POLYTECHNIC UNIVERSITY OF CATALONIA

MASTER THESIS

SafeDM a light-lockstep approach

Author:
Francisco BAS JALÓN

Supervisor:
Dr. James SMITH

A thesis submitted in fulfillment of the requirements
for the degree of Master's degree in Electronic Engineering

in the

Research Group Name
Department or School Name

June 30, 2022

Declaration of Authorship

I, Francisco BAS JALÓN, declare that this thesis titled, "SafeDM a light-lockstep approach" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

Polytechnic University of Catalonia

Abstract

Faculty Name
Department or School Name

Master's degree in Electronic Engineering

SafeDM a light-lockstep approach

by Francisco BAS JALÓN

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

List of Abbreviations

LAH	**L**ist **A**bbreviations **H**ere
WSF	**W**hat (it) **S**tands **F**or

Physical Constants

Speed of Light $c_0 = 2.997\,924\,58 \times 10^8\,\mathrm{m\,s^{-1}}$ (exact)

List of Symbols

a	distance	m
P	power	W ($\mathrm{J\,s^{-1}}$)
ω	angular frequency	rad

For/Dedicated to/To my…

Chapter 1

Introduction

1.1 Motivation

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam ultricies lacinia euismod. Nam tempus risus in dolor rhoncus in interdum enim tincidunt. Donec vel nunc neque. In condimentum ullamcorper quam non consequat. Fusce sagittis tempor feugiat. Fusce magna erat, molestie eu convallis ut, tempus sed arcu. Quisque molestie, ante a tincidunt ullamcorper, sapien enim dignissim lacus, in semper nibh erat lobortis purus. Integer dapibus ligula ac risus convallis pellentesque.

1.2 Contribution

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam ultricies lacinia euismod. Nam tempus risus in dolor rhoncus in interdum enim tincidunt. Donec vel nunc neque. In condimentum ullamcorper quam non consequat. Fusce sagittis tempor feugiat. Fusce magna erat, molestie eu convallis ut, tempus sed arcu. Quisque molestie, ante a tincidunt ullamcorper, sapien enim dignissim lacus, in semper nibh erat lobortis purus. Integer dapibus ligula ac risus convallis pellentesque.

1.3 Structure of the Thesis

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam ultricies lacinia euismod. Nam tempus risus in dolor rhoncus in interdum enim tincidunt. Donec vel nunc neque. In condimentum ullamcorper quam non consequat. Fusce sagittis tempor feugiat. Fusce magna erat, molestie eu convallis ut, tempus sed arcu. Quisque molestie, ante a tincidunt ullamcorper, sapien enim dignissim lacus, in semper nibh erat lobortis purus. Integer dapibus ligula ac risus convallis pellentesque.

Chapter 2

Background

2.1 Faults, Failures and Errors

During this thesis, the common terminology in fault-tolerant systems add reference is employed:

Faults, failures and errors are abstract concepts that can be applied to different systems. Since the scope of this work is computing systems, we will restrict the provided examples to this kind of systems.

Any electronic system delivers a service that the user of that system perceives. This service comprises all the external states of the system. A service failure or system failure occurs when the delivered service (i.e. one or more external states) deviates from the correct service state. The correct service is defined by the functional specification of the system. A failure in safety-critical systems can endanger lives or produce high economic losses. Thus, the main goal of safety-critical systems is to minimize the probability of a system failure.

The deviation between the correct internal or external state and the real state is called an error. The cause of an error is called a fault. Thus, a fault is a defect within the system. A fault first causes an error in one of the components that form the system, altering the system's internal state. If this error propagates to the system's output altering the external state and the service provided, we will say that the error led the system to a failure. However, not all faults produce errors and not all the errors reach the external estate of the system producing a failure.

For instance, consider a two-inputs AND gate inside a system. If one gate input is '1' and the other is '0', the expected output will also be '0'. In this scenario, a fault that flips the input driving the '0' input to a '1' will produce an error because the output of the gate will be '1' instead of '0'. However, if a fault flips the other input from '1' to '0', the output will still be '0', the expected value.

Following the same logic, an AND gate, whose inputs are driven from two registers, one of them with an incorrect value (error), could correct the error preventing it from spreading to other registers and reaching the output of the system.

Faults can be classified into two main categories: Systematic faults that are related in a deterministic way to a certain cause and are avoidable by construction i.e. taking into account possible faults during the first step of the design or investing enough resources into verification and validation processes (examples....). Random faults that occur unpredictably following a probabilistic distribution and are unavoidable. This work focuses on addressing a method for handling Common Cause Faults (CCF) a especial type of random faults that will be explained later.

2.2 Safety Related Systems

Safety-critical systems are those systems that need to work properly because otherwise a failure or malfunctioning could put in jeopardy peoples life or health, produce losses in expensive equipment or environmental harm. For this reason, these systems must have mechanisms to lower the failure rates until they happen with a negligible likelyhood. For instance, in the standards of the aircraft industry an acceptable failure rate is

$$10^{-}9$$

accdintes per hour reference

Some erros, like systematic errors can be found and corrected during the development process or can be mitigated appliying some qualitative measures depending on the system integrity level SIL [libro?] that is desired. However, random faults could not be avoided and require especial mechanisms to prevent these faults from producing a system failure or at least to minimize the likelyhood of these happening until a reasonable stent.

Faults also can be classified into permanent, intermitent and transient faults[contanstinescu]. Permanent faults are those ones that produce irreversible physical changes to the hardware. Intermitent faults are those faults that appear in irregular intervals. Transient faults occur because of temporary environmental conditions.

Transient faults are also called soft errors, these faults alter the normal behaviour of the system momentaneously. Transient faults produce a loss of data but they do not produce any damage to the circuit. They are random by nature and they can appear at any time in some parts of the system causing a deviation from the espected behavior. Several sources of transient faults exists: neutron and alpha particles, power supply variations and interconnect noise, electromagnetic interference and electrostatic discharge. These sources can affect one or several transistors momentanially modifying their behavior. Loss of reliabitlity in digital safety-critical systems is produced mainly by transient faults [citar libro].

According to Moore's Law the number of transistors that fits in the same are increase by a factor of two every year. The industry has followed this trend many years and even though the trend is slowing down, transistors are samller every year. Incresing the number of transistors in a system also increases the probability of any of them experience a soft error. But also lower power voltages, higher frequencies and srinking transistors geometries make them more vulnerable to other sources of faults having a negative impact in the system relaivaility. For instance, higher frequencies and smaller interconect features increase the possibility of violatin the timiming sefety margins of the system. Also, lower voltages along with smaller transistors make systems more vulnerable against neutron and alpha particles. [Constantinescu]

Also, with smaller transistors, variations in the manufacturing process (in widths, lenghts, oxide thicknes, etc), are more likely to produce systematic failures in the integrated circuits.

Safety-critical systems must have the capacity of operate properly even in the presence of faults. The systems that integrate mechanisms to allow a correct operation even when faults appear are called fault-tolerant. Fault-tolerant systems must include two basic mechanisms: fault detection and recovery.

The system has to be equiped with components able to detect the errors and prevent them from propagaiting to other components. When the error is spoted this components alert the system to trigger a recovery mechanism that put the system in a previous error-free state.

When the sytem detects an error it can recover the last known error-free state, reset the system by powering off or enter in a safe state mode for example in the event of a permanent fault. However, detecting when a transient fault is ocurring in our system is not always easy. Reseting the system is not always possible and to recover to the last error-free state requires a mechanism to store those states.

2.3 Redundancy

Errors are detected usually employing redundancy. Redundancy is applied differenly for different parts of the system. For instance, Error Correction Codes (ECC) are employed to protect the stored data [reference sefede]. The data is encoded whit rendundant information which allows to detect errors. In the case of the computing elements two different kinds of diversity is employed:

time redundancy space redundancy

When time redundancy is applied, the same operation (an instruction or a set of instructions) is executed several times (more than once) in the same processing unit. On the other hand, space redundancy is achieved by replicating several times (more than once) a given processing unit that performs the same operation. In a free-error execution, all the outputs must coincide. Therefore, by comparing the different generated outpus, either by the different executions or in the different processing units, the system is capable of detecting possible errors.

When the different outputs do not coincide, the system has to activate a recover mechanism to restore the system to a safety state and re-execute from there. This safety state can be a previous free-error state stored in memory or could be achieved by dropping the taks if the time contraints allow it. For instance, a system executing a task with a small period (e.g. every 50 ms) such as braking and steering must perform the task before its Fault Tolerant Time Interval (FTTI) (e.g. 200 ms). In this example, the FTTI is big enough w.r.t the taks period to allow the system to drop the task and execute it all over again as long as two consecutive faults do not take place.

Time redundancy is usuful to proctect agains transien faults since it is very unlikely that a transient fault affect both consecutive executions in the same way, and the free-error execution output and the errouneous output will differ, allowing the comparator to detect the error. However, a premanent fault will cause the same error in both executions making impossible to the comparator to detect the error. On the other hand, space redundancy is more suitable to effectively detect permanent faults since it is likely that a systematic fault affect only one of the replicas of the hardware.

Space redundancy has a big area penalty becacuse of the replication of processing units, but the performance lost is negligible since all the operations are performed in parallel. On the ohter hand, time redundancy has no area overhead but the performance degradation is high since the same operation has to be performed serveral time secuentially.

Redundancy is based on the idea that the probability that all the replicas produce the same error is near zero. This is true most of the cases since most of the cases faults are independent. However, if the same cause is responsible for different faults in the different replicas, faults are not independent anymore and the preivous assumption is not valid.

2.4 Sphere of Replication

The Sphere of Replication (SoR) is the granularity at which the outputs of the replicated elements are compared to detect errors. In figure [] several levels of granularity are illustrated, from the finest granularity (a) to the most coarse one (d). Comparison between stages of the pipeline is not practical due to the huge hardware and performance overhead to perform the communication and the comparison between the stages of the cores each cycle. Instruction granularity can be applied as shown in [15] where the authors propose to execute redundant threads in a superescalar microprocessor with the capacity of executing several threads in different functional units. However, this approach is not extent of drawbacks since especial communication channels are required to perform the comparison at intruction-level granularity. On the ohter hand, a finer granularity means a faster error detection and faster and easier recover mechanisms.

However, most of the approaches used in the industry rely on a off-core-level SoR. In this case, only the data written to system memory or I/O interface is compared. This approach take advantage of the fact that a task finishes without errors when the service provided (state of the I/O and memory) is correct, and provided that the external states are correct, the internal states (e.g. in-core activity) can be ignored. Also, the overhead is much lower compared to the overhead of intrusction-level SoR. With off-core-level SoR, all the information required to perform the comparisson are the addresses and values sent by the cores through the communication-network. Therefore, snooping the information flowing through the communication-network is enough and the intrusivness of this solution is much lower. However, the time elapsed from a fault occurrence and the error detection is unbound. For instance, an error could be confined to the register file during most of the execution of a program without reachgin the communication-network. These is an issue for safety-critical real-time systems where each task has a strong time constraints.

2.5 Dependent failures and Common Cause Fauilures

Dependent failures are characterized by their ocurrence probability. Whereas the probability of the ocurrence of several failure events produced by indpenedent faults can be computed as the product of all their ocurrence probabilities, the ocurrence probability of dependent failures can not be modelled in the same way.

$P(A) \cdot P(B \mid A) = P(B) \cdot P(A \mid B) 6 = P(A) \cdot P(B)$.

Usually the probability of two dependent failures is bigger than the probability of two independent failures caused by independent faults. This issue has to be taken into account since redudancy is based in the assumption that the likelyhood of two replicas experiencing the same failure is virtually zero.

$P(A) \cdot P(B \mid A) > P(A) \cdot P(B)$

Concretely, Common Cause Failures CCF is a type of dependent failure that arise simultaneously in redundant elements from a single shared cause. In safety-ciritcal systems implementing space redundancy CCFs may cause the same failure in all the replicas. If this happen, the error detection mechanism will fail since both erroneus outputs will coincide. For this reason, CCFs are a hazard for safety critical systems and all fault tolerant systmes standars take into account the effects of CCFs.

For a CCF to occurs, the systems has to have at least two channels (two replicated elements). A CCF arises when a fault, that is the root case, spreads through a

coupling mechamism to all the channels of the system, figure[]. For instance, suppose a fault-tolerant tow-channel system (i.e. two redundant cores in the same die) where the cooling system fails. The root case is the failure of the cooling system. The heat is not disipated anymore and the temperature in the whole die increases. In this example, the thermal coupling becomes the copuling mechanism that is the responsible for the fault cause affecting both channels.

Notice that a CCF can be also caused by both systematic and random faults. For instance a common example is a soft error affecting clock logic of the two cores or a voltage droop. Defects on the design of the hardware of the replicas or in the manufacutring process (e.g. identical phsysically low gates in the replicas) could affect all the channels in the same way producing a CCF.

In figure [bla bla] the most relevant fault coupling mechanisms are shown.

Coupling by similar design or fabrication process: Usually same software and hardware is employed for all of the channels of the system (e.g. all the cores have the same design and the software they execute is also the same). Employing different hardware or software for each channel will daramatically increase the design and test costs. Therefore, a fault in the design or in the manufacuturing process must be prevented during the design process or detected during the testing or it will be missed by the fault-detection mechanism even if redundancy is applied.

Mechanical and Thermal coupling: The effects of the mechanical and thermal coupling are transmitted slowly over the SoC die compared to the processor operation speed. It is assumed that a CCF can arise only if the effects of the mechanical stress or the heating affect the same gate/transistor of all of the replicas at the same time. However, this is very unlikely due to the slow propagation of these physical effects. The most hazardous scenario would consist on a shor circuit producing a very focus and abrupt temperature increase. This could result in a CCF if the heat source is situated if the affected point present a perfect simetry w.r.t the affected gates/transistors. It is assumed that mechanical stress affects the whole die and therefore it does not represent a CCF risk.

Electromagnetic coupling: Electromagnetic coupling can affect the layout when the pahts of the cores act as antenna for the electromagnetic field. This could result in voltage changes that could produce soft errors in both replicas. If the layout of all the replicas is identical, the effect of the electromagnetic fields is very likely to be similar in all the channels of the system. However, the small dimension of the VLSI works protect them against frequencyes below 100GHz, since the circuit pahts only work as antennas for higher frequencies. PCB traces are much more vulnerable against electromagnetic fields. Therefore, it is high ulikely that electromagnetic fields produce a CCF in a integrated circuit.

Electrical coupling: Usually both cores share the power supply and some signals such as the clock. Therefore, a perturbation in the power supply, e.g. voltage droop or noise or a soft error in the clock logic can affect analogously all the replicas of the system. Unlike the mechanical or thermal coupling, electrical coupling affect the whole system concurrently, and therefore, in case of similar effects, a CCF which can not be detected by the comparator unit of the fault detection mechanism will ocurr. Thus, diversity is not enough to prevent system failures when a fault affect the different replicas in the same way and extra measures have to be applied.

Notice that even though single event upsets (SEUs) caused by particle radiation are one of the major causes of failures in electric sytems [14], they usually effect a very small area (usually a single register or cell RAM) making very unlikely for them to produce a CCF.

To protect the system against CCF and be compliant with the safety standars, the redundant elements in the system have to show diversity among them. Diversity be acieved in different ways:

Design diversity diversity which consist of achieving hardware procuring the same functionality but implemented in a different way. Design diversity can be achieved at different abstracion levels. For instance, the same functionality can be performed with two different architectures, or the same architecture can be implemented employing different gate libraries or a different technology. Design diversity is very effective against preventing transient and systematic faults to cause a CCF since is very unlikely that any fault affect analogously all the replicas of the system. However the cost of implementing two diverse components is associated with a considerable increase in the design costs.

Time diversity is reached ensuring that the redundant software executions in the replicas are staggered i.e the redundant cores are executing always different instructions and thus their internal states are always different. Being the internal states different, a transient fault affecting all the cores analogously will produce different results and thus, the errors will be detected by the fault-detection mechanism. Appliying time diversity imply lower costs and design efforts. However, time diversity is vulnerable against systematic faults that affect analogously all the recplicas.

2.6 Lockstep execution

Lockstep architecture consist of two identical processor implemetantions that execute the same software with some staggering between the cores. Lockstep execution provide a mechanism to detect CCF by implementing diverse redundance in the system. Two different apporaches for achieving a locksteped execution are shown in the next section.

2.6.1 Lockstep schemes

Tight hardware-based lockstep execution:

We can see this approach implemented in the Infineon AURIX family [referencia. In a hardware-based lockstep implementation two identical instances of the same processor are employed. Both processors execute the same instructions with a small time shift of N cycles (usually 2 or 3 cycles). The output of both processors are compared, introducing redundancy in the system. Also, since both cores have some cycles of staggering by desing, time diversity is introduced in the system protecting it against CCFs.

Both cores assume a different roll in this technique. The core that goes ahead in the program execution will be the head core while the core that is behind in the execution will be the trail core. As shown in the figure[figura lockstep], the inputs are the same for both cores although the trail core needs to queue the inputs in a buffer for N cycles. The external requests (data load/store, interrupts, etc) of the head core are stored in a buffer during N cycles until the trail core performs the same requests. Requests of both cores are compared before leaving the core complex. In the event of a mismatch the error is detect and the proper recorer mechanisms can be applied.

As mention before, errors can be only detected once they leave the core complex and are visible at the outputs. Therefore, any fault can go undetected for an arbitrary number of cycles which dependens only in the workload. Some mechanisms can

be applied to reduce the error time detection as described in paper jaume. In this work is proposed to periodically send the file registers values through the system communication network to detect errors before they reach to the ouputs of the cores.

This lockstep implementeation hides its complexity to the user that precives everything as one single processor, easeing the development process. However, for the same reason, the performance of halve since both processors can not be used to execute different non-critical tasks. At the same time, this approach is very intrusive and important hardware modificaitons are required to implement this solution in a couple of cores.

Light-weight software-based lockstep execution:

This approach is proposed in referencia 3. The idea is to create diverse redundancy at software level by running a program twice on different cores. To implement this approach no hardware modifications are needed and thus the intrusiveness of this approach is null in hardware terms and it can be implemented with Commerical off-the-shelf (COTS) products.

In this approach three cores are required: one for monitoring the execution of the critical taks and ensure that the minimum required staggering between the redundant executions is maintained, and two identical cores to perform the redundant execution of the critical task. The monitor core executes the monitor thread which is in charge of schedule the redundant processes in the two different cores and periodically obtain the number of executed intructions by each redundant core (#instr in the figure). The monitor only allows to the trail core to make progress if staggering (#instr_head - #instr_trail) is bigger than a given threshold TH_stag. This condition is checked by the monitor every T_check cycles. TH_stag and T have to be selected such as if the trail core execute in T the maximum possible number of instructions while the head core is stalled the staggering still is bigger than zero. Only when the head core finishes the execution of the critical task, the monitor allows to the trial core to run without exercising any controll over it.

The monitor thread is in charge of keeping a proper staggering between both redundant execution to procure time diversity. However, the monitor do not provide a mechanism to detect faults during the execution. This checking mechanism has to be implemented in software so execution results are compared after task execution is complete.

The core executing the monitor thread is unprotected against CCFs. Therefore, its execution requires a hardware-based lockstepping. The thread monitor is also in charge of handling the inputs and the outputs of the system. Two different readings of an input from the redundant processors at different times could result in different read values which would make the results of the execution differ.

This apporach achieves a lockstepped execution with a null intrusiveness in hardware terms. However, there is a trade-off between the period T_check and the minimum staggering allowd TH_stag. The smaller is the T_check period the bigger is the computational overhead that the thread monitor experiences. Also, the smaller is the T_check period, the trail core can execute less number of instructions in that period, and the TH_stag is smaller. This trade-off is usually resolved choosing a staggering equal to the number of instruction that the core can execute in 100μs. Therefore, the biggest drawback of this approach is its high staggering.

Notice that with the light-weight approach and namely with the light-weight software-based approach, both cores can execute either the same redundant task when the level of criticallity is high, ensuring a safe execution or two different non-critical tasks. Therefore, this approach doubles the performance when non-critical tasks are executed.

2.7 Other Fault detection Approaches

Chapter 3

SafeDE

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

3.1 SafeDE Motivation

3.2 Architecture

SafeDE is built on the light-weight lockstepping concept. As expalined in the section referencia sección this approach has already been implemented in software referencia paper sergi. However, the implemented software light-ewight lockstep needs a third core running the monitor thread and the frequency at which the monitor is able to retrieve the executed intruction, compute the staggering and apply corrective measures is low. As a consecuence the long feedback loop impose a large staggering.

SafeDE is a tiny hardare module that monitors the execution of the redundant cores imposing some staggering to reach time diversity. As shown in image imagen referncia SafeDE collects every cycle the intructions executed by the two cores (#instr_head and #instr_trail) and computes the staggering that exists between both cores (instr_head - instr_trail). If the head core is not at least TH_stag instructions ahead of the trail core, SafeDE will raise the trail core stall signal that freezes its pipeline registers (registers keep the same value). The stall signal will be set to zero whenever the head core makes enough progress so the staggerin is again bigger than TH_stag instructions.

TH_stag is the minimum staggering (in terms of number of instructions) that SafeDE enforces between both cores. Typically, TH_stag has a low value. For instance, a value bigger than the pipeline stages can be chosen (e.g. 10 instructions) to ensure that the content of both pipelines is always completele different. Hence, the enforced staggering by SafeDE is much smaller compared to the on enforced by the software approach, and it is comparable to the hardware-based lockstep execution staggering. Since SafeDE is implemented directly on hardware is capable of computing the staggering and stalling any of the cores every cycle overcoming the light-weight lockstepping limitations while maintaining its advantages.

As in the light-weight software-based lockstep execution, a software mechanism has to be implmented to compare the results of the executions once they finish.

Notice that execution of the redundant critical tasks have to be independet. For that porpuse, each critical task has to be allocated in a different memory space address. SafeDE is controlled by means of internal registers. Each core has to configure SafeDE once it reaches the critical section that needs to be protected. Configuration of SafeDE register is perform using the corresponding driver. An API is also needed

to schedule both redundant processes to the corresponding cores in case that critical task are runing on top of a operative system. Later in the section añadir seccion these software components are described both in the contex of bare metal and Linux integrations.

Also note, that neither of the cores has predefined the roll of head or trail core. The first core indicating SafeDE that has reached the critical section assumes the head core roll while the other core assumes the trail roll.

3.3 Features and limitations analysis

In this section the main SafeDE limitations and features are presented. They are also compared against the limitations and features of the software-only approach and the tight-lockstep approach.

SafeDE features:

* Low cost: The light-weight software-based lockstep approach needs a third monitor to run the monitor thread. On the contrary, SafeDE is a tiny hardware module that monitors the execution preventing the system from needing a third core. SafeDE implementation require few resources. In a SoC that integrates four RISC-V cores, SafeDE employs only x% of the FPGA resources of the SoC. A more detailed view of the FPGA resources employed by SafeDE is shown in the section add section.

* Low staggering: SafeDE checks every cycle the executed instructions by both cores and computes the staggering. Having such a short feedback allows SafeDE to impose very small staggeringns (e.g. few instructions 10-20). On the contrary, as disscussed, the software-only solution needs staggering values of many thousands of instructions.

* Flexibility: SafeDE can be easily enable and disable through its configuration registers. Therefore, SafeDE can be used at very fine granularity. However, the granularity is dictated by duda

* Low intrusiveness: SafeDE needs a few signals from the cores internal pipeline. First, it needs a signal that indicates SafeDE each time that a new instruction is commited. Second, it needs a signal that provides a mechanism to stall the pipelin of the core when the signal is risen. These modifications are much smaller compared with the modifications that a tight lockstep implementation would require. The software-only implementation does not require any hardware modification, but, unlike SafeDE, it may require modification in the operating system to allow reading the instruction count of the redundant cores from the monitor thread.

SafeDE limitations:

* Non-null intrusiveness: Even though the hardware modifications required by SafeDE are light, they are not null, and unlike the software-only solution, SafeDE can not be built with COTS products.

* Limited applicability: Light-weight lockstepping (either software or hardware) assumes that both redundant executions follow identical instruction streams. Both the hardware and the software-only approach relay on the count of the executed intructions by both cores, if the execution paths diverge, different number of executed instructions would not prevent both cores from exposing the same internal state. This limitation restricts the use of the light-weight lockstepping approach to programs whose control path is deterministic. This restricts SafeDE from being used in applications that make decistions based on random variables and in parallel programs that for instance could differ in the number of executed instructions due to the synchronization primitives. Also, programs performing I/O operations would

perform these I/O operations twice at different time instants. These could affect the functionality of the system or two different values could be read causing different results between redundant executions. Note that these limitations apply for light-weight lockstepping and thus they do not apply only to SafeDE but also to the software-only solution.

* Limited diversity: SafeDE allows to force time diversity between two redundant cores forcing a staggered execution. However, even though SafeDE protects the system against some key CCFs, those CCFs whose coupling channel is related to the hardware design or fabrication process (e.g. identical physically weak gates in both cores) will represent a hazard for the system. The system only can be protected against these CCFs appliying other types of diversity such as layout diversity. As mentioned in the section referencia sección, this kind of diverstiy only can be reached by different hardware designs of applaying different designs at any of the abstraction layers of the ASIC design.

* SafeDE hardening: SafeDE is also suceptible to single faults. For instance, a transient fault could affect SafeDE in such a way that the fault propagates to the ouput leading SafeDE to a failure. If this is the case, both cores could reach the same internal state being vulneable in the event of a CCFs. To preven this situation, SafeDE must be hardened replicating the tight lockstepping scheme shown in Figure add figura, but substituting the cores with SafeDE.

Scope of applicability:

As mentioned before, light-weight lockstepping (either hardware or software) has limited applicability (e.g. same instructions streams or no I/O operations). Therefore, two redundant cores coupled by SafeDE can be employed to execute critical code regions rather than entier programs. For instance, a in a multicore system implementing eight cores, two must implement tight hardware-based lockstepppping to handle the I/O operations while the rest can be coupled with SafeDE. With this configuration the system is capable of executing several combinations of lockstepped and non-locksstepped tasks. Variying from 4 critical tasks to 1 critical task and 6 non-critical tasks. Therefore, the user sees 7 cores instead of 4 cores that would see if all the cores were coupled with a tight lockstepping mechanism. This limits the user that only is able to execute four tasks regardless its level of criticallity.

3.4 N-modular redundancy

For this work we have developed, implemented and assessed SafeDE in the context of dual-modular redundancy (DMR). However, SafeDE concept can be easily extended to N-modular redundancy. Some domains (e.g. avionics or medical domains) could require a safety level that is only achieved through 3-modular redundacy or even 5-modular redundancy.

In order to extend SafeDE functionality to a system needing N-modular redundancy N-1 SafeDE modules are required. For instance, in the case of triple-modular redundancy (TMR), two SafeDE modules would couple the cores 1 and 2 (SafeDE 1-2) and the cores 2 and 3 (SafeDE 2-3). In this escenario, assuming we want core 1 to be ahead in the execution, core 1 must be the first one indicating SafeDE 1 that it has reached the critical section, becoming the head core. The core 2 would be the second one entering the critical section, becoming the trail core w.r.t core 1, and the head core w.r.t the core 3. Finally core 3 would enter the critical section the last, becoming the trail core w.r.t the core 2. This scheme can be extended for N-modular redundancy, each core i will always exibit a staggering > TH_staggering w.r.t the

core i+1. Note that unlike in DMR, in N-modular redundancy provided N>2, the order in which redundant cores access the critical section must be controlled.

Figure add reference shows the concep of flexible N-modular redundancy in a 8-core multicore setup. Seven (N-1) SafeDE modules are developed to pair all the consecutive cores in the system. By activating the appropiate SafeDE modules we can obtain eny possible combination of N-modular redundancy. For instances, as shown in the figure, TMR is implemented in the cores 1-3 while two core couples (4-5 and 7-8) exibit DMR. Finally, core 6 run independently. This is achieved activating a given subset of the implemented SafeDE modules. In Figure referencia activated SafeDE modules are the blue-colored ones while the inactive ones are the black ones.

3.5 SafeDE Implementantion and Integration

SafeDE has been implemented and tested in two MultiProcessor

We have integrated SafeDE in two different multiprocessor platforms based on CAES Gaisler RISC-V NOEL-V cores. In this section both platforms are described and detailed information of the SafeDE hardware implementation and integration is provided. Later, is explained how SafeDE should be configured. Finally, SafeDE software integration is explained both for a bare metal setup and for a platform running Linux.

3.5.1 De-RISC and SELENE Platforms

citar paper derics

De-RISC platform: The derisc platform is developed in the scope of a european project motivated by the lack of high performance Multiprocessor System on a chip (MSPSoC) suitable for space applications. Most of the existent platforms do not supply the necessary performance required by spacecrafts, are not reliable enough and do not complaint with the safety requirements for space applications or face export restrictions like the use of propeitary Instruction Set Arquitectures (ISA).

The project tries to overcome these limitations by adopting multicore processors in the space domain that provide the required performance but face some chanllenges related with space safety regulations, predictability and reliability. To avoid export limitations and propietary ISAs the platform is based on the open source RISC-V ISA.

As a proof of concept, we have integrated SafeDE in the De-RISC industrial space MPSoC based on CAES Gaisler RISC-V NOEL-V cores. This platform is composed of different reusable IP cores developd by CAES Gaisler. Those IP cores are contained in the GRLIB IP library, which is an integrated set or reusable IP cores designed for SoC development. The IP cores have a common on-chip bus interface. SafeDE has been added to the GRLIB IP library as another reusable IP core. The library includes cores for AMBA AHB/APB control. The library is provided under the GNU GPL license.

The platform in which SafeDE is integrated and evaluated instanciates 2 Gaisler's NOEL-V 64-bit cores. NOEL-V cores implement the RISC-V ISA, they are dual-issue, and implement a 7-stages pipeline. Both cores have a private L1 data and instructions caches. The IP cores are centered around several AMBA AHB and APB busese. Cores are connected to the L2 cache and other peripherals through a 128-bit AMBA Advanced High-performance BUS (AHB). A 32-bit low-bandhwidth Advanced Peripheral Bus (APB) is instanciated for components requiring low bandwidth and is

connected to the main AHB Bus through a AHB/APB bridge. The L2 cache is shared for all of the cores and it connected to the Memory controller through another 128-bit AHB bus.

SELENE platform: SELENE is another European project that focuses on developing a High-Performance Computing (HPC) Multicore platform which is capable of delivering the computitation capabilities needed by autonomous systems in safety-critical domains such as space, avionics, robotics and factory automation. HPC platforms impose some difficulties when being certified for safety-critical systems due including support for functional and timing isolation, as well as testability. The project tries to overcome these limitations developing a open-source Safety-critical Cognitive Computing Platform (CCP) with self-awareness, self-adapting, and autonomous capabilities.

SELENE computing platform builds upon a combination of a multicore and accelerators, that will be prototyped on a FPGA SoC, so that they can be extended and upgraded. SELENE platform is also developed using the IP CORES from GRLIB IP CAES Gaisler and other GPL IP cores developed by project partners. SafeDE is one of the IP cores envised to be integrated in the SELENE platform to be constraint with the certification requirements for the different safety-critical domains.

SELENE platform is very similar to the De-RISC platform since the basic IP cores used to build the platform are implemented from the GRLIB IP library. However, SELENE platform offers more complexity since it instanciates more NOEL-V cores and shared L2 cache interconnects with a Network on a Chip including several Artificial Intelligence (AI) accelerators.

Although, results and conclusions of this work come from the experiments perfromed in the De-RISC platform, it is not integrated in the official version of the De-RISC project and the integration was made only as a research exercise. However, SafeDE it is integrated in the SELENE platform as part of the final delivered platform.

3.5.2 Hardware Implementation and Integration

add AMBA protocol in anexex or bibliography Since SafeDE only has to inferface with the system to be configured with its internal registers, it demands low banwidth and for to avoid unnecesary complexity is connected to the system through an APB interface. As shown in figure add reference, SafeDE is built in three independent hardware layers developed in VHDL.

* The first layer (the inner one) called "staggering_handler" instanciates SafeDE's core functionality. This layer is in charge of calculate the instructions executed by each core, compute the staggering and asserting the stall signal when needed. This layer is completely portable since it is compeletely independent of the platform interface. * The second layer implements the APB protocol. Configuration and statitical registers are implemented in this layer. This layers allows to the AHB masters (mainly cores) to modify SafeDE internal registers. The values stored in the internal registers act as inputs for the staggering_handler layer. * Finally the outter layer is an AMBA APB wrapper. This wrapper converts the defined APB AMBA types from the GRLIB IP library to standard VHDL types. In this way, the first two layers are completely portable for any platform implementing APB interface.

As mentioned previously, SafeDE needs an internal signal from the pipeline of the cores that provides information about how many instructions each core has executed. SafeDE also needs a signal capable of stalling the cores, to stop their progress whenever the staggering is smaller than the set threshold.

To provide the information related to the executed instructions, we employed a 2-bit signal "inst_cnt" from the pipeline. Each of the bits of the signal provides information of each issue of the core. Whenever the issue commits one instruction, the corresponding bit is set to '1' during one cycle. Each core is modified so this signals are fed to SafeDE as an input. With these signals, SafeDE can compute the executed instructions by both cores from a given time instant. Although the De-RISC and the SELENE platforms implement dual-issue NOEL-V cores, SafeDE can be configured through a generic to work with cores with an arbitrary number of issues.

To stall the pipelines, the output stall signals from SafeDE and ORed with a signal driven in the caches controller. This signal is fed to the pipeline as an input and whenever is risen, it prevents the values in the registers of the pipeline to update. Therefore, when the stall signal is risen, all the activity in the pipelin is stoped until the signal is set to '0'.

Although it is not strictly needed to protect against CCFs, SafeDE also can be configured to set a maximum staggering threshold TH_stag_max. Whenever the staggering between both cores is bigger or equal than the configured threshold (#instr_head - #instr_trail >= TH_stag_max) , the head core is stalled. This is useful to reduce the error detection time when intermedate results of the execution are compared between both cores.

The figure add reference shows an image of the De-RISC platform with SafeDE controlling the staggering of both cores. As shown in the figure add reference, SafeDE has 3 32-bit configuration registers (Configuration, CritSec1 and CritSec2) plus several statistic registers to gather execution information.

The bit 31 of the configuration register is used to perform a reset. When this bit is set to 1, all the SafeDE registers are set to their initial value except for the configuration register that keeps it value except for the bit 31 which is set to '0' again. The bit 30 of the configuration register is anded to the output stall signals. Thus, SafeDE is totally neutral and uncapable of perfoming any action over the pipeline if this bit is not set to '1'. The rest 30 bits are used to configure a maximum and a minimum thresholds for the staggering (15 bits each). Whenever the staggering is bigger or equal than the maximum threshold or smaller or equal than the minimum threshold, the trail or head core will be stalled in order to keep the staggering within the stablished limits.

The first bit of the register CritSec1 should be set to '1' when the core 1 enters in the code region needing lockstepped execution. From the moment in which this bit is set to '1', SafeDE will start counting the instruction executed by the core 1. Anlogously, the core 2 will set the first bit of the register CritSec2. The count of executed instructions will be reset for both cores once the first bit of both CritSec registers are set to '0'. Two separate registers (one for each core) are dedicated to indicate SafeDE the execution start of the region needing proctection. If two different bits of the same register are employed, writes from different cores could override the other core's bit.

SafeDE statistic registers gather information of: total number of cycles SafeDE has been active, number of executed instructions by each core, number of times each core has been stalled during the execution, number of cycles each core has been stalled during the execution and maximum, minimum and average staggering during the execution.

talck about the annex for the SafeDE documentation

3.5.3 Configuration and Operation

When using SafeDE the first step is to configure the desired staggering thresholds TH_stag_max and TH_stag_min. In case the TH_stag_max is not configured leaving the reset value which is 0, TH_stag_max will be internally set close to its maximum possible value (32750). Afeter configuring the staggering thresholds, the next step is to set to '1' the activation bit (bit 32 of the configuration register) otherwise it will not be able to stall the piplines. The first core reaching the code section that needs lockstep protection will set its CritSec register first bit to '1'. From that moment, this core will assume the head roll and SafeDE will start counting each commited instruction. Later, the second core will reach the code region needing protection and it will set again its CritSec register assuming the trail roll. Once both cores have set their CritSec registers, SafeDE will check that the staggering is within the limits (TH_stag_min < staggering < TH_stag_max) and will stall any of the cores if needed until the previous condition is met.

Once the head cores has finished the exeuciton, it will set its CritSec register bit to '0'. SafeDE will let the trail core finish the execution without intervention. Once the trail core finishes the execution it will set its CritSec register bit to '0' analogously to the head core. When both CritSec registers first bits have value '0', SafeDE will set the instruction counts of both cores to 0.

3.5.4 Software Integration

3.6 SafeDE Evaluation

This section asseses SafeDE by different methods in the context of De-RISC MPSoC. To perform the evalution process we simulated the VHDL model of the MPSoC including SafeDE IP using the simulator QuestaSim. We also synthesized the platform into a Xilinx Kintex UltraScale KCU105 evaluation kit and performed several tests.

3.6.1 Functional Validation

Several strategies are applied to validate the correct SafeDE functionality:

*Testbench:

The first approach to validate SafeDE is by means of a VHDL testbench that simulates the behavior of the system generating SafeDE inputs and reacting to its outputs. For that porpuse, a component simulating the behavior of the cores is developed. This component randomly generates the inst_cnt signals for both cores, randomly setting its bits to '1' as if the cores where committing instructions. If the stall signal of one core is set to '1' this prevents the inst_cnt signal bits of that core to be set to '1' as it would do in a real core.

A part from this component, at the top of the testbench, two procedures are define to simulate APB read and write transactions. This allows to configure the internal SafeDE registers during the simulation. As mention before, SafeDE has some statistic register that can be read trhough the APB interface. During the testbench execution, the same statistics are computed at the top of the testbench so they can be compared with the results recover from the APB interface.

Therefore, the testbench execution starts configuring internal SafeDE registers. After that, the testbench run for a configurable number of cycles simulating two cores executing instructions that hold when the stall signals are risen. Finally, SafeDE is stopped thorugh the configuration registers and the statistics are retreived. The

testbench pass provided that the statistics read from SafeDE registers and the statistics computed in the top of the testbench coincide and the staggering keeps all the time between the limits (TH_max_stagg > stag > TH_min_stag)

The testbench completition is the first step for SafeDE design validation and it also ease the development process since the testbench is automatically run each time a new Git push is performed, informing of potential errors each time the design is modified.

*RISC-V ISA tests: add reference to isa tests https://github.com/riscv-software-src/riscv-tests

RISC-V ISA tests are a group of test designed by the Universtity of California to test the correct functioning of the RISC-V ISA instructions. The ISA tests are written in single assembly language file and contain a self-checking code to test the result. Each ISA test tests one operation from the RISC-V ISA forcing corner cases occurrence.

To load the binaries into the platform, control the execution flow and debugging the application we used GRMON. GRMON is a general debug monitor for the LEON (SPARC V7/V8) processor, NOEL-V (RISC-V) and for SOC designs based on the GRLIB IP library developed by CAES Gaisler. add reference to GRMON

We modified the asembly of the RISC-V ISA tests to configure and activate SafeDE prior to the test execution. We also performed some modifictions to allow the exeuction in two different cores. The linker script is modified to load two ISA tests in two different addresse spaces. GRMON is emploed to activate both cores and determine the correct entry point. Once the test has finished, the results are stored into one register file register in each core. Those registers are read using GRMON and it is checked that the test passed.

This process is automated via Makefiles (for the compilation), Bash scripts and GRMON TCL scripts. Therefore, a simple linux command compiles the binaries, uploads the binaries to the FPGA, controls the exectuion flow and checks the results for all of the selected RISC-V ISA tests. The correct completition of these tests proves that neither the internal modifications performed to the integer pipeline of the cores nor the SafeDE action produce any system malfunctioning.

TACLe Benchmarks: TACLe Benchamrks suite is a set of self-contained and open source benchmarks of variying types and sizes. They are specifically designed for the evaluation of cirtical real-time embedded sytems. Since they are self-contained they have their inputs hardcoded together with the source code making them perfect for experimenting in a bare metal setup. Their execution times range from thousand of cycles to a few millions of cycles, making them in many cases easy to simulate and debug to understund unexpected results.

We ported some TACLe Benchmarks to RISC-V ISA and added some C funcion form the bare metal drive to configure SafeDE and gather execution statistics. As with the RISC-V ISA tests TACLe Benchmarks have also a self-checking function that performs a comparison between the obtained results (or any kind of signature summarizing the final result) and the expected results. TACLe Benchmarks are executed over the FPGA loaded with the De-RISC platform. Execution is controlled agian by GRMON. All the compiation, loading and result checking processes are automated using different scripts.

We checked for all the executed TACLe Benchmarks that the results with SafeDE forcing 20 instructions of staggering (TH_stag_min = 20; TH_stag_max = 32750) coincide with the expected results. Also, results from the statistic registers are gathered for every benchmark. We checked that in every execution that the number of executed instructions by both cores coincide and that the minimum staggering reached

during the execution is never below that the expected one (stag >= TH_min_stag = 20).

Actually, since a 1-cycle latency exists between the moment the stall signal is asserted until the core pipeline stops making progress, the former expresion does not hold during the benchmarks execution. Since the cores are dual-issue they can commit two instructions in one cycle being the worst escenerio the one in which with a existing staggering of 21 instructions, the trail core commits two instruction while the head core commits none. When this happens, the staggering reaches 19 instructions, and trail core stall signal is risen. Since that signal will take effect the next cycle, the trail core can commit another two instructions while the head cores commit again none, leaving the staggering at 17 instructions. The same happens for TH_max_stagg. The real minimum staggering that is kept is dependent of the architecture, namely on the number of issues of the core and the number of cycles that elapses from the moment the stall signal is asserted until it stalls the pipeline.

3.6.2 Fault Injection

3.6.3 Time Overhead

3.6.4 Hardware Costs

We have employed the Vivado 2018.1 toolchain to syntheize and generate the bistream targeting the Xilinx UlstraScale KCU105 Evaluation Kit featuring the Kintex XCKU040-2FFVA1156E FPGA. SafeDE implementation required 261 LUTs and 417 registers. Those numbers are really low compared with the resources required by each core (approximately 38,000 LUTs and 17,000 registers) or with the hardware resources required by the whole MPSoC (approximately 114,000 LUTs adn 74,000 registers). Thus, SafeDE hardware costs are negligible, namely just 0.23% of the LUTs and 0.56% of the registers of the whole MPSoC are employed to implement SafeDE. Hardware overhead could be limited even more by removing the statistics registers which are only useful for debugging porpuses.

add image of the pie maybe?

3.7 Conclusions

Chapter 4

Conclusions and Future Work

Appendix A

Published Work

Write your Appendix content here.