. Thus, Block Division must be used. First, we compute the Block Division Factor (BDF) based on the current register requirements and the registers available (line 11), and then we apply Block Division accordingly (line 12). This

1: $\#SM_{available} \leftarrow \lfloor TotalSM/2 \rfloor$
2: $\#Register_{available} \leftarrow \lfloor Register_{SM}/2 \rfloor$
3: $\#SM_{Used} \leftarrow \#ThreadBlocks$
4: $\#Register_{UsedpBlock} \leftarrow Y$
5: **while** Kernels not concurrent **do**
6: $TCF \leftarrow \lceil \frac{\#SM_{used}}{\#SM_{avail}} \rceil$
7: $ApplyThreadCoarsening(TCF)$
8: Recompute($\#SM_{Used}$), $1 \leq \#SM_{Used} \leq \#SM_{avail}$
9: Recompute($\#Register_{UsedpBlock}$)
10: **if** ($\#Register_{UsedpBlock} > \#Register_{available}$) **then**
11: $BDF \leftarrow \lceil \frac{\#Register_{UsedpBlock}}{\#Register_{available}} \rceil$
12: $ApplyBlockDivision(BDF)$
13: Recompute($\#SM_{Used}$)
14: **else**
15: Done (Kernels concurrent)
16: **end if**
17: **end while**

Figure 6-8: Proposed protocol, where TCF = Thread Coarsening Factor and BDF = Block Division Factor

second step can increase the number of SMs used (line 13), which will require performing the thread coarsening technique again. However, as shown next, the registers of the thread block are reduced with respect to the previous iteration, guaranteeing the convergence of the process.

Formal validation

We validate our protocol showing that a kernel can be made to fit in a single SM, so that if the number of SMs per kernel is higher, the protocol can converge faster. Let us consider a kernel containing B blocks, where each block has T threads and each thread uses R registers. An SM of the GPU contains S registers being $S \geq R$. Then, the number of registers used is the total number of threads ($B \cdot T$) multiplied by the registers per thread (R), so $B \cdot T \cdot R$

If $B \cdot T \cdot R < S$, all thread blocks can be allocated in a single SM. Otherwise, we would use thread Coarsening with a TCF (α), where $\alpha \in \mathbb{N}$ and $\alpha \geq 2$, to reduce the total number of registers. Now the total registers used are: $\frac{B \cdot T \cdot Y}{\alpha}$ where Y is the registers used per thread after applying Thread Coarsening.

Due to register reuse, Y can be at most $\alpha \cdot R$ (worst case), but it will be typically lower. In fact, appropriate compilation constraints may make merged threads run purely sequentially, thus reusing the same output registers for each instruction, thus ensuring that $Y < \alpha \cdot R$. Therefore:

$$\frac{B \cdot T \cdot Y}{\alpha} < \frac{B \cdot T \cdot (\alpha \cdot R)}{\alpha} = B \cdot T \cdot R$$

Since $Y < \alpha \cdot R$, in the first step, we reduce the total number of threads, but also the total number of registers used. Now let us apply block division by a factor β, where $\beta > 1$ and $\beta \in \mathbb{Z}$. Now, the total number of registers is:

$$\frac{\beta \cdot B \cdot T \cdot Y}{\beta \cdot \alpha} = \frac{B \cdot T \cdot Y}{\alpha}$$

As seen, Block division does not affect the total number of registers used, only the registers per block ($\frac{T}{\beta} \cdot Y$). Therefore, at each iteration of our loop, which includes both Thread Coarsening and Block division, we reduce the total register requirements. Considering that the kernel could be executed in the CPU, where the register file is smaller than the one of the SM, the extreme case of just using one thread would be valid. Thus, our protocol will always allow two kernels to be executed concurrently in a GPU with at least two SMs (one for each redundant kernel).

Complexity

The protocol is guaranteed to terminate, as explained above, since, in every iteration, we reduce the total number of threads of the kernels and, eventually, we will reach the case with one thread per kernel, which is always a functionally valid option. However, such a degenerate case may be far from being the optimal solution in terms of performance.

The number of iterations required to reach a solution where both threads can be executed concurrently depends on the particular kernel being considered. The factors affecting the number of iterations relate to (1) the requirements of the initial kernels (number of SMs used, the total number of threads, number of thread blocks, ...), and (2) the resources available (number of SMs, the maximum number of registers per block ...).

Generally, kernels with higher requirements per block will need more iterations since thread coarsening will easily make kernels exceed the resources per block limits and will take more iterations to find a valid pair of values for TCF and BDF. To reduce the number of iterations, one could be more aggressive when calculating the two factors by selecting higher values for both factors. Once a valid solution is found, it would be a matter of checking intermediate values not evaluated to search for a potentially better solution with a higher number of threads. Instead, kernels with small requirements will take fewer iterations to find a valid solution. In fact, as we see in the evaluation section, in most cases, one iteration of the loop is enough to achieve a valid solution.

Formally, after adjusting the number of SMs used to be at most half of those available in the GPU, the number of registers required per SM is:

$$\#Register_{UsedpBlock} = R \cdot \left\lceil \frac{\#SM_{used}}{\#SM_{avail}} \right\rceil$$

The worst case occurs when R matches the total number of registers per SM, thus leading to the largest register per SM exceedance after thread coarsening. As explained before, in each iteration, the total number of registers per thread decreases at least by 1 given that $Y < \alpha \cdot R$. Thus, the worst case would be to move from $\#register_{UsedpBlock}$ to R in $\#Register_{UsedpBlock} - R$ iterations assuming that the number of SMs used is fixed. In practice, since the number of SMs that can be used to have

a friendly kernel is between $\#SM_{avail}/2$ and $\#SM_{avail}/4$ SMs[1], the number of steps could almost double if the worst case occurs (i.e., if we end up using $\#SM_{avail}/4+1$ SMs), thus leading to up to $2 \cdot (\#Register_{UsedpBlock} - R)$ steps.

6.2.8 Diversity Limitations: from DMR to TMR

Both the software and hardware contributions enable a diverse redundant execution can still be vulnerable to two sources of CCF; The first type relates to physical layout effects that may make some specific mask patterns prone to faults. Since all SMs are, in general, identical, errors induced by layout effects may produce the same error in all redundant copies despite occurring in different SMs at different times. Note, however, that having these effects manifesting for the first time almost simultaneously and causing *exactly* the same effects in two different physical locations in the chip is unlikely. This relates to effects like process variation, both random and systematic, that may affect different locations differently. In the case of TMR, having three such almost simultaneous first manifestations is even more unlikely to occur. Hence, while this CCF is not avoided completely, TMR is expected to be much less exposed to it than DMR.

The second source of CCFs relates to the use of non-redundant components such as the kernel scheduler, which is unique in GPUs. The serial offloading of the kernels brings some diversity, as well as the fact that the second kernel may find a different scheduler state to that of the first kernel, thus reducing the chances of a fault causing identical errors in both kernels for DMR. In the case of TMR, as for the case of layout effects, the use of 3 redundant copies instead of 2, further decreasing the chances of this type of CCF. However, it is not completely avoided and, as in the case of layout effects, hardware support is convenient to guarantee that CCFs are avoided.

6.3 Experimental Validation

In this section, we evaluate different aspects of the software-only solution discussed in the previous section. We start by showing real slack measurements observed in a real COTS GPU. Next, we show results of executing Rodinia [7, 173] benchmarks with our approach for DMR and TMR. We follow with two examples of the usage of the transforming protocol, in two Rodinia *heavy* benchmarks transforming them to *friendly*. Later, we discuss the results of a fault injection campaign showing the fault-detection capabilities of our solution. Finally, we perform a comparison side by side of the software-only solution and the hardware one described in the previous chapter 5.

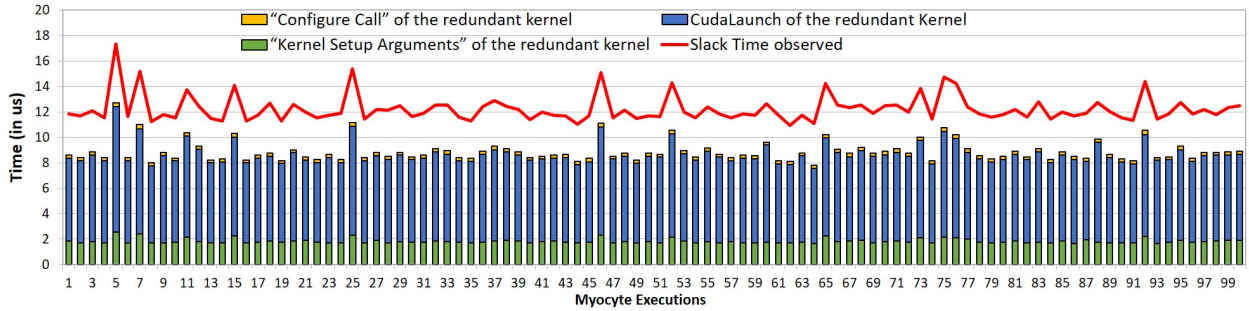


Figure 6-9: Slack observed and subprocedures of the kernel launching for the consecutive executions of the Myocyte kernel.

6.3.1 Slack Measurements Results on a COTS GPU

The kernel offloading process executes the following routines: *Configure Call, Kernel Setup Arguments, and CUDALaunch.* In this experiment, we modify the myocyte benchmark (part of the Rodinia Benchmark Suite) to make its kernels redundant and execute it 100 times using the setup explained in 3.6.1. We use the NVIDIA profiler and obtain the results shown in Figure 6-9. The thick dark line corresponds to the time elapsed between the start time of both kernels, the original and redundant ones. Stacked bars show the individual contribution of each one of the kernel offloading routines for the second kernel, which are executed serially on the CPU. There is some code between those routines (CUDA calls), whose execution time covers the gap between the stacked bars and the total execution time (thick line). However, the NVIDIA profiler does not provide information about this non-CUDA code.

Kernels are launched on the GPU only after these CUDA calls and surrounding code are executed on the CPU. The dominant routine (CUDALaunch) takes around 6μs (if not more), and its execution time is independent of the characteristics of the kernel to be launched. This guarantees that there always be such staggering across kernels, although it will be typically higher due to the remaining code executed for offloading purposes. Hence, the staggering across redundant kernels is guaranteed to exist. Since this behavior is not specific to this GPU but, instead, is intrinsic to the CPU-GPU relation, we can expect similar behavior for different GPUs and even other runtimes (e.g., OpenCL).

6.3.2 COTS GPU Results for diverse DMR

Since the software-only solution does not require any hardware modification, we can directly evaluate it on the COTS platform. We have characterized the different parts of the redundant execution process in the Rodinia benchmarks as can be seen in

[1]Our mechanism decreases $\#SM_{used}$ by an integer factor to not exceed $\#SM_{avail}/2$. If the value obtained was not exceeding $\#SM_{avail}/4$, then the factor used could be doubled without exceeding $\#SM_{avail}/2$.

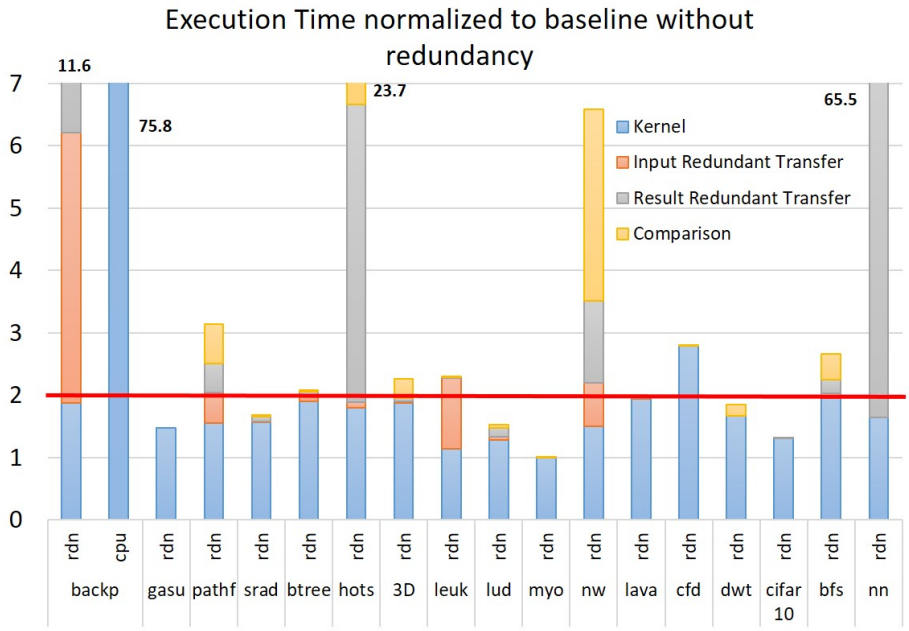


Figure 6-10: Redundant execution times characterization for the Rodinia benchmark suite. Backprop and gaussian are short kernels; *nn* and *bfs* are heavy kernels; and the rest are friendly kernels.

Figure 6-10[2]. Execution times are normalized per each benchmark where "1" corresponds to the original benchmark's normalized execution time without redundancy. More precisely, we have characterized the execution into kernel execution, the redundant transfers, and the comparison phase, thus, showing all the overheads created by our strategy. The backprop benchmark also includes a CPU-only version, which we also include (2nd leftmost bar), and discussed next.

Short Kernels (backprop and gaussian): The backprop benchmark (the leftmost one) is a short kernel. As shown, the redundant version of this benchmark leads to an execution time above 2x the execution time without redundancy since redundant threads do not overlap. Due to the short duration of the kernel, the relative impact of needing redundant data transfers and having to compare results is huge w.r.t. GPU execution without redundancy. The CPU version of this benchmark, which is included in the benchmark suite, has an execution time 75.8x higher than the one for the GPU baseline version in our setup. Despite being huge in relative terms, such a slowdown is low in absolute terms and, hence, affordable. In the case of *gaussian*, the overhead due to running it redundantly is below 2x since the execution time includes both kernels launching in the CPU and kernel execution in the GPU. Hence, while the kernel execution of both redundant copies does not overlap, the kernel launching of the second kernel overlaps with the kernel execution of the first

[2]In this experiment, we changed the inputs of the bfs benchmark to obtain another *heavy* kernel for the later evaluation of the heavy-to-friendly kernel transformation protocol.

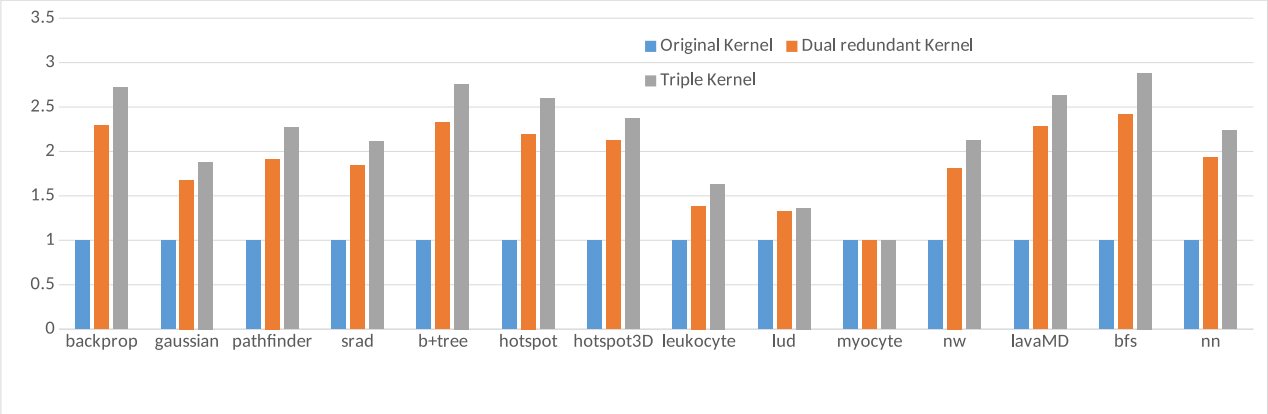


Figure 6-11: Execution time of diverse DMR and TMR normalized w.r.t. non-redundant execution.

one and the kernel launching is implicitly included in "Kernel" time in the Figure. Therefore, we categorize it as a *short* kernel despite having an execution time below 2x because there is no kernel (understood as GPU computation) execution overlapping.

Friendly Kernels: The overlap of these kernels is large, and thus, the overall execution time to run both redundant kernels is below 2x the execution time of the non-redundant kernel. Redundant execution for friendly kernels causes small overheads in the GPU computation part. Instead, overheads due to comparison and data transfers are benchmark-dependent. Those overheads could be reduced by performing comparisons redundantly in the GPU to reduce the amount of data to be transferred back to the CPU and to parallelize the comparison. However, while this would be possible, it has not been explored explicitly in this work.

Heavy Kernels (nn and bfs): Since the redundant kernels for these benchmarks barely overlap, the impact of running them redundantly is above 2x in terms of execution time. However, the protocol introduced before allows converting these heavy kernels into friendly ones. Their friendly versions are evaluated in section **??**.

6.3.3 COTS GPU Results for diverse TMR

We implement TMR by manually tripling memory allocations, data transfers and kernel offloading and performing the output comparison back in the CPU side, similarly to the DMR. The objective of our evaluation is to assess the execution time impact of implementing software-only diverse TMR in GPUs.

For this experiment, we use the other GPU of our setup, the NVIDIA 1080Ti [172] detailed in chapter 3.6.3. We took this decision since we were afraid that the amount of available memory of the NVIDIA 1050Ti (4GB) could limit the concurrency of three kernel instances executing at the same time. Instead, the NVIDIA 1080Ti has a total of 11GB available, more than enough based on the results obtained.

Results are shown in Figure 6-11. In this Figure, we show the timings of end-to-end execution including both CPU and GPU execution as well as data transfers. As we can observe, TMR increases execution time w.r.t. DMR, as expected. However,

Default (heavy) configuration				
Benchmark	Thread Blocks	Threads x Block	Total Threads	Registers Block
bfs	[1954 , 1]	[512 , 1]	1,000,448	K1:8,192 K2:7,680
nn	[2560 , 1]	[256 , 1]	655,360	3,840
Final (friendly) configuration				
bfs	[3 , 1]	[512 , 1]	1,536	K1:8,192 K2:7,680
nn	[2 , 1]	[256 , 1]	768	3,840

Table 6.1: Default configuration of the applications that produces *heavy* kernels on the NVIDIA GTX 1050 Ti, and final friendly configuration obtained using the Heavy-to-friendly protocol

while DMR causes a nearly-linear slowdown w.r.t. the baseline execution time, TMR generally leads to execution times clearly below 3X w.r.t. the non-redundant case. Further investigation reveals that some relatively low contention causes a large impact, and additional contention has a lower impact mostly due to further serialization of the execution.

6.3.4 Evaluation of the Heavy-to-friendly Protocol

As seen, some of the benchmarks turn out to be either friendly or short, and we only observe one *heavy* benchmark (`nn`) with the 1050 Ti setup. Thus, in order to test the protocol with more workloads, we have modified the input variables of the benchmark `bfs`, which changes the number of resources used by the benchmark to make it also *heavy*.

The default grid configurations of the two applications are shown in the Table 6.1 (top rows) (information obtained through NVIDIA's profiler *nvprof*).

To have a complete evaluation, we have created a small script for the *bfs* benchmark. This script (written in *bash*) tries different argument configurations and launches the *bfs* application with the NVIDIA profiler (nvprof). Logs of the results are stored and later parsed to identify the configurations that serialized the kernel execution, which is a characteristic expected for heavy kernels (necessary but not sufficient). These configurations are later inspected with the visual profiler to (1) check that kernels are serialized and (2) the serialization is because they are *heavy* kernels, and not *short*.

The benchmark `bfs` contains two kernels with the same grid and block configuration, one using 16 registers per thread and the other 15. Instead, `nn` contains only one kernel that uses 15 registers per thread. Both applications use a high number of thread blocks, which results in a big TCF, 653 for `bfs` and 854 for `nn`. We obtain these factors by dividing the number of thread blocks by half of the GPU's SMs (thus, three SMs for each redundant kernel), line 6 in Figure 6-8.

To ease the application of the protocol, we adapt the code to enable Thread Coarsening and Block Division as follows:

1. Add a new parameter to the kernel function, the TCF.

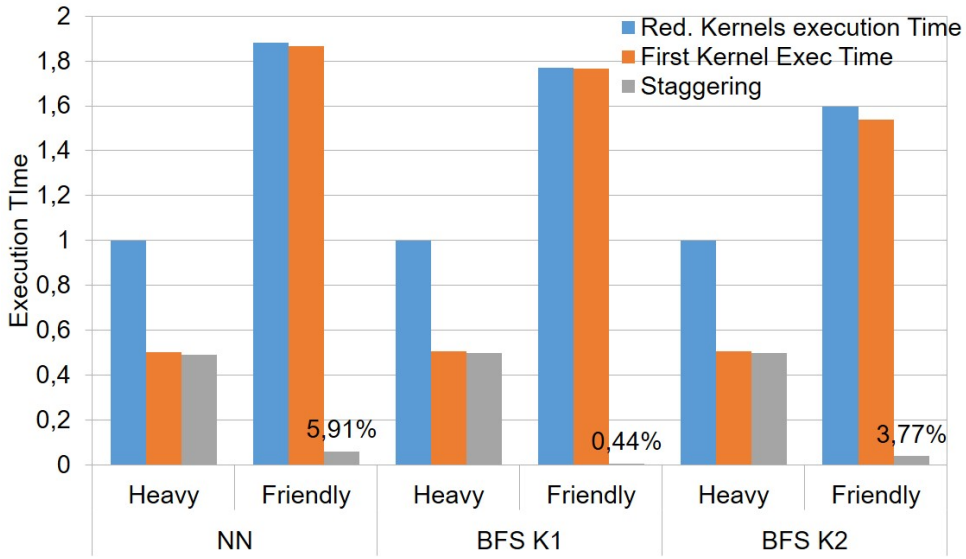


Figure 6-12: Normalized execution times of the total redundant execution (grey), the execution time of the first kernel launched (white) and the initial staggering (black) between the redundant kernels w.r.t. total kernel redundant execution on the default configuration (*heavy*) of each benchmark.

2. Apply Thread Coarsening to the kernel code. In order to facilitate programming, we add an outer loop that iterates TCF times. However, while this automates the application for Thread Coarsening, this solution may not benefit from some optimizations. For example, memory instructions from originally different threads, accessing the same data, will not be performed closely because of the loop's body, which may lose potential cache hits. In order to improve performance, we recommend using the compiler technique in [219] whenever possible, although it makes it less straightforward to apply Thread Coarsening.
3. Modify the kernel launching, in the CPU code, to launch the kernel with the grid according to the Thread Coarsening and Block Division factors.

Using the calculated TCF, the benchmarks are launched. As expected, the execution of the kernels finishes correctly, and the execution of the redundant kernels overlaps. In particular, we measure the execution time of the first launched kernel (white bar), the total kernel execution time (from the start of the first until the completion of the second, grey bar), and the staggering time between them at the launching (black bar). Results for the two benchmarks are shown in Figure 6-12 normalized w.r.t. the execution time of the redundant kernels on the baseline configuration (*heavy*). The results shown for each benchmark are the average of 500 executions in the same COTS GPU used before, an NVIDIA GTX 1050 Ti.

As shown, in all heavy configurations of both benchmarks, the first kernel takes half of the total execution time, matching with the staggering time, meaning that both redundant kernels take a very similar amount of time to execute and are fully serialized. In the case of the friendly versions of the benchmarks, we observe that the first and total execution times nearly match, thus meaning that both redundant

kernels finish virtually at the same time. Also, the fact that the staggering time is tiny indicates that both of them start almost simultaneously, thus overlapping their execution completely, with just some little staggering.

Note, however, that making kernels friendly impacts total execution time, which grows by a factor of 1.9x for `nn` and 1.7x and 1.6x for `bfs` kernels. While this effect is undesirable, it is the price to pay to guarantee diverse redundancy on a COTS GPU without explicit lockstep support and only by software means. Such performance loss relates to (1) worse cache access patterns that lead to an increased miss rate and, thus, less efficient DRAM bandwidth utilization, whose access latency cannot be effectively hidden. And (2) the loss of parallelism since we reduce the number of parallel threads per kernel. For the sake of completeness, Table 6.1 (bottom rows) shows the final kernel configurations for both benchmarks after applying the protocol to make them friendly.

6.3.5 Fault-detection capabilities evaluation using fault injection

To test the fault tolerance capabilities of our solutions, we use the NVBitFi [180, 181] a framework that is built on top of NVidia Binary Instrumentation Tool (NVBit) [182] that performs error injection campaigns for GPU application resilience evaluation. NVBitFI injects errors into the destination register values of a dynamic thread instruction by instrumenting instructions after they are executed. Only one injection is done per run. A dynamic instruction is selected randomly from all dynamic kernels of a program for error injection. This tool allows four different instructions to inject errors:

- Instructions that write to general-purpose registers
- Single-precision floating-point instructions
- Double-precision floating-point instructions
- Load instructions.

Additionally, the tool supports four different fault models (errors that can be injected):

- Single bit-flip: one bit-flip in one register in one thread
- Double bit-flip: bit-flips in two adjacent bits in one register in one thread
- Random value: random value in one register in one thread
- Zero value: zero out the value of one register in one thread

We have selected one of the Rodinia benchmarks (backprop) to perform the injections. In particular, this application contains two kernels. For each fault model available and instruction type, we perform 10k injections on the baseline application

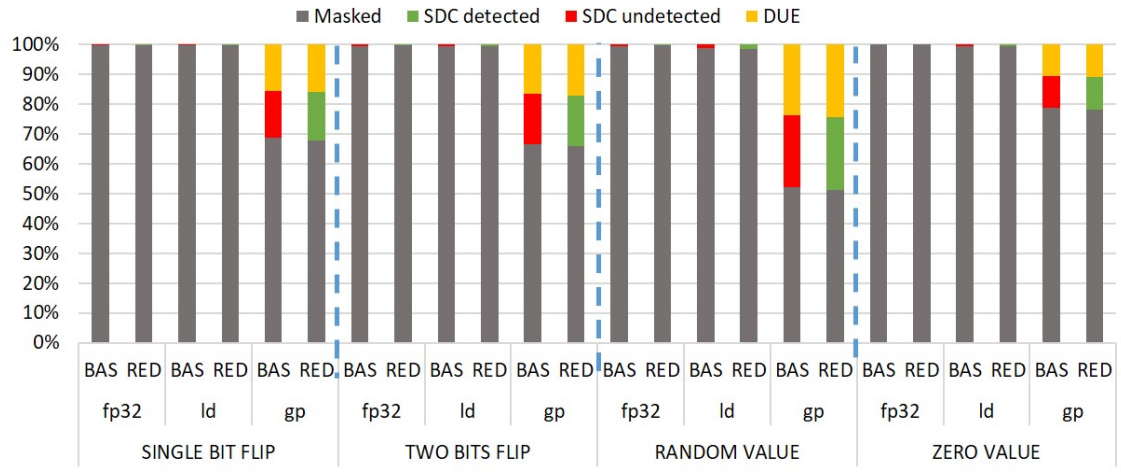


Figure 6-13: Fault injection results for each fault model. **Masked**: The output of the execution was correct. **SDC detected**: An error was found by the detection mechanism and reflected in the output of the application. **SDC undetected**: A mismatch in the output was found which was not detected. **DUE**: A detected error prevented finishing the execution.

and 10k for the application with our redundant kernel software approach. Since this application only uses single-precision floats, double precision injections have been discarded. For each execution, the output is analyzed and compared against a golden output (an output from an error-free execution).

Results from the fault injection can be observed in Figure 6-13. We can see three pairs of columns for each fault model, each corresponding to a different instruction type targeted by the fault injection tool. From left to right, single-precision (32-bits) floating-point instructions (fp32), load instructions (ld), and instructions that write into general-purpose registers(gp)[3]. For each pair, we have results on the baseline application on the left (*BAS*) and the application with our software-only redundant strategy on the right (*RED*).

Results follow similar behavior for all fault models. Baseline applications reported no Silent Data Corruption (SDC) for floating-point injections or load instructions. Instead, injections on general-purpose registers experienced a slight percentage of SDCs undetected from 10.7% to 24% on the baseline approach, which are translated into SDC detected by our fault detection mechanism (comparison of kernel results). Detected Uncorrectable Errors (DUE) only have significant importance for general-purpose registers and have similar values for the baseline and redundant approaches. The tool detects them by using the *dmesg* command, but the application could also detect them if the appropriate CUDA error handling procedures are called (e.g., *cudaGetLastError()*) after the kernel calls. In summary, we observe that our redundant diversity scheme provides protection against the single-point faults evaluated.

[3]64-bit (double) floating point instructions are not used by the application targeted, so we did not perform a fault injection campaign aiming at them.

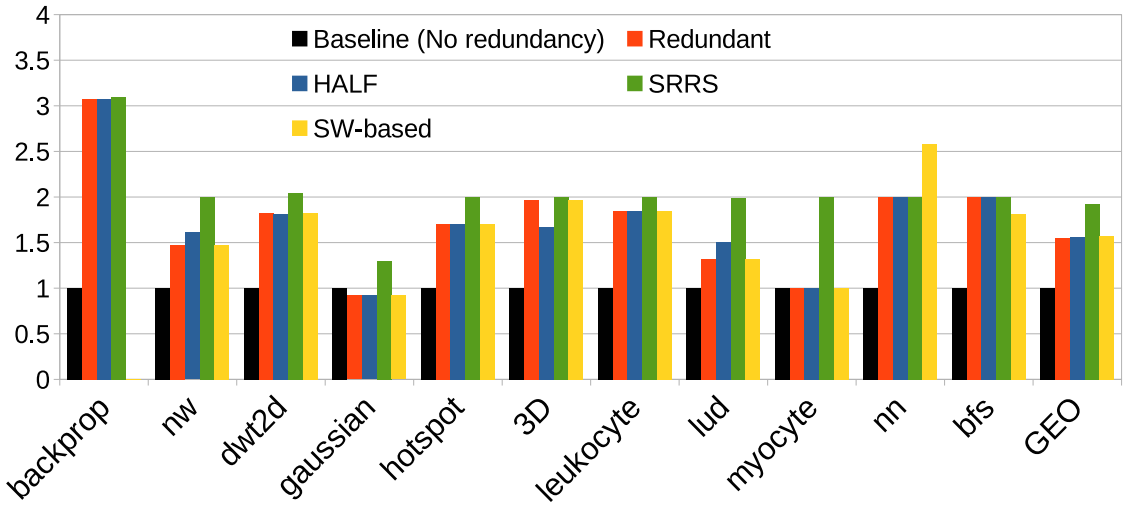


Figure 6-14: Simulator Cycles of all the solutions

6.3.6 HW and SW-only solutions side by side

Last but not least, we show the execution times of all the solutions side by side. The hardware solutions could not be integrated into a COTS platform since we cannot modify the kernel scheduler (at least to mimic HALF timing behavior). Instead, we perform the evaluation side by side on the simulator GPGPUSim, where we can compare them fairly. In this experiment, we show our solutions' execution time, the baseline non-redundant version, and the simple redundant version, which only guarantees diversity for the *friendly* kernels. Results are shown in Figure 6-14.

Note that there is not an SW-based bar for backprop. Backprop is a *short* kernel, so this software-only solution is to be executed only on the CPU, but the simulator only models the GPU cycles. For this reason, we do not consider the backprop results when calculating the geometric mean, which are the rightmost bars labeled as GEO.

Generally, the HALF approach is the one that suits best since most of these kernels are *friendly* and serializing them (SRRS) ends up in a longer execution time. The software-only solution also fits well when dealing with *friendly* kernels since the simple redundant version is applied. For heavy kernels (nn and bfs), nn execution time increases up to 2.6x, whereas bfs obtains 1.8x, similar results to those tested on the real platform. Bfs takes advantage of the coalesced memory accesses, and fewer threads compete to access memory. With this, the software solution obtains the best performance in this particular workload.

6.4 Related Work

ASIL-D capable processors like the Infineon AURIX [1] and the ST Microelectronics SPC56XL70 [74] deployed in current cars implement DCLS. DCLS may not suffice for some fail-operational ASIL-D systems with tight FTTI [80]. To improve the reaction time in case of error detection, several works have proposed mechanisms to achieve

timely error detection [220] and recovery by means of low-latency checkpointing and roll-back recovery [221]. However, computational power requirements of AD systems greatly exceed the ones of current ASIL-D applications and thus, more powerful – yet safe – computing platforms are needed to realize AD systems [12].

NVIDIA has recently announced the first functionally safe autonomous driving platform [28], which includes support to achieve fail-operational capabilities by allowing complex software algorithms run on the CPU, the CUDA GPU, a deep-learning accelerator and a programmable vision accelerator to enhance redundancy and diversity. According to the announcement, ASIL-D rating is achieved by an NVIDIA DRIVE Xavier GPU and an ASIL-D rated safety microcontroller with appropriate safety logic. However, to the best of our knowledge, ASIL-D compliance for functionalities requiring high performance can only be achieved with diverse software implementations, which ultimately increase drastically design and V&V costs.

Some authors evaluate the use of GPU, FPGA and ASIC designs for AD applications, showing that each design provides a different performance and power tradeoff, so the best hardware platform may change across different AD applications [186]. However, GPUs have already been suited to automotive systems, which provides GPUs with an advantage w.r.t. other hardware platforms [22, 207].

While redundancy is a well-known reliability measure to combat random (independent) faults, such as radiation, either employing time-redundancy [222, 223], space-redundancy [224, 119] or both indistinctly [124], none of those works considers the case of CCFs, which may lead the system to failure despite redundancy. Differently to those works, in this contribution, we consider specifically CCFs, which are the faults of relevance for the highest ASIL in automotive, and show how CCFs can be avoided – and to what extent – by enforcing diverse redundancy.

6.5 Conclusions

The use of GPUs for highly-critical Autonomous Driving (AD) software poses a number of functional safety requirements for GPUs' design and utilization. In this contribution, we propose to exploit the intrinsic redundancy inside GPUs to achieve diverse redundancy, as needed for ASIL-D software components. With this idea, we present multiple solutions to achieve it by software-only means.

We have identified the different kernel patterns when trying to execute them redundantly and classified them accordingly in three different categories, *friendly, heavy* and *short*. The software-only solution requires an early inspection of the kernel and its behavior on the desired platform since only *friendly* kernels are guaranteed to be executed in a diverse redundant manner. To solve this, we also presented a protocol to transform any *heavy* kernel into a *friendly* one based on the specification of the COTS GPU targeted. Smaller kernels (*short*) can be executed directly on the safe CPU side. Thus, this work delivers a full software-only solution for any given kernel. Later, we proved that this solution also works for a TMR strategy. Staggering is naturally created at the beginning of the execution during kernels offloading, as discussed and observed in a real COTS GPU. We also performed a fault injection campaign

with a single-fault fault model, in which we observed that SDC that were undetected using the baseline application, were detected by our redundant mechanism. Finally, we have compared the software-only solution together with the hardware solutions of the previous proposal side by side on a GPU simulator. As expected, hardware solutions generally offer lower execution times, but software solutions can instead be used right away in COTS GPUs.

Chapter 7

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

7.1 Introduction

As explained in previous chapters, Autonomous Driving (AD) frameworks require high-performance platforms with some form of support for diverse redundancy since some AD tasks need to run on those platforms have ASIL-D requirements.

Several chip vendors have already commercialized several processors and platforms for AD systems, such as the RENESAS R-Car H3 [22] and the NVIDIA Xavier [18] platforms, to name a few. In general, those platforms include some general-purpose computing cores (e.g., ARM-based) and/or additional accelerators for some specific applications. However, eHPC platforms usually lack explicit hardware support for enforcing some form of diverse redundancy, and software strategies are needed. Analogously, the European Processor Initiative (EPI) [225], which includes an ARM-based multicore, is intended for HPC and automotive applications. Hence, it inherits the same requirements of the other eHPC platforms for the automotive domain: implementing diverse redundancy.

This contribution tackles this challenge by proposing a *software-only* flexible and efficient approach to achieve diverse redundancy by design for eHPC multicores in general, and for ARM-based multicores (i.e., like the EPI one) 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 very relevant fault models are avoided. In particular, the contributions presented in this chapter are as follows:

1. A flexible and efficient software-based approach to enforce diverse redundancy on high-performance multicores, with requirements met by the most popular processor families.
2. A tailoring of the approach for a 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 of around 4% on average and up to 10% in one case for a variety of automotive benchmarks.

7.2 Software-based Diverse Redundancy Approach

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

7.2.1 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, 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. In any case, how to manage CCFs in the memory system without duplication is an already solved problem, since this is the default solution for current safety-critical systems in domains such as automotive, avionics, and space, among others.

7.2.2 Specification of the Execution Strategy

The approach we propose to achieve diverse redundancy consists of the following main steps, illustrated in Figure 7-1:

1. The MCU replicates input and output data and buffers for the task to be executed on the Embedded High-Performance Computing Multicore (eHPCM). We refer to redundant processes as head and trail processes.

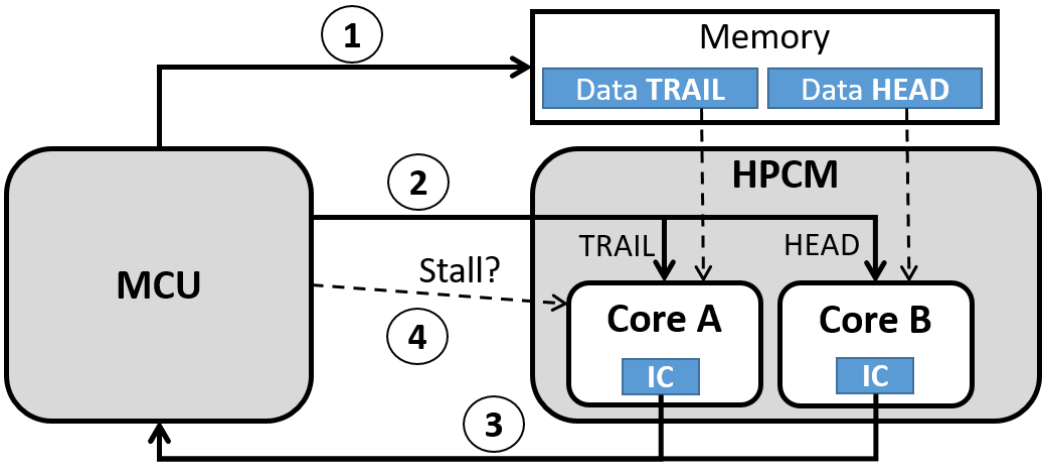


Figure 7-1: Schematic of the diverse redundancy execution strategy.

2. The MCU offloads both processes onto different cores in the eHPCM.

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.

   4.1. 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).

   4.2. 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 eHPCM, ensuring that the staggering is large enough so that the trail process cannot catch up with the head one. Particular care must be taken to 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.

As shown in Figure 7-1, the strategy relies on several properties that must be provided by the hardware/software platform, which we list as follows:

- The MCU must be entitled to specify the eHPCM core where a process is offloaded. This is a feature already provided by common Linux implementations. Therefore, it is expected that the specific OS or hypervisor deployed onto the eHPCM for the automotive segment delivers this support.

- The MCU must be entitled to program the Performance Monitoring Counters (PMCs) of the eHPCM cores to count instructions. If this is not possible, then such actions must be embedded into the processes themselves so that they program their PMCs appropriately on their own.

- The MCU must be entitled to access the PMCs (to retrieve the IC) of the eHPCM cores remotely. If this is not possible, then the processes must be deployed with capabilities to post such information in specific memory locations periodically so that the MCU can retrieve such information.

- The MCU must be entitled to send inter-process communication signals to the eHPCM cores. In particular, the MCU needs to stop and resume cores' execution to control the execution progress of the trail process. This can be achieved with the `SIGSTOP` signal to stall execution if the trail process progresses too fast, and the `SIGCONT` to resume its execution whenever the head process has progressed enough not to be caught by the trail process.

  In our implementation, we use POSIX signals to carry out this task.

- The response time of the remote PMC readouts, as well as of the signal transmissions, must be low enough so that T_{check} is also kept low enough. Values in the order of microseconds are at an affordable scale for automotive systems where tasks' execution times are in the order of tens to hundreds of milliseconds.

- All the memory hierarchy needs protection (e.g., ECC or parity). In fact, modern COTS products already provide these protection mechanisms for memories.

The pseudocode executes periodically in the MCU for monitoring purposes is shown in Listing 7.1 for completeness.

```
void On_timer(){
  IC_{head} = remoteRead(Core_{head},IC)
  IC_{trail} = remoteRead(Core_{trail},IC);
  if ((IC_head - IC_trail) < I_stagger){
    kill(SIGSTOP, Core_trail);
  }
  else{
    kill(SIGCONT, Core_trail);
  }
}
```

Listing 7.1: Monitor routine to preserve staggering across redundant processes

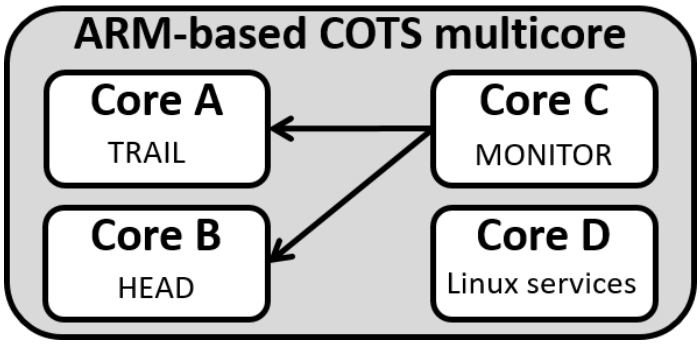


Figure 7-2: Strategy used in the 4-core Jetson TX2

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.

7.2.3 Realization on an ARM-based Multicore

We integrate our strategy on a 4-core ARM Cortex-A57, part of the NVIDIA Jetson TX2 platform [127], see Section 3.1.4 for detailed information on the 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 realizing our strategy on different multicores.

The current platform used for implementing our strategy lacks some features, such as an MCU processor mastering the eHPCM and AUTOSAR support[1]. Therefore, we run the monitoring process in a core in the eHPCM, 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.

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, we concentrate as many Linux services as possible in a specific core not used neither by the monitor nor by the redundant application processes (core D in Figure 7-2).

Our monitor is scheduled in one core, thus having to access application cores remotely. Remote PMCs are accessed using the *perf* API version 4.4.38 and are set to capture only user-space instructions. In particular, we access those PMCs mapping the instruction count, which have been configured properly prior to the offloading of the redundant application processes. As indicated before, redundant application pro-

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

cesses are spawned employing MPI, which is also used to detect when their execution is completed and to retrieve output data back to the monitor core, where output data is compared, and the faults leading to an erroneous result can be detected. Interestingly, we notice that the signal processing, when receiving a SIGSTOP, executes one instruction at the user space, which is captured by the counter. Thus, we count the number of signals sent to the Trail and subtract it from the PMC value.

Given a specific T_{check} monitor interval, we must make sure that between two monitor checks, the trail process cannot catch up with the head process. For that purpose, we first compute the maximum number of instructions that can be executed in one interval as follows (Equation 7.1):

$$maxI_{period} = maxIPC \cdot \frac{T_{check}}{minT_{cycle}} \tag{7.1}$$

where $maxIPC$ stands for the maximum number of instructions per cycle, and $minT_{cycle}$ for the cycle time at the maximum operating frequency (so minimum cycle time). Thus, $maxI_{period}$ stands for the maximum number of instructions that could execute in a single monitoring period, $maxIPC$ is set to 3 according to the processor specification since the ARM A57 core can fetch up to 3 instructions per cycle. $minT_{cycle}$ is calculated based on the maximum frequency of the processor (1.2GHz).

Additionally, we measure the time needed to retrieve PMCs remotely, perform monitor calculations, and send a signal to another core. Such time has always been below 20μs in total. Thus, for the sake of simplicity and to account for potential measurement inaccuracies, we set $I_{stagger}$ to be twice $maxI_{period}$, so that, in practice, if for instance T_{check} is 1 ms, we enforce a staggering of no less than 2 ms. Of course, this value could be tightened to reduce performance cost but, as shown in the evaluation, the performance overhead of such monitoring frequency is mostly related to the order of magnitude of T_{check}, being a fraction of $T_{check} \leq 1ms$. Still we left below (equation 7.2) the general equation including these two latencies, where $Latency_{SendSignal}$ is the latency in cycles to send a SIGSTOP signal and the $Latency_{ReadRemotePMC}$ is the latency to retrieve both PMC values.

$$maxI_{period} = maxIPC \cdot \left(\frac{T_{check}}{minT_{cycle}} + \frac{Latency_{SendSignal}}{minT_{cycle}} + \frac{Latency_{ReadRemotePMC}}{minT_{cycle}} \right) \tag{7.2}$$

7.2.4 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 7-1) for the recovery actions (e.g., reset and restart the eHPCM or part of it) whenever a fault is detected. Thus, we rely on current recovery mechanism techniques to perform the recovery of the execution once a fault is detected.

The proposed solution provides fault detection through software-only diverse redundancy for any type of transient fault (e.g., voltage droops or radiation effects) and permanent faults randomly affecting core components. 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 a watchdog by monitoring the progress of both processes. Eventually, a fault is detected if one of them does not make any progress during a predefined time (i.e., its instruction count does not vary). 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.

7.3 Evaluation

This section evaluates our strategy to achieve diverse redundancy on COTS multicores. First, we introduce the evaluation framework. Second, we assess the sensitivity of overheads to T_{check} duration for a representative benchmark. Finally, we evaluate our approach on a number of benchmarks for an appropriate T_{check} (1 ms).

7.3.1 Framework

As a representative platform, we build upon the CPU complex of an NVIDIA Jetson TX2 board, which includes two multicores: a 4-core ARM Cortex-A57 and a 2-core NVIDIA Denver, both of them implementing ARM 64-bit (A64) instruction set architecture. In particular, we use the Cortex-A57 multicore for our experiments and power off the NVIDIA Denver multicore, see Section 3.1.4 for deeper insides on the SoC used and Section 3.7 for a detailed description of the setup used and the required modifications we implemented.

As representative real-time automotive benchmarks, we use the EEMBC Autobench suite [185], (see Section 3.2.2) which reflects some functionalities relevant to 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 such as, for instance, Apollo [226], build upon neural networks whose execution time is mostly devoted to executing such types of operations (e.g., object detection, fusion, tracking, trajectory prediction, etc.). To capture potential performance variations, each benchmark is run 500 times for each setup considered. In some plots, we depict all measurements with boxplots, whereas, in others, we summarize results by averaging execution times across measurements.

7.3.2 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 consider four setups, which we name as follows:

- BASELINE: This one corresponds to the original non-redundant version of the benchmark.
- No monitor (NM): This one corresponds to the redundant version of the benchmark without any monitoring in place, thus reflecting the impact on execution time due to sharing multicore hardware resources (e.g., bus and memory controller).
- Passive (P): This one includes the monitor, which performs all actions needed, such as retrieving PMCs from the head and trail processes, but taking no action on the trail process so that it never gets stalled to preserve diversity.
- Safe (S): This setup is the complete implementation of the approach where the trail process is delayed whenever it could catch up with the head process, thus preserving diversity.

For each setup but the baseline one, we consider different T_{check} values, ranging between 0.0001 and 0.01 seconds, so between 100μs 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.

Results of the different setups and T_{check} values for EEMBCs and `matmul` are shown in Figure 7-3. The figure summarizes the normalized results for the 8,500 runs (500 runs for each of the 17 benchmarks). As shown, introducing redundancy (NM setup) creates a slight execution time increase for some runs due to the contention experienced in shared resources by running processes redundantly. These overheads are not intrinsic to our approach but to the need to run processes redundantly. The Passive monitor (P in the figure) causes no meaningful differences in execution time since reading PMCs remotely and performing some local calculations causes negligible interference in the computing processes. Finally, 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 on the 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 to be scheduled in the MCU, where computation resources are scarce and shared across a number of safety-critical activities. Thus, we regard $T_{check} = 0.001$ $(1\ ms)$ as an appropriate tradeoff, and thus, this is the configuration we use in the rest of this section. Note, however, that other platforms with different latencies may require a different T_{check} value.

While our approach guarantees diversity by construction, we further assess it by comparing the passive (**P**) and active (Safe, **S**) monitors across all benchmarks. In particular, we check whether the head process has always been ahead of the trail process in terms of executed instructions.

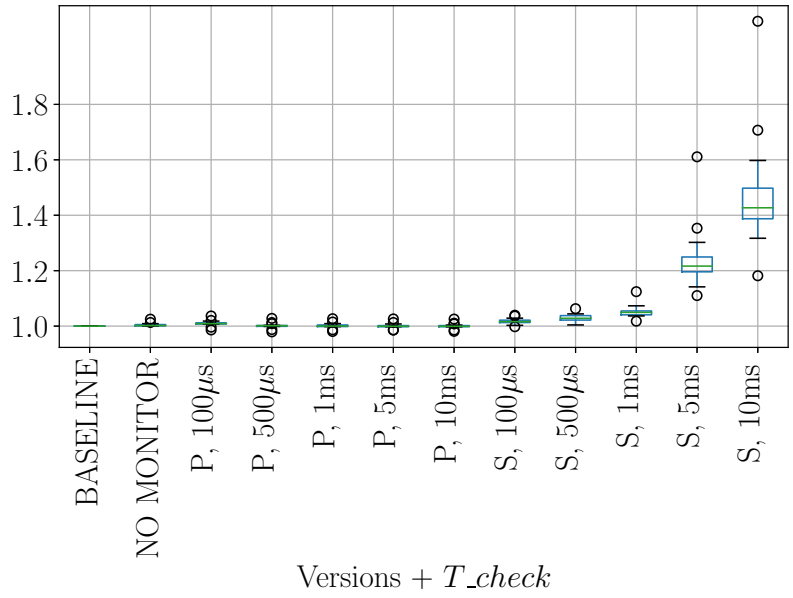


Figure 7-3: Execution times, in the form of a boxplot, for the different setups (**Baseline**, **No-Monitor**, **P** = Passive, **S** = Safe) and T_{check} values (including EEMBCs and matrix multiplication).

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, it is quite common having the trail process executed 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.

7.3.3 Performance Assessment

In Figure 7-4 we show boxplots for all benchmarks with $T_{check} = 0.001s$ (1 ms) setup, normalized w.r.t. their respective (non-redundant) baseline cases. Average execution times increase only by a few percent, 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 eHPCM.

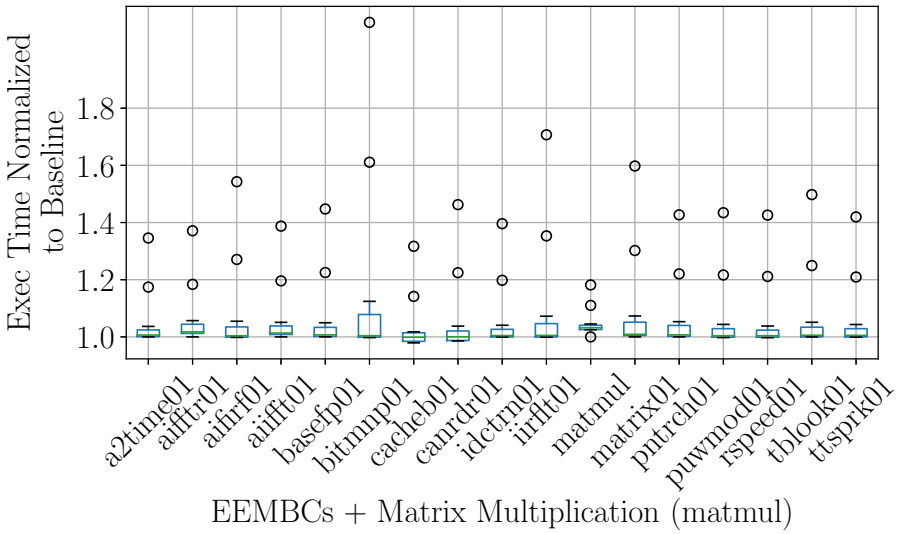


Figure 7-4: Boxplot of the relative execution times of our approach for $T_{check} = 0.001s$ (1 ms) in the EEMBCs benchmarks.

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 7-3. For completeness, we further illustrate this effect in Figure 7-5 with average results for the *basefp01* EEMBC benchmark, whose overheads are close to the average behavior across all benchmarks. In the Figure, we show the execution time of the application as *Binary Time*, plus the *data transfer* and the *comparison* times, where the latter two are negligible in practice (largely below 1% across setups and T_{check} values evaluated). As shown, in the five rightmost columns, decreasing T_{check} down to $0.1ms$ decreases overheads from 5% to 2% only (with increased monitoring costs in the MCU), whereas increasing T_{check} up to $10ms$ increases overheads to 46%.

Overly high execution times in Figure 7-4 are some tens of milliseconds above their median, and such large variation can only be attributed to uncontrolled resource sharing. This indicates the convenience of setting appropriate setups in the eHPCM to limit the impact of worst-case contention in shared resources.

7.4 Related Work

ASIL D compliant ST Microelectronics SPC56XL70 [74] and Infineon AURIX processor family [1] implement DCLS, whereas some Arm Cortex-R5 designs implement Triple-Core Lockstep [80], but fails to provide enough performance for AD systems [245].

Some improvements shorten time-to-detection for errors [220] or enhance recovery processes [221] but do not improve performance. Another family of solutions for high-performance CPUs builds upon thread redundancy inside a single core [228, 65], or

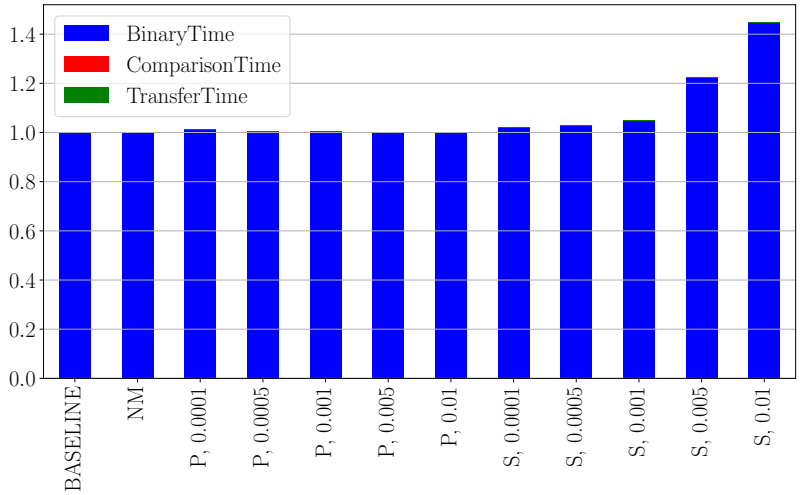


Figure 7-5: Execution times of the different setups for basefp01 baseline normalized.

Strategy	Target	Diversity (CCF considered)	Approaches
HW	CPU	Yes	[74, 80, 227]
		No	[65, 228, 229, 230, 231, 232]
	GPU	Partially	[47]
		No	[222, 233, 223, 234]
SW-Only	CPU	Yes	**This proposal**
		No	[235, 236, 237, 238], [239, 240, 241, 242]
	GPU	Partially	[48]
		No	[243, 124, 206, 244]

Table 7.1: Classification of HW and SW-only fault-tolerant techniques

across cores [229, 230, 231], even with only partial redundancy [227, 232]. However, those solutions require hardware support for thread synchronization and, differently from DCLS, do not guarantee diversity.

SW-only solutions also exist for CPUs, introducing redundancy at compiler levels [238, 239], building on transactional memory [235] or creating a monitoring process to check for errors [236, 241], among other solutions [242]. However, none of them guarantees staggering execution for redundant threads/processes, so CCFs may cause the same error in both threads/processes, leading to a failure. For example, authors in [241] consider process granularity when applying for redundancy, but they do not impose control on the pace of the redundant execution, which made them vulnerable to CCFs since their cores may have exactly the same state when the fault affecting both cores identically occur.

Analogous solutions for accelerators (e.g., GPUs or the Kalray MPPA family) have been proposed, either with hardware support [124, 233, 223, 234] or with software-only support [243, 124, 206], but none of them guarantees diversity to protect against CCFs. Some preliminary solutions guarantee diversity to some extent for GPUs either with [47] or without hardware support [48].

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 7.1.

7.5 Conclusions

This proposal 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 eHPCM and MCU can be traded off easily by increasing/decreasing the monitoring frequency.

Chapter 8

Conclusions

8.1 Contributions

The work done in this thesis advances the state of the art of the safety-critical domain in several aspects. The main challenges tackled focus on devising diverse redundant executions to enable the usage of HPC hardware components for the highest criticality levels in safety-critical applications. Two different approaches are presented for HPC GPUs, each one with different trade-offs and one for the Multicores. These challenges are tackled in the different contributions of the thesis, which are the following:

- Our first contribution focuses on an early analysis of the GPUs usage in the automotive domain. In particular, we analyze the fundamental properties for a correct operation under automotive's safety regulations. This initial contribution builds as a starting point for the rest of the contribution towards the contribution focused on enabling GPUs in the Automotive domain.

- Our second contribution proposes minor modifications of the internal scheduling policies of the GPUs that guarantee diverse redundancy by construction. Thus, reaching ASIL-D compliance efficiently without the need to increase design and/or procurement costs. In particular, we show how the explicit control of the SMs used for a given kernel, together with the serialization of redundant execution, in some cases, allows achieving diverse redundancy with low-cost w.r.t uncontrolled redundancy for all types of kernels.

- Our third contribution presents an alternative way to achieve diverse redundancy executions in COTS GPUs by software-only means. In this contribution, a different approach is required based on the type of kernel identified. A different solution is offered for each, from where we highlight the *reshape* protocol employed for the *heavy* kernels to transform them into *friendly*. We analyze the slack observed when two consecutive kernels are launched to the GPU, and we extend the initial approach to achieve Triple Modular Redundancy (TMR) as well. To finish with the GPU contributions, we perform a fault injection campaign to analyze the improvements in reliability terms, and we compare the two GPU contributions' execution times side by side in the simulator.

- The last contribution of this Thesis is the proposal focusing on Multicore systems. In this contribution, we develop a strategy to achieve a diverse redundant execution in these systems based on the PMC. One of the system's cores is in charge of orchestrating and monitoring the execution on two other cores (or accelerators), which will execute the task similarly to lockstep. By reading the PMC to monitor their progress and pausing/resuming them in case of potential loss of *diversity*. We evaluate and verify the proposal in a COTS ARM multicore system.

8.2 Impact

As we have seen, the automotive industry intends to shift towards the usage of GPUs due to Autonomous Driving (AD) performance requirements. As we analyze in the first contribution, multiple gaps exist from the HPC domain and the safety-critical domain. In particular, those related to the functional safety and timing guarantees are still open challenges and threats to certification, as well as the lack of internal controllability and observability. GPUs will need to address these challenges before they can be integrated into automotive SoC and be commercialized. The work done in this thesis has an impact on the following aspects:

- The GPU contributions introduce one of the challenges mentioned above, to enable a *diverse redundant* execution, which enables them to provide functional safety, although the certification of GPUs for the automotive domain is a long journey. With the contributions of this thesis, we made an initial step and devise two different strategies to achieve a *diverse redundant* execution in a single GPU, hence with lower costs than default solutions using two GPUs.

- The GPU hardware proposal may have an impact on the GPU architecture community since it is a relatively easy implementation that enables a *diverse redundant* execution in a single GPU.

- The realization of this thesis has been the seed of another scientific article. A paper that studies the reliability properties of one of the contributions under the effect of proton radiation. The paper was presented in the poster session of the RADECS 2022 conference and has been selected to be published in a special issue of the IEEE Transactions on Nuclear Science:

  Assessment of redundant kernel execution on embedded GPU under proton irradiation

  S. Alcaide, A. Serrano-Cases, A. Romero Maestre, Y. Morilla, S. Cuenca-Asensi

- The software-only GPU contribution of this thesis has been used as a basis for some successful projects proposals and tasks within projects such as the project **ASIL2ECSS** or the European Space Agency (ESA) co-funding Ph.D. thesis of Ivan Rodriguez has used the contribution as a starting point.

- We have recently started a bilateral collaboration with Intel Corporation for the integration of our software-only solution for GPUs on Intel GPUs due to the relevance of the topic and the effectiveness and efficiency of our solution. So far a solution with some constraints has been developed, and the work continues towards removing those constraints and maturing the solution so that it can be transferred to product units within Intel in the short or mid-term.

On the other side, we also analyze multicore systems that can also be employed to improve the performance demanded by some tasks in safety-critical systems. The work done in this thesis also impacts the following aspects:

- The Multicore contribution has been later improved in the context of the **FRACTAL** project in a task, in the form of a software library to enable a *diverse redundant* execution on multicores. The library has been ported successfully to RISC-V and x86 ISAs. The library is open-source and can be found here: https://gitlab.bsc.es/caos_hw/software-diverse-redundancy-library
- The Multicore contribution has also been used as the basis for a small hardware module named *SafeDE* developed in the **SELENE** project. Similarly to our contribution, this tiny module reads the instruction counters of two cores in order to guarantee a *diverse redundant* execution. In a few words, this module is the hardware counterpart of our software-only solution, hence providing much higher efficiency, yet at the cost of requiring hardware modifications. The module is open-source and can be found here: https://gitlab.bsc.es/caos_hw/hdl_ip/bsc_lightlock

Overall, the global contribution of this thesis consists of enabling the use of HPC parts even for applications with the highest integrity levels by providing solutions to realize diverse redundancy – a key safety requirement for those applications.

8.3 Future Work

The results and contributions of this thesis can be further extended in several directions. We list some of these directions and present them in order of feasibility, from short-term to long-term:

- The GPU software-only contribution lacks automation, as it requires different solutions based on the type of kernel being dealt with. However, the kernel type depends on the hardware and the software. On one side, the software ran and, in particular, the resources required on the kernel launch will affect the characterization of the kernel since bigger resource requirements will generally point towards a *heavy kernel*. On the other side, the larger the number of resources available in the GPU, the less likely a kernel will be *heavy*. Thus, a foreseen line of research is automatizing kernel-type detection. Once this is achieved, a second phase is to automate the modifications into the diverse-redundant approach based on the outcome of the first part since it will differ based on the kernel type (e.g., requires a transformation for *heavy* kernels).

- Related to the previous point, another interesting line of research is to extend the proof of concept to GPUs from other vendors such as Intel or AMD. Although we already commented on the feasibility of this approach to other GPU vendors, we believe is still interesting to achieve it. This line has already produced a publication that is currently pending to appear (Publication number 10 in 1.6) which explores this approach on Intel GPUs.

- Another aspect that we just barely touch in this Thesis is radiation testing. This topic is particularly interesting for the space domain. It is a common practice in the space domain to reuse automotive components due to the similarities of both domains, with the particularity of radiation, which is more relevant in the space domain. With this, we think that radiation testing of the GPU software-only approach can be very convenient to extend this approach into the space domain. Publication number 14 in Section 1.6 goes in this direction and could be easily extended to other radiation types (e.g., electrons).

- Related to the last contribution, one interesting line of research is to extend to approach to multi-threaded applications. The current approach is limited to applications with only one thread. However, due to the number of cores available in multicore systems, it could be very interesting to use all of them and use all that computing power. A more ambitious research line could be extending this approach to be used in other specific accelerators apart from GPUs.

Bibliography

[1] Infineon. AURIX Multicore 32-bit Microcontroller Family to Meet Safety and Powertrain Requirements of Upcoming Vehicle Generations. http://www.infineon.com/cms/en/about-infineon/press/press-releases/2012/INFATV201205-040.html. viii, 1, 37, 85, 95, 108, 120

[2] Xabier Iturbe, Balaji Venu, Emre Ozer, Jean-Luc Poupat, Gregoire Gimenez, and Hans-Ulrich Zurek. The Arm Triple Core Lock-Step (TCLS) Processor. *ACM Trans. Comput. Syst.*, 36(3), 2019. viii, 22

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. xv, 16, 17, 18

[4] G.A. Klutke, P.C. Kiessler, and M.A. Wortman. A critical look at the bathtub curve. *IEEE Transactions on Reliability*, 52(1):125–129, 2003. xv, 42, 43

[5] ARM. ARM Cortex A57 specifications. https://developer.arm.com/Processors/Cortex-A57. xv, 48, 49

[6] Dustin Franklin - NVIDIA. NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge, 2017. https://developer.nvidia.com/blog/jetson-tx2-delivers-twice-intelligence-edge. xv, 49

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009. xvi, 48, 53, 79, 91, 96, 100

[8] The Economist. Nvidia plays the diversification game. https://www.economist.com/business/2019/02/23/nvidia-plays-the-diversification-game. 1

[9] NVIDIA. Crypto mining gpu for professional miners. https://www.nvidia.com/en-us/cmp/, 2021. 1

[10] Edoardo Calia and Davide D'Aprile. *Industry4.0*, pages 309–333. Springer International Publishing, 2020. 1

[11] Xi Chen, Chen Wang, Shanjiang Tang, Ce Yu, and Quan Zou. Cmsa: a heterogeneous cpu/gpu computing system for multiple similar rna/dna sequence alignment. *BMC Bioinformatics*, 18, 06 2017. 1

[12] ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade, 2015. https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php. 1, 60, 109

[13] Cobham Gaisler. Section Processors - Category LEON CPU Family. https://www.gaisler.com/index.php/products/processors/leon-examples, 2021. 1

[14] Cobham Gaisler. Section Processors - Category NOEL CPU Family. https://www.gaisler.com/index.php/products/processors/noel-v-examples, 2021. 1

[15] G. Durrieu and M. Faugère and Sylvain Girbal and D. G. Pérez and C. Pagetti and W. Puffitsch. Predictable flight management system implementation on a multicore processor. In *Proceeding of the 7th European Congress on Embedded Real Time Software and Systems*, 2014. 1, 5

[16] Xilinx. Xilinx Zynq UltraScale+ MPSoC Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf. 1, 41

[17] Alejandro Serrano-Cases, Juan M. Reina, Jaume Abella, Enrico Mezzetti, and Francisco J. Cazorla. Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems, ECRTS 2021, July 5-9, 2021, Virtual Conference*, volume 196 of *LIPIcs*, pages 3:1–3:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. 1

[18] Danny Shapiro. Introducing Xavier, the NVIDIA AI Supercomputer for the Future of Autonomous Transportation. *NVIDIA blog*, 2016. 2, 3, 33, 111

[19] CoreAVI. GPUs/SoCs for Safety Critical. https://coreavi.com/product_category/gpus-socs-for-safety-critical, 2021. 2

[20] M. Benito et al,. Comparison of GPU computing methodologies for safety-critical systems: An avionics case study. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*, pages 717–718. IEEE, 2021. 2

[21] Gary Hicok. NVIDIA Xavier Achieves Industry First with Expert Safety Assessment. https://blogs.nvidia.com/blog/2020/05/20/xavier-achieves-industry-first-safety-assessment/, 2020. 2, 3

[22] RENESAS R-Car H3. https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html. 2, 4, 82, 109, 111

[23] Guardian News & Media Limited. Florida Tesla crash which killed two will be investigated by federal board. https://www.theguardian.com/technology/2021/sep/18/florida-tesla-crash-autopilot-fire-national-federal-transportation-board, 2021. 2

[24] Euronews. Tokyo 2020 driverless buses lose self-driving functions after hitting Paralympic athlete. https://www.euronews.com/next/2021/08/30/toyota-halts-autonomous-e-palette-buses-after-one-hits-paralympic-athlete-in- 2021. 2

[25] NHTSA. Nhtsa report on toyota unintended acceleration investigation. https://one.nhtsa.gov/About-NHTSA/Press-Releases/ci.NHTSA%E2%80%93NASA-Study-of-Unintended-Acceleration-in-Toyota-Vehicles.print. 2

[26] Intel Corporation. Intel® Core™ i9-11900KF Processor. https://ark.intel.com/content/www/es/es/ark/products/212321/intel-core-i9-11900kf-processor-16m-cache-up-to-5-30-ghz.html. 3

[27] NVIDIA Corporation. Nvidia geforce rtx 3080. https://www.nvidia.com/es-es/geforce/graphics-cards/30-series/rtx-3080-3080ti/. 3

[28] NVIDIA. NVIDIA Announces World's First Functionally Safe AI Self-Driving Platform. https://nvidianews.nvidia.com/news/nvidia-announces-worlds-first-functionally-safe-ai-self-driving-platform. 3, 72, 82, 109

[29] Imagination Technologies Limited. Graphics processors. https://www.imaginationtech.com/graphics-processors/, Mar 2021. 4

[30] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. CARLA: an open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017. 4

[31] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018. 4, 34, 51

[32] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006. 5, 37

[33] V.P. Nelson. Fault-tolerant computing: fundamental concepts. *Computer*, 23(7):19–25, 1990. 5, 22

[34] Infineon. *AURIX-TC27xB-Step, 32-bit Single Chip Micro-controller, User's Manual, vl4.1*, feb 2014. 5, 6

[35] PROXIMA. Probabilistic real-time control of mixed-criticality multicore and manycore systems. http://www.proxima-project.eu/, oct 2014. 5

[36] T. Moseley, J. L. Kihm, D. A. Connors, and D. Grunwald. Methods for modeling resource contention on simultaneous multithreading processors. In *IEEE ICCD*, 2005. 5

[37] H. Kim, Dionisio de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *RTAS*, 2014. 5

[38] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *RTSS*, 2009. 5

[39] International Standards Organization. *ISO/DIS 26262. Road Vehicles — Functional Safety*, 2009. 6, 16, 22, 25, 26, 28, 34, 41, 59, 85

[40] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, 2010. 6, 49, 52, 64

[41] NVIDIA. NVIDIA CUDA Toolkit 10.0.130. https://docs.nvidia.com/pdf/CUDA_Toolkit_Release_Notes.pdf, 2018. 6, 49, 52, 64

[42] Matina Maria Trompouki and Leonidas Kosmidis. Brook auto: High-level certification-friendly programming for gpu-powered automotive systems. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018. 6, 33, 64

[43] Leonidas Kosmidis, Iván Rodriguez, Álvaro Jover, Sergi Alcaide, Jérôme Lachaize, Jaume Abella, Olivier Notebaert, Francisco J. Cazorla, and David Steenari. Gpu4s: Embedded gpus in space. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 399–405, 2019. 7, 34

[44] Leonidas Kosmidis, Jérôme Lachaize, Jaume Abella, Olivier Notebaert, Francisco J. Cazorla, and David Steenari. Gpu4s: Embedded gpus in space. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 399–405, 2019. 7, 34

[45] Leonidas Kosmidis, Iván Rodriguez, Álvaro Jover, Sergi Alcaide, Jérôme Lachaize, Jaume Abella, Olivier Notebaert, Francisco J. Cazorla, and David Steenari. Gpu4s: Embedded gpus in space - latest project updates. *Microprocessors and Microsystems*, 77:103143, 2020. 7, 12

[46] Sergi Alcaide, Leonidas Kosmidis, Hamid Tabani, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Safety-related challenges and opportunities for gpus in the automotive domain. *IEEE Micro*, 38(6):46–55, 2018. 10

[47] Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. High-integrity gpu designs for critical real-time automotive systems. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 824–829, 2019. 10, 121, 122

[48] Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. Software-only diverse redundancy on gpus for autonomous driving platforms. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 90–96, 2019. 11, 121, 122

[49] Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. Software-only triple diverse redundancy on gpus for autonomous driving platforms. In *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pages 82–88, 2020. 11

[50] Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. Achieving diverse redundancy for gpu kernels. *IEEE TETC (2022) Transactions on Emerging Topics in Computing - Special Section on Defect and Fault Tolerance in Nanoscale Systems for Emerging Computing Paradigms and Applications*, pages 1–1, 2022. 11

[51] Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. Software-only based diverse redundancy for asil-d automotive applications on embedded hpc platforms. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–4, 2020. 11

[52] Leonidas Kosmidis, Iván Rodriguez, Alvaro Jover-Alvarez, Sergi Alcaide, Jérôme Lachaize, Olivier Notebaert, Antoine Certain, and David Steenari. GPU4S: major project outcomes, lessons learnt and way forward. In *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, pages 1314–1319. IEEE, 2021. 11

[53] Francisco Bas, Sergi Alcaide, Ruben Lorenzo, Guillem Cabo, Guillermo Gil, Oriol Sala, Fabio Mazzocchetti, David Trilla, and Jaume Abella. Safede: a flexible diversity enforcement hardware module for light-lockstepping. In *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–7, 2021. 12

[54] Jaume Abella, Sergi Alcaide, Jens Anders, Francisco Bas, Steffen Becker, Elke De Mulder, Nourhan Elhamawy, Frank K. Gürkaynak, Helena Handschuh, Carles Hernandez, Mike Hutter, Leonidas Kosmidis, Ilia Polian, Matthias Sauer,

Stefan Wagner, and Francesco Regazzoni. Security, reliability and test aspects of the risc-v ecosystem. In *2021 IEEE European Test Symposium (ETS)*, pages 1–10, 2021. 12

[55] Francisco Bas, Sergi Alcaide, Guillem Cabo, Pedro Benedicte, and Jaume Abella. Safede: A low-cost hardware solution to enforce diverse redundancy in multicores. *IEEE Transactions on Device and Materials Reliability*, 22(2):111–119, 2022. 12

[56] Sergi Alcaide, Guillem Cabo, Francisco Bas, Pedro Benedicte, Francisco Fuentes, Feng Chang, Ilham Lasfar, Ramon Canal, and Jaume Abella. Safex: Open source hardware and software components for safety-critical systems. In *2022 Forum on Specification & Design Languages (FDL)*, pages 1–4, 2022. 13

[57] Ramon Canal, Francisco Bas, Sergi Alcaide, Guillem Cabo, Pedro Benedicte, Francisco Fuentes, Feng Chang, Ilham Lasfar, and Jaume Abella. Safedx: Standalone modules providing diverse redundancy for safety-critical applications. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 22nd International Conference, SAMOS 2022, Samos, Greece, July 3–7, 2022, Proceedings*, page 383–393, Berlin, Heidelberg, 2022. Springer-Verlag. 13

[58] Fabio Mazzocchetti, Sergi Alcaide, Francisco Bas, Pedro Benedicte, Guillem Cabo, Feng Chang, Francisco Fuentes, and Jaume Abella. Safesoftdr: A library to enable software-based diverse redundancy for safety-critical tasks, 2022. 13

[59] Alvin Reyes. Mercedes-benz wins world's first approval for level 3 autonomous cars: What's that mean?, Feb 2022. 15

[60] J3016c: Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. Standard, SAE International. 15

[61] Nico DeMattia. Level 4 autonomous driving could come by 2030 per audi lawyer, Feb 2022. 15

[62] International Electrotechnical Commission. *IEC61508Functional safety of electrical/electronic/programmable electronic safety-related systems*, 2010. 16, 25, 43, 44

[63] Wafa Gabsi and Bechir Zalila. Fault tolerance for distributed real time dynamically reconfigurable systems from modeling to implementation. In *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2013. 16

[64] Mukherjee, S.S. and Kontz, M. and Reinhardt, S.K. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 99–110, 2002. 20

[65] Rotenberg, E. Ar-smt: a microarchitectural approach to fault tolerance in microprocessors. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No.99CB36352)*, pages 84–91, 1999. 20, 120, 121

[66] M. Agostinelli, J. Hicks, J. Xu, B. Woolery, K. Mistry, K. Zhang, S. Jacobs, J. Jopling, W. Yang, B. Lee, T. Raz, M. Mehalel, P. Kolar, Y. Wang, J. Sandford, D. Pivin, C. Peterson, M. DiBattista, S. Pae, M. Jones, S. Johnson, and G. Subramanian. Erratic fluctuations of sram cache vmin at the 90nm process technology node. In *IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest.*, pages 655–658, 2005. 20

[67] Carles Hernandez and Jaume Abella. Timely error detection for effective recovery in light-lockstep automotive systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1718–1729, 2015. 21

[68] Sergi Alcaide, Carles Hernandez, Antoni Roca, and Jaume Abella. Dimp: A low-cost diversity metric based on circuit path analysis. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017. 21

[69] S. Mitra, N.R. Saxena, and E.J. McCluskey. Efficient design diversity estimation for combinational circuits. *IEEE Transactions on Computers*, 53(11):1483–1492, 2004. 21

[70] S. Mitra, N.R. Saxena, and E.J. McCluskey. A design diversity metric and analysis of redundant systems. *IEEE Transactions on Computers*, 51(5):498–510, 2002. 21

[71] S. Mitra, N.R. Saxena, and E.J. McCluskey. Techniques for estimation of design diversity for combinational logic circuits. In *2001 International Conference on Dependable Systems and Networks*, pages 25–34, 2001. 21

[72] S. Mitra and E.J. McCluskey. Design diversity for concurrent error detection in sequential logic circuits. In *Proceedings 19th IEEE VLSI Test Symposium. VTS 2001*, pages 178–183, 2001. 21

[73] Peter Tummeltshammer and Andreas Steininger. Power supply induced common cause faults-experimental assessment of potential countermeasures. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 449–457, 2009. 21

[74] STMicroelectronics. 32-bit Power Architecture microcontroller for automotive SIL3/ASILD chassis and safety applications. `https://www.st.com/en/automotive-microcontrollers/spc570s40e3.html`, 2014. 22, 95, 108, 120, 121

[75] 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 2017. 22

[76] Radio Technical Commission for Aeronautics (RTCA). *DO-178C - Software Considerations in Airborne Systems and Equipment Certification*, 2011. 22, 25

[77] Bernick, D. and Bruckert, B. and Vigna, P.D. and Garcia, D. and Jardine, R. and Klecka, J. and Smullen, J. Nonstop/spl reg/ advanced architecture. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 12–21, 2005. 22

[78] IBM. *PowerPC 750GX Lockstep Facility. Application note*, 2008. 22

[79] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. Fault-Tolerant Platforms for Automotive Safety-Critical Applications. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, page 170–177. Association for Computing Machinery, 2003. 22

[80] X. Iturbe et al. Addressing Functional Safety Challenges in Autonomous Vehicles with the Arm Triple Core Lock-Step (TCLS) Architecture. *IEEE Design and Test*, PP(99):1–1, 2018. 22, 108, 120, 121

[81] Wikipedia contributors. Ram parity — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=RAM_parity&oldid=1042196119, 2021. [Online; accessed 23-November-2021]. 23

[82] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950. 23

[83] Marco Ottavi, Dimitris Gizopoulos, and Salvatore Pontarelli. *Dependable Multicore Architectures at Nanoscale*. Springer, Cham, 1 edition, 2018. 24, 37

[84] European Committee for Electrotechnical Standardization (CENELEC). *Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS) Part 2: Systems Approach to Safety*, 2019. 25

[85] IEC - International Electrotechnical Commission. *IEC 62304:2006/AMD 1:2015 Medical device software — Software life cycle processes — Amendment 1*, 2015. 25

[86] IEC - International Electrotechnical Commission. *IEC 62061:2021 Safety of machinery - Functional safety of safety-related control systems*, 2021. 25

[87] IEC - International Electrotechnical Commission. *IEC 61513:2011 Nuclear power plants - Instrumentation and control important to safety - General requirements for systems*, 2006. 25

[88] International Standards Organization. *ISO/PAS 21448:2019 Road vehicles — Safety of the intended functionality*, 2019. 25, 29, 34

[89] SAE International. *J3016: Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems*, 2014. 27

[90] Rory Cellan Jones. Uber's self-driving operator charged over fatal crash, Sep 2020. https://www.bbc.com/news/technology-54175359. 30

[91] E. Wyrwas. Body of knowledge for graphics processing units (gpus). Technical report, National Aeronautics and Space Administration (NASA), 2018. 31, 34

[92] Daniel Alfonso Gonçalves Gonçalves de Oliveira, Laercio Lima Pilla, Thiago Santini, and Paolo Rech. Evaluation and mitigation of radiation-induced soft errors in graphics processing units. *IEEE Transactions on Computers*, 65(3):791–804, 2016. 31, 34

[93] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011. 31

[94] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004. 33

[95] Marc Benito, Matina Maria Trompouki, Leonidas Kosmidis, Juan David Garcia, Sergio Carretero, and Ken Wenger. Comparison of gpu computing methodologies for safety-critical systems: An avionics case study. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 717–718, 2021. 33, 34

[96] Olmedo, Ignacio Sañudo and Capodieci, Nicola and Cavicchioli, Roberto. A perspective on safety and real-time issues for gpu accelerated adas. In *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, pages 4071–4077, 2018. 34

[97] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, and Gagandeep S. Sachdev. Compute solution for tesla's full self-driving computer. *IEEE Micro*, 40(2):25–35, 2020. 34

[98] Junko Yoshida. Unveiled: BMW's Scalable AV Architecture. *EE—Times*, 2020. https://www.eetimes.com/unveiled-bmws-scalable-av-architecture/. 34

[99] Jyotika Athavale, Andrea Baldovin, and Michael Paulitsch. Trends and functional safety certification strategies for advanced railway automation systems. In *2020 IEEE International Reliability Physics Symposium (IRPS)*, pages 1–7, 2020. 34

[100] Jyotika Athavale, Andrea Baldovin, Ralf Graefe, Michael Paulitsch, and Rafael Rosales. Ai and reliability trends in safety-critical autonomous systems on ground and air. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 74–77, 2020. 34

[101] Irune Agirre, Jaume Abella, Mikel Azkarate-Askasua, and Francisco J. Cazorla. On the tailoring of cast-32a certification guidance to real cots multicore architectures. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–8, 2017. 34

[102] Hamidreza Ahmadian, Roman Obermaisser, and Jon Perez. *Distributed Real-Time Architecture for Mixed-Criticality Systems*. CRC Press, 2018. 34, 44

[103] Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J. Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multicore devices for safety-critical systems: A survey. *ACM Comput. Surv.*, 53(4), 2021. 34

[104] Guoqi Xie, Yanwen Li, Yunbo Han, Yong Xie, Gang Zeng, and Renfa Li. Recent advances and future trends for automotive functional safety design methodologies. *IEEE Transactions on Industrial Informatics*, 16(9):5629–5642, 2020. 34

[105] Fernando dos Santos, Luigi Carro, and P. Rech. Kernel and layer vulnerability factor to evaluate object detection reliability in gpus. *IET Computers & Digital Techniques*, 13, 09 2018. 34

[106] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 249–258, 2017. 34, 55

[107] Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, and F. Donelson Smith. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In *ECRTS*, 2018. 34, 79, 82, 94

[108] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017. 34, 82

[109] Daniel A. G. Oliveira, Paolo Rech, Heather M. Quinn, Thomas D. Fairbanks, Laura Monroe, Sarah E. Michalak, Christine Anderson-Cook, Philippe O. A. Navaux, and Luigi Carro. Modern gpus radiation sensitivity evaluation and mitigation through duplication with comparison. *IEEE Transactions on Nuclear Science*, 61(6):3115–3122, 2014. 34

[110] Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–10, 2017. 34

[111] Alessandro Vallero, Dimitris Gizopoulos, and Stefano Di Carlo. Sifi: Amd southern islands gpu microarchitectural level fault injector. In *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 138–144, 2017. 34

[112] Alessandro Vallero, Sotiris Tselonis, Dimitris Gizopoulos, and Stefano Di Carlo. Multi-faceted microarchitecture level reliability characterization for nvidia and amd gpus. In *2018 IEEE 36th VLSI Test Symposium (VTS)*, pages 1–6, 2018. 34

[113] Sotiris Tselonis and Dimitris Gizopoulos. Gufi: A framework for gpus reliability assessment. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 90–100, 2016. 34

[114] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, 2009. 34, 52, 55, 72, 79

[115] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020. 34, 52, 55, 79

[116] Atieh Lotfi, Saurabh Hukerikar, Keshav Balasubramanian, Paul Racunas, Nirmal Saxena, Richard Bramley, and Yanxiang Huang. Resiliency of automotive object detection networks on gpu architectures. In *2019 IEEE International Test Conference (ITC)*, pages 1–9, 2019. 35

[117] Mohammad Abdel-Majeed, Waleed Dweik, Hyeran Jeon, and Murali Annavaram. Warped-re: Low-cost error detection and correction in gpus. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 331–342, 2015. 35

[118] Josie E. Rodriguez Condia, Pierpaolo Narducci, M. Sonza Reorda, and L. Sterpone. A dynamic hardware redundancy mechanism for the in-field fault detection in cores of gpgpus. In *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 1–6, 2020. 35

[119] Martin Dimitrov, Mike Mantor, and Huiyang Zhou. Understanding software approaches for gpgpu reliability. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, page 94–104. Association for Computing Machinery, 2009. 35, 83, 109

[120] Abhyankar, Ameya V. . Performance-cost analysis of software implemented hardware fault tolerance techniques. Technical report, University of Wisconsin Madison, 2010. 35

[121] Jingweijia Tan and Xin Fu. Rise: Improving the streaming processors reliability against soft errors in gpgpus. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, page 191–200. Association for Computing Machinery, 2012. 35

[122] Marcio M. Goncalves, Ivan Peter Lamb, Paolo Rech, Raphael M. Brum, and Jose Rodrigo Azambuja. Improving selective fault tolerance in gpu register files by relaxing application accuracy. *IEEE Transactions on Nuclear Science*, 67(7):1573–1580, 2020. 35

[123] L. L. Pilla, P. Rech, F. Silvestri, C. Frost, P. O. A. Navaux, M. Sonza Reorda, and L. Carro. Software-based hardening strategies for neutron sensitive fft algorithms on gpus. *IEEE Transactions on Nuclear Science*, 61(4):1874–1880, 2014. 35

[124] Jack Wadden, Alexander Lyashevsky, Sudhanva Gurumurthi, Vilas Sridharan, and Kevin Skadron. Real-world design and evaluation of compiler-managed gpu redundant multithreading. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 73–84, 2014. 35, 83, 109, 121, 122

[125] Josie E. Rodriguez Condia, Marcio M. Goncalves, Jose Rodrigo Azambuja, Matteo Sonza Reorda, and Luca Sterpone. Analyzing the sensitivity of gpu pipeline registers to single events upsets. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 380–385, 2020. 35

[126] Jon Perez-Cerrolaza, Jaume Abella, Leonidas Kosmidis, Alejandro J. Calderon, Francisco Cazorla, and Jose Luis Flores. Gpu devices for safety-critical systems: A survey. *ACM Comput. Surv.*, 55(7), dec 2022. 35

[127] Nvidia jetson tx2: High performance ai at the edge. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/. 36, 48, 115

[128] Jongtaek Han, Michael Deubzer, Jin Park, Jens Harnisch, and Patrick Leteinturier. Efficient Multi-Core Software Design Space Exploration for Hybrid Control Unit Integration. In *SAE Technical Paper 2014-01-0260, 2014*, 04 2014. 37

[129] Samarjit Chakraborty and S. Ramesh. Guest Editorial Special Section on Automotive Embedded Systems and Software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1701–1703, 2015. 37

[130] Jon Perez, David González, Salvador Trujillo, Ton Trapman, and Jose Garate. A safety concept for a wind power mixed-criticality embedded system based on multicore partitioning. 05 2014. 37, 44

[131] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. The shift to multicores in real-time and safety-critical systems. In Gabriela Nicolescu and Andreas Gerstlauer, editors, *2015 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2015, Amsterdam, Netherlands, October 4-9, 2015*, pages 220–229. IEEE, 2015. 37

[132] Jyotika Athavale, Riccardo Mariani, and Michael Paulitsch. Flight Safety Certification Implications for Complex Multi-Core Processor based Avionics Systems. In Dimitris Gizopoulos, Dan Alexandrescu, Panagiota Papavramidou, and Michail Maniatakos, editors, *25th IEEE International Symposium on On-Line Testing and Robust System Design, IOLTS 2019, Rhodes, Greece, July 1-3, 2019*, pages 38–39. IEEE, 2019. 37, 39

[133] Geoffrey Blake, Ronald G. Dreslinski, and Trevor N. Mudge. A survey of multicore processors. *IEEE Signal Process. Mag.*, 26(6):26–37, 2009. 37

[134] Giulio Corradi. Tools, architectures and trends on industrial all programmable heterogeneous MPSoC (KeyNote). In *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS'17)*, 2017. 37, 42

[135] Ali Hayek and Josef Börcsök. Safety chips in light of the standard IEC 61508: Survey and analysis. In *2014 International Symposium on Fundamentals of Electrical Engineering (ISFEE)*, pages 1–6, 2014. 37

[136] Georg Macher, Andrea Höller, Eric Armengaud, and Christian Kreiner. Automotive embedded software: Migration challenges to multi-core computing platforms. In *13th IEEE International Conference on Industrial Informatics, INDIN 2015, Cambridge, United Kingdom, July 22-24, 2015*, pages 1386–1393. IEEE, 2015. 37

[137] Xilinx. Xilinx Zynq UltraScale+ MPSoC Data Sheet. https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf. 37

[138] Irune Agirre, Mikel Azkarate-askasua, Asier Larrucea, Jon Pérez, Tullio Vardanega, and Francisco J. Cazorla. Automotive Safety Concept Definition for Mixed-Criticality Integration on a COTS Multicore. In Amund Skavhaug, Jérémie Guiochet, Erwin Schoitsch, and Friedemann Bitsch, editors, *Computer Safety, Reliability, and Security - SAFECOMP 2016 Workshops, ASSURE,*

DECSoS, SASSUR, and TIPS, Trondheim, Norway, September 20, 2016, Proceedings, volume 9923 of *Lecture Notes in Computer Science*, pages 273–285. Springer, 2016. 37

[139] Viacheslav Izosimov, Antonis M. Paschalis, Pedro Reviriego, and Hans A. R. Manhaeve. Application-Specific Solutions. 2018. 37

[140] Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Comput. Surv.*, 52(3):56:1–56:38, 2019. 37

[141] Xavier Jean, Marc Gatti, Guy Berthon, and Marc Fumey. The Use of Multicore Processors in Airborne Systems (EASA 2011.C31). Technical Report. Technical report, EASA, Thales Avionics, 2011. 37

[142] Irune Agirre, Mikel Azkarate-askasua, Asier Larrucea, Jon Pérez, Tullio Vardanega, and Francisco J. Cazorla. A safety concept for a railway mixed-criticality embedded system based on multicore partitioning. In *CIT/IUC-C/DASC/PICom*, pages 1780–1787. IEEE, 2015. 37

[143] Ainara Bilbao, Irune Yarza, Jose Luis Montero, Mikel Azkarate-askasua, and Nera González Romero. A railway safety and security concept for low-power mixed-criticality systems. In *INDIN*, pages 59–64. IEEE, 2017. 37

[144] Jaume Abella and F.J Cazorla. *Harsh computing in the space domain*, pages 267–293. 12 2017. 37

[145] Jon Pérez, David González, Salvador Trujillo, and Ton Trapman. A Safety Concept for an IEC-61508 Compliant Fail-Safe Wind Power Mixed-Criticality System Based on Multicore and Partitioning. In Juan Antonio de la Puente and Tullio Vardanega, editors, *Reliable Software Technologies - Ada-Europe 2015 - 20th Ada-Europe International Conference on Reliable Software Technologies, Madrid Spain, June 22-26, 2015, Proceedings*, volume 9111 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2015. 37, 44

[146] Wikipedia contributors. Apple m1 — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Apple_M1&oldid=1064052900, 2022. [Online; accessed 26-January-2022]. 39

[147] Jim Salter. Intel's Alder Lake big.little CPU design, tested: It's a barn burner. https://arstechnica.com/gadgets/2021/11/intels-alder-lake-big-little-cpu-design-tested-its-a-barn-burner/, 2021. [Online; accessed 26-January-2022]. 39

[148] Wikipedia contributors. ARM big.LITTLE — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=ARM_big.LITTLE&oldid=1061988292, 2021. [Online; accessed 26-January-2022]. 39

[149] XILINX. Rockwell Collins Uses Zynq UltraScale+ RFSoC Devices in Revolutionizing How Arrays are Produced and Fielded: Powered by Xilinx. 2018. 39, 41

[150] Jingyi Bin, Sylvain Girbal, Daniel Gracia Pérez, Arnaud Grasset, and Alain Merigot. Studying co-running avionic real-time applications on multi-core cots architectures. 02 2014. 39

[151] Imanol Allende, Nicholas Mc Guire, Jon Pérez, Lisandro Gabriel Monsalve, Nerea Uriarte, and Roman Obermaisser. Towards linux for the development of mixed-criticality embedded systems based on multi-core devices. In *EDCC*, pages 47–54. IEEE, 2019. 39

[152] Sylvain Girbal, Xavier Jean, Jimmy Le Rhun, Daniel Gracia Pérez, and Marc GATTI. Deterministic platform software for hard real-time systems using multi-core cots. pages 8D4–1, 09 2015. 39

[153] Patrick Huyck. Arinc 653 and multi-core microprocessors — considerations and potential impacts. In *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pages 6B4–1–6B4–7, 2012. 39

[154] Santosh Kumar Jena and M. B. Srinivas. On the suitability of multi-core processing for embedded automotive systems. In *CyberC*, pages 315–322. IEEE Computer Society, 2012. 39

[155] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *RTSS*, pages 259–268. IEEE Computer Society, 2010. 39

[156] Sara Royuela, Alejandro Duran, Maria A. Serrano, Eduardo Quiñones, and Xavier Martorell. A functional safety OpenMP for critical real-time embedded systems. In *Proceedings of the International Workshop on OpenMP (IWOMP)*, September 2017. 39

[157] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. 40

[158] ITRS. International roadmap for devices and systems—executive summary. technical report. Technical report, ITRS, 2018. 41

[159] Sylvain Girbal, Miquel Moretó, Arnaud Grasset, Jaume Abella, Eduardo Quiñones, Francisco J. Cazorla, and Sami Yehia. On the convergence of mainstream and mission-critical markets. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, New York, NY, USA, 2013. Association for Computing Machinery. 41

[160] Riccardo Mariani, Gabriele Boschi, and Federico Colucci. Using an innovative soc-level fmea methodology to design in compliance with iec61508. In *2007 Design, Automation Test in Europe Conference Exhibition*, pages 1–6, 2007. 43

[161] Cobham Advanced Electronic Solutions. Leon3ft fault-tolerant processor. https://www.gaisler.com/index.php/products/processors/leon3ft. 44

[162] Dimitris Gizopoulos, Mihalis Psarakis, Sarita V. Adve, Pradeep Ramachandran, Siva Kumar Sastry Hari, Daniel Sorin, Albert Meixner, Arijit Biswas, and Xavier Vera. Architectures for online error detection and recovery in multicore processors. In *2011 Design, Automation Test in Europe*, pages 1–6, 2011. 44

[163] Daniel J. Sorin. *Fault Tolerant Computer Architecture*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009. 44

[164] Albert Meixner, Michael E. Bauer, and Daniel J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, 2008. 44

[165] Shidhartha Das. Razor: A variability-tolerant design methodology for low-power and robust computing. 01 2009. 44

[166] Alfons Crespo, Patricia Balbastre, José Simó, Javier Coronel, Daniel Gracia Pérez, and Philippe Bonnot. Hypervisor-based multicore feedback control of mixed-criticality systems. *IEEE Access*, 6:50627–50640, 2018. 44

[167] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143, 2012. 44

[168] Asier Larrucea, Imanol Martinez, Jon Perez, Vicent Brocal, Salva Peiró, Hamidreza Ahmadian, and Roman Obermaisser. Dreams: Cross-domain mixed-criticality patterns. 2016. 44

[169] C. Hilton and B. Nelson. A flexible circuit switched noc for fpga based systems. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 191–196, 2005. 44

[170] H. Ahmadian, R. Obermaisser, and J. Perez. Distributed real-time architecture for mixed-criticality systems. 2018. 44

[171] NVIDIA Corporation. Nvidia geforce gtx 1050 ti specifications. https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1050-ti/specifications/ and https://www.techpowerup.com/gpu-specs/geforce-gtx-1050-ti.c2885. [Online; accessed 11-April-2022]. 46, 52, 79, 82

[172] NVIDIA Corporation. Nvidia geforce gtx 1080 ti specifications. https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1080-ti/specifications/ and https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877. [Online; accessed 05-May-2022]. 48, 103

[173] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. pages 1–11, 2010. 48, 53, 79, 96, 100

[174] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007. 50

[175] EEMBC. EEMBC AutoBench Data Book. https://www.eembc.org/techlit/datasheets/autobench_db.pdf. 50

[176] Rapita Sytems Ltd. RapiCover [Online]. https://www.rapitasystems.com/products/rapicover. 51, 67

[177] NVIDIA. CUDA Toolkit 8.0. https://developer.nvidia.com/cuda-80-ga2-download-archive, Feb 2017. 52

[178] NVIDIA. Nvidia Pascal Architecture whitepaper. https://www.nvidia.com/en-us/data-center/resources/pascal-architecture-whitepaper/. 52

[179] GNU. GCC Documentation:3.10 Options for Debugging Your Program. https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html. 53

[180] NVLabs. Nvbitfi: An architecture-level fault injection tool for gpu application resilience evaluations. https://github.com/NVlabs/nvbitfi, 2020. 54, 106

[181] Timothy Tsai, Siva Kumar Sastry Hari, Michael Sullivan, Oreste Villa, and Stephen W. Keckler. Nvbitfi: Dynamic fault injection for gpus. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 284–291, 2021. 54, 106

[182] Oreste Villa, M. Stephenson, David W. Nellans, and S. Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019. 54, 106

[183] Fang, Bo and Pattabiraman, Karthik and Ripeanu, Matei and Gurumurthi, Sudhanva. GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 221–230, 2014. 55

[184] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. 56

[185] Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the eembc benchmark suite. *IEEE Micro*, 29(5):18–29, 2009. 57, 117

[186] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. pages 751–766, 03 2018. 60, 69, 82, 109

[187] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, abs/1410.0759, 2014. 60, 67

[188] Francisco J. Cazorla, Jaume Abella, Enrico Mezzetti, Carles Hernandez, Tullio Vardanega, and Guillem Bernat. Reconciling time predictability and performance in future computing systems. *IEEE Design & Test*, 35(2):48–56, 2018. 61, 69

[189] Justyna Zander. Functional safety for autonomous driving. In *Proc. Auton. Veh. Mach. Conf*, 2017. 63

[190] MIRA Ltd. MISRA-C:2004 Guidelines for the use of the C language in critical systems. www.misra.org.uk, 2004. 64

[191] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016. 66

[192] NVIDIA. CUTLASS. https://docs.nvidia.com/cuda/cublas/index.html. 67

[193] NVIDIA. CUTLASS. https://devblogs.nvidia.com/cutlass-linear-algebra-cuda. 68

[194] R. Wilhelm et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36:1–36:53, May 2008. 68

[195] Adam Betts and Alastair Donaldson. Estimating the wcet of gpu-accelerated applications using hybrid analysis. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 193–202, 2013. 68

[196] E. Mezzetti, L. Kosmidis, J. Abella, and F. J. Cazorla. High-Integrity Performance Monitoring Units in Automotive Chips for Reliable Timing V&V. *IEEE Micro*, 38(1):56–65, January/February 2018. 69

[197] NVIDIA. Fermi. NVIDIA's Next Generation CUDA Compute Architecture. White paper., 2009. 73

[198] Imagination and Ambarella partner on Autonomous Vehicle human-machine interface visualisations with ASIL functional safety. 73

[199] Wikipedia contributors. Round-robin (document) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Round-robin_(document)&oldid=1045906114, 2021. [Online; accessed 1-April-2022]. 76

[200] Wikipedia contributors. Round-robin scheduling — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Round-robin_scheduling&oldid=1034376468, 2021. [Online; accessed 1-April-2022]. 76

[201] Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017. 79

[202] Jacob T. Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. The case for gpgpu spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, 2012. 83

[203] D. Black-Schaffer J. Janzen and A. Hugo. Partitioning GPUs for Improved Scalability. In *Proceedings - Symposium on Computer Architecture and High Performance Computing*, 2016. 83

[204] B. Wu et al. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the International Conference on Supercomputing*, 2015. 83

[205] M. Thazhuthaveetil S. Pai and R. Govindarajan. *Improving GPGPU Concurrency with Elastic Kernels*. 2013. 83

[206] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41, 2019. 83, 121, 122

[207] TESLA. Full Self-Driving Hardware on All Cars. https://www.tesla.com/autopilot. 85, 109

[208] IEEE. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. 90

[209] Nathan Whitehead and Alex Fit-Florea. Precision & performance: floating point and ieee 754 compliance for nvidia gpus. *rn (A+ B)*, 21, 01 2011. 90

[210] H. Naghibijouybari, K. N. Khasawneh, and N. Abu-Ghazaleh. Constructing and characterizing covert channels on gpgpus. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017. 92

[211] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on gpus. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014. 92

[212] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *2012 Innovative Parallel Computing (InPar)*, 2012. 93

[213] T. Allen. Improving Real-Time Performance with CUDA Persistent Threads (CuPer) on the Jetson TX2. Technical report, Concurrent Real-Time, March 2018. https://www.concurrent-rt.com/wp-content/uploads/2016/09/Improving-Real-Time-Performance-With-CUDA-Persistent-Threads.pdf. 93

[214] N. Capodieci and P. Burgio. Efficient Implementation of Genetic Algorithms on GP-GPU with Scheduled Persistent CUDA Threads. In *Proceedings - International Symposium on Parallel Architectures, Algorithms and Programming, PAAP*, 2016. 93

[215] V. Vlkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC*, 2008. 96

[216] Y. Yang et al. A unified optimizing compiler framework for different gpgpu architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(2), 2012. 96

[217] A. Magni et al. A large-scale cross-architecture evaluation of thread-coarsening. In *SC*, 2013. 96

[218] Bruce Merry. Faster gpu-based convolutional gridding via thread coarsening. *Astronomy and Computing*, 16, 05 2016. 96

[219] Nicolai Stawinoga and Tony Field. Predictable thread coarsening. *ACM Trans. Arhcit. Code Optim.*, June 2018. 96, 105

[220] Carles Hernandez and Jaume Abella. Timely Error Detection for Effective Recovery in Light-Lockstep Automotive Systems. *IEEE TCAD*, 34(11), 2015. 109, 120

[221] Carles Hernandez and Jaume Abella. Low-cost checkpointing in automotive safety-relevant systems. In *DATE*, 2015. 109, 120

[222] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. Optimizing software-directed instruction replication for gpu error detection. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 842–854, 2018. 109, 121

[223] Michael B. Sullivan, Siva Kumar Sastry Hari, Brian Zimmer, Timothy Tsai, and Stephen W. Keckler. Swapcodes: Error codes for hardware-software cooperative gpu pipeline error detection. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 762–774, 2018. 109, 121, 122

[224] Daniel A. G. Oliveira, Paolo Rech, Heather M. Quinn, Thomas D. Fairbanks, Laura Monroe, Sarah E. Michalak, Christine Anderson-Cook, Philippe O. A. Navaux, and Luigi Carro. Modern gpus radiation sensitivity evaluation and mitigation through duplication with comparison. *IEEE Transactions on Nuclear Science*, 61(6):3115–3122, 2014. 109

[225] European Processor Initiative. European Processor Initiative. https://www.european-processor-initiative.eu/, 2019. 111

[226] Apollo, an open autonomous driving platform. http://apollo.auto/, 2018. 117

[227] Brett H. Meyer, Benton H. Calhoun, John Lach, and Kevin Skadron. Cost-effective safety and fault localization using distributed temporal redundancy. In *2011 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 125–134, 2011. 121

[228] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. *SIGARCH Comput. Archit. News*, 28(2):25–36, may 2000. 120, 121

[229] S.S. Mukherjee, M. Kontz, and S.K. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *Proceedings 29th Annual International Symposium on Computer Architecture*, pages 99–110, 2002. 121

[230] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. *SIGARCH Comput. Archit. News*, 31(2):98–109, may 2003. 121

[231] Christopher LaFrieda, Engin Ipek, Jose F. Martinez, and Rajit Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 317–326, 2007. 121

[232] Jian Fu, Qiang Yang, Raphael Poss, Chris R. Jesshope, and Chunyuan Zhang. On-demand thread-level fault detection in a concurrent programming environment. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 255–262, 2013. 121

[233] Hyeran Jeon and Murali Annavaram. Warped-dmr: Light-weight error detection for gpgpu. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 37–47, 2012. 121, 122

[234] Ralph Nathan and Daniel J. Sorin. Argus-g: Comprehensive, low-cost error detection for gpgpu cores. *IEEE Computer Architecture Letters*, 14(1):13–16, 2015. 121, 122

[235] Florian Haas, Sebastian Weis, Theo Ungerer, Gilles Pokam, and Youfeng Wu. Poster: Fault-tolerant execution on cots multi-core processors with hardware transactional memory support. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 421–422, 2016. 121

[236] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A. Connors. Using process-level redundancy to exploit multiple cores for transient fault tolerance. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 297–306, 2007. 121

[237] Daniel J. Scales, Mike Nelson, and Ganesh Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *SIGOPS Oper. Syst. Rev.*, 44(4):30–39, dec 2010. 121

[238] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. Swift: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, 2005. 121

[239] Hwisoo So, Moslem Didehban, Yohan Ko, Aviral Shrivastava, and Kyoungwoo Lee. Expert: Effective and flexible error protection by redundant multithreading. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 533–538, 2018. 121

[240] Mohammad Shadi Alhakeem, Peter Munk, Raphael Lisicki, Helge Parzyjegla, Helge Parzyjegla, and Gero Muehl. A framework for adaptive software-based reliability in cots many-core processors. In *ARCS 2015 - The 28th International Conference on Architecture of Computing Systems. Proceedings*, pages 1–4, 2015. 121

[241] Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. Plr: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, 2009. 121

[242] Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. Efficient software-based fault tolerance approach on multicore platforms. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 921–926, 2013. 121

[243] Martin Dimitrov, Mike Mantor, and Huiyang Zhou. Understanding software approaches for gpgpu reliability. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, page 94–104. Association for Computing Machinery, 2009. 121, 122

[244] Vanessa Vargas, Pablo Ramos, Jean-Francois Méhaut, and Raoul Velazco. Nmr-mpar: A fault-tolerance approach for multi-core and many-core processors. *Applied Sciences*, 8(3), 2018. 121

[245] Xabier Iturbe, Balaji Venu, Juergen Jagst, Emre Ozer, Peter Harrod, Chris Turner, and John Penton. Addressing functional safety challenges in autonomous vehicles with the arm tcl s architecture. *IEEE Design Test*, 35(3):7–14, 2018. 120

Appendices

Example of Rodinia modifications

```
1  //... Redundant input (if needed) and output replication done above
2
3  //Create CudaStreams
4  #ifdef REDUNDANT
5  cudaStream_t streams[NUM_STREAMS];
6  for (int i = 0; i < NUM_STREAMS; i++)
7      cudaStreamCreate(&streams[i]);
8  #endif
9
10 //CUDA Kernels call
11 #ifdef REDUNDANT
12 solver_2<<<blocks, threads, 0, streams[0] >>>(workload, xmax, d_x, d_y,
       d_params, d_com, d_err, d_scale, d_yy, d_initvalu_temp,
       d_finavalu_temp);
13 #ifdef SERIALIZE
14 cudaDeviceSynchronize();
15 #endif
16 solver_2<<<blocks, threads, 0, streams[1] >>>(workload, xmax,
       d_x_redundant, d_y_redundant, d_params_redundant, d_com_redundant,
       d_err_redundant, d_scale_redundant, d_yy_redundant,
       d_initvalu_temp_redundant, d_finavalu_temp_redundant);
17 #else
18 solver_2<<<blocks, threads>>>(workload, xmax, d_x, d_y, d_params, d_com,
        d_err, d_scale, d_yy, d_initvalu_temp, d_finavalu_temp);
19 #endif
20
21 //Waiting until both kernels finished
22 #ifdef REDUNDANT
23 for (int i = 0;i < NUM_STREAMS; i++)
24     cudaStreamSynchronize(streams[i]);
25 #endif
26
27 //Get results back from GPU Memory to CPU memory
28 cudaMemcpy(x, d_x, x_mem, cudaMemcpyDeviceToHost);
29 cudaMemcpy(y, d_y, y_mem, cudaMemcpyDeviceToHost);
30 #ifdef REDUNDANT
31 cudaMemcpy(x_redundant, d_x_redundant, x_mem, cudaMemcpyDeviceToHost);
32 cudaMemcpy(y_redundant, d_y_redundant, y_mem, cudaMemcpyDeviceToHost);
33 #endif
34
35
36 //Checking the results of both executions
```

```
37 #ifdef REDUNDANT
38 int j, k;
39 bool correct = true;
40 for(i=0; i<workload and correct; i++){
41     for(j=0; j<(xmax+1) and correct; j++){
42         for(k=0; k<EQUATIONS and correct; k++){
43             correct = float_equals(y[i*((xmax+1)*EQUATIONS) + j*(
    EQUATIONS)+k], y_redundant[i*((xmax+1)*EQUATIONS) + j*(EQUATIONS)+k
    ]);
44         }
45     }
46 }
47 #endif
```

Listing 1: Myocyte code with the modifications required

Nvprof example

First, we list the options of the nvprof used together with a small description:

- *CUDA_VISIBLE_DEVICES=1:* To use the GPU 1 from the server, the GTX 1050 Ti.
- *–print-gpu-trace:* Print individual kernel invocations (including CUDA memcpy's/memset's) and sort them in chronological order. In event/metric profiling mode, show events/metrics for each kernel invocation.
- *–cpu-profiling on:* Turn on CPU profiling (Used to identify the function executed by the NVIDIA Runtime API)
- *–concurrent-kernels on* Allow concurrent kernel execution
- *–print-api-trace:* Print CUDA runtime/driver API trace.
- *–csv:* Format it in csv (easier to parse)
- *-o profiler.nvvp* Export the result file, which can be imported later or opened by the NVIDIA Visual Profiler

Next, we can see a template of the calling of the nvprof tool in Listing 2

```
nvprof nvprof_options binary binary_arguments
```

Listing 2: Template call of nvprof tool

Last, we can see an example of binary call together with the nvprof, its options and the binary arguments in Listing 3

```
CUDA_VISIBLE_DEVICES=1 usr/local/cuda-9.2/bin/nvprof --print
 -gpu-trace --cpu-profiling on --concurrent-kernels on
--print-api-trace --csv -o profiler.nvvp bin/myocyte.out 100
  1 1
```

Listing 3: Nvprof call used for the myocyte application

Modifications on the Tegra TX2

We start by showing the command to turn off the Denver cores, Listing 4.

```
echo 0 > /sys/devices/system/cpu/cpu1/online
echo 0 > /sys/devices/system/cpu/cpu2/online
```

Listing 4: Commands that turn off the two Denver cores

Next, we detail the commands to fix the frequency, at 2.04GHz, for the rest of the cores in Listing 5.

```
sudo cpufreq-set --cpu 0 -g Performance -u 2.04Ghz -d 2.04
 Ghz
sudo cpufreq-set --cpu 3 -g Performance -u 2.04Ghz -d 2.04
 Ghz
sudo cpufreq-set --cpu 4 -g Performance -u 2.04Ghz -d 2.04
 Ghz
sudo cpufreq-set --cpu 5 -g Performance -u 2.04Ghz -d 2.04
 Ghz
```

Listing 5: Commands to deactivate the frequency scaling by setting the maximum and minimum frequency equal