Mixed-Criticality Systems on COTS MPSoCs

Ph.D. Dissertation by Stefano Esposito
Advisor: prof. Massimo Violante
-- Commission --
prof. Marco Di Natale – referee
prof. Alberto Bosio – referee
prof. Graziano Pravadelli
prof. Paolo Bernardi
prof. Maurizio Rebaudengo

EMBEDDED SYSTEMS

Embedded systems are everywhere

- Including aircrafts, cars, medical equipment…

Some applications cannot fail under any circumstance

- or else, someone might get hurt or fired

Dependability is the justified trust in the correct behavior of a system

It is described by Reliability, Availability, Maintainability, Safety, Testability

THREATS TO DEPENDABILITY

Design errors

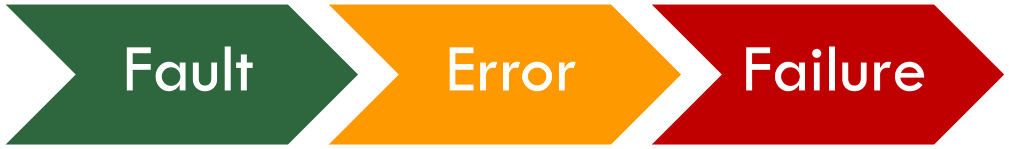
Users error

Environmental interferences

A threat may cause a fault

The fault may evolve in an error

When the error reaches the external interface, there is a failure or misbehavior



Your PC ran into

handle, and now

You can search for the error onli

SAFETY

Safety is a property of a system

- The property of being protected from harm or non-desirable outcomes in general
- It can be negated by faults and subsequent failures

Several standards deal with safety and safety-critical systems

- ISO26262
- DO-178C and DO-254
- ECSS standards
- …

SAFETY CRITICAL SYSTEMS

A system is safety critical if its failure can produce serious harm to its user or to the environment

It is often involved in the control of a physical system

It <u>must</u> meet a given deadline.

- Hard real-time (HRT): catastrophic consequences to a deadline miss
- Firm real-time (FRT): non-catastrophic consequences, but results are useless
- Soft real-time (SRT): just service degradation; results usefulness decrease with time but is not immediately 0

DEALING WITH SAFETY: ISO26262

Specific for the automotive industry

Defines the Automotive SIL (ASIL)

Each *item* has an assigned ASIL based on

- Severity: what happens if the item fails
- Controllability: how the driver can control the outcome of a failure
- Exposure: how probable is a failure

To each ASIL corresponds a set of guidelines for hardware and software development and testing

DEALING WITH SAFETY: ECSS STANDARDS

European Cooperation for Space Standardization

Define standards for design and development of space applications

- Safety requirements
- RAMS analysis
- FMEA
- Testing

Separate standards for both software and hardware

DEALING WITH SAFETY: DO-178C

DO-178C for avionic Software

It is the most relevant for this work

It defines a set of Design Assurance Levels (DALs)

The DAL is conceptually similar to ASIL

The real difference is the certification

- Must be provided by a third party
- Hard and expensive process

DEALING WITH SAFETY: ARCHITECTURES

Federated Architecture

- Each functionality has its own computer
- Virtually no resource sharing
- SWaP waste

Integrated Modular Avionics (IMA)

- Functions can share the same hardware platform
- Time multiplexing
- Supported by the ARINC-653 API standard
- Single core computers

MIXING CRITICALITIES

Functions sharing the same hardware can have different DALs

Reduce SWaP by increasing sharing through TDM

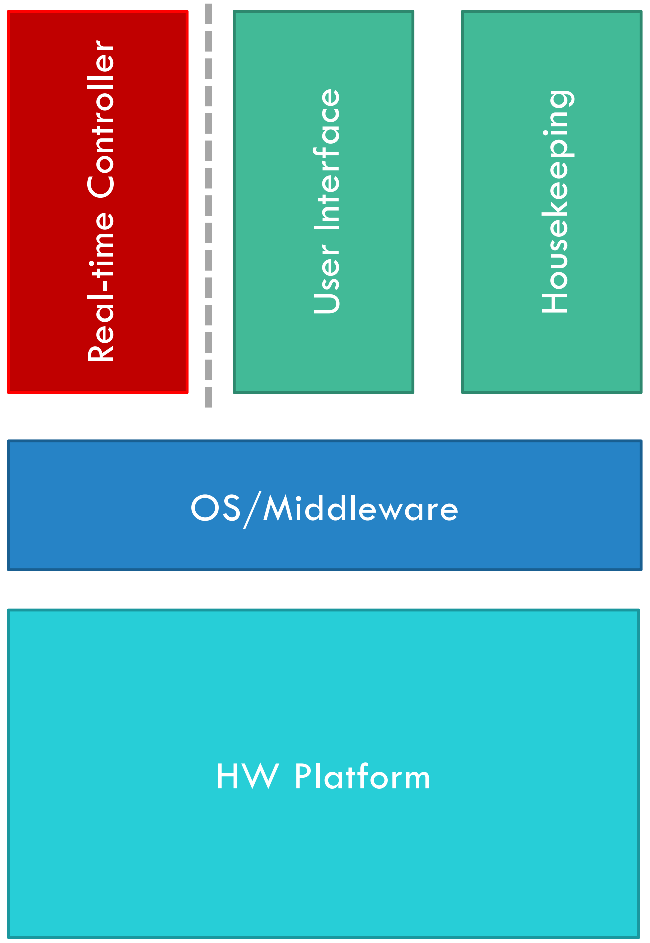
- More space for payload or less fuel consumption

Hard to prove safety

- low criticality should not interfere with high criticality
- low criticality has a higher probability of being affected by bugs

Enters the MPSoC

- ability to simultaneously process several workloads of different criticalities
- further reduction of SWaP!
- **Even harder to prove safety!**



MCA AND RESOURCE CONTENTION

When several application use the same resource the main issue is contention

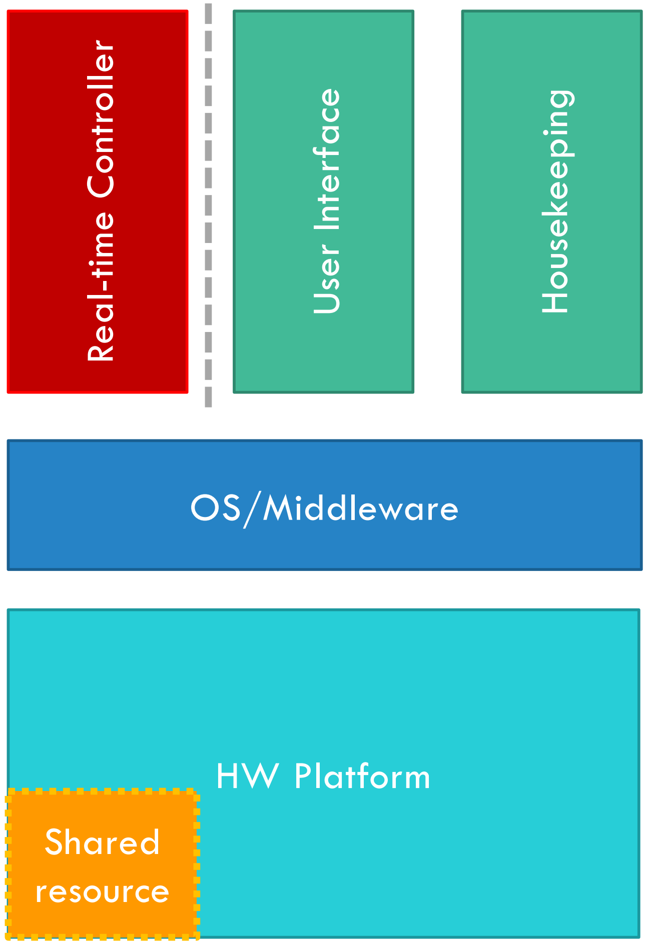
In a safety-critical application, contention must be bounded

- Otherwise WCET estimation is very difficult and imprecise

When considering mixed-criticality other issues arise

- Applications at low criticality might be subject to errors and corrupt data and resources used by high criticality applications
- WCET is not the only concern!

Resource partitioning in both space and time must be granted in order to achieve a certifiable architecture



DIVIDE ET IMPERA

Resource Partitioning in space is a very different problem with respect to resource partitioning in time

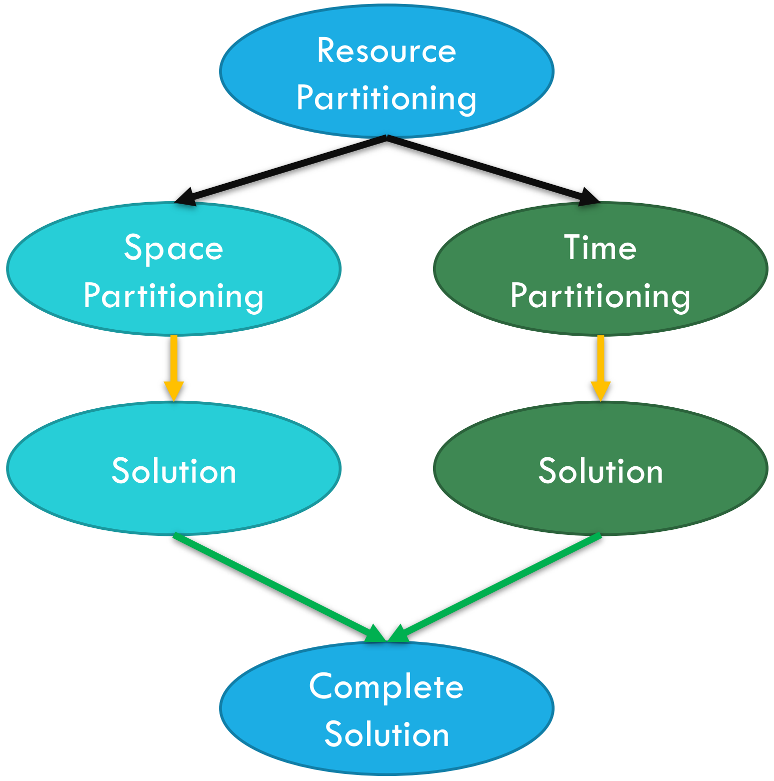
The two issues can be solved separately and linearly composed

Resource Partitioning in space

- Enforce isolation of the data used by each application

Resource Partitioning in time

- Ensure that no interference in the execution time can ever result by an application abusing a shared resource



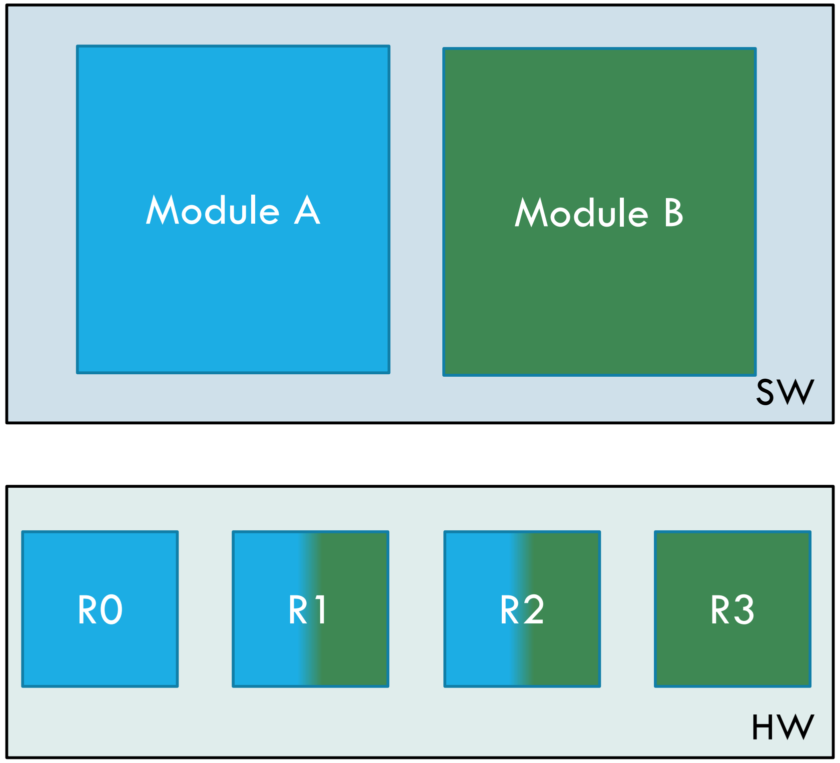
SPACE PARTITIONING

Each application has its own set of reserved resources

It does not concern the temporal aspects of sharing, just the spatial

- Data provided by any resource should not be corrupted by misbehaviors in other modules

Space partitioning should ensure that no application can corrupt data belonging to a different application



SPACE PARTITIONING VIOLATIONS

A faulty module can corrupt resources used by other modules

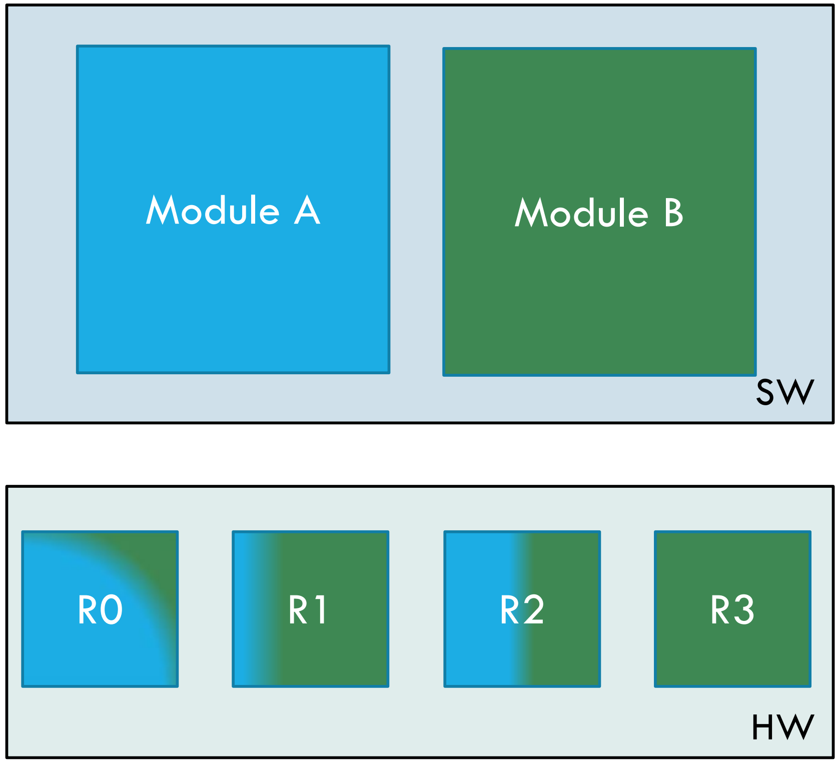
Shared resources are most vulnerable

- The module can use the resource, thus it can modify the data

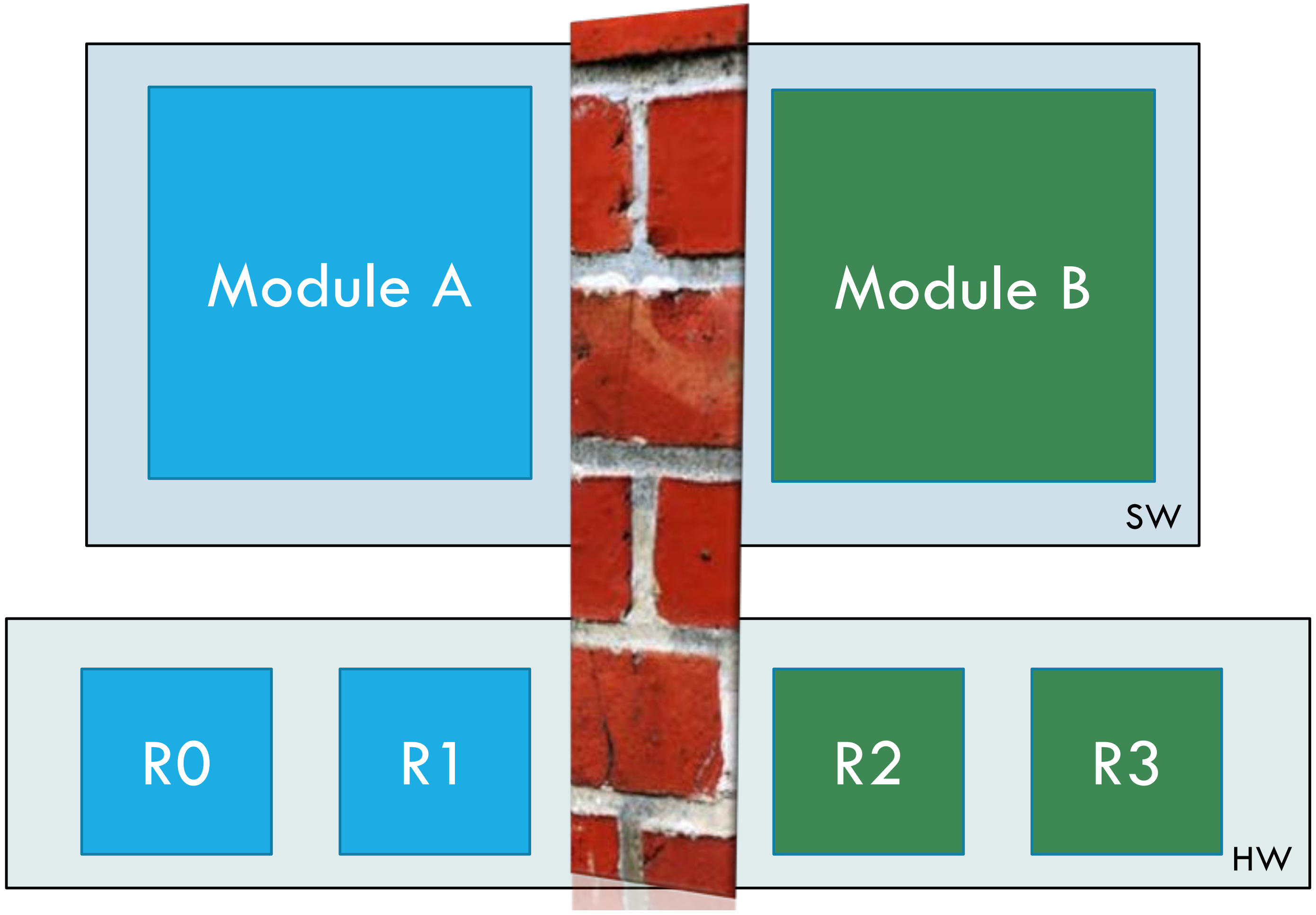
This includes also changing resource configuration

- Change configuration of the memory controller
- Change source/destination address of a DMA controller

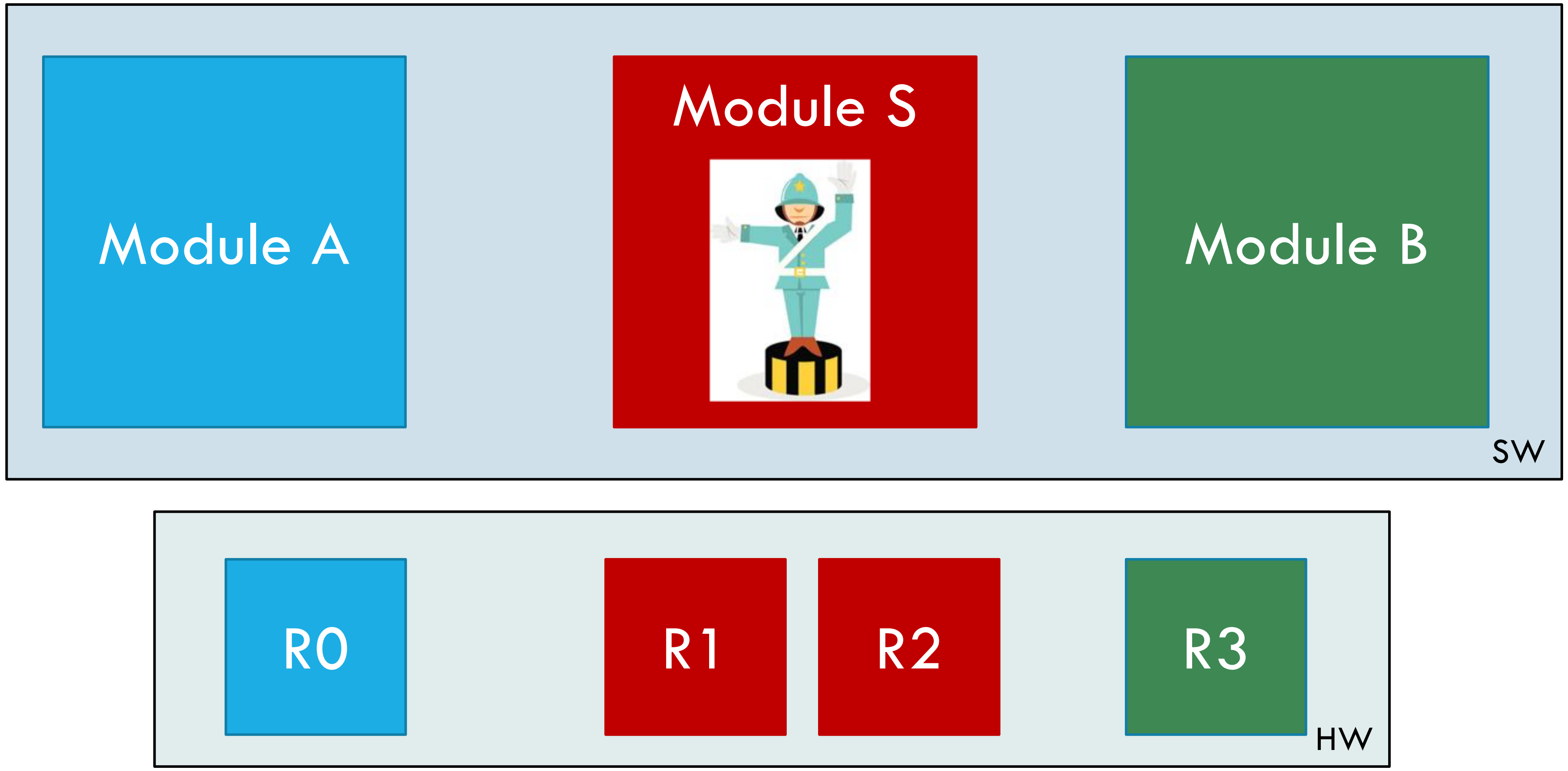
Shared memory is the easier victim



HOW TO AVOID VIOLATIONS: NO SHARING!



HOW TO AVOID VIOLATIONS: CONTROLLED SHARING



TIME PARTITIONING

Absence of interference on the execution time must be granted among different applications

- Ensure that WCET estimation is meaningful

Hard to enforce when the applications run on the same CPU

Even harder when applications are not at the same criticality level

- Different guarantees about correct behavior

Mandatory for safety-crtical hard-real-time applications

TIME PARTITIONING: VIOLATIONS

Shared resource abuse

- Access latency increases
- Unexpected delays
- WCET estimation is no longer valid
- Timing violation!

Safety Critical HRT applications are very sensible to this

Must ensure no violations can happen

SCHEDULING MCS

First solutions proposed more than 10 years ago

- Vestal, 2007

Solve mixed-criticality issues by better system scheduling

- Originally on single-cores
- later extended to MPSoCs

Some assumptions are not directly applicable to actual systems

- oversimplification of rules and standards
- high criticality does not necessarily imply high priority

SCHEDULING MCS

It is hard to estimate WCET on MPSoC

- Issue shared by all scheduling approaches
- Any scheduling approach for real-time systems rely on a WCET estimation

Some attempts have been made to provide a better WCET for MPSoC

- For instance: isWCET by Nowotch and Paulitsch, 2015
- Based on the increased access latency due to parallel accesses

This approaches only work under a bug-free assumption

- Something is needed to cope with possible bugs

MONITORING

Runtime safety can be enforced by monitors

A monitor is a device that observe a subset of the states of the system

Incorrect behavior is detected if states differ from expectations

Expectations can be set by profiling the system or by model-based approach

- Use a model of the system to evaluate intermediate internal states to be monitored
- Use the same model to design the actual monitor

WHAT'S NEW?

This work proposes a new comprehensive system architecture for MCAs

Based on the concept of partitioning

- Space partitioning
- Time partitioning

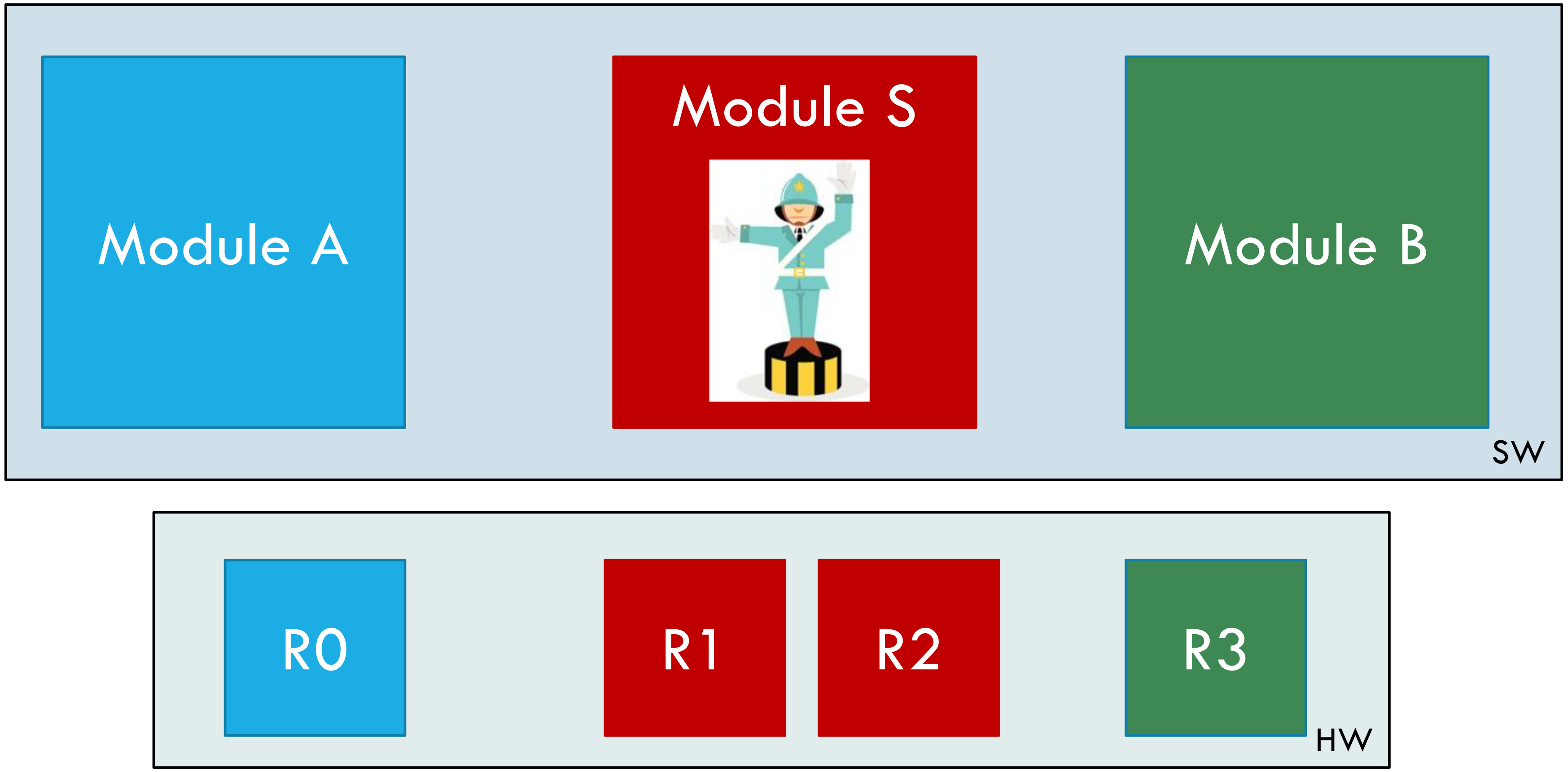
Able to ensure absence of interference among applications sharing the same hardware.

Designed to be used with COTS MPSoC platforms

- Tested on the Zynq and the i.MX6Quad

Certification is the final objective

HOW TO AVOID VIOLATIONS: CONTROLLED SHARING



TYPE-1 HYPERVISORS

Can be used in the controller role

Hardware abstraction layer

Virtual Machines (Resource Partitions)

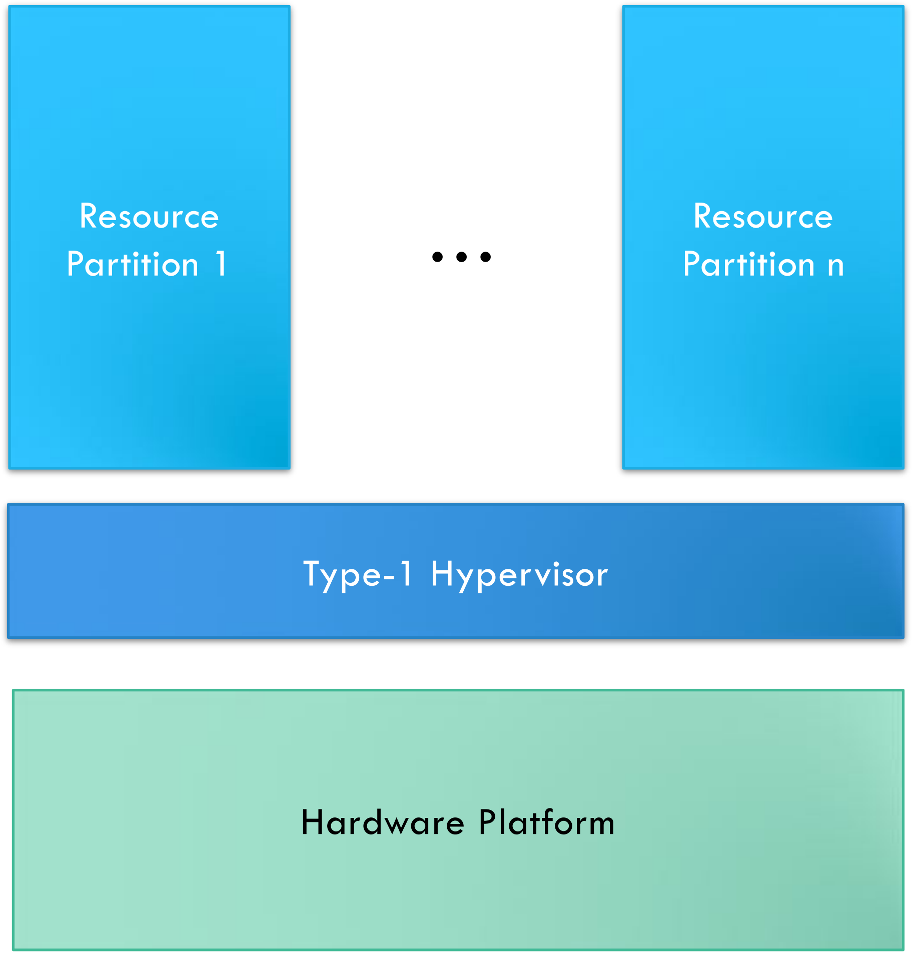
- Each application sees only the resources it uses

Similar to AMP but better…

- Easier to integrate, easier to synchronize (if needed)

…unless something goes horribly wrong

- SMP is subject to common mode errors



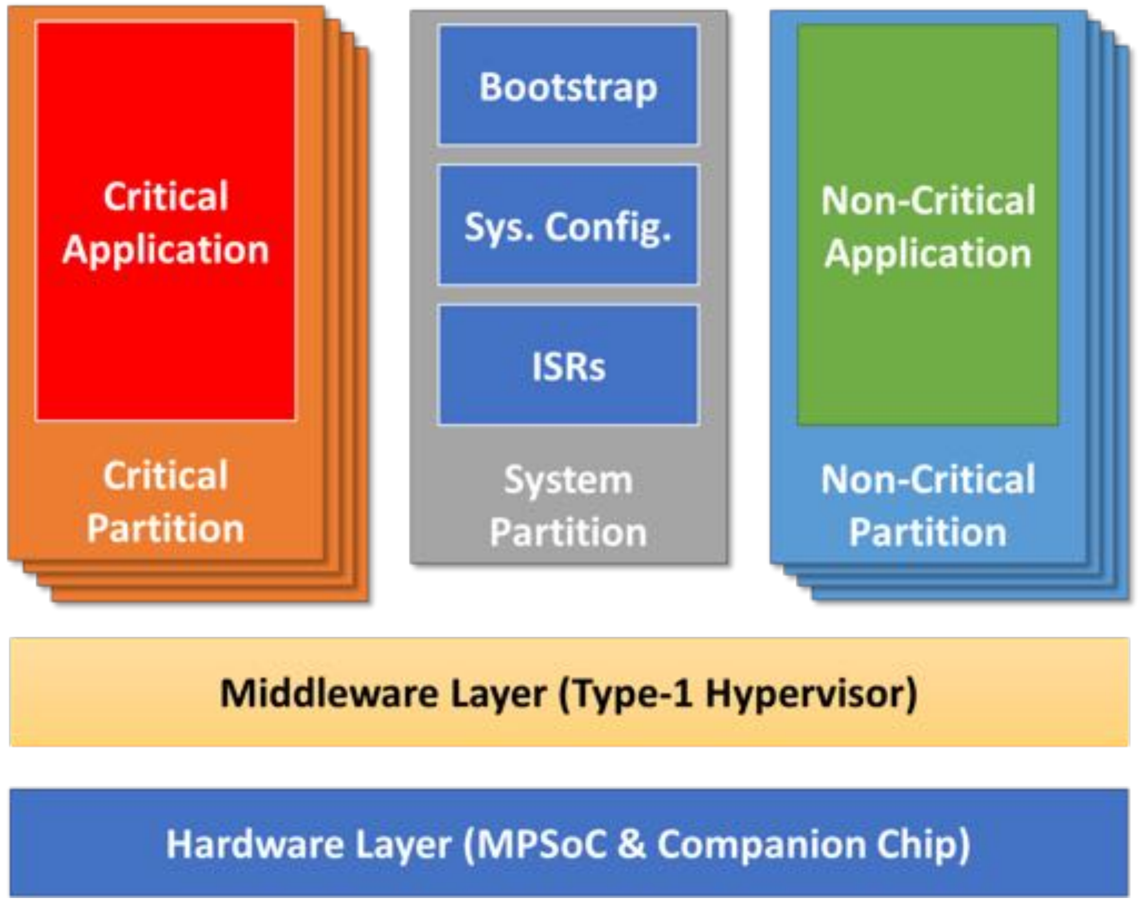
SPACE PARTITIONING SOLUTION

Use a Type-1 Hypervisor to define resource partitions

Each partition runs a separate application
- Each partition has a set of reserved resources
- Resource sharing can only happen at the middleware layer through provided IPC mechanisms

The system partition contains bootstrap and configuration code

ISRs run in the system partition



TEMPORAL PARTITIONING

It is hard to prove temporal partitioning a priori

More convenient to prove safety at runtime

- Even if an application misbehaves, the system shall survive

The proposed architecture is based on multiple monitors

- Monitor performance metrics for fast response
- Monitor execution flow for CFEs
- Monitor overall time to react to functional interruptions

PERFORMANCE MONITOR UNIT

Hardware available in most MPSoCs

- Can have different names, PMU is the ARM implementation

Can be used to monitor a set of performance metrics

- Including cache hit/miss, stall cyclces, data write/read…

Usually can monitor more than one metric at the same time

- For instance, Cortex-A9's PMU can monitor up to 6 metrics

Mostly used during application profiling

COMPUTE THRESHOLDS

To detect temporal interference not all metrics are good

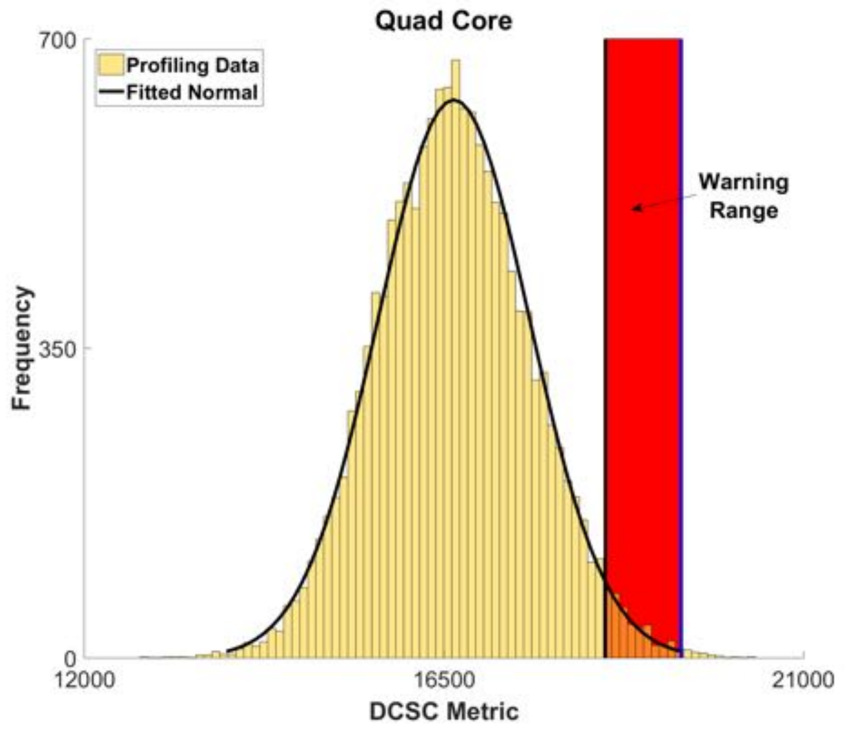
- Must be sensible to interference
- Must be measurable at runtime

An interference metric should be selected

- Can be composed by multiple metrics

Once the metric has been selected, thresholds should be computed

- Profile the metric
- Perform statistical analysis
- Extract the thresholds



USE THE THRESHOLDS

Detection Threshold

- If the metric is above this, something is going horribly wrong

Warning Range

- If the metric is in this range, it can be symptom of an error, or it may not…

Counter Threshold

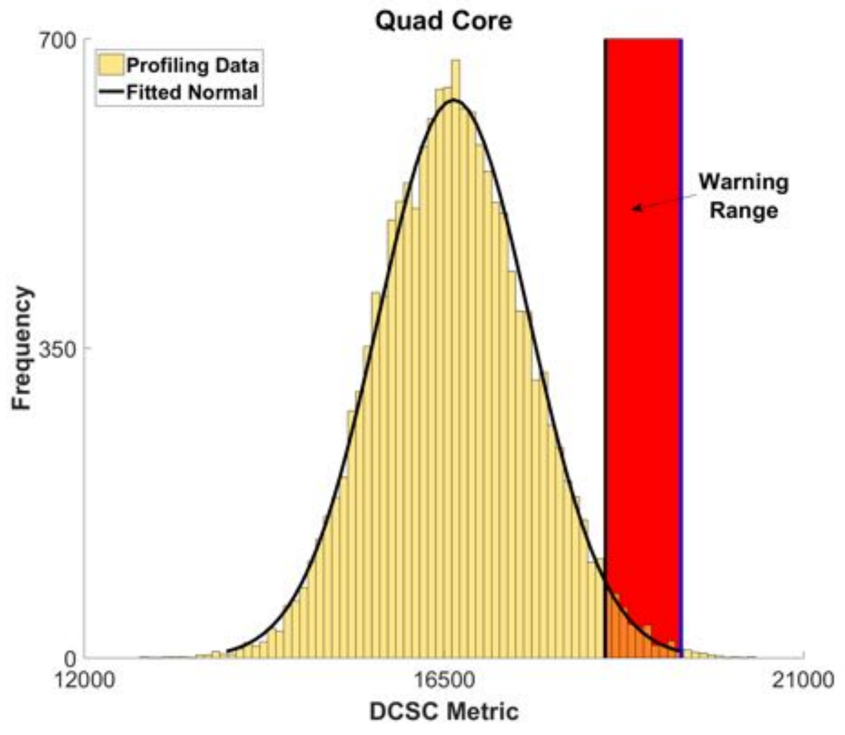
- If the metric is in the warning range more than this many times, something is going wrong

Panic Rule: violation of the detection threshold

- Reset, switch to hot stand-by spare

Warning Rule: violation of the counter threshold

- Graceful degradation



IMPLEMENTATION DETAILS

Different granularities

1. Core level
   - Monitor activities of a core
   - No OS support strictly needed
2. Task level
   - Monitor each task separately
   - Different tasks sharing one of the cores
   - OS should include PMUs in the task context registers and save/restore them on context switch

IMPLEMENTATION DETAILS

The PMU is a hardware unit available in the ARM Cortex-A9
- Similar units (with different names) are available in most MPSoCs

It must be configured at bootstrap to measure any selected metrics
- Specific ASM instructions are available for this
- A driver can be added to the OS to manage the PMU

The PMU can trigger an IRQ when it reaches a threshold
- ISR implementing the recovery action for the panic rule

Value of the PMU can be read by software
- To implement the warning rule

WATCHDOG PROCESSORS

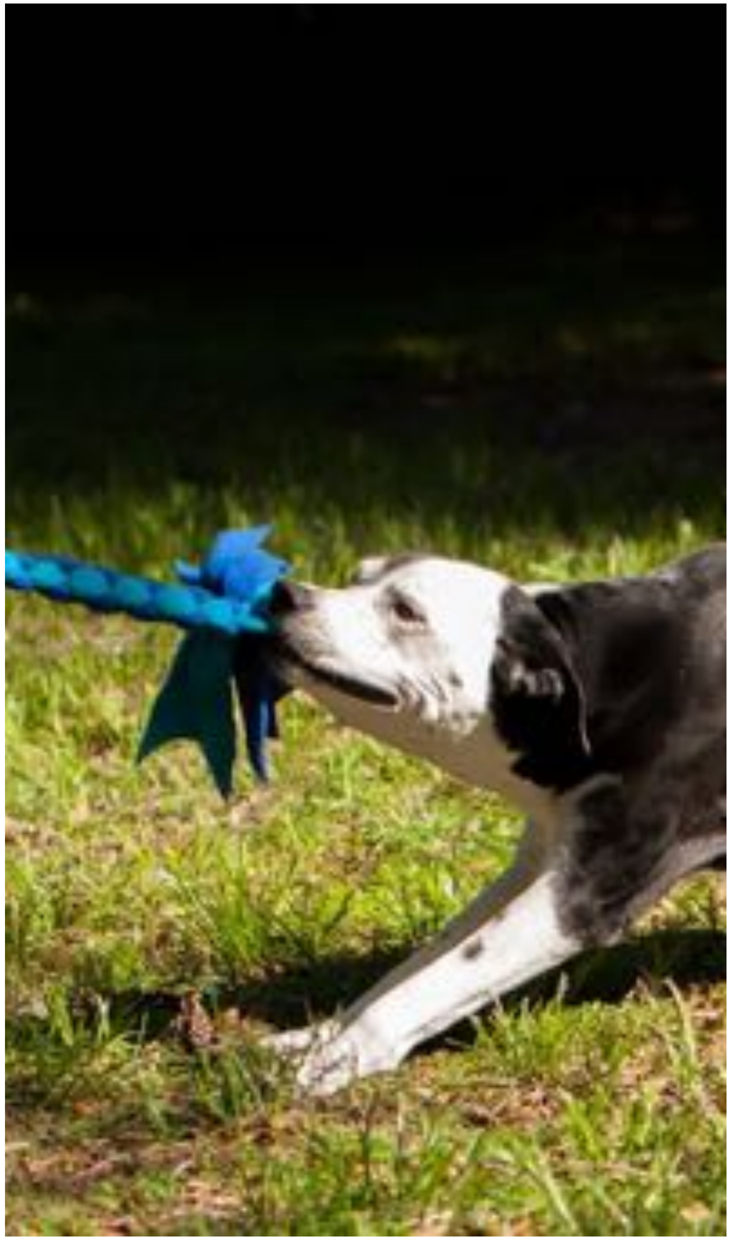
Special purpose hardware to monitor the execution flow

They detect some CFEs

- A CFE is an error that affects the execution flow
- A tipical example of CFE is an infinite loop caused by a SW bug
- Wrong results or deadline miss

WDPs used in this work use a signature approach

- A program is subdivided into blocks, each block is identified by a unique signature
- The valid sequence of signatures is unique.
- WDPs expect reception of such sequence at given time intervals
- Trigger an error upon wrong/unexpected signature or on timeout



SYSTEM WATCHDOG TIMER

The SWDT is a device available on almost every MPSoC

It should be able to send an external signal

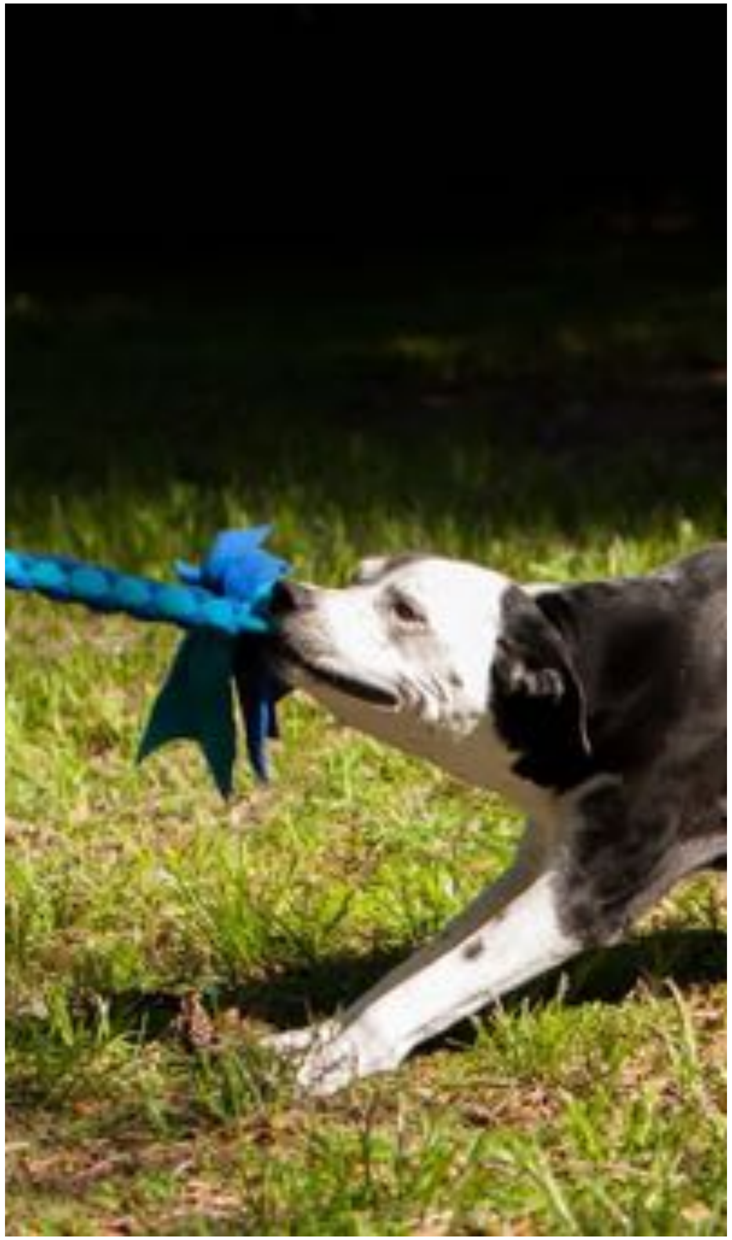
- To trigger a system reconfiguration
- E.g., switch to hot stand-by spare

It is configured at bootstrap

- Applications cannot change its configuration

It is re-armed by the critical application

It triggers a system reconfiguration when timeout happens



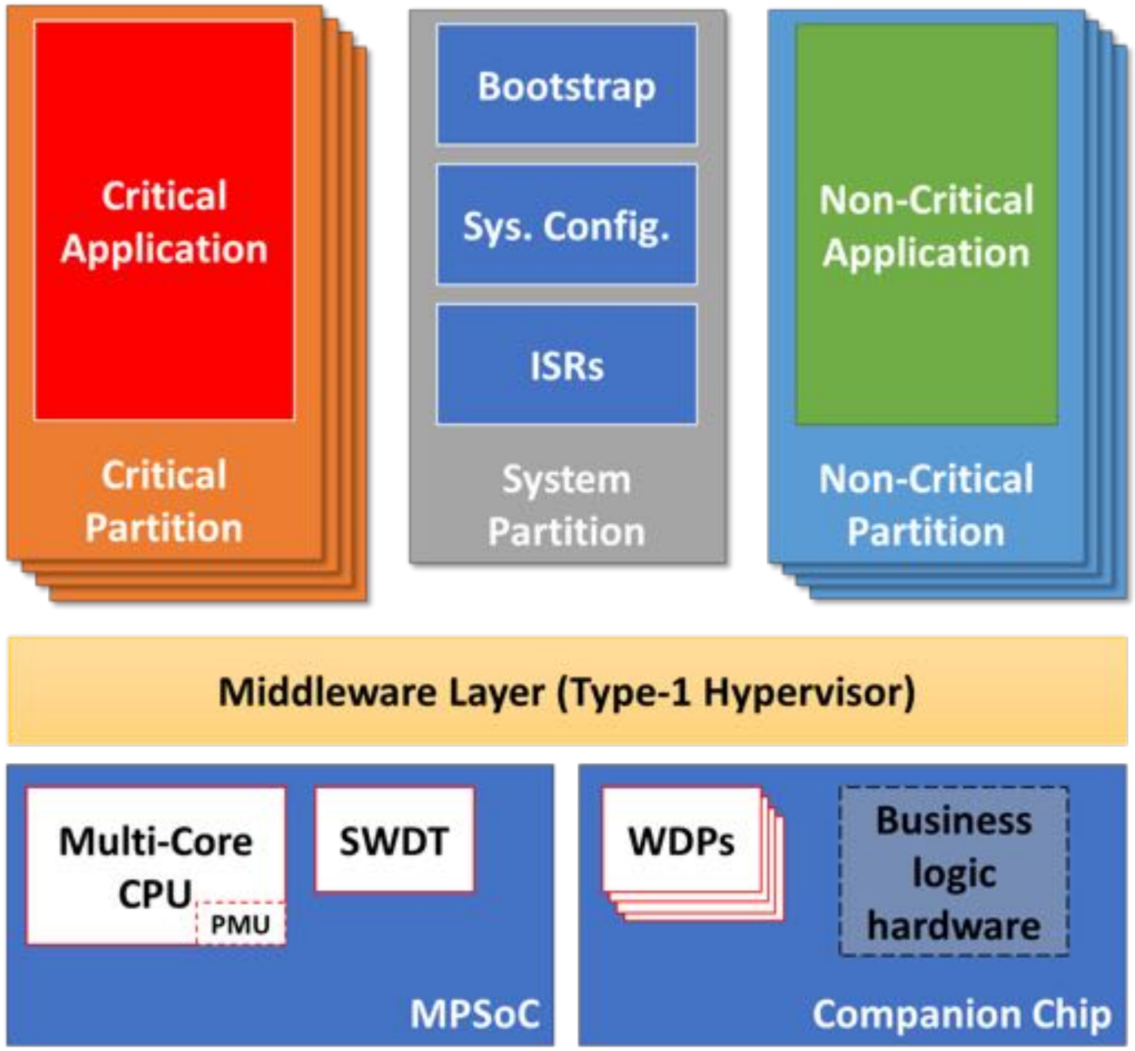
PUTTING ALL TOGETHER: ODIn-A (avionic)

Space Partitioning

- Type-1 Hypervisor

Temporal Partitioning

- PMU
- WDP
- SWDT



PUTTING ALL TOGETHER: ODIn-S (space)

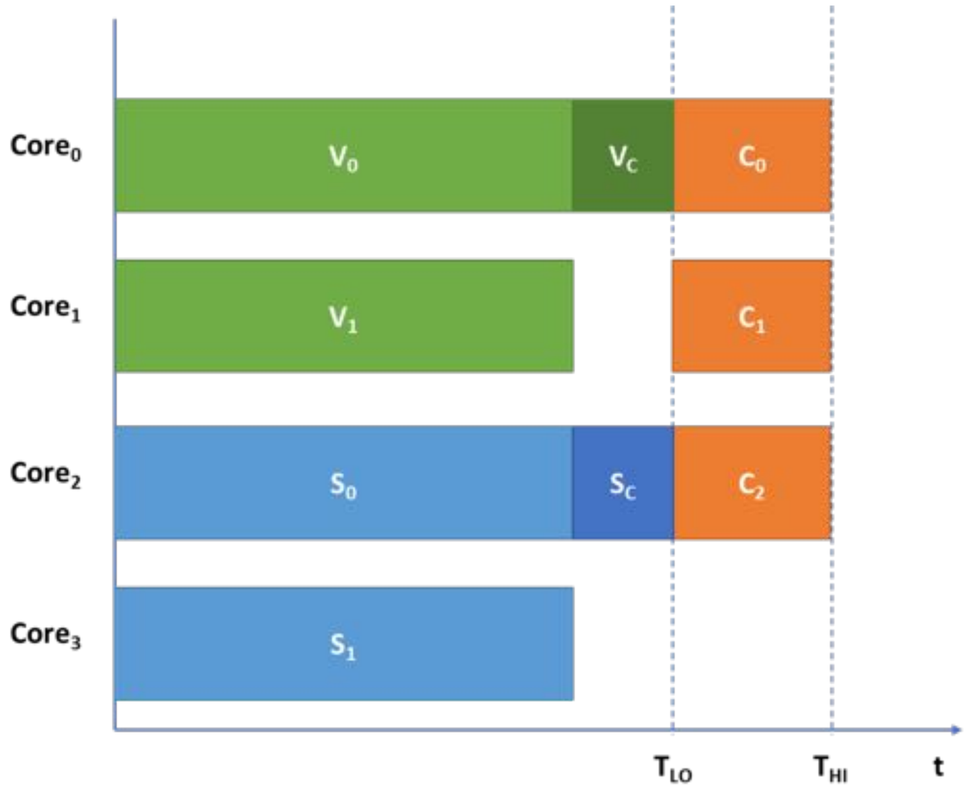
Harden ODIn against radiation effects

Use TMR for the critical software

- Supported by a HW voter implemented in the FPGA

Use TTMR for the non-critical software

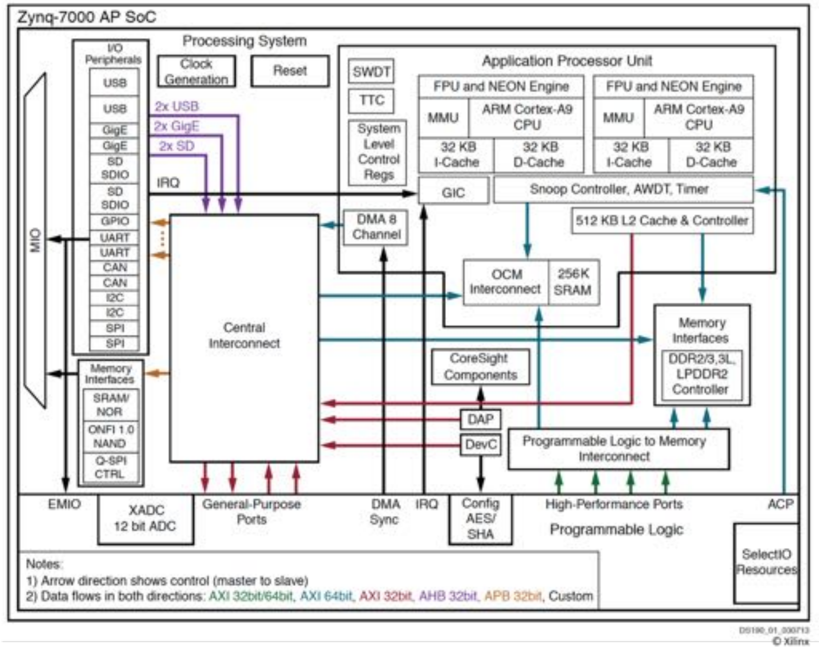
- Schedule two execution in parallel
- Check for agreement at the end of both
- If no agreement is found execute a third time or discard computation



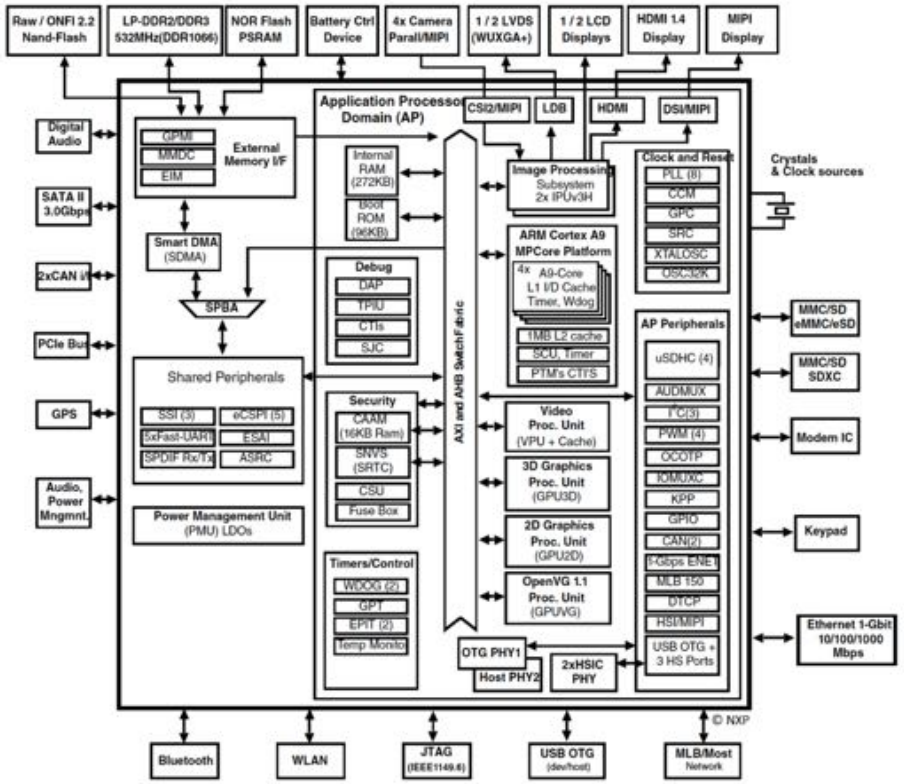
EXPERIMENTAL VALIDATION

Two flavors of ODIn

- Avionic ODIn: ODIn-A
- Space ODIn: ODIn-S



Zynq-7000 AP SoC
Processing System
Application Processor Unit
Central Interconnect
MIO
EMIO
XADC 12 bit ADC
General-Purpose Ports
High-Performance Ports
Programmable Logic
ACP
Notes:
1) Arrow direction shows control (master to slave)
2) Data flows in both directions: AXI 32bit/64bit, AXI 64bit, AXI 32bit, AHB 32bit, APB 32bit, Custom
© Xilinx



Application Processor Domain (AP)
ARM Cortex A9 MPCore Platform
AXI and AHB Switch Fabric
Shared Peripherals
AP Peripherals
© NXP

Each implemented on two hardware platforms

- Xilinx Zynq APSoC (dual-core with integrated FPGA)
- i.MX6Q MPSoC with Lattice EPP5U FPGA (connected through PCIe)

BENCHMARK APPLICATIONS

Realistic workload provided by Leonardo in the scope of the EMC^2 project

Dedicated workloads for avionic and space use cases

- For both dual and quad-core architectures

Each workload is a composition of a set of programs

- Control application
- Sensor data compression (RICE compression)
- Image processing (Edge detection)

AVIONIC BENCHMARK

Dual-core benchmark

- Control application – Critical
- Sensor data compression – Non Critical

Quad-core benchmark

- Control application – Critical
- Sensor data compression 1 – Non Critical
- Sensor data compression 2 – Non Critical
- Image processing – Non Critical

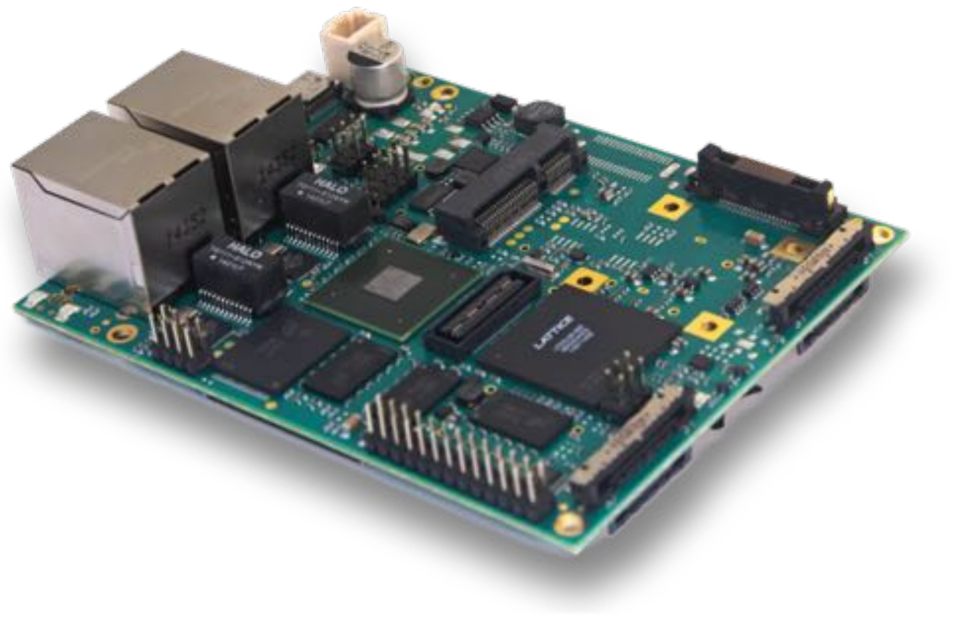
SPACE BENCHMARK

Dual-core benchmark

- Control application – Critical
- Sensor data compression – Non Critical

Quad-core benchmark

- Control application – Critical
- Sensor data compression – Non Critical
- Image processing – Non Critical



FAULT INJECTION

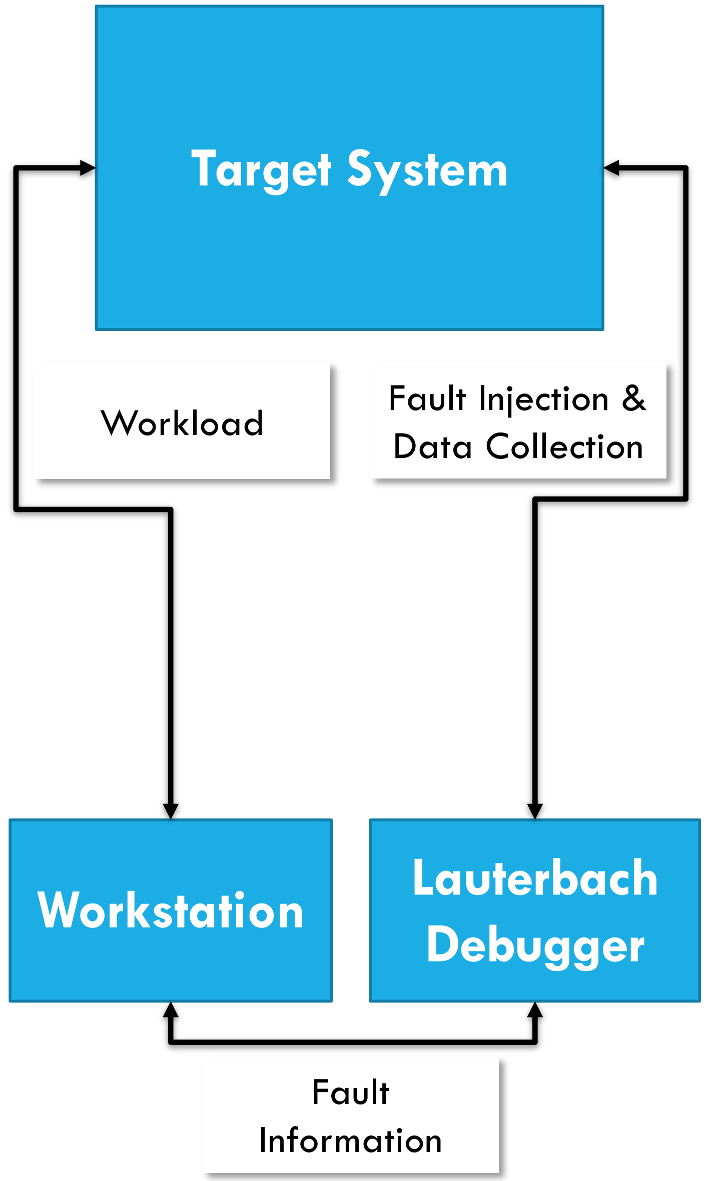
Used to evaluate system's response to faults

Inject a fault either in the hardware or in the software

Observe the behavior of the system

The fault injection system is based on the Lauterbach debugger

- Stop the execution
- Inject a fault in the system
- Resume execution
- Download results and classify



FAULT INJECTION: FAULT MODELS

SEU

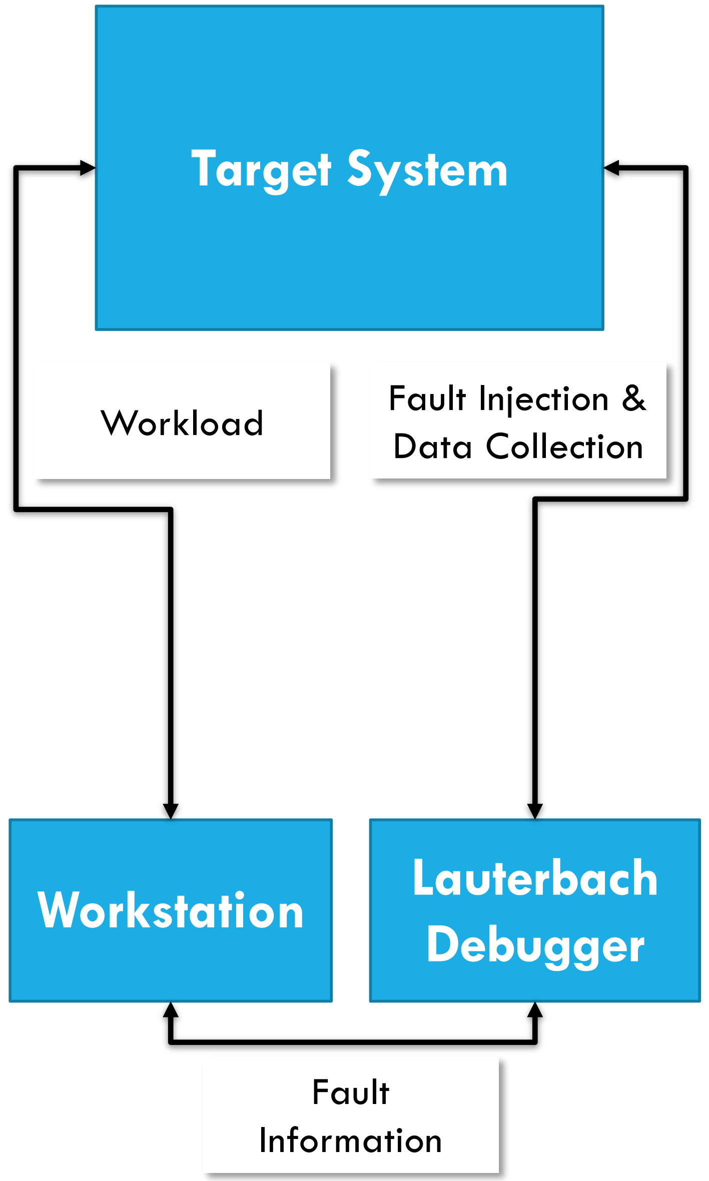
- Bitflip in a memory element, either CPU RF or CFG Regs.

Software bug

- Bitflip in a random word in the code memory area

Artificial bug

- Designed to stress the interconnect to enhance observability
- The metric selected to detect this fault through the PMU is the Data Cache-dependent Stall Cycles (DCSCs)
- Based on the assumption that low-criticality applications can have software bugs, due to the lower design effort



BITFLIP INJECTION CLASSIFICATION

No Effect

- the fault had no effect on the system

Detected CFE

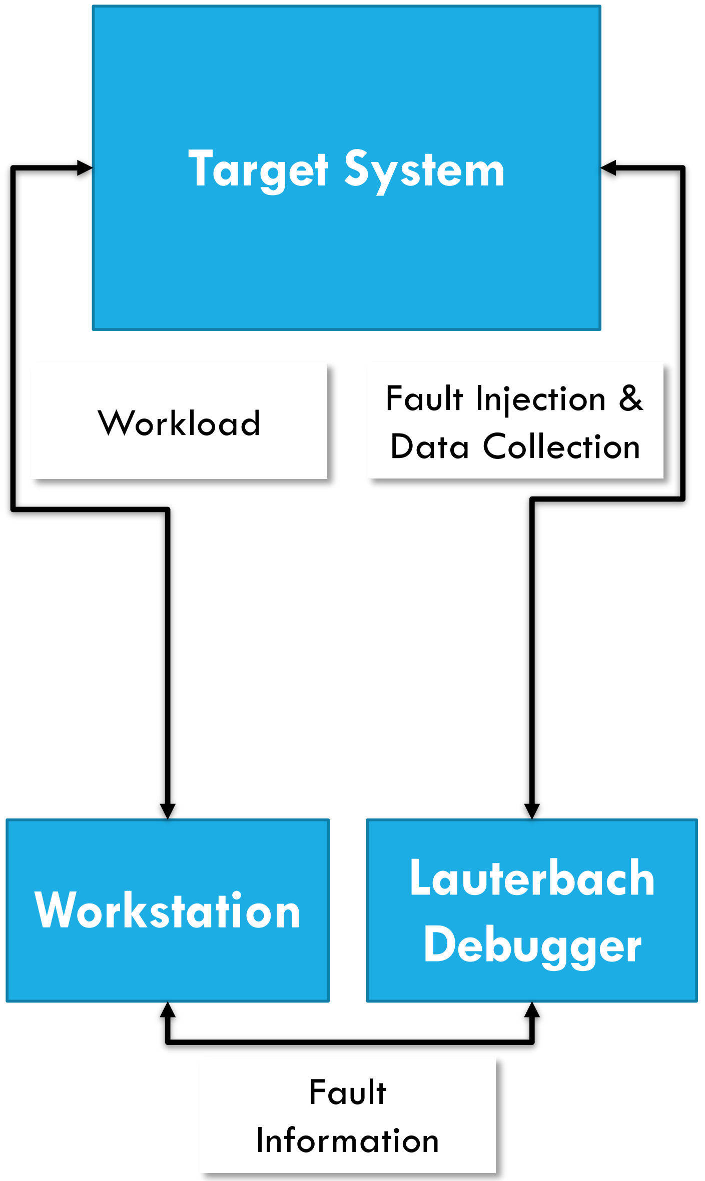
- the fault resulted in a control flow error detected by WDPs or PMUs

Detected TO

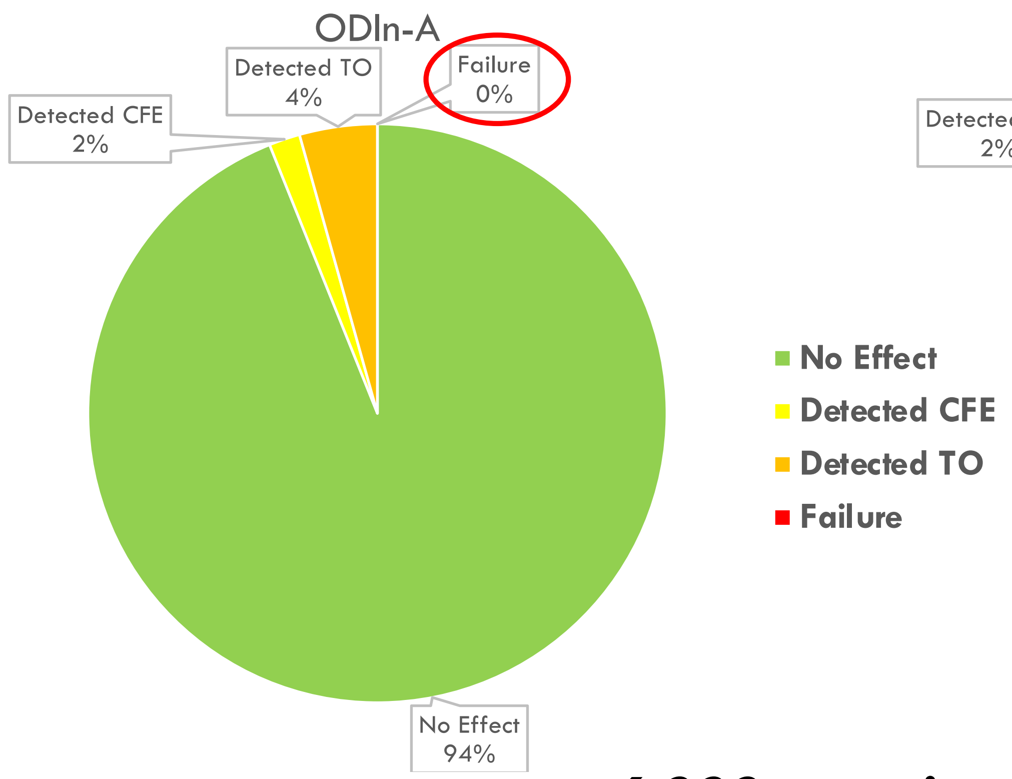
- the fault resulted in a functional interruption detected by the SWDT

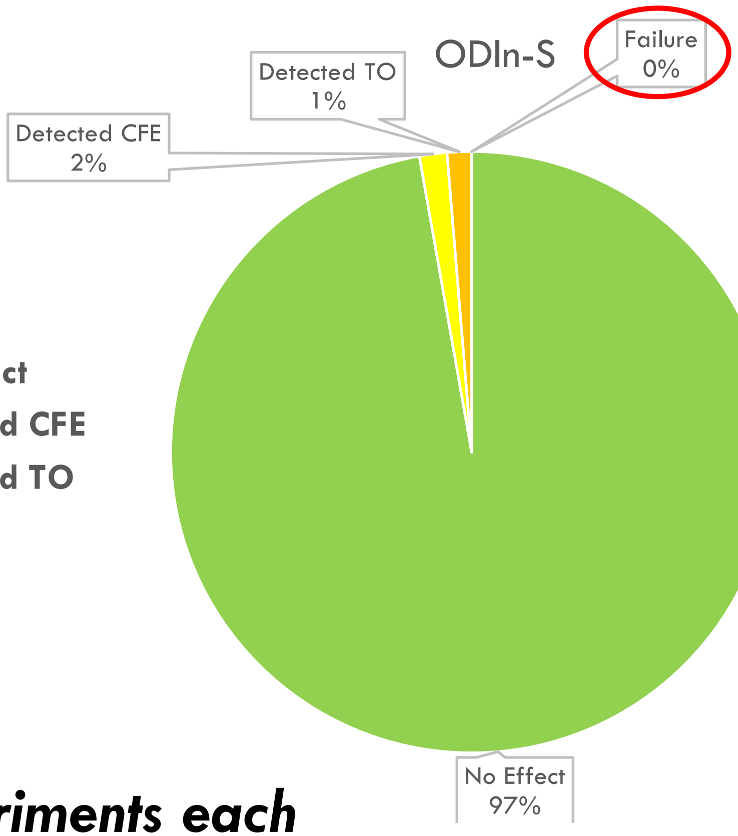
Failure

- the fault resulted in a misbehavior



BITFLIP INJECTION RESULTS





6,000 experiments each

SOFTWARE BUG INJECTION

Critical Error

- error in the safety critical application

Non Critical Error

- error in a non-critical application

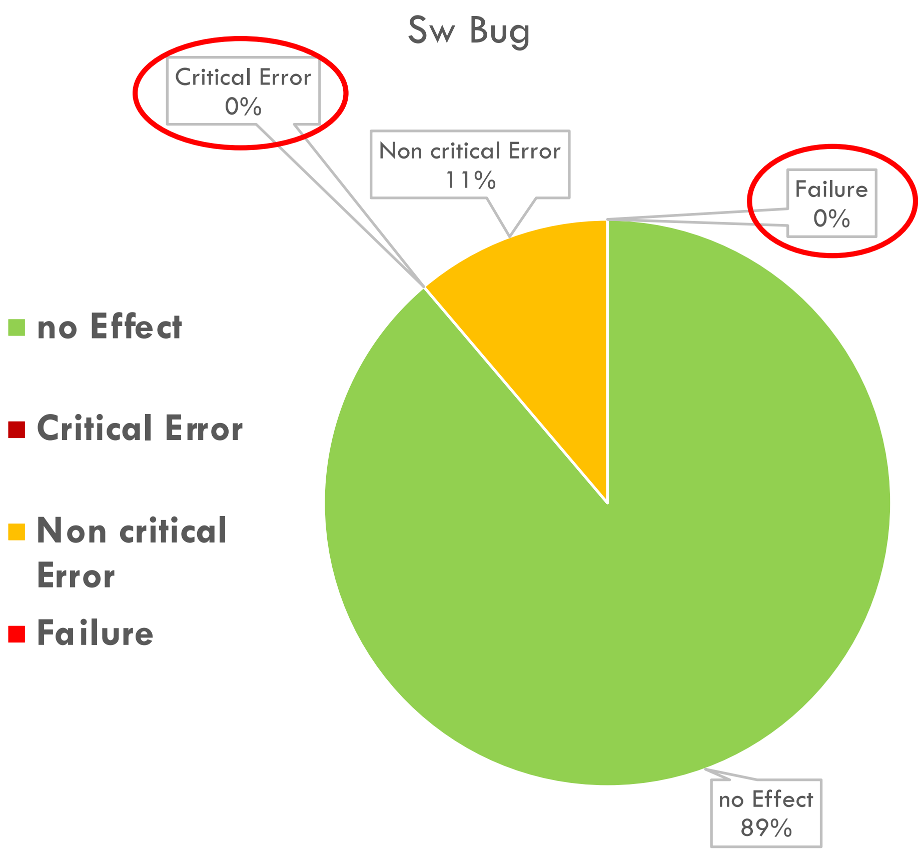
Failure

- undetected error causing a misbehavior

No Effect

- the fault had no effect on the system

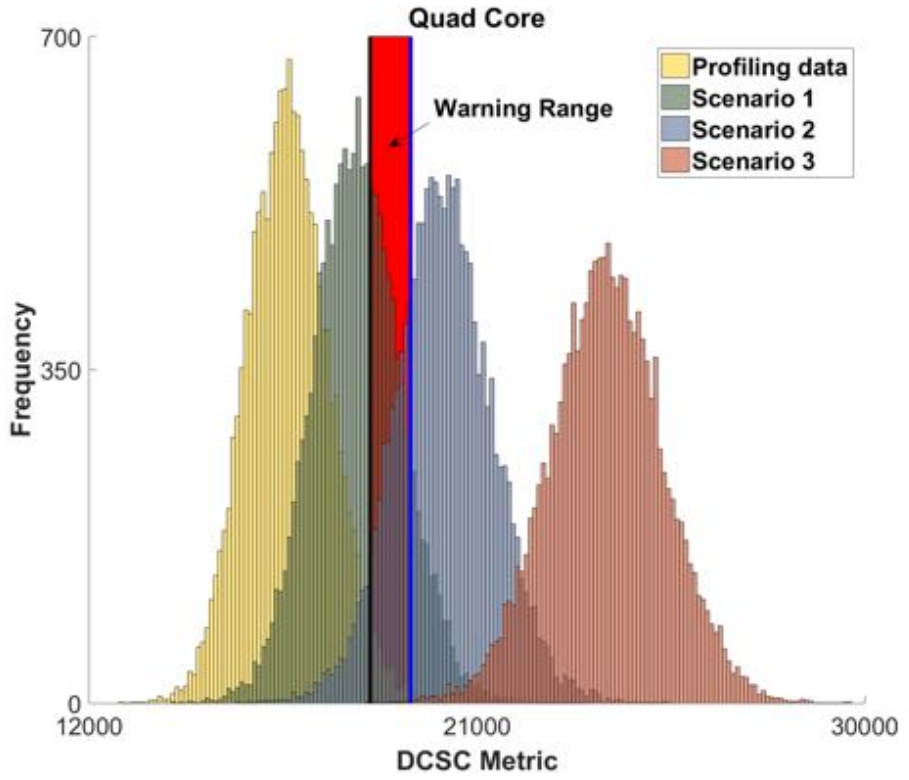
NE	Critical Error	Non-Critical Error	Failure
8880 (88.8%)	0	1120 (11.2%)	0



10,000 experiments

ARTIFICIAL BUG INJECTION

Platform	Buggy Tasks	Warning	Panic	NE
Dual-Core	1	2 (0.01%)	14859 (99.06%)	139 (0.93%)
Quad-Core	1	1114 (7.43%)	1668 (11.12%)	12218 (81.45%)
	2	528 (3.52%)	11348 (76.65%)	3124 (20.83%)
	3	0	14490 (99.93%)	10 (0.07%)



15,000 experiments in each scenario

CONCLUSIONS

The proposed architecture is suitable for implementing mixed-criticality on MPSoC

Experimental results proved that critical applications are never affected by errors in non-critical applications

Final demonstrator presented at the EMC^2 final review

Results published in several outlets including

- ACM TECS
- Springer JETTA
- IEEE IOLTS'15, IOLTS'16
- IEEE LATS'16

QUESTIONS?