Incentive Scheme / EXPRO+ and GSTP activity
4000124897/18/NL/CBi

SoC-HEALTH

On-Chip Health Monitoring and Fault Management for SoC Health-Awareness in Space Missions

Final Report

Document revision 1.1

2021

Testonica Lab

Notices

This document is intended to fulfil the contractual obligations of the SoC-HEALTH project concerning the Final Report – FR, described in ESTEC Contract no 4000124897/18/NL/CBi.

For more information, please contact Testonica Lab at email: info@testonica.com

About Testonica Lab. Founded in 2005, Testonica Lab has become a world-wide pioneer and leader in developing automated synthetic and virtual embedded instrumentation. Currently it offers cutting-edge technologies and tools for high-speed test access and at-speed test application based on JTAG and FPGAs, which are used by leaders in consumer electronics, telecom, automotive, military, aerospace, industrial electronics, entertainment and fundamental science segments. Testonica Lab has deep competence in the area of building highly optimized hierarchical system health management networks for in-field error detection and diagnosis. Testonica Lab focuses on technologies like embedded CPUs, microcontrollers, reconfigurable FPGAs, and SoC-FPGAs.

Table of Revisions

Version	Date	Description and reason	Author	Affected sections
0.1	May 29, 2021	Initial structure created	A. Jutman	All sections
0.2	May 30, 2021	Writing Introduction	A. Jutman	1
0.3	May 31, 2021	Updating Introduction	A. Jutman	1
0.4	June 4, 2021	OCFM Hardware chapter	A. Jutman	2
0.5	June 5, 2021	OCFM HW update	A. Jutman	2
0.6	June 6, 2021	OCFM HW update	A. Jutman	2
0.7	June 7, 2021	OCFM SW chapter	A. Jutman	3
0.8	June 11, 2021	SW and experiments	A. Jutman	3, 4
0.9	June 13, 2021	Summaries, conclusions	A. Jutman	Exec, 5, 6
1.0	June 14, 2021	Preparing for submission	A. Jutman	All sections
1.1	June 21, 2021	Fixing several typos	A. Jutman	All sections

Authors

Konstantin Shibin, Testonica Lab
Sergei Devadze, Testonica Lab
Anton Tšertov, Testonica Lab
Artur Jutman, Testonica Lab

Executive Summary

This document summarizes the development and evaluation done in frames of the SoC-HALTH project, which aimed at integration as well as the evaluation in terms of performance and HW overhead of the OCFM technology that comprises an original HW architecture, fault handling and system health monitoring methodology, and respective software functions.

The OCFM technology TRL 4 demonstrator is based on 8-core LEON3 soft CPU implemented on Kintex7 FPGA and running Linux OS. The data collection and emergency signaling infrastructure based on an IEEE Std. 1687 (IJTAG) network that collects real-time data from 1262 checkers embedded right inside the CPU offers immediate fault detection and diagnosis.

The experimental evaluation has demonstrated that the OCFM hardware can handle over 100000 faults per second with the average single fault processing time being in the order of 10^{-3}s, which confirmed the expected high efficiency of IJTAG-based fault and fitness data collection approach. The OCFM achieves full performance with low overhead in terms of latency, system load and used area (around 10% area in HW overhead).

Future work in fault management and active fault tolerance has to be primarily focused on further improvement of the fault handling functions and scenarios in the software and applications to heterogeneous high-performance computing platforms being currently developed by ESA.

List of Abbreviations

ASIC	Application-Specific Integrated Circuit
ADAS	Advanced Driver Assistance Systems
AFPN	Asynchronous Fault Propagation Network
AHB	Advanced High-Performance Bus
AM	Asymmetric Multi-Processing
APB	Advanced Peripheral Bus
API	Application Programming Interface
ASIP	Application-Specific Instruction-Set Processor
BIST	Built-in Self Test
BRAM	Block RAM
BUFG	Global clock buffer
CAM	Core Affinity Mask
CLI	Command Line Interface
CMOS	Complementary Metal Oxide Semiconductor
COTS	Commercial off-the-shelf
CPS	Cyber-Physical System
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSU	Capture, Shift, Update
DAC	Digital to Analog Converter
DFF	D-type Flip-flop
DFT	Design for Test
DI	Data In
DM	Dependability Manager
DO	Data Out
DR	Data register
DSM	Deep Sub-Micron
DSP	Digital Signal Processor
ECC	Error-correcting code
EDAC	Error detection and correction
EEPROM	Electrically Erasable Programmable Read-Only Memory
ELF	Early-life failure
ESIB	Extended SIB
FCX-SIB	SIB extended with F, C and X emergency flag handling logic
FDIR	Fault Detection, Isolation and Recovery
FF	Flip-flop
FM	Fault Manager
FMI	Fault Management Infrastructure
FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
GPU	Graphics Processing Unit
HCI	Hot carrier injection

HM	Health Map
HMP	Heterogeneous Multi-Processing
HSIB	Hierarchical SIB
HW	Hardware
IC	Integrated Circuit
ICL	Instrument Connectivity Language
IJTAG	Internal JTAG, denotes IEEE Std. 1687 infrastructure as a whole
IM	Instrument Manager
IMCS	Instrument Manager command/status
IO	Input/Output
IOCTL	Input Output ConTroL
IP	Intellectual Property
IPMI	Intelligent Platform Management Interface
IRQ	Interrupt Request
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group, also a short name for IEEE Std. 1149.1
LAN	Local area network
LKM	Loadable Kernel Module
LUT	Look-up Table
LUTRAM	LUT as memory
MCU	Microcontroller Unit
MMCM	Mixed-Mode Clock Manager
MMU	Memory Management Unit
MPSoC	Multiprocessor system-on-chip
MQTT	Message Queue Telemetry Transport
NBTI	Negative bias temperature instability
NMT	Network Map Table
NoC	Network-on-Chip
OCFM	On-chip Fault Management
OS	Operating System
PC	Program Counter
PID	Process ID
PLL	Phase-Locked Loop
POST	Power-on Self-Test
QoS	Quality of Service
RAM	Random Access Memory
RM	Resource Map
ROM	Read-Only Memory
RSN	Reconfigurable Scan Network
SBST	Software-based Self-Test
SDC	Silent data corruption
SDK	Software Development Kit
SEE	Single-event effect
SEL	Single-event latch-up
SET	Single-event transient
SEU	Single-event upset
SFI	System Fitness Index

SHM	Serialized Health Map
SI	Scan In
SIB	Segment Insertion Bit
SMBUS	System Management Bus
SMP	Symmetric Multi-Processing
SNMP	Simple Network Management Protocol
SO	Scan Out
SoC	System-on-chip
SSH	Secure Shell protocol and respective software
SW	Software
TAM	Test Access Mechanism
TAP	Test Access Port
TCK	Test Clock
TFI	Task Fitness Index
TL	Task List
TMR	Triple Modular Redundancy
TNS	Total Negative Slack
TRL	Technology Readiness Level
TSIB	Terminal SIB
VLIW	Very long instruction word
WNS	Worst Negative Slack
XML	Extensible Markup Language

Table of Contents

1 Introduction

Modern complex electronics has various reliability concerns in particularly caused by aging and radiation effects, which are especially pronounced in space applications due to high radiation and non-existent serviceability. Therefore, the space electronics is normally radiation hardened, being equipped with well-established but often costly fault tolerance[1] mechanisms. Still the radiation dose and wear out accumulating over time would eventually lead to non-tolerable permanent defects that render the target application dead. The results of this project promise to extend the useful lifetime of such electronics by promoting carefully planned active in-situ fault management actions.

Moreover, the space industry is increasingly seeking today for opportunities to leverage non-hardened commercial off-the-shelf (COTS) electronic components, which normally offer much better throughput at much lower cost while also providing much wider functionality scope.

It is therefore essential to *manage the health* of space electronics by closely monitoring the status of hardware resources and carefully planning their utilization. This allows attaining the highest possible performance using the remaining resources when some have already gone out of order. This is commonly referred to as *graceful degradation*. When these procedures are executed inside the system itself, it becomes self-health aware.

Indeed, the presence of permanent defects doesn’t necessarily mean the end of life if the electronic system is aware of its health and fitness status and thereby can cope with the reduced processing capacity. Moreover, modern multi-core SoCs used in data processing applications provide extensive natural redundancy that together with a smarter task scheduling provide an inherent commodity for active health and fault management.

The project has resulted in development of a demonstrator of a fine-grain in-situ On-Chip Fault Management (OCFM) architecture and framework that acts as the backbone of the system’s self-health awareness. The SoC Health-Awareness (SoC-HEALTH) conception relies on reuse of multitude of typical sensors and checkers already embedded deep into the hardware to measure operating parameters of the target IP core, detect and correct errors before they manifest at the application software level. Still the SoC-HEALTH project *did not* aim at development of any specific checkers or monitors as plethora of such solutions is well-known to the industry[1].

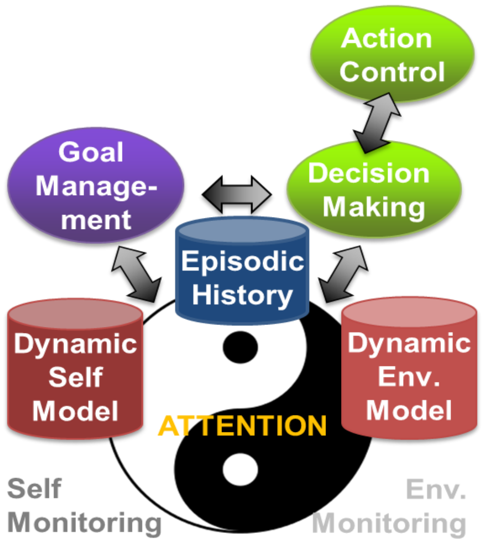
The self-health aware system would then use the OCFM infrastructure to collect health statistics and maintain a so-called health map so that the OS and application software can avoid using faulty components, thus adapting to the reduced capacity of SoC sub-modules and sub-systems (adapting to the damage), hence keeping the mission alive as long as possible.

The key to a successful OCFM is the ability to simultaneously collect and process data from dozens or even hundreds of on-chip sensors and checkers in real time independent of the system size and configuration, which is not a trivial task. The SoC-HEALTH project has proven this challenge realistic

[1] Space product assurance: Techniques for radiation effects mitigation in ASICs and FPGAs handbook, ESA-ESTEC ECSS-Q-HB-60-02, European Space Agency, 2016.

by demonstrating the ability to simultaneously handle many hundreds of instruments on board and even on a SoC. Moreover, the resulting HW overhead of the fault management infrastructure as well as fault detection latency are impressively low.

1.1 Background



Self-Health Awareness as is the ability of the system to monitor its fitness in context of its own state and the state of the environment. The monitoring data being put into an event-driven episodic historical perspective would then allow a self-aware system to maintain a dynamic self-model that includes its fitness assessment data. Combining this data with goal management under the target desirability scale allows the self-aware system perform decision making and action control in context of its actual fitness status. The general perspective on a wider topic of system self-awareness is given in a series of publications[2] and tutorials by A. Jantsch, N. Dutt et al.

Known approaches towards in-field monitoring and self-health awareness can be very roughly classified into the following categories.

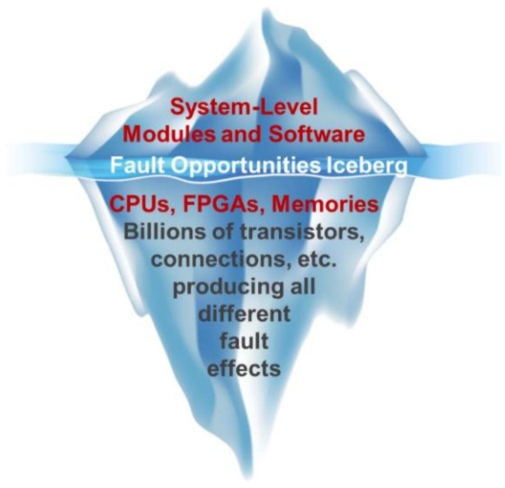
- Pure conceptual studies still trying to define the research canvas or evaluate the existing approaches.
- Works focusing on the system-of-systems level, mainly assuming communication between agents and the global system adaptation.
- Model based approaches looking at the system as a black box that has to exhibit a certain behavior detect anomalies and analyze failure modes.
- Domain-specific low-level fault mitigation, quality of service, FDIR and hardening techniques.

Most of the approaches evolving today tend to abstract from the low-level fine grain data, due to presumed overheads and complexity. Instead, a behavioral validation and model checking are used to detect anomalous behavior and assess the system fitness. In a typical flight scenario, a software would monitor the real trajectory and compare it against the nominal one. As soon as the system behavior is out of the tolerance limits, the software starts corrective actions. This approach can be compared to a mitigation of patient's high fever by immersing him/her into a cold bath. The temperature is down, but the cause of fever as well as a harmful effect of the immersion remain unknown.

In any complex electronic system, fault opportunities represent an iceberg where the modelled behavioral part resembles just its "visible" tip, whereas the vast "underwater" section corresponds to the data processing units, such as microcontrollers, CPUs/GPUs, memories, FPGAs, etc. Passive fault tolerance techniques, used today to harden electronic assemblies, allow to simply ignore the "underwater" part of the fault opportunities iceberg, which wouldn't work for COTS components. The SoC-

[2] A. Jantsch, N. Dutt and A. M. Rahmani, "Self-Awareness in Systems on Chip- A Survey," in *IEEE Design & Test: Special Issue on Self-Awareness in SoCs, Vol 34, No. 6*, 2017, pp. 8-26.

HEALTH project addresses exactly this issue by elaborating the bottom-up approach towards higher levels of Health Awareness, aiming at development a full-stack hierarchical self-health awareness framework based on the OCFM infrastructure that enables health/fitness data collection at the micro level and its immediate handling at the system level.


System-Level
Modules and Software
Fault Opportunities Iceberg
CPUs, FPGAs, Memories
Billions of transistors, connections, etc. producing all different fault effects

The SoC-HEALTH framework collects and processes two types of data: a) general fitness information for prognostics and damage-aware task scheduling, b) error alarms for instant fault recovery. The latter requires ultra-low-latency highly-optimized alarm delivery and system recovery mechanisms that have been developed in the course of project and implemented both on-chip at hardware level as well as at the OS kernel level.

Today, ESA is actively stimulating research on AI-assisted FDIR. Here, an efficient cross-layer monitoring data collection framework reaching deepest levels of the electronic components like the OCFM can improve the efficiency of this ML-based methods. Still, most of state-of-the-art approaches keep focusing on the top system level. We provide a few typical examples in the next chapter.

1.2 State of the Art

Today, the topic of in-field health monitoring is gaining momentum, which wasn't still the case, when SoC-HEALTH project started in 2018. The core set of recent academic publications on this topic, especially those focusing on the hardware-related aspects, has originated from a few research groups. Most of the researchers focus on development of particular checkers[3] and monitors[4] rather than on higher-level health management functions and the infrastructure[5] as such. Others address certain applications, like e.g. NoC[6]. Several research groups picked up the ideas formulated earlier[7] by us and eventually implemented in SoC-HEALTH project. These works help to further improve[8] and generalize[9] the original concepts.

From the industrial perspective, the aerospace sector has been historically driving the fault-tolerance and FDIR technology forward due to enormous cost of failures and strict safety regulations applied.

[3] S. P. Azad, B. Niazmand, A. Kaur Sandhu, J. Raik, G. Jervan, T. Hollstein, Automated Area and Coverage Optimization of Minimal Latency Checkers, IEEE European Test Symposium, ETS 2017.
[4] H. Ebrahimi, A. Rohani, H. G Kerkhoff, "Detecting intermittent resistive faults in digital CMOS circuits," IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp. 87-90, September 2016.
[5] A. Ibrahim, H.G. Kerkhoff, "Analysis and Design of an On-Chip Retargeting Engine for IEEE 1687 Networks," in Proc, European Test Symp. (ETS), Amsterdam, The Netherlands, May 23-26, 2016, pp. 1-6.
[6] S. P. Azad, B. Niazmand, K. Janson, N. George, A. S. Oyeniran, T. Putkaradze, A. Kaur Sandhu, J. Raik, G. Jervan, R. Ubar, T. Hollstein, From Online Fault Detection to Fault Management in NoC Routers: A Ground-up Approach, DDECS 2017.
[7] A. Jutman, S. Devadze and K. Shibin, "Effective Scalable IEEE 1687 Instrumentation Network for Fault Management," Design & Test, IEEE, vol. 30, no. 5, pp. 26-35, 2013.
[8] F. G. Zadegan , D. Nikolov, E. Larsson "A Self-Reconfiguring IEEE 1687 Network for Fault Monitoring," in Proc, European Test Symposium (ETS), Amsterdam, The Netherlands, 2016, pp. 1-6.
[9] E. Larsson and F. Ghani Zadegan, "Accessing on-chip instruments through the life-time of systems", Latin-American Test Symposium (LATS), 2016.

Still, the range of ESA actions that contribute to at least coarse-grain health awareness technology is currently very limited.

The so-called Standard Platform for Monitoring (SPMON2) (G617-195GI) was the most up to date Health Management framework at ESA prior to SoC-HEALTH. SPMON is an SNMP-based system, which retrieves health and performance attributes of managed hosts, system resources and applications such as: MCS task processes, MCS TM/TC/Admin connections, CPUs, memory, swap space, disks file systems partitions, Network Stats, Sockets, interface traffic, pre-defined TCP connections, etc. with respect to Mission Control Systems. According to authors, the main benefit is that it is a generic system easily configurable to any type of mission (e.g. SWARM, Sentinel, Bepi Colombo, EXOMARS, Euclid, etc.) and flexible enough to also monitor and control similar Data Systems (e.g. Flight Dynamics Systems, ESTRACK Management System). Still the SPMON works at the system level and is not designed to support the full cross-layer hierarchy down to SoC and microcontrollers.

There are, however, some isolated activities at ESA, targeting low-level fine-grain data collection, but orchestration and general strategy seems to be missing. In this context, it is worth mentioning the following two actions:

- Microcontroller Softcore for Space Applications (G617-251ED)
- Network on Chip (NoC) for many-core System on Chip in Space applications (T201-005ED)[10]

The first one lists "high priority interrupt on parity error in order to signal the fault to the application" activity as a part of the development plan. The second one, among other project activities, targets "graceful degradation mechanisms" based on sensor data processing. None of the two projects targets the ability to simultaneously collect and process data from dozens or even hundreds of on-chip sensors.

Another big class of activities at ESA targeting system health are called Failure Detection Isolation and Recovery (FDIR). Most of them are performed at the software level (application), involve high-level system modelling, and embrace the complete satellite functionality. An example activity is given below:

- Generic AOCS/GNC techniques and design framework for Failure Detection Isolation and Recovery (G617-035EC).

This activity, according to authors, is contributing to mitigation of "the current lack of systematic approach and the lack of engineering transparency and guidance of the FDIR engineering process, with also the aim to decrease overall complexity." It is neither focusing on SoC level fine-grain FDIR nor addressing permanent faults or a damage handling.

A very comprehensive compendium of available classical fault-tolerance techniques (both on-chip and off-chip), like radiation hardening at process and layout levels, redundancy (spatial, temporal, SW), encoding, ECC and parity checking, filtering and many others is collected in ESA's "Space product assurance: Techniques for radiation effects mitigation in ASICs and FPGAs handbook."[11] The book, however, does not provide any hint of extending the fault-tolerance techniques used by ESA towards Fault Management and Health Awareness solutions.

[10] Activity: *Space SoC Network–on-Chip IP options study-case* by Recore (NL); contract No. 4000115252.

[11] Space product assurance: Techniques for radiation effects mitigation in ASICs and FPGAs handbook, ESA-ESTEC ECSS-Q-HB-60-02, European Space Agency, 2016.

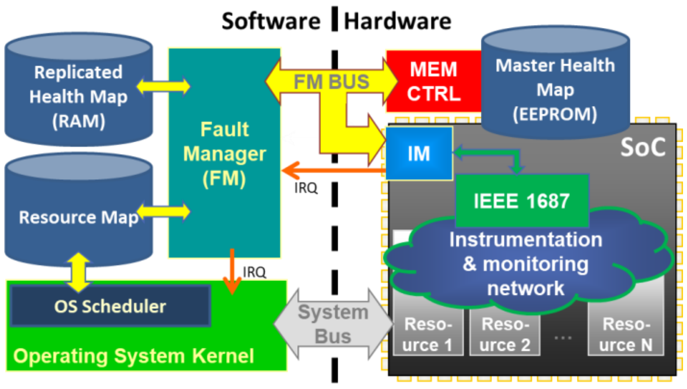
Apart from space applications, there are some domain-specific health-awareness and fault management techniques like the well-known S.M.A.R.T.[12] (Self-Monitoring Analysis and Reporting Technology) developed specifically for computer hard drives, which measures temperature, spin-up time and data error rates. Somewhat less-known is SNMP (Simple Network Management Protocol) for managing (incl. fault management) devices such as cable modems, routers, switches, servers, workstations, printers, and more over the local area network (LAN). It operates over the LAN and does not address the needs of lower system levels.

The Intelligent Platform Management Interface (IPMI) is a good example of an application-neutral industrial system management technology, which is also OS-independent and uses a special System Management Bus (SMBUS) interface as well as a dedicated "side-band" management LAN connection. IPMI, however, does not support real-time immediate reaction on faults. Neither it supports the cross-layer data collection and handling hierarchy.

All in all, the SoC-HEALTH project is a timely action that addresses the gap in cross-layer fine-grain systematic health data collection in SoC-based systems that takes advantage of a current technological trend towards natural redundancy of modern multi-core data processing architectures. Among other benefits, this redundancy enables numerous interesting low-cost alternatives to traditional expensive TMR.

1.3 The SoC-HEALTH Approach

Testonica Lab (TL) has started developing the fine-grain in-situ On-Chip Fault Management (OCFM) concept[13] for multi-core SoC devices as a *system-neutral systematic approach* back in 2010. The TL's OCFM technology represents the backbone of SoC-HEALTH approach, which extends the lifetime of multi-core data processing and communication modules by adapting the software to run on a partially broken hardware. Conceptually, the cross-layer OCFM acts as a middleware between HW checkers/sensors and mission applications. Theoretically, it is capable of handling both transient and permanent faults as well as the system's degradation and is based on the following four major components.



Embedded monitors and sensors as well as built-in self-test facilities and various checkers called collectively Embedded Instrumentation form the fundament of the framework and are responsible for collecting service information (*whereas the reuse of fault tolerance features is practiced*). The next layer based on IEEE Std. 1687 networks appended with a special-pur-

[12] ANSI X3.298-1997 AT Attachment-3 Interface (ATA-3), American National Standards Institute, Inc.
[13] K. Shibin, S. Devadze, A. Jutman, M. Grabmann, R. Pricken, "Health Management for Self-Aware SoCs based on IEEE 1687 Infrastructure", in *IEEE Design & Test: Special Issue on Self-Awareness in SoCs*, *Vol 34, No. 6*, 2017, DOI: 10.1109/MDAT.2017.2750902, pp. 27-35.

pose asynchronous emergency signaling infrastructure is responsible for efficient data transportation mainly from the instruments towards monitoring and fault management software[14].

An important property of the SoC-HEALTH approach is that the data exchange is done in very rare occasions, based on thresholds or fault detection conditions. As a result, simple mechanisms allow to disregard the vast majority of irrelevant data. We employ a set of flags asynchronously transported through the hierarchy of modules resulting in a carefully prioritized interruption reaching the system software. This is used for both ensuring an emergency reaction to an uncorrected fault event as well as routine background transportation of system health information.

Fault management software represents the third major component of the SoC-HEALTH framework. It takes responsibility for proper reaction on interrupts, handling received information and initiating follow-up service actions, e.g. scheduling diagnostic procedures in case of faults, scheduling tasks for re-execution, updating health-map data[15]. The latter is the fourth component of the framework. It reflects the actual status of the system marking some blocks as unusable, while holding error-occurrence statistics for the others.

1.4 List of SoC-HEALTH Requirements

The list of initial requirements gives a good one-stop source of the framework implementation details, its properties as well as technical challenges we've faced in the course of SoC-HEALTH project.

- The demonstration system shall implement at least 4 embedded processing cores capable of running full-featured Linux operating system (OS) including system kernel, kernel tasks and user processes (e.g. arbitrary user tasks).
- Each processing core should be supplemented with at least 125 embedded health monitors or error checkers implemented in FPGA logic. The total number of embedded health monitors / error checkers should be not less than 500. The error checkers should either be capable of detecting real faults inside processing logic and/or software or emulate fault detection.
- Fault injection mechanism should be provided capable of activating a fault condition that is detectable by one or more error checkers. This mechanism may inject error either into the processing logic under monitoring (preferably) or directly excite fault detection condition inside an error checker.
- Fault injection mechanism should be capable of simultaneously activating multiple faults detectable by health monitors / fault checkers associated with at least 2 different processing cores.
- In addition, there should be capability to periodically perform fault injection into various locations of the design.
- On-chip health sensors, error checkers and other online test instruments embedded into various parts of multi-core design should be united by a scalable health monitoring network infrastructure for OCFM.

[14] K. Shibin, S. Devadze, A. Jutman, "Asynchronous fault detection in IEEE P1687 instrument network", in *Proc of IEEE North Atlantic Test Workshop (NATW'2014)*, Binghamton, NY, USA, May 14-16, 2014.

[15] K. Shibin, S. Devadze, A. Jutman, "On-line Fault Classification and Handling in IEEE1687 based Fault Management System for Complex SoCs" in *Proc. 17th IEEE Latin-American Test Symposium (LATS'2016)*, Foz do Iguaçu, Brazil, April 6-8, 2016, pp. 69-74.

- Theis OCFM network should be based on industry-approved IEEE1687 standard, hence capable of interfacing different types of checkers and sensors in a standardized way.
- The provided network should have low latency in propagation of detected fault events, locating the fault source. These latencies should be less than the time normally required by an operating system to service hardware IRQ request.
- The online health monitoring infrastructure should allow extended interaction with error checkers or test instruments. This ensures that checker may pass any kind of diagnostic information in addition to reporting of fault event. In case of embedded test instrument (e.g. Built-in Self-Test – BIST module), the network should provide access for configuring and running different kinds of tests and fetching the test results and diagnosis.
- Flexible software sub-system for effective collection and processing of health information coming from health monitoring infrastructure should be delivered. The software should perform analysis of detected alarms, classify faults and, hence, reason on overall system health status as well as grade the health state of individual processing blocks.
- The information about health state of the multi-core processing system should be accessible for reading by human operator (via console) or third-party autonomous system monitoring facility (via network protocol, e.g. SNMP or IPMI).
- The developed sub-system for health information processing should be made adaptable with respect to hardware architecture and requirements of a target platform. In particular, fault classification and health reasoning algorithms should be easily adaptable with respect to abilities of particular system to detect different kinds of faults.
- The operating system of the demonstrator should be enhanced with the mechanism that considers availability and health status of processing resources while performing task scheduling. For example, if a certain processing core is partially broken the scheduler should decide whether a particular task may still be scheduled on this resource (e.g. depending on task's computational requirements, task criticality, etc.).
- The demonstrator OS should be capable of reacting to detected fault events (reported by health monitoring infrastructure) by stopping the task(s) that were potentially affected by the fault. The OS should automatically re-start the stopped/failed task on a newly allocated healthy resource. In this way, the demonstrator should maintain the system operability even on a partially damaged hardware.
- The OCFM task scheduling software should take into account the requirements of managed tasks, such as required modules (e.g. FPU), resource reliability, etc. This information should be given together with the executable in form of a text file (e.g. XML format). This information should be used to decide which hardware resources could be used to run a particular task on a partially damaged hardware.

All requirements have been successfully closed, while some of them being significantly outperformed. The resulting OCFM technology demonstrator developed in SoC-HEALTH project, which was implemented on Kintex7 FPGA is based on 8-core LEON3 soft CPU running Linux OS and featuring 1262 embedded checkers embedded right inside the CPU. The data collection and emergency signaling infrastructure based on hierarchical IEEE Std. 1687 compliant IJTAG network offers immediate fault detection and diagnosis.

At the SW layer, the Linux OS runs fault-managed tasks on health-managed resources w.r.t. system health and task requirements. The fault management system handles the faults real time by dynamic task relocation and re-execution. The human operator readable telemetry interface (MQTT client) provides a convenient way to observe the general system operation, its health as well as follow the OCFM actions and performance.

Fault injection mechanisms and respective scenarios are available for demonstration and OCFM performance evaluation purposes. Faults could be injected into different cores and resources using SSH channel from external PC.

The SoC-HEALTH OCFM technology demonstrator corresponds to TRL 4.

2 The OCFM Hardware

Rapid emergence of embedded instrumentation as an industrial paradigm and adoption of respective IEEE Std. 1687-2014[16] - *IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device* by key players of semiconductor industry opened new horizons in developing efficient on-line health monitoring frameworks for prognostics and fault management.

The OCFM monitoring, diagnostic and fault management functions in SoC-HEALTH project are implemented based on underlying IEEE Std. 1687 infrastructure and instrumentation, achieving the high-performance operation at the cost of a reasonable overhead impact on the underlying system performance and cost. Since, the approach is majorly based on the reuse of the DFT (Design for Testability) infrastructure (including IEEE Std. 1687 networks) normally implemented in semiconductor devices for manufacturing testing and diagnostic purposes, the additional overhead is minimal, and it is mainly caused by the need to protect this service infrastructure during the mission. In the course of the project, we have demonstrated that the value of the proposed framework is especially evident for modern high complexity devices and this value is expected to grow along with system complexity.

2.1 Embedded Instrumentation

Embedded instruments are hardware modules which are designed to be inserted into the hardware design of an IC and perform a certain activity usually related to non-functional requirements. This may include device testing, measurement of operating parameters like temperature and voltage, monitoring of available slack of certain critical paths in the design, bus activity monitoring, adjustment and calibration of internal parameters, setup of modules (e.g. Digital to Analog Converter (DAC), Phase-Locked Loop (PLL)) and other applications.

In the context of fault and health management, online embedded instruments that monitor the correctness of hardware operation as well as critical parameters that may indicate the high probability of failure happening soon are most useful. Some concurrent correctness monitors also called checkers are based upon the principle of continuous monitoring of hardware module’s inputs and outputs to make sure certain logical properties are not violated. Others can detect errors in data - like Cyclic Redundancy Check (CRC) and optionally correct them like ECC.

Another type of embedded instrumentation - parametric monitors are very useful for estimating the "health" of hardware, prognosing and possibly avoiding a fault. They monitor essential parameters of a circuit like temperature, voltage, delay or slack in digital paths, frequency of ring oscillators. The values and tracked changes of these parameters may indicate aging and performance degradation which has already occurred or will be accelerated at certain conditions.

We have implemented **1262 embedded instruments** (checkers and monitors) in SoC-HEALTH technology demonstrator as follows:

- Property and assertion checkers in larger modules and automata

[16] IEEE Standard for Access and Control of Instrumentation Embedded within a Semiconductor Device, IEEE Std. 1687-2014, 2014.

    - 1088x register checkers 136 per core, which are
    - 8 checkers for global registers
    - 16*8 checkers for register windows in SPARC architecture
- Watchdogs on IOs and peripherals
    - 1x AHB/APB bus idle checker (per system)
    - 1x simple address/value pattern checker on AHB/APB bus (per system)
    - 8x push-button linked checkers activating fault signal on button press (1 per core)
- IC integrity and environmental monitors
    - Temperature sensor (per system)
    - 3x voltage sensors (per system)
    - 32x aging sensors that are located near specific core (4 per core)
- CPU specific monitors
    - 64x Program Counter (PC) idle detection (8 per core, one per pipeline stage)
    - 64x PC out of range detection (8 per core, one per pipeline stage)

Embedded instruments can generate broadly two sorts of information with different properties. The first one is emergency data about a current critical event (such as hardware fault). This type requires very fast delivery of quite limited amount of information (e.g. an uncorrected fault has been detected) because this information has the highest value during a very short period of time. The second type is diagnostic data which has larger volume, but does not require urgent delivery (e.g. parameter measurement data). This difference determines varying approaches to handling these types of data.

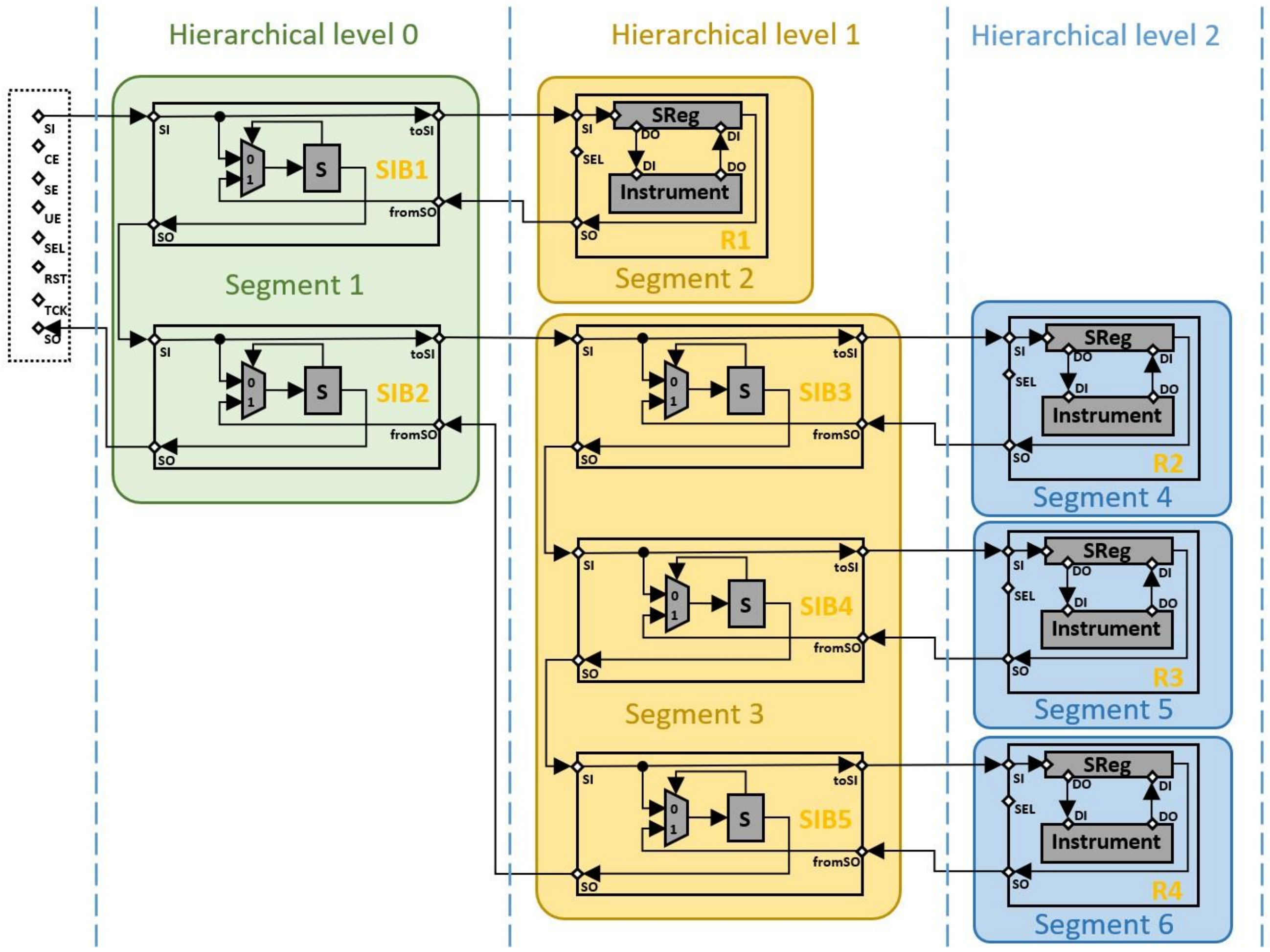
In order to address these needs and achieve performance targets, the OCFM imposes certain requirements onto embedded instrumentation. Firstly, instruments need to be connected to a unified access mechanism for efficient data transfer. In SoC-HEALTH we employ IEEE Std. 1687 Reconfigurable Scan Networks (RSNs) for this purpose. Secondly, emergency information about detected faults must reach the fault management system as quickly as possible. This is done by introducing specific extensions to the typical IJTAG commodity (which are still IEEE Std. 1687 compliant).

2.2 IEEE Std. 1687 Based Reconfigurable Scan Networks

By definition, the IEEE 1687 standard (a.k.a IJTAG) offers a methodology for accessing instrumentation embedded within a semiconductor device, without defining the instruments or their features themselves. This methodology and access mechanism are standardized, which allows for fast adoption by various vendors and interoperability between their products.

Besides fulfilling the initially intended target of regulating the external access to internal embedded instruments, IEEE Std. 1687 access mechanism (i.e. network) can be used also to continuously handle fault detection information as well as to manage test and system resources as a system-wide background process during the system operation. The advantage of IJTAG networks is a low overhead in terms of chip area and relaxed routing and timing requirements between nodes due to serial and hierarchical nature of these connections, which becomes more important in complex SoCs with large physical die sizes. For these reasons, the IEEE 1687 standard approach was chosen as an efficient embedded instrument access mechanism for the OCFM architecture.

A typical IEEE Std. 1687 instrumentation network is a serial scan chain which can be divided into segments that are connected together in a hierarchical manner as shown on the picture. In such hierarchical network the segments are connected in host- client (parent-child) manner where child segments must be accessed from parent segment through Segment Insertion Bits (SIBs), which are controllable and allow inserting the child segments into the active scan chain (see figure).



SIBs are among a few key building blocks of the OCFM as they allow manipulating the active configuration of the IJTAG network by dynamically including and excluding segments of the network as required. The standard SIB has a client interface with SI and SO ports and a host interface with toSI (to Serial Input) and fromSO (from Serial Output) ports, a scan multiplexer, and the SIB register. Depending on the value of the SIB register, the scan multiplexer will:

Value 0 - connect its client interface signals SI and SO together, thus skipping the child segment - SIB is closed.

Value 1 - connect the output of child network segment fromSO to SIB's output SO, effectively inserting the child segment into the scan chain - SIB is open.

The IEEE Std. 1687 based RSNs provide a flexible, adjustable, efficient, and standardized instrumentation polling mechanism, well-suitable for low intensity periodic monitoring or targeted communication with a particular selected instrument (preferably one at a time).

2.2.1 *Instrument polling versus event-triggered target instrument access*

When it comes to emergency data delivery, polling becomes slow and inefficient, as this involves successive opening of hierarchical levels of the RSN until the polled instrument is reached. This, in turn, involves several Capture, Shift, Update (CSU) cycles and a lot of bits to be shifted through the network, which may take a considerable amount of time depending on the network configuration.

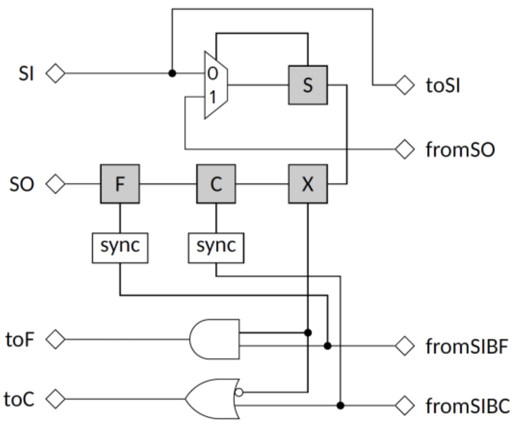
Drawbacks of polling have been known for a very long time in other domains of computer engineering. For examples, CPUs employ the system of interrupts (IRQ signals and a handler) allowing to suspend the normal CPU operation to process a request from a peripheral device in real time. This saves the CPU from having to poll the peripherals and reduces the time needed to react to an event. Similar approach is used by OCFM to solve the issue of high fault handling latency when faults are detected by embedded instruments: the instrument manager, instead of polling each instrument, would simply get a direct access through the RSN to the target instrument that triggered the communication.

However, due to hierarchical and serial nature of IJTAG networks, this concept requires more than one signal to carry the essential information as well as a method for signal propagation and aggregation which would be compatible with the hierarchical structure of IJTAG networks. As faults may be detected asynchronously and the information about them should be propagated as quickly as possible, this method must also be asynchronous. Hence, it is called Asynchronous Fault Propagation Network (AFPN).

2.3 AFPN Extensions to IJTAG Networks

In order to deliver an indication of a fault detection event as fast as possible from the embedded instrument to the fault handler, the SoC-HEALTH approach relies on three key components: (1) the addition of two flags which carry information about fault detection status in embedded instruments, (2) additional hardware inside SIBs and (3) an asynchronous aggregation and propagation network which would quickly deliver the information.

2.3.1 *Extended SIB*



The SIB architecture as such is not mandated by the IEEE Std. 1687, but its typical implementation and function discussed above is widely used. The AFPN extension to IJTAG networks adds extra features and functionality to the original SIB. The AFPN's SIB helps propagating emergency signals originating at the instruments that detect a fault up to the fault handling controller and respective software.

Such an extended FCX-SIB has three additional special purpose registers F, C and X and a logic, which implements the asynchronous emergency signal propagation and readout between hierarchical levels (segments) in IJTAG networks. The F bit, C bit, X bit and special ports toF, toC, fromSIBF, fromSIBC are components of the AFPN used to connect the Fault and Correction signals hierarchically between the network segments (signals come from the child segment to the parent segment) and in a daisy-chain fashion within one network segment.

2.3.2 AFPN flags

Two additional flags, implemented as binary digital signals, are added to indicate the presence of a fault and its severity. These flags are generated by all embedded instruments and get propagated up in the network hierarchy to the root – the IJTAG network controller. The function of the flags is defined as follows:

Fault (F) flag Indicates that a fault is detected. Whenever an embedded instrument detects a fault in the functional resource it is attached to, it sets F flag to the active state, which corresponds to logic value "1" in the HW. Otherwise, this flag should be kept inactive at logic value "0".

Corrected (C) flag Indicates that there is no uncorrected fault. This flag is set to the active state (logic "0") only when a fault has been detected (F flag active), but was not automatically corrected or tolerated. Otherwise, this flag should be kept inactive at logic value "1".

The table below summarizes the interpretation of the flag combinations by the fault handling controller – the Instrument Manager (IM).

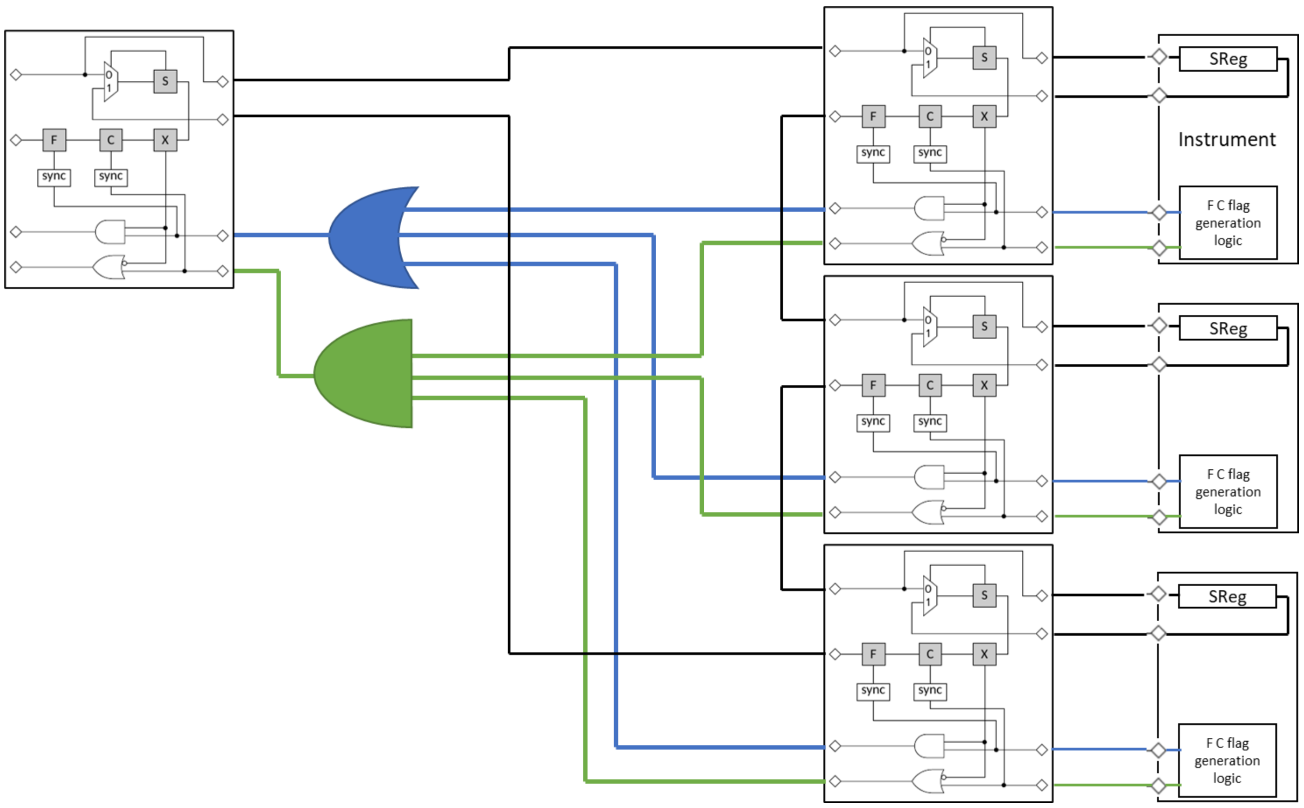
Case	Fault (F)	Corrected (C)	Interpretation
I	0	1	Normal operation, no fault
II	1	0	Error detected, not corrected
III	1	1	a. Error detected, but automatically corrected b. Some other non-fault benign event happened
IV	0	0	Reserved for AFPN self-test

Although flag values are initially generated by the embedded instruments, as flags are propagated through the AFPN to a higher level of hierarchy, they have to be aggregated to produce one composite value at the root. Such aggregated value handled by the controller, therefore represents the status of the whole IJTAG network segment. Assuming a single fault detected at a time, as soon as any instrument sets its flag F, the aggregated root flag F has to be set to logic "1". If that fault was also corrected, the aggregated root flag C has to be set to logic "0". Hence the general (simplified) flag signal aggregation rules are:

$$F_{root} = F_1 \lor F_2 \lor \ldots \lor F_n$$
$$C_{root} = C_1 \land C_2 \land \ldots \land C_n$$

Sometimes an embedded instrument can generate unwanted Fault flag values, for example when it constantly detects a fault or becomes faulty itself. This will congest AFPN and lead to miss relevant fault detection from other embedded instruments. Such situation could be avoided by masking (disabling) the Fault and Corrected flag signals by forcing them to an inactive state. In AFPN, an additional X bit is introduced for this purpose. The decision whether the F and C flags should be masked at a particular FCX-SIB is taken by the fault handling software. It then gives the corresponding instruction to the Instrument Manager which will automatically perform necessary shift operations to write an active (logic "0") value to the X bit in that SIB.

The following figure gives a general understanding how a segment in the IJTAG network hierarchy would look like if it is built using the AFPN-defined special-purpose infrastructure elements: FCX-SIBs, fault status flags (F, C) and the aggregation network. The figure depicts one of the two aggregation possibilities – the parallel aggregation, which offers an easier to grasp impression. Here, the aggregation gates are located outside the SIBs and combine all asynchronous signals together (F and C flags) in a given network segment at once, using one logical gate per flag. The resulting signal is then fed to the next aggregation gate on a higher hierarchical level of the IJTAG network.


S
F
C
X
sync
sync
SReg
Instrument
F C flag generation logic
SReg
F C flag generation logic
SReg
F C flag generation logic

In certain cases, when the detected fault is very short-lived and an embedded instrument does not store the fault detection state automatically, the resulting Fault and Corrected flags are set to the respective values only for a very limited period of time. This may cause the valuable fault detection information to be lost. In order to avoid such risk, the state of flags is latched into "sticky bits" when a fault is detected and F flag is set to 1.

2.3.3 Generating and handling the AFPN flags

In order to make the AFPN system efficient, the instruments, which are the sources of F and C flag signals must follow certain requirements:

- Flags are set to active state only once per one distinct event.
- When the flags are latched inside FCX-SIBs and acknowledged (by `afpn_ack`[17]), the instrument must reset the corresponding signal to an inactive state even if the detected fault (or other condition that raised the F flag) is still present. In this way, the faults will not congest the AFPN while they will still be registered.

In order to simplify the requirements of F C flag signal generation an asynchronous acknowledge method is used to make sure that corresponding signals are asserted by embedded instruments for the time not longer, but also not shorter than needed. This method also helps avoiding clock domain

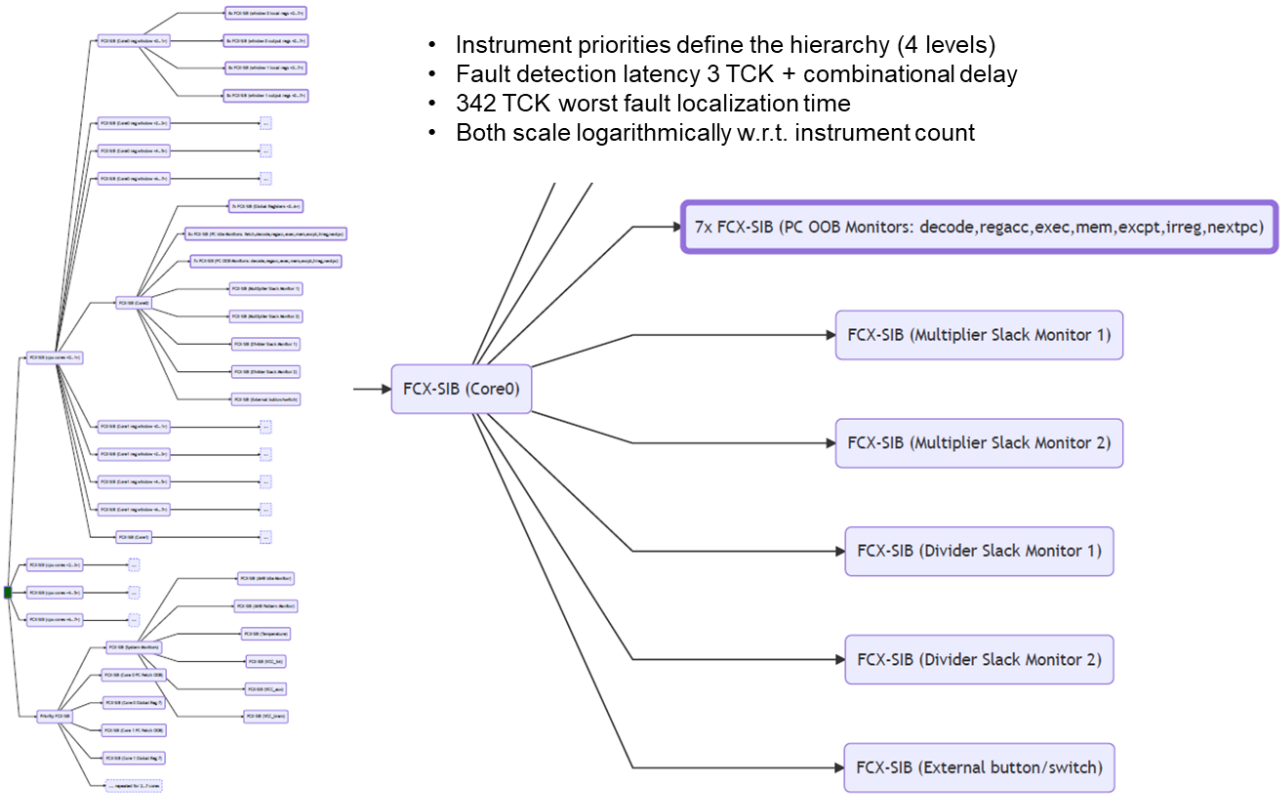
[17] `afpn_ack` - asynchronously acknowledge for F and C sticky flag latching. Active state is high.

crossing issues, when flag generation logic inside an instrument is not clocked by IJTAG network clock (TCK). This method works as follows:

1. An instrument detects a fault or other condition which requires attention from FM. It sets Fault (F) flag to the active state.
2. F and C flags are propagated through AFPN and simultaneously are fed to ESIBs.
3. Due to signal propagation and resynchronization to TCK clock, some time is required before flags are actually latched to "sticky" bits inside ESIBs.
4. When active F flag state is latched, it is acknowledged by setting `afpn_ack` signal to high.
5. `afpn_ack` signal is received by the instrument which should deassert its F and C flags in response.
6. `afpn_ack` signal is deasserted in response.

2.3.4 *Resulting SoC-HEALTH OCFM architecture*

The fragment of the RSN that handles 1262 embedded instruments integrated into the 8-core LEON3 CPU in SoC-HEALTH is shown on the figure below. The resulting OCFM network has 4 hierarchical domains defined by our network optimization procedure based on instrument priorities. This network provides fault detection latency of 3 clock cycles (TCKs) the worst case fault localization time of 342 TCKs.



- Instrument priorities define the hierarchy (4 levels)
- Fault detection latency 3 TCK + combinational delay
- 342 TCK worst fault localization time
- Both scale logarithmically w.r.t. instrument count

7x FCX-SIB (PC OOB Monitors: decode,regacc,exec,mem,excpt,irreg,nextpc)
FCX-SIB (Core0)
FCX-SIB (Multiplier Slack Monitor 1)
FCX-SIB (Multiplier Slack Monitor 2)
FCX-SIB (Divider Slack Monitor 1)
FCX-SIB (Divider Slack Monitor 2)
FCX-SIB (External button/switch)

2.3.5 *Limitations imposed on IEEE Std. 1687 infrastructure to be usable in OCFM*

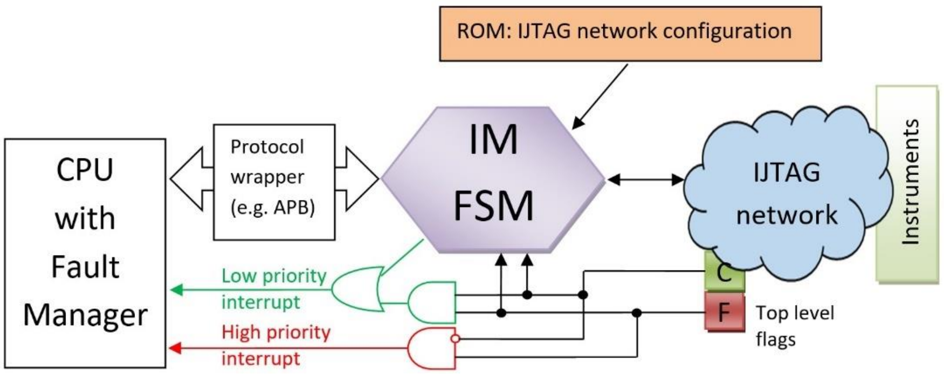
In SoC-HEALTH, we are not just adding some OCFM-specific structures to support efficient fault handling, but also impose some limitations to in-field usable IJTAG networks. These limitations help to organize the instrument access network in a regular fashion which aids in reducing fault detection latency and making it more deterministic. IEEE 1687 IJTAG standard supports a great variety of possible network configurations and ways to switch the active scan chain, which is good for manufacturing test, but hinders the efficiency of in-field scan chain manipulation.

The most important limitation is related to the scan chain switching, whereas the only SIB type supported is the extended FCX-SIB. More details can be looked up in SoC-HEALTH project's technical deliverables.

Still some RSN segments may be located outside of the scope of the in-field usable OCFM infrastructure, e.g. in the deeper levels of the network hierarchy or simply aside. Within such segments, the networks and respective embedded instruments (for example scan-test networks used in IC production test) can have arbitrarily complex organization. To some extent, these segments can also be used in-field, but in this case they have to be manipulated by a dedicated software.

2.4 Instrument Manager (IM)

IM is a hardware module and a part of OCFM which controls the IJTAG network and acts as an interface between the embedded instruments and the host – the Fault Manager (FM) software running on a CPU. Whenever the FM needs to access the instruments to get the diagnostic information, it gives a read/write command to IM which in turn opens the path to the requested instrument through the hierarchical IJTAG network and performs the requested operation. In communication between IM and FM, embedded instruments are referenced by a node address (row number) in the Network Map Table (NMT). In its turn, the node address depends on the position of the instrument inside the network.



Besides the instrument access, the IM is responsible for reacting to the fault flags propagated through the AFPN. IM can automatically open the path to an instrument which has raised the fault flag and provide the information about that instrument's location (node address) inside the IJTAG network.

2.4.1 IM operation principle

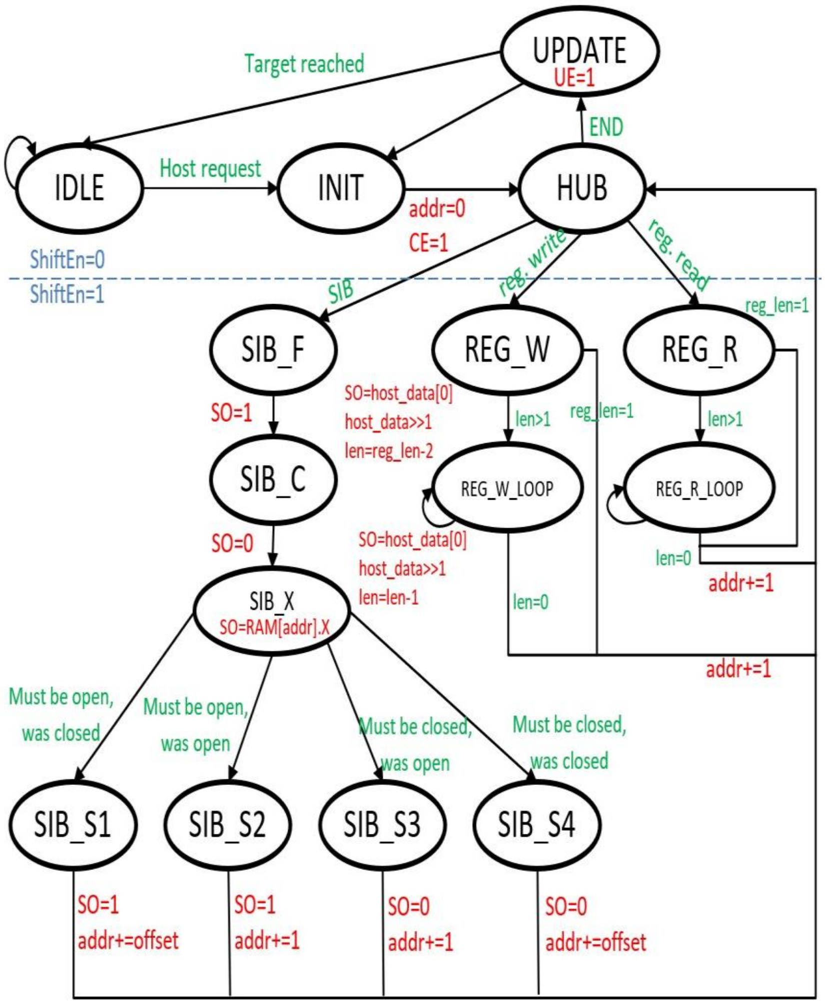
IM is directly connected to the IJTAG network through a standard set of serial scan and control signals and operates as an embedded controller of the IJTAG network. No intermediate layer with JTAG TAP controller FSM is required as it only adds unnecessary complexity and limits the flexibility of the network control. IM takes care of shifting data in and out, settings control signals to needed values, receiving or supplying data through registers to the software running on the CPU.

The IM has two basic operation modes which differ in purpose and the way the communication over the IJTAG network is initiated:

Access request from the software. When the FM software needs to access an embedded instrument and read or write some data into it, it instructs the IM to do this by providing the NMT address, data (for write operation) and a command. If IM was idle, it would start executing the requested access operation.

Fault detection. If IM is idle and the Fault flag at IM's input becomes active (delivered from an embedded instrument through the AFPN), IM will send an IRQ and automatically start the fault localization procedure. This involves opening SIBs in the IJTAG network hierarchy until the source of Fault flag is found. The NMT address of the latter is then provided to the FM software to facilitate the fault classification.

The core logic of IM is implemented as an FSM with 15 states (see the respective figure). IM starts the operation in IDLE state and upon a request from the host or AFPN it moves to INIT state. In INIT state it asserts the CaptureEn signal to copy the actual values of the network's nodes to the shift registers. It also resets the address counter since the first node for which the serial scan bits need to be generated is at address 0. Then it moves to HUB state which is a helper state to manage the node address handling. From there, depending on the node type located in ROM at the current address and the requested operation, it will proceed to SIB_F, REG_W, REG_R or UPDATE states. SIB's S register value depends on whether the SIB needs to be opened or closed for the current retargeting goal. For registers, IM will first shift in the first bit (IM treats the data from host in such a way that LSB is shifted in first) and if there are more bits, it will stay in the loop state while there are bits to shift in or out. When the network node closest to network's SI is handled, FSM will reach and END entry in ROM. This means that it should assert Update signal and check whether additional CSU operations are still needed, e.g. if the target register hasn't been reached yet or if the network must be closed after the access is finished. This is decided in the UPDATE state.



2.4.2 IM interrupts

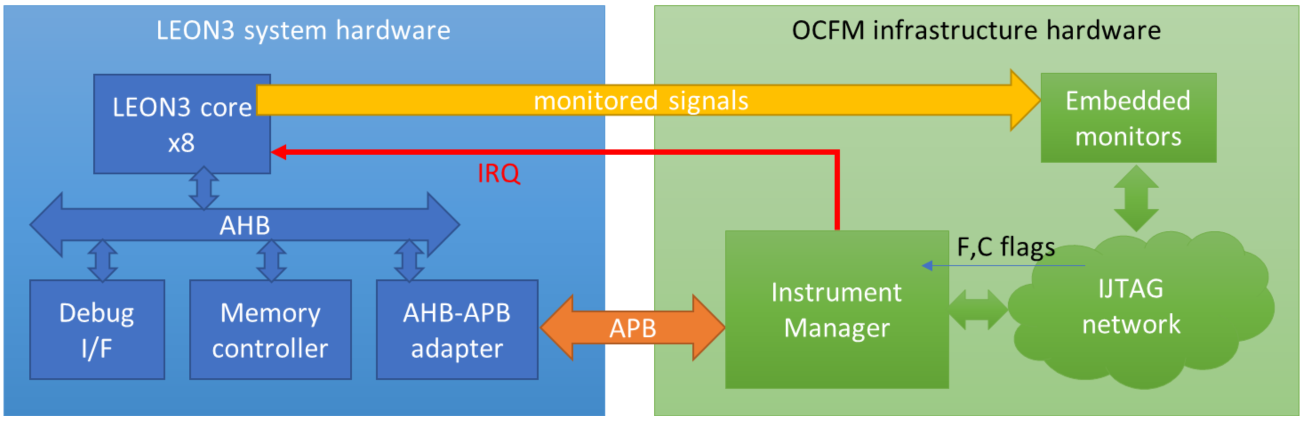
There are two types of interrupts generated by the IM to the host CPU: high and low priority ones. The Interrupt Service Routine (ISR) software identifies the actual interrupt type and initiates respective actions:

High priority interrupt corresponds to fault/error detection event. In this case, ISR initiates fault isolation to prevent possible error propagation. By default, it stops all cores to pause the task running on a faulty core, when an unrecovered fault has been detected by an instrument. After the location of the faulty resource becomes known, the healthy cores are released and can continue their operation.

Low priority interrupt is invoked when IM has finished the localization procedure or wants to transfer other information to the FM driver. In case of the localization procedure, the ISR may add an entry to the HM and call the fault classification procedure to interpret the information received from IM and update the system health status. The affected processes are rescheduled if required.

The IRQ's actual type can be determined by reading specific IRQ status register bits. At least one of them must be set to 1 if an interrupt is triggered. The fault detection bit corresponds to the high priority interrupt type, while others to the low priority type. In case two IRQ lines are not available in the system all requests could be combined into one line with the disadvantage of slightly longer response time to critical events.

IM uses a special watchdog that monitors the ability of the system to respond to the fault occurrence event. The system reaction to the IRQ may be missing due to specific faults in the CPU HW or in the OS SW.



2.4.3 *IM host interface*

On the host side, IM is treated as a peripheral and requires several 32-bit registers for controlling its operation. The list of registers accessible from software (FM kernel module) is given below:

IM command/status register Host writes to this register to control the behavior and start the operations in the IM and IJTAG network. Host reads this register to get the current status of IM.

IRQ control/status register Allows to enable, disable and read the status of different interrupt sources: fault detection, fault localization, instrument data register read/write operation completion.

Data register Stores the data which has to be written to or read from an embedded instrument through the IJTAG network. At least one data register is required and it supports read/write operations to data scan registers up to 32 bits in length. In case larger scan registers need to be supported, several 32-bit interface data registers could be used in concatenation.

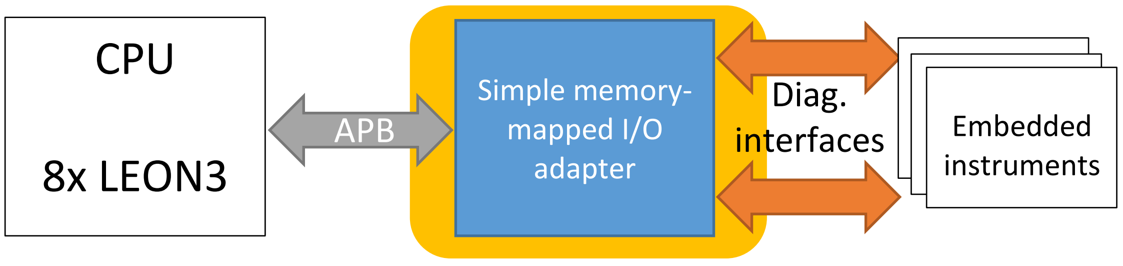
In the simplest case, IM registers can be mapped to the memory address space of the host CPU by means of a bus (e.g. using an Advanced Peripheral Bus (APB) wrapper). In our case, two 32-bit registers are used for total of maximum 64-bit data access operations, but this number can be customized for support of larger IJTAG scan registers of the instruments.

Offset	Register description
0x00	Instrument Manager command/status (IMCS)
0x04	IRQ control/status
0x08	Data, lower 32 bits
0x10	Data, upper 32 bits

2.5 Evaluation of the OCFM-Induced Hardware Overhead

SoC-HEALTH activity integrates the OCFM architecture into an 8-core implementation of LEON3 softcore CPU on Kintex7 KC705 FPGA Evaluation Platform with **XC7K325TFFG900-2 FPGA** device. In order to assess the overhead introduced by the OCFM infrastructure one can either compare the health-aware version of the base system against the initial "pure" non-hardened version of the CPU or with a version that has checkers and monitors already part of the system. The SoC-HEALTH project follows the latter scenario as it targets the evaluation of the data collection framework itself, i.e. provided that we have some instrumentation to detect faults and measure system parameters, there must be some data collection mechanism in place to compare to. Moreover, we want to establish some new practical reference point in terms of HW overhead and fault detection latency so that other researchers could evaluate their new results against a practical baseline, which is now offered by the OCFM architecture developed in frames of SoC-HEALTH project.

Therefore, besides the target *IJTAG RSN-based OCFM*, we have also implemented a simple memory-mapped access mechanism, which becomes the initial *baseline system* for the SoC-HEALTH project. As a result, we were able to compare the OCFM hardware utilization for both versions. The results show that the overhead mainly consists of flip-flops (FF) required for scan-chains and that it is not high: just 8.86% of all FFs in the target FPGA. Both designs have been successfully implemented in the target FPGA.

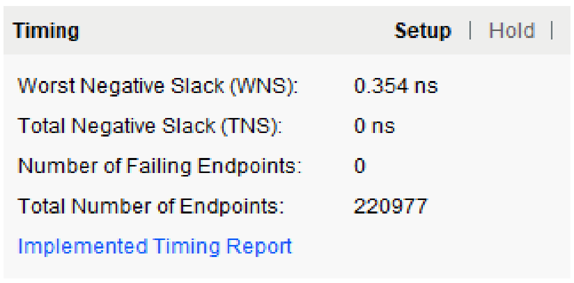


Embedded instrument access infrastructure in the baseline system

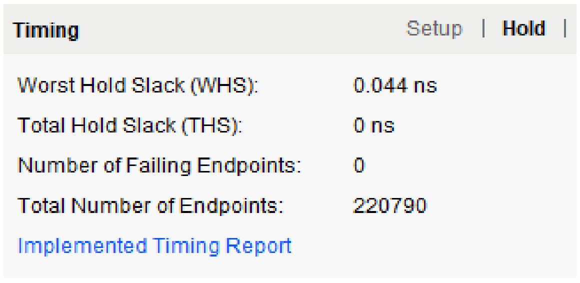
2.5.1 The baseline system

The health data collection framework in the baseline system uses a trivial memory-mapping infrastructure (memory controller, debug port, etc.) as well as exactly the same set of embedded instruments as it is in the final system, all of them having the same unified interface to the diagnostic system. All the instruments are accessed by the host (LEON3 CPU) through a memory-mapped APB peripheral which aggregates diagnostic interfaces of all embedded instruments. This access infrastructure is highlighted in yellow.

The setup and hold timing summaries below show that the design was successfully placed and routed while respecting given timing requirements.



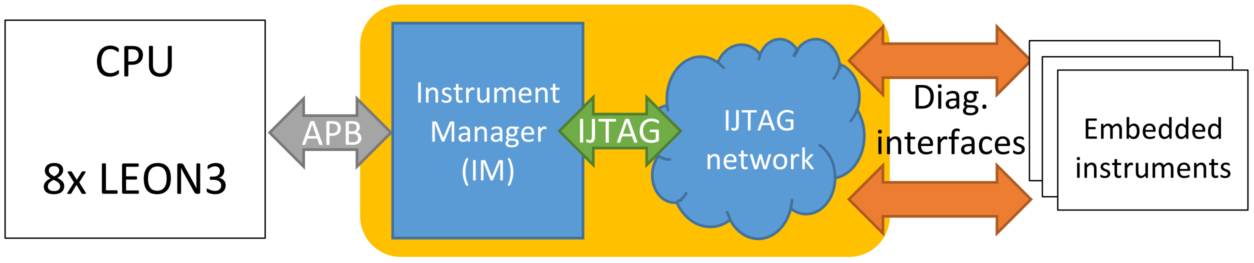
Worst Negative Slack (WNS):	0.354 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	220977



Worst Hold Slack (WHS):	0.044 ns
Total Hold Slack (THS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	220790

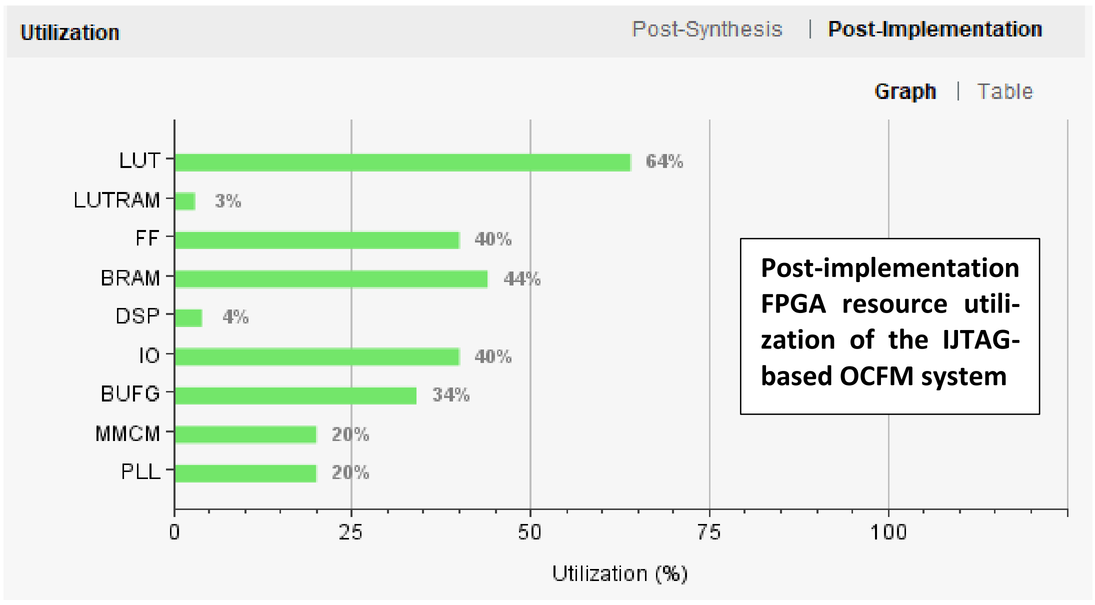
2.5.2 *The target SoC-HEALTH IJTAG RSN-based OCFM system*

The SoC-HEALTH IJTAG-based target OCFM replaces the baseline/reference embedded instrument access peripheral with a combination of the IJTAG RSN and the IM (highlighted in yellow).



SoC-HEALTH OCFM instrument access infrastructure based on IJTAG

Due to the nature of IJTAG networks, somewhat more hardware resources (flip-flops in scan chains, interconnection) is used per each accessed bit of diagnostic interface of an embedded instrument. Therefore, resource utilization of the IJTAG-based access infrastructure is higher.



Post-implementation FPGA resource utilization of the IJTAG-based OCFM system

2.5.3 *Comparison of the infrastructure resource utilization for both implementations*

In order to evaluate the overhead of the IJTAG-based OCFM infrastructure w.r.t to the baseline, resource utilization of both versions of the access infrastructure was first extracted in Xilinx Vivado software. The following table provides the utilization data (absolute and percentage of total FPGA resources) in absolute numbers (columns 2 and 3) and the percentage of total FPGA resources (columns 4 and 5). The data only reflects the access infrastructure part of the design only (highlighted with yellow in figures above). The last column shows the difference between the target and the baseline system. The calculated difference shows that resource utilization overhead of IJTAG-based access infrastructure mainly consists of extra required flip-flops, which can be attributed to flip-flops in scan chains which are the main component of any IJTAG network.

Resource type	Baseline Utilization	OCFM Utilization	Baseline % of FPGA	OCFM % of FPGA	Difference wrt. baseline
LUT	21522	22089	10.56%	10.84%	0.28%, rel. +2.65%
LUTRAM	0	1	0%	<0.01%	<0.01%
FF	0	36136	0%	8.87%	8.87%
BRAM	0	0	0%	0%	0%
BUFG	0	0	0%	0%	0%
PLL	0	0	0%	0%	0%

2.6 Chapter Conclusions

The OCFM infrastructure is the key component of the SoC-HEALTH hardware that provides flexible access to embedded instruments via IJTAG-based reconfigurable scan network (RSN). This allows for fast, efficient, and scalable communication with embedded instruments. Inevitably, it occupies some hardware resources. The results show that IJTAG-based OCFM introduces insignificant overhead in area and no impact on functional performance compared to the baseline system.

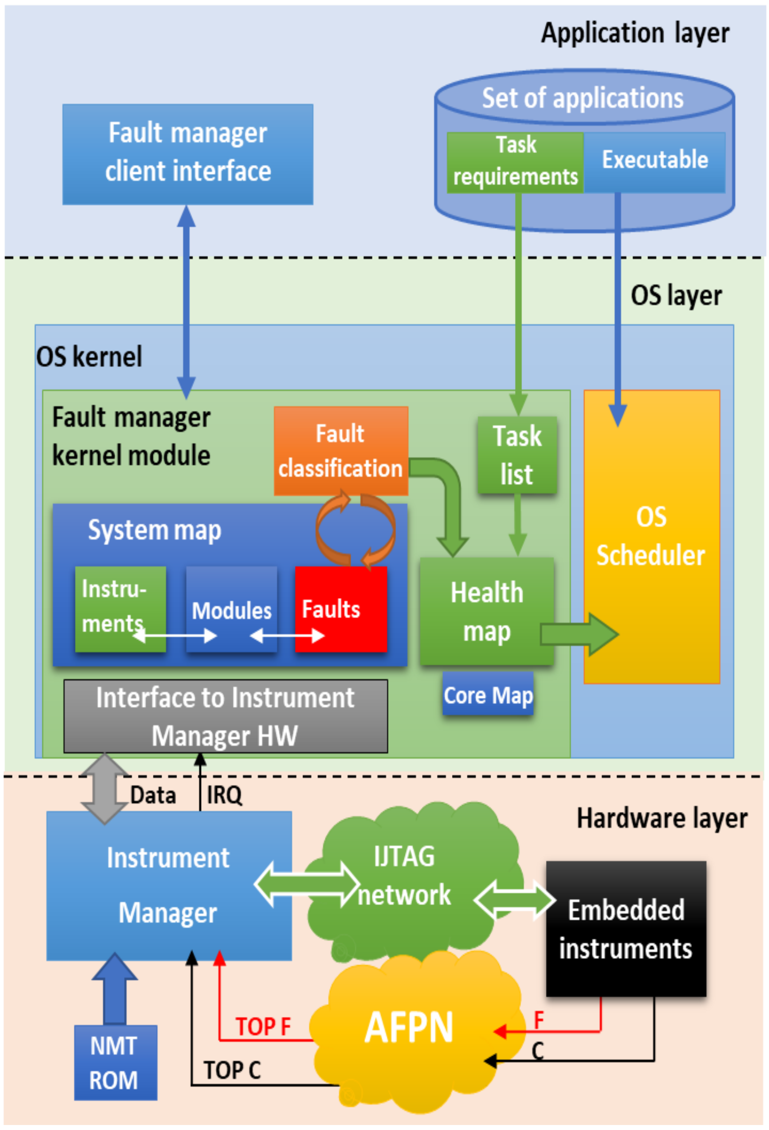
Since the fault handling is performed at higher system levels, the hardware must provide an efficient way to read and write data to the embedded instruments, as well as provide a notification about detected fault as fast as possible. All these functions are implemented in an FSM-based IJTAG network controller called Instrument Manager. It can receive and forward AFPN signals and automatically localize the source of fault signal in the RSN hierarchy. This offloads the OS by shifting these operations from software to hardware and therefore reduces fault localization and fault handling latency.

3 Fault Handling and Health Monitoring Software

This section provides an overview on how the OCFM system collects, stores, analyzes health data and performs the Fault Detection, Isolation and Recovery (FDIR) functions. Besides the embedded instruments in hardware, the health data collection can also use additional data sources, like BIST embedded into a functional resource or a software test procedure (e.g. SBST). Based on the acquired information, it can also execute additional diagnostic procedures for a fine-grained localization and classification of faults.

The OCFM is intended to work with such an OS where the scheduler could be modified to take advantage of the information provided by the fault management system and implement a health-aware or damage-aware task scheduling.

3.1 Interaction Between HW and SW in the OCFM System



While a fault could be timely detected in hardware by embedded instruments and passively tolerated by hardening and mitigation techniques (e.g. ECC, TMR), it can't be actively handled or processed in-situ. The SoC-HEALTH technology targets this aspect by enabling immediate passing of the emergency data from the HW embedded instruments to where this data could be used and interpreted – the Fault Manager (FM) software and the OS so that the latter could reschedule the affected (failed) task immediately to another available resource. After detecting the fault, this event has to be diagnosed and analyzed by the software in order to update the system Health Map (HM) and isolate the resource in case of permanent fault detection.

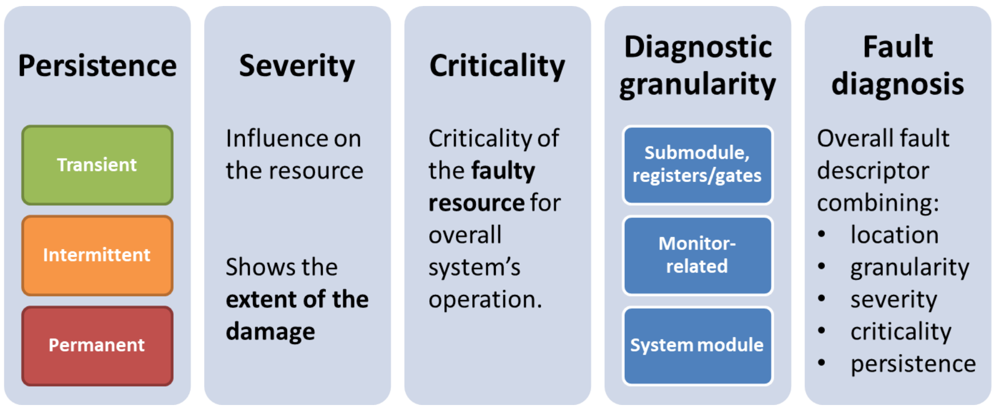
The OCFM core parts – FM and IM are closely coupled to the functional part of the system: FM is a service software that maintains the Health and Resource Maps and exchanges data with IM which has to be implemented as a dedicated hardware. Besides regular instrument access requests from the software, IM is responsible for reacting to the fault flags set by the instruments and propagated as an asynchronous emergency signal. IM automatically opens the path to the instrument that raised the fault flag and provides the information about its location to the software. However, IM can only provide a coarse-grain fault diagnosis in this manner (down to the core affected). It is the task of the FM to handle the fault, perform its classification and optionally find out the location of the fault as precisely as possible. The respective information like the fault class and its occurrence statistics are reflected in the HM. If the resource usability is compromised, the FM must also update the Resource Map (RM), which is the basis of health-aware scheduling. The SW functions also include processing the application requirements, matching them to available resources and interfacing the OS scheduler.

3.2 Health-Aware Scheduling

The health management software that is a part of the OS (as a kernel module) is the core of OCFM architecture as it is responsible for maintaining Health and Resource Maps, performing fault classification, and communicating with IM hardware. This kernel module includes several key components:

Instrument Manager driver responsible for communication with the IM HW and providing corresponding Interrupt Service Routine (ISR).

Health map (HM) is a linked list data structure that is central to the architecture and holds the detailed information about the system's functional and diagnostic resources (embedded instruments) as well as data about faults that have occurred previously. The HM maintains the statistics of fault occurrences to enable reliability prediction capability (prognostics). Since hardware faults may not disappear after system restart, HM also should not be lost. HM should be stored in a reliable non-volatile memory to maintain the prediction capability across power cycles. To facilitate that HM structure is organized in a way that is easy to serialize for storage and where new fault detection entries are always appended to the end of the occupied memory.


Persistence
Transient
Intermittent
Permanent
Severity
Influence on the resource
Shows the extent of the damage
Criticality
Criticality of the faulty resource for overall system's operation.
Diagnostic granularity
Submodule, registers/gates
Monitor-related
System module
Fault diagnosis
Overall fault descriptor combining:
location
granularity
severity
criticality
persistence

Fault classification is important for fault and resource management as well as effective system recovery. Faults are classified according to their severity levels and their contribution to the permanent malfunction of system's components and modules. In OCFM, certain properties are assigned to the faults (persistence, severity, criticality, diagnostic granularity and location).

Resource map is a data structure in the system memory that holds the information about the currently available (healthy) resources of the system. It is updated on the fly during the system's normal operation, should a fault be detected by an instrument or a diagnostic routine. FM updates the resource map based on the fault classification data. This enables health-aware scheduling based on the current health status of each module:

- *Available*: the module is available, no faults registered.
- *Own fault*: the module might have limited functionality due to a fault detected in it.
- *Propagated fault*: limited functionality due to a fault detected in child modules.
- *Maintenance*: the module is not available due to ongoing maintenance (BIST, SBST etc).

Task scheduling is performed by an OS scheduler that takes into account the information from the Resource Map (RM) when assigning tasks to resources that are healthy enough to run these tasks. RM relies of the following information:

- sub-resource availability (e.g., Floating Point Unit (FPU) inside a CPU),
- information in the task descriptor,
- reliability of the resource based on the fault statistics, and
- mission-criticality of the task.

Core Map provides mapping of the cores described as system modules to actual core IDs understood by the OS. In multi-core systems, several cores can execute several programs in parallel and each of the latter is assigned to a certain processing core ID. The assignment of core IDs to actual physical processing cores is performed by the OS. There are many possible configurations of processing cores and therefore, their enumeration in the SoC. In case of a traditional Symmetric Multi-Processing (SMP) system, all cores are identical and equivalent in function and are enumerated in hardware. In more complex cases, like Asymmetric Multi-Processing (AMP) systems, the core ID enumeration may be more complicated.

Task List is a data structure which keeps the list of all fault-managed tasks with their process IDs (PIDs), task requirements (an array of requirements in binary form) and CPU affinity masks (Binary bitfield that encodes CPU cores allowed to run a given task). This data is constantly compared to current hardware resource availability in the RM by the FM LKM, therefore relevant information needs to be kept inside the LKM memory during runtime.

Task Requirements is an array of resource requirements that describe the necessary hardware resources needed to run that task. In addition to the type and quantity of resources, task requirements can define minimum health status of a functional resource, by means of the worst acceptable fault class present in the resource. These requirements are the cornerstone of the health-aware scheduling. For example, if it is known that a task requires a specific CPU core feature (e.g., FPU), then it should not be scheduled to a core that lacks this functionality (e.g., due to a permanent fault). These dependencies are stored in an XML-format task descriptor file that contains the information about task's resource requirements. Then for each task, based on a snapshot of HM, the following task requirement evaluation procedure is executed (yielding the CAM for that task):

1. Verify that the resources of the system are sufficient to run the task and their health requirements are met.
2. Calculate the Core Affinity Mask (CAM) that shows which CPU cores can be picked to execute the task.
3. Calculate Task Fitness Index (TFI) that shows how redundant the avail- able resources are for the task.

In case of Linux, for each running process it is possible to set an affinity mask that defines which CPU cores can be used to execute a given process. This mask can be set and read using the following system calls:

`sched_setaffinity(pid, *mask)` – sets the affinity mask for a given task
`sched_getaffinity(pid, *mask)` – returns the affinity mask for a given task

Using these system calls, OCFM LKM can forbid execution of certain task(s) on certain cores if a fault was detected in the respective hardware. It should be noted that default Linux scheduler has certain limitation in the context of OCFM purposes. It will move tasks from one core to another according to the given CAM, but it will not do it immediately by stopping the task as soon as an updated CAM is received. This will limit the reaction speed of fault handling procedure. In order to overcome this limitation, the default scheduler must be modified, or a new scheduler can be designed.

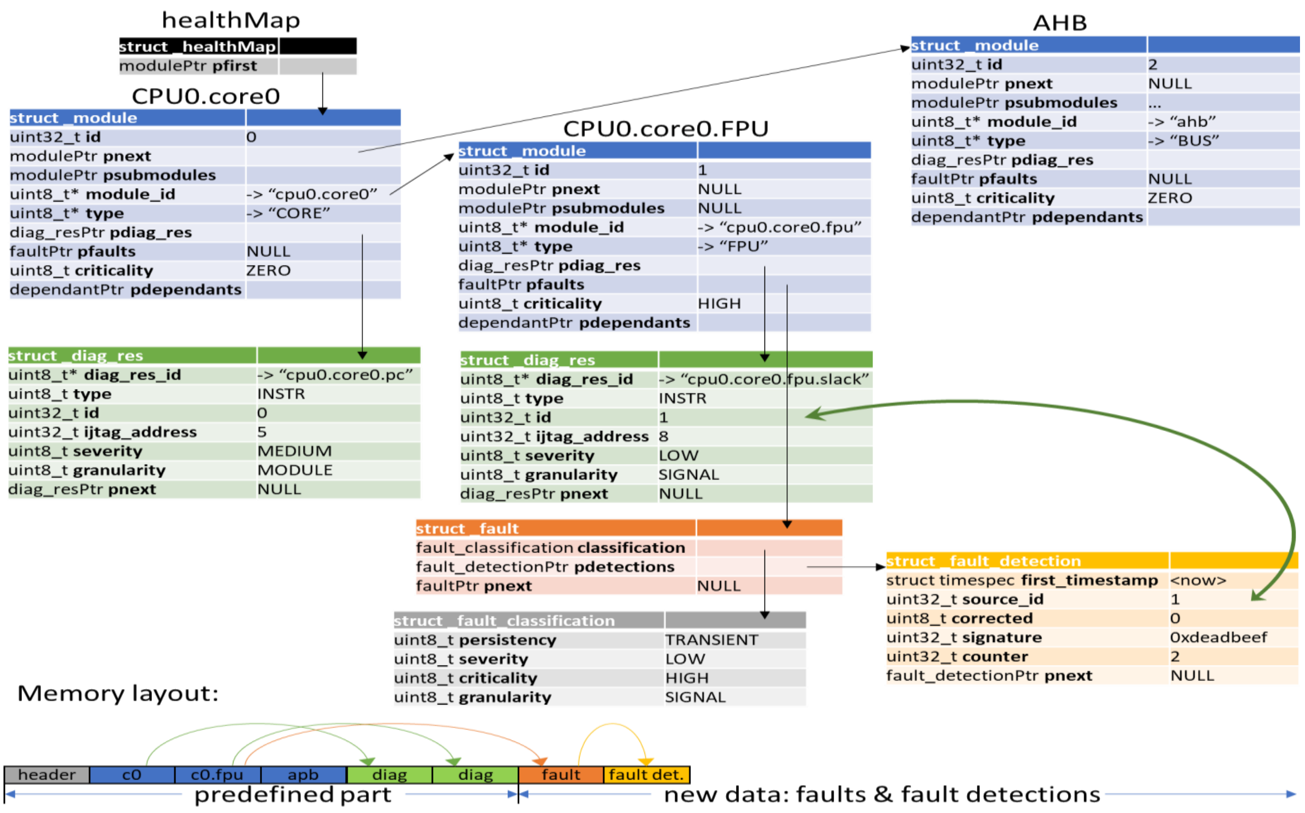
Fitness Indexes. The OCFM presumes that there is in general some redundancy in terms of used resources. In order to estimate the "*fitness*" i.e. how far the system and each particular task are from failing to function due to the lack of resources, the OCFM calculates corresponding metrics or indices: System Fitness Index (SFI) and Task Fitness Index (TFI). When the fitness index is 1, there is exactly as many resources available as it is needed for the function of a task or the system. Higher fitness index values show that there is some redundancy available, and some functional resources can still fail without compromising the functionality. Fitness indices are calculated based on the "redundancy ratio" of top-level system resources needed to execute the tasks, such as CPU cores. If a certain task requires just 2 cores to run while there are 8 cores allowed to execute this task, then the "redundancy ratio" and the TFI of this task is 4. If there are other top-level resources (e.g. communication interface) required, then TFI is equal to the lowest "redundancy ratio" among all required top-level resources. It is important to track the lowest TFI value among all running tasks because it shows which task has the least amount of available resource redundancy, and how much of it. When the TFI reaches 1 for some task, a significant fault in respective resources will lead to inability to run this task and, therefore, a system failure. SFI, is in its turn calculated as the average of TFI values of all fault-managed tasks. SFI does not correlate with current system load or performance, it shows the average redundancy of system resources w.r.t. currently running tasks.

User Command Line Interface (CLI) provides a way to control the kernel module from a user-space client program. Although FM Loadable Kernel Module (LKM) is normally working without the user intervention or control, in some cases there may be a need for to inspect system's health status or control the FM operation. The CLI application is a user space application. It expects that the system is properly booted, and FM LKM is installed and running. This software uses driver interface to send user commands to the kernel module. It allows the user to get the HM snapshots from the FM kernel module, see faults saved in HM; list, add and remove fault-managed tasks.

The health-aware task scheduling is then performed by an augmented or modified OS scheduler which takes into account the information from the Resource Map (RM) selecting the resources to be used for task execution based on 1) sub-resource availability (e.g. FPU inside a CPU), 2) task requirements, 3) reliability of the resource itself based on the fault statistics. The task requirements file contains the information about resource requirements for the task execution. This is represented by a description of the required set resource types, optionally with some reliability or environmental requirements.

3.3 The Health Map Composition

Raw data which is collected from embedded instruments is in most cases not immediately usable nor useful. Often it needs to be put into a context of previous events. A fault detected once is difficult to classify into classes like e.g. transient, intermittent or permanent, or in other words the persistence of this fault. When the same fault starts to appear periodically it is already a sign of degradation and it's then necessary to measure the frequency of fault appearance. For such a purpose, the OCFM maintains a centralized database, called the Health Map, which holds the information about functional resources, occurring faults and the relationship between those two. Moreover, HM preserves the statistics of fault occurrences in a resource, which can be then used for estimating which of the processor cores is currently more stable than the others in order to pick it for the most critical tasks.



struct _healthMap	
modulePtr **pfirst**	

CPU0.core0

struct _module	
uint32_t **id**	0
modulePtr **pnext**	
modulePtr **psubmodules**	
uint8_t* **module_id**	-> "cpu0.core0"
uint8_t* **type**	-> "CORE"
diag_resPtr **pdiag_res**	
faultPtr **pfaults**	NULL
uint8_t **criticality**	ZERO
dependantPtr **pdependants**	

struct _diag_res	
uint8_t* **diag_res_id**	-> "cpu0.core0.pc"
uint8_t **type**	INSTR
uint32_t **id**	0
uint32_t **ijtag_address**	5
uint8_t **severity**	MEDIUM
uint8_t **granularity**	MODULE
diag_resPtr **pnext**	NULL

CPU0.core0.FPU

struct _module	
uint32_t **id**	1
modulePtr **pnext**	NULL
modulePtr **psubmodules**	NULL
uint8_t* **module_id**	-> "cpu0.core0.fpu"
uint8_t* **type**	-> "FPU"
diag_resPtr **pdiag_res**	
faultPtr **pfaults**	
uint8_t **criticality**	HIGH
dependantPtr **pdependants**	

struct _diag_res	
uint8_t* **diag_res_id**	-> "cpu0.core0.fpu.slack"
uint8_t **type**	INSTR
uint32_t **id**	1
uint32_t **ijtag_address**	8
uint8_t **severity**	LOW
uint8_t **granularity**	SIGNAL
diag_resPtr **pnext**	NULL

struct _fault	
fault_classification **classification**	
fault_detectionPtr **pdetections**	
faultPtr **pnext**	NULL

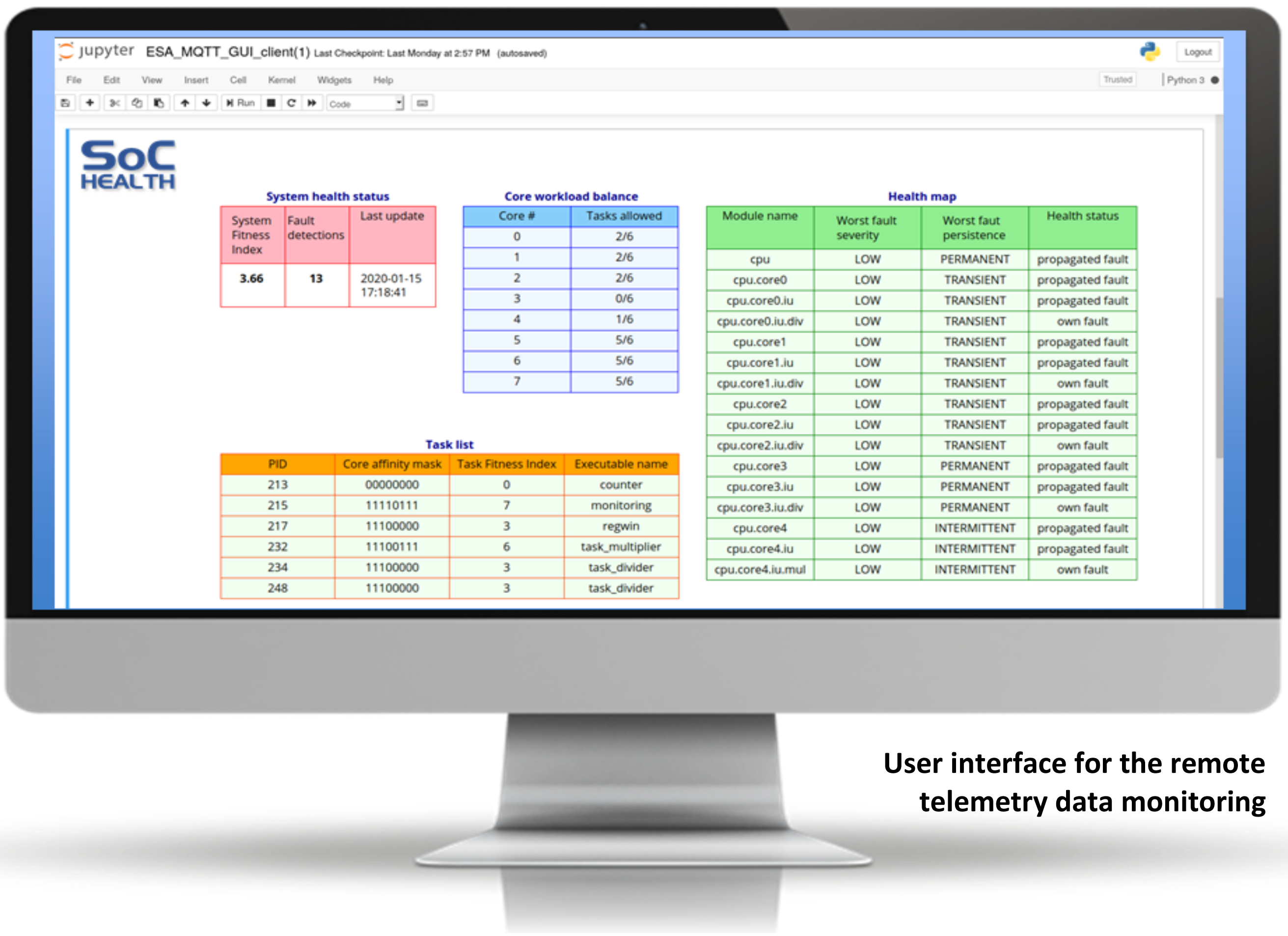
struct _fault_classification	
uint8_t **persistency**	TRANSIENT
uint8_t **severity**	LOW
uint8_t **criticality**	HIGH
uint8_t **granularity**	SIGNAL

AHB

struct _module	
uint32_t **id**	2
modulePtr **pnext**	NULL
modulePtr **psubmodules**	...
uint8_t* **module_id**	-> "ahb"
uint8_t* **type**	-> "BUS"
diag_resPtr **pdiag_res**	
faultPtr **pfaults**	NULL
uint8_t **criticality**	ZERO
dependantPtr **pdependants**	

struct _fault_detection	
struct timespec **first_timestamp**	<now>
uint32_t **source_id**	1
uint8_t **corrected**	0
uint32_t **signature**	0xdeadbeef
uint32_t **counter**	2
fault_detectionPtr **pnext**	NULL

The Health Map is a collection of data structures that represent different types of entities. The connections between different types represent their actual relationships (e.g. a fault detected in a functional resource). An array of similar entities is organized as a linked list. The figure shows a simple example of a HM with different entities: CPU cores with an FPU, faults and fault detection events. The upper part in the figure shows the relationship between different data structures (tables with different background colors), the structure members together with data types and example values. The lower part (the row) shows the memory layout of these data structures.


jupyter ESA_MQTT_GUI_client(1) Last Checkpoint: Last Monday at 2:57 PM (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code
SoC HEALTH

System health status

System Fitness Index	Fault detections	Last update
3.66	**13**	2020-01-15 17:18:41

Core workload balance

Core #	Tasks allowed
0	2/6
1	2/6
2	2/6
3	0/6
4	1/6
5	5/6
6	5/6
7	5/6

Health map

Module name	Worst fault severity	Worst faut persistence	Health status
cpu	LOW	PERMANENT	propagated fault
cpu.core0	LOW	TRANSIENT	propagated fault
cpu.core0.iu	LOW	TRANSIENT	propagated fault
cpu.core0.iu.div	LOW	TRANSIENT	own fault
cpu.core1	LOW	TRANSIENT	propagated fault
cpu.core1.iu	LOW	TRANSIENT	propagated fault
cpu.core1.iu.div	LOW	TRANSIENT	own fault
cpu.core2	LOW	TRANSIENT	propagated fault
cpu.core2.iu	LOW	TRANSIENT	propagated fault
cpu.core2.iu.div	LOW	TRANSIENT	own fault
cpu.core3	LOW	PERMANENT	propagated fault
cpu.core3.iu	LOW	PERMANENT	propagated fault
cpu.core3.iu.div	LOW	PERMANENT	own fault
cpu.core4	LOW	INTERMITTENT	propagated fault
cpu.core4.iu	LOW	INTERMITTENT	propagated fault
cpu.core4.iu.mul	LOW	INTERMITTENT	own fault

Task list

PID	Core affinity mask	Task Fitness Index	Executable name
213	00000000	0	counter
215	11110111	7	monitoring
217	11100000	3	regwin
232	11100111	6	task_multiplier
234	11100000	3	task_divider
248	11100000	3	task_divider

User interface for the remote telemetry data monitoring

3.3.1 *Accumulation of fault event data in HM*

When the system is fresh, it is considered fault-free. Hence, an initial "clean" HM generated from the module's XML description is used. It would only contain data structures for the functional modules and respective diagnostic resources. It may also contain dependencies. During the system's lifetime, faults may occur and if detected by OCFM, would trigger creation in the HM new data structures dedicated to faults, their classification and occurrences.

3.3.2 *HM entities*

Functional hardware resources (modules) e.g. CPU cores, their sub- modules like FPU, shared re-sources like buses and so on represent a large group of entities. All sibling modules are organized into a linked list, while each module can also have submodules in order to represent actual hardware organization. Each module also has a type, a set of associated diagnostic resources (e.g. embedded instruments), a list of detected faults and several other fields in the data structure used to store information about the module.

Diagnostic resources are various embedded instruments that supply diagnostic information about system health and fault events. Other types of diagnostic resources, like e.g. SBST, BIST, or system watchdogs are also supported. Diagnostic resources always belong to a certain functional hardware resource. This relationship and other properties of diagnostic resources are stored in the HM.

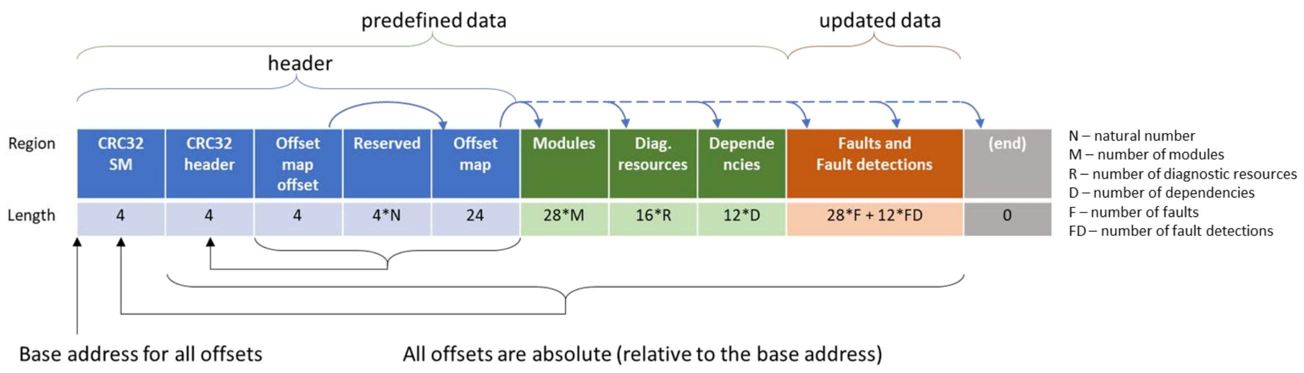
Faults and their classification status are reflected in the respective HM structure. As system modules accumulate faults during the lifetime, all relevant data is being captured and stored permanently in the HM. This ensures that faults, their occurrence patterns, and other meta-data can be analyzed and extracted at any time later.

Fault detections. The same instrument may detect a fault several times and potentially with different properties. In HM fault detections are always associated to particular faults and have numerous parameters. In the event of fault, software creates or updates a fault detection record to keep the details of the event. In case of subsequent identical fault events that land into predefined time period (e.g. 10 milliseconds) the system increments the fault detection counter.

Functional dependencies. In some cases, i.e. when correct functionality of one module depends on operation of another module in a system, such a relationship between functional modules (system resources) needs to be stored in the HM. It can be a peripheral connected to a certain bus: e.g. when AHB or its part fails, the Ethernet (a peripheral) may become inaccessible and hence unusable. In order to keep track of the effect of faults towards other modules, a special data structure in HM provides a link to a dependent resource and the strength of that dependency.

3.3.3 Safe storage of the serialized Health Map

The Health Map is intended to contain the information about the status of all system's resources and since faults are occurring in hardware and do not disappear after system restart, HM also should not be lost. To retain the information about the known faults in the system when the system is powered off, HM should be stored in a reliable non-volatile memory to maintain the prediction capability across the power cycles. However, the access to the memory is performed by FM which is executed as a part of the OS kernel. To help protect the contents of this memory in general, as well as from an error-prone behavior of the OS, HM memory itself should not be directly accessible, but rather a special error-tolerant controller should be employed to control the access to the memory, which either could be a binary file or an EEPROM.



HM software is capable of serializing system map objects. Serialized Health Map (SHM) occupies subsequent uninterrupted memory region. This allows to incrementally update the saved safe version of the SHM as well as unroll it into manipulatable data structures when reading it back into the RAM.

SHM has specific layout in memory that facilitates (de-)serialization and updating with new fault information. It is largely divided into two parts. In the predefined (constant) part, the header and information about system's resources have to be prepared once at hardware design stage. This part would be the same for all systems with identical hardware. Updated (dynamic) part contains data about faults occurred throughout the system lifetime. This data is unique for each individual physical system. HM is designed in such a way that new data structures are appended to the end of HM. This allows to store SHM in a continuous memory.

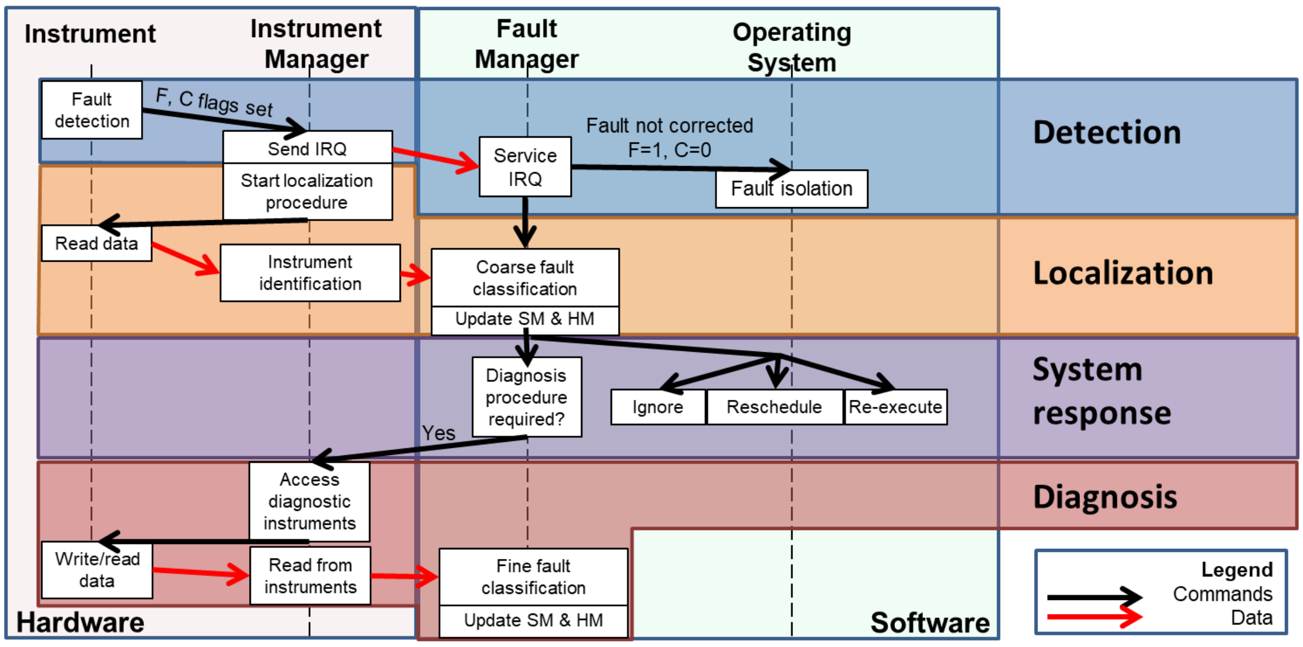
All memory offsets (shown as blue arrows above the layout structure) are stored as absolute byte offsets using the beginning of HM as the base address. This ensures easy relocation of the HM (at any address in RAM, in a regular file, in an EEPROM, etc.) because the offset values do not depend on the type of the memory used.

3.4 Fault Handling Flow

When a fault occurs in a complex SoC working under the control of an OS, it is necessary that the latter becomes aware of the fault as quickly as possible. The OS must then take actions to isolate and mitigate the effects of the fault. In OCFM, the fault event is processed by four actors: Instrument, Instrument Manager, Fault Manager and the OS (see the figure).

Fault detection. Whenever a fault is detected by an instrument, the information about this event is quickly passed to the IM through AFPN. In response, IM sends an interrupt request to a CPU which is executing the OS kernel, which, in turn invokes the FM to service this interrupt. Since the nature of the fault is not known at this stage yet, the FM software executes necessary procedures in order to isolate the fault and contain possible error propagation.

Fault localization. Concurrently with sending the interrupt request, the IM starts the instrument localization procedure. During this procedure, the IM will subsequently open those hierarchical IJTAG network segments with the Fault flag set. As soon as the instrument, which has raised the flag has been reached, the IM reports the location of the fault expressed as the position in the IJTAG network.



Coarse-grained fault classification. Based on the information about which instrument has raised the Fault flag, the FM can perform coarse classification of the fault. Since the detailed diagnostic information about the fault is not available at this time, the diagnostic granularity is limited by the granularity of the instrument location (e.g. it is attached to a CPU or FPU, or some particular checker). However, in some cases the criticality of the resource can be determined by the type of affected system resource. In any case, the Health and Resource Maps are updated accordingly.

System response. Based on the information derived in the previous step, the OS may need to take actions to mitigate the effects of the fault on the functional operation of the system. The fault can be ignored if it does not affect the operation, or the task was not a critical one. Alternatively, the task can be rescheduled to another resource or re-executed on the same one later. When the required actions are taken, CPU cores are released.

Diagnosis. Depending on the outcome of the coarse classification step, the FM system may decide to get more detailed diagnostic information by executing a diagnostic procedure. The IJTAG network is used to communicate with instruments (such as BIST or other DfT hardware) to fetch additional information about the fault event. This communication channel is managed by the IM.

Fine-grained fault classification. The diagnostic procedure updates the FM with the new information used to perform the fine fault classification and update both Health and Resource Maps.

3.5 Fault Management Use Cases

Below are several use case examples involving health-aware scheduling and reaction on fault events.

Task start. Whenever a new fault-managed task must be started, user-space FM CLI utility will send the information about the task and its requirements to FM LKM. The latter will then add this task to TL, evaluate the requirements and return the result to the user space.

Task ended. Task's entry must be removed from the TL as soon as the task has ended. The task may finish normally or can be terminated prematurely externally.

Task deleted. A running fault-managed task can be deleted from the FM CLI by the user. The CLI will terminate this task and send a message to FM LKM to remove the task from the TL.

New fault event. Whenever a new fault is classified and added to the HM, as a result, the RM is updated. After that, the requirement evaluation procedure should be run for the affected tasks. In order to avoid unnecessary calculations, a simple optimization is applied: in case new fault detection affected a module or submodule of a CPU core, only tasks which have that core allowed in their affinity mask should have their requirements re-evaluated.

Resource Map update. As a result of the RM update, CAMs of the tasks can:

- *Remain unchanged*. The fault did not significantly affect the resources required by the task and there were no changes in the set of cores that can run the task. According to the task requirements, task may need to be re-executed.

- *Change*. The set of cores which are allowed to run the task has changed and the new task CAM must be sent to OS scheduler. According to the task requirements, task may need to be re-executed or re-scheduled.
- *Deplete*. There are no more cores left allowed to run the task and the latter must be terminated by the FM LKM.

3.6 Chapter Conclusions

In the SoC-HEALTH OCFM concept, the fault management software has two important functions: the health-aware task scheduling (slow-paced activity) and the immediate fault handling (fast emergency activity) with subsequent task recovery. Both functions fully rely on the underlying OCFM hardware described in the previous chapter.

In order to perform these two key functions, the OCFM SW also maintains specific data structures reflecting the underlying system architecture, location of diagnostic instrumentation, system health information, fitness status, etc. as well as it implements a number of data processing functions, such as fault localization and classification, health map and resource map updating, system fitness calculation, servicing IRQs from the IM and many others.

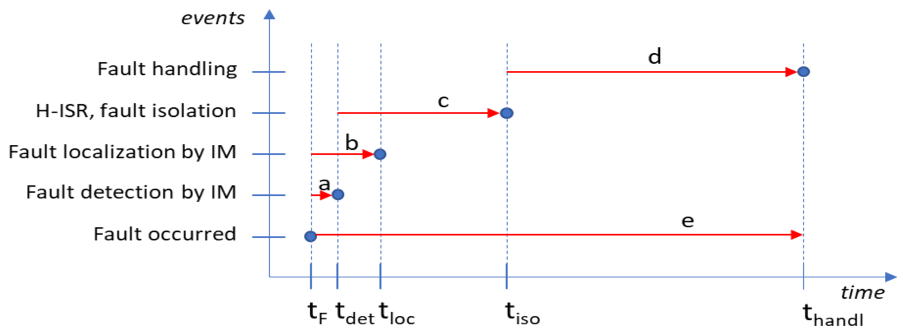
In SoC-HEALTH project we aimed at minimum intrusion into the Linux OS, being actually able to master available resources, implementing the health-aware scheduling based on the original unmodified Linux scheduler. It is possible using system call `sched_setaffinity()` that allows to explicitly define a subset of cores that should be used to execute a certain process. Therefore, in this setup, the Resource Map exists in an implicit form of the CPU affinity information that is provided by FM to Linux scheduler.

The key fault management tasks are carried out by specially developed FM software in the form of a loadable kernel module, which is the central part of the architecture. The list of managed processes is stored in the kernel module, it contains process identifiers, and the associated record of resources (modules/submodules) needed for execution of each managed process. With the information from the process list and HM, the system is always aware of how to schedule the tasks to available resources.

4 OCFM Performance Assessment

This chapter presents a performance assessment of the implemented OCFM in terms of system reaction latency. Critical latency parameters were measured: fault detection and localization, software response to interrupts and fault handling time. The results show that the reaction latency of the OCFM system in SoC-HEALTH FPGA based demonstrator is dominated by the software part including the OS and interrupt handling. The overall fault handling latency is in the order of 10^{-3}s.

In order to evaluate the performance of this system, we have conducted a measurement of reaction times of the solution proposed for this case study to find out how fast the system can react to an unrecovered fault. For the experiment, fault in one of the CPUs was emulated in a controlled manner, by injecting a fault into different instruments (one at a time). We measured the time required for different stages of the system's reaction to this fault on the real hardware.



Fault detection, isolation, and handling latency timeline

Several timing parameters were measured using logic analyzer embedded into the hardware. The measurement time starts when the Fault flag is raised, and it is denoted by t_F. The following measurements were made for respective fault handling stages:

a) **Fault detection latency (in HW)** - the time needed for the Instrument Manager (IM) to raise high priority interrupt ($t_{det} – t_F$).
b) **Fault localization latency (in HW)** - the time needed for the IM to automatically localize the instrument which has raised the F flag and raise the low Interrupt Request (IRQ) ($t_{loc} – t_F$).
c) **OS interrupt latency** - the time needed for the OS to react to high priority interrupt and stop the CPU cores to isolate the fault ($t_{iso} – t_{det}$).
d) **Localization and classification latency** - the time needed for the FM kernel module to handle the fault (localize and coarsely classify) and resume the execution ($t_{handl} – t_{halt}$).
e) **Total fault handling time** – from fault occurrence to resumed operation ($t_{handl} – t_F$).

Measurement		**Time, µs**	**Time, clock cycles**
a) Fault detection latency	*const*	**0.074**	3
b) Fault localization latency	min–max	0.97–8.42	37–342
c) OS interrupt latency	min–max	38.45–52.75	
	average μ_c	**47.39**	
	std.dev. σ_c	3.62	
d) Localization and classification latency ($\mu_e – \mu_c – a$)	average μ_d	**5156**	
e) Total fault handling time	min–max	4922–5800	
	average μ_e	**5203**	
	std.dev. σ_e	213	

CPU cores were running at 81.25 MHz while IM and IJTAG network at half that frequency. Since the interrupt latency of standard Linux kernel is nondeterministic and can vary from run to run, the results of the third and fourth stages are averaged over 10 runs. The measurements are given in the table above that shows fault detection latencies for every stage in TCK cycles and microseconds (F_{TCK}=40.625MHz).

5 OCFM Technology Applications and Future Work

The European Space Technology Harmonization roadmap addresses the continuous miniaturization and speed/power optimization for the microelectronics by a number of ongoing activities, which will collectively produce an enabling ecosystem for European high-performance on-board computing. Today, space missions, especially those serving telecommunication and Earth observation purposes are in need for more processing power. According to the Technical Dossier on Microelectronics: ASIC and FPGA[18], the ESA has a strategic interest in minimizing the dependency on export restrictions, while improving the competitiveness, availability and performance of European space ASIC and FPGA technology, which is always needed at the very heart of all space avionics. Today, the key strategic challenge for Europe is to make advances in the performance and functional capabilities of European satellites (*e.g. deep sub-micron (DSM) ASIC for next generation telecom payloads*), while achieving high levels of miniaturization and speed/power optimization for the microelectronics. These drives future developments in the domain of microprocessors, FPGAs and ASICs (both standard and proprietary) to address the increase of processing needs in terms of throughput, bandwidth, and performance of platform and payload equipment, while still decreasing power, size, and cost.

The Eurospace RDT Priorities[19] consolidates emerging needs of the European space industry with regard to research development and technology. In the domain of EEE components and electronics building blocks Eurospace sees high speed processing for intensive image and data processing as a key area for action, including activities such as new DSPs, co-processing, and DSM multi-core processors.

5.1 The OCFM Technology Application

The OCFM technology follows these trends by exploring the natural redundancy of the multi-core devices for health-aware task scheduling purposes. Acting as a middleware between HW checkers/sensors and mission applications, the OCFM thus naturally enables Health Awareness functions, valorizing thereby the developments of next generation multi-core processors both in the space industry but also in the terrestrial economy, especially in the domains that require high availability.

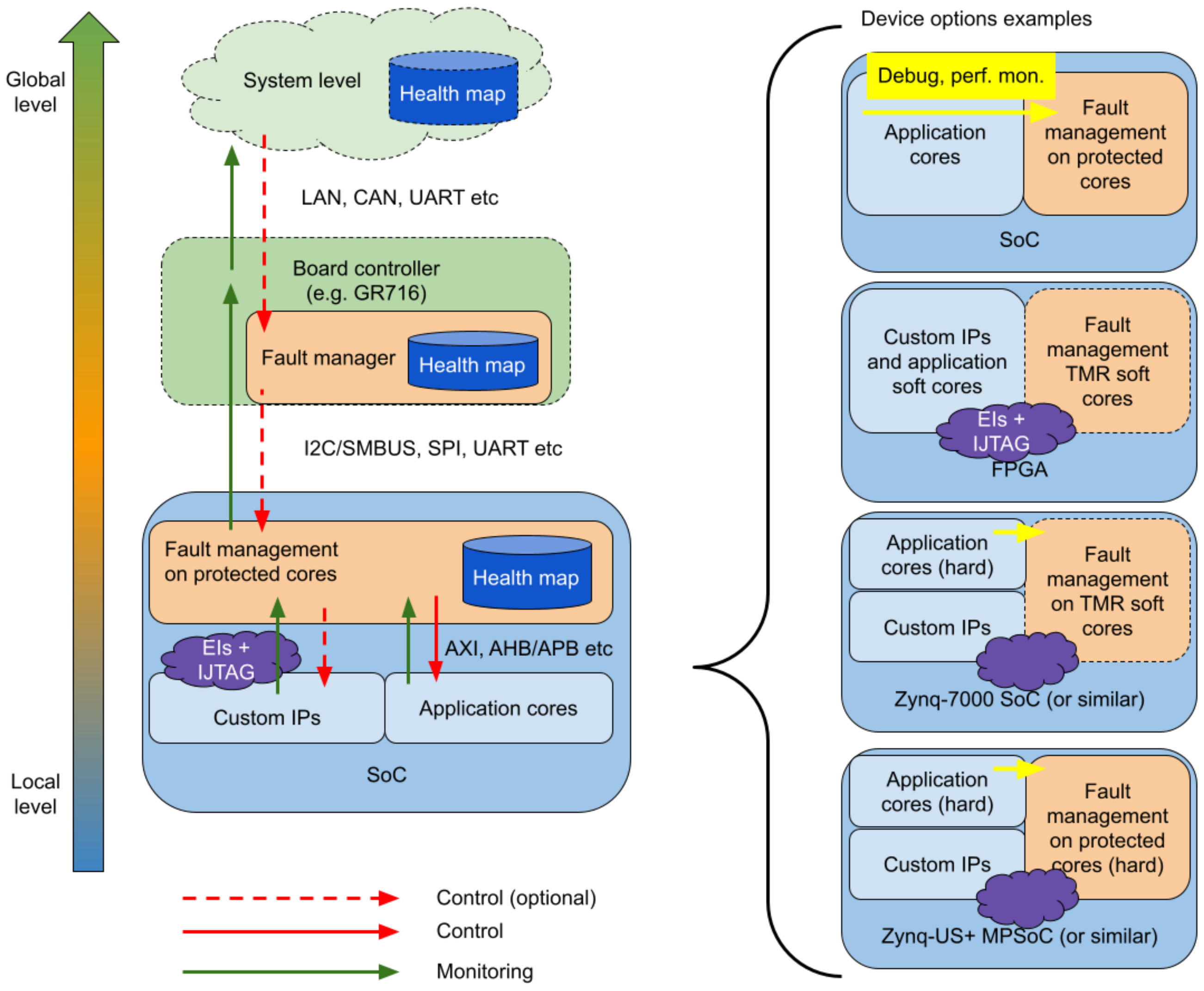
The first-priority target customers for the OCFM technology are the companies producing high-cost electronic equipment for mission-critical systems (e.g. automotive, aerospace, and health-care) or systems targeted for long-term continuous operation (telecom). For instance, the Advanced Driver Assistance Systems (ADAS) are expected to revolutionize the automotive market, elevating the demand for highly-reliable electronics, including sensors and processing power. The growth of combined ADAS and automotive safety systems market is expected to create a demand towards system health monitoring and fault management solutions like the OCFM developed in frames of the SoC-HEALTH.

[18] Technical Dossier on Microelectronics: ASIC and FPGA; ESA/IPC/THAG(2016)
[19] Eurospace (2016); Space RDT priorities 2020: the incremental roadmap of technology research and development activities for space; https://eurospace.org/wp-content/uploads/2018/05/eurospace-rdt_2020_web.pdf

The OCFM health management system and respective methodology is highly flexible and adaptable to a very broad range of target applications across market segments. The software can be plugged-into an existing operating system without kernel modification and the health management functions can be easily switched on and off on the fly, which simplifies its adoption and safety certification.

5.1 Adapting SoC-HEALTH Project Results to Heterogeneous Systems

In the near-term, we foresee the application of SoC-HEALTH project results by adaptation of the OCFM framework for systems like the High-Performance Computing Board (HPCB), which is now being developed at ESA as a primary data processing platform for space satellites. The HPCB platform represents a heterogeneous system comprising of the central controller, communication module, computation units, and combines such diverse components like MCUs, FPGAs and application-specific accelerators.



The health monitoring architecture in heterogeneous systems can be implemented within a single SoC or span across several hierarchical levels, especially in a large complex system where many devices are connected to each other on one or several boards. The figure shows an abstract example of such relationships between hierarchical levels in a complex system and respective devices where it could be implemented. Depending on the level of hierarchy and the purpose of a device, they can contain execution units and/or local health management controllers or board/system level controllers. These devices fall into several classes that have to be considered in health management architecture:

- FPGA with user IP cores, e.g. Kintex UltraScale, etc
- CPU application cores, e.g. Zynq-7000 / UltraScale+ Cortex-A, NXP Layerscape LS1046A
- CPU protected cores (TMR or lock-step execution), e.g. ZynqUS+ Cortex-R, TMR uBlaze
- Board/system level controllers, e.g. GR716

The following table lists the roles of these device classes in the health management architecture.

	FPGA fabric with user IP cores	**CPU application cores**	**CPU protected cores (TMR, lock-step execution)**	**Board/system controller**
Examples	- Any FPGA	- Cortex-A9 (Zynq7000) - Cortex-A53 (Zynq-US+) - Soft cores (Microblaze, RISC-V etc)	- Cortex-R5 cores in Zynq-US+ - TMR Microblaze	- GR716 Protected MCU
Monitoring	- Embedded instruments and sensors	- Debug/fault registers - Performance monitors - Interrupts - Temporal redundancy - BIST	- Fault registers - BIST	- On-board sensors - Aggregated health data from other devices lower in the hierarchy
Active control in case of faults	- IP-specific controls - Reconfiguration	- Task scheduling according to hardware resource health - Task re-execution or re-scheduling	- Restart - Disable access to faulty resource	- System reboot - Mode Control (Normal, Low-Power, Safe Mode etc) - Disable access to faulty resource
Type of collected health data	- Raw sensor/monitor values	- Performance counters - Fault/exception flags - BIST results	- Fault/exception flags	- Sensor readings - Health data from units lower in hierarchy
Type of reported health data	- IP-specific health data	- Core health status - Core sub-resource health status	- Core health status - Core sub-resource health status	- Component health status - Sub-component health status - Parameters (temperature, voltage etc)
Health management modules	- Embedded instruments	- Local fault manager + Health map - Health-aware task scheduler	- Local fault manager + Health map	- System/global fault manager + health map

The columns here represent the classes of components. From the left to right, the ordering of the component classes also represents the location in the hierarchy from execution units to system-level controllers while CPU application cores potentially can perform both task execution and health monitoring. The rows of the table show the properties of component classes:

- **Examples**: representatives of the class.
- **Monitoring**: means used for monitoring/detection of faults in functional resources for this class.

- **Active control in case of faults**: means used for control of functional resources in case they are found to be faulty in order to avoid erroneous behavior (active failure avoidance).
- **Type of collected health data**: the types of data collected by the health management system for this component class.
- **Type of reported health data**: the types of information which can be extracted from the collected health data.
- **Health management modules**: parts of the health management system which could be accommodated by this component class.

The potential results of such follow-up activities can be directly applied to aerospace applications (i.e. those being today developed in ESA, NASA or by other space agencies/companies) as well as to other market segments such as companies producing high-cost electronic equipment for safety-critical systems (e.g. automotive, health-care) or systems targeted for long-term continuous operation (telecom).

5.2 Higher Level OCFM Functions for AI-Based Systems

Another large and promising OCFM application domain is the Artificial Intelligence (AI) in the broad sense. There is no doubt that today we are witnessing the dawn of the AI age. Along with general-purpose Machine Learning and Deep Learning platforms such as e.g. TensorFlow, the AI paves its path to our everyday life. Smart homes, smart cities, smart factories are a few examples of intelligent automation and control technologies boosting comfort and productivity of industries and human beings, improving their quality of living. Automotive industry is undergoing a revolutionary leap in intelligent transportation, while the aerospace segment players including ESA consider employing AI in future space missions for a wide range of applications such as automated decision making, plan and schedule execution, monitoring and control, image classification and data analytics.

Coming back to Earth, we see increasing usage of AI in big data analysis for various purposes such as medical treatments, calculation of insurance and loan rates, weather forecasting, crisis prediction, etc. All in all, the humankind already relies on technologies, where electronic processing, data collection and control systems play the key role. Essentially, high-performance computing systems is the cornerstone behind the AI, it's their body and the brain. Just like us, the humans, the AI systems need healthy "organs" to operate correctly. The OCFM methodology offers a bottom-up approach for in-situ big data collection and analysis, enabling instant failure recovery, predictive maintenance, self-healing and self-adaptation to damage.

Today, with the rise of AI, the electronic systems gradually become intelligent enough to become self-aware and situation aware. These two concepts are being extensively studied today with active R&D work ongoing both in academia and industry. An important aspect in Self-Awareness is the ability of the system to comprehend and maintain its fitness in context of its goals and decisions as well as to adapt the decisions in accordance with the system fitness and the current situation. This specific ability, the Self-Health Awareness is the next emerging paradigm on the path starting from the classic passive Fault Tolerance through Error Resilience and Fault Detection Isolation and Recovery (FDIR) towards intelligent reliability management solutions.

6 Conclusions

This document summarizes the development and evaluation done in frames of the SoC-HALTH project, which aimed at integration as well as the evaluation in terms of performance and HW overhead of the OCFM technology that comprises an original HW architecture, fault handling and system health monitoring methodology, and respective software functions.

The SoC-HEALTH OCFM framework is a semiconductor technology independent, abstract, and universal solution, which mainly aims at multi-core processing SoC (ASIC) applications. The future work section summarizes foreseen developments needed to further extend the OCFM technology application scope towards heterogeneous architectures.

The spectrum of faults targeted by the OCFM framework spans from transients to wear-out, whereas the capability of handling these faults fully depends on the embedded instrumentation used in each particular target system. Still the general philosophy behind assumes a scenario of gradual degradation of the protected system. Hence during the normal lifetime, the Health Map is just updated with statistics of transients, but as the damage accumulates, the Health Awareness functions would help the system to cope with permanent faults by employing health-aware task scheduling as well as immediate fault handling and operation recovery.

Active fault protection methods, such as OCFM, would normally go beyond a simple fault masking and need to be applied at different levels. Therefore, a cross-layer approach including higher system levels as well as in-situ hardware checkers has been implemented. The OCFM framework collects the diagnostic information from embedded instruments through IEEE 1687 IJTAG network in hardware as well as from testing and diagnostic routines in software. Such a statistical information as well as fault detection events are stored in system's Health Map helping to track the underlying system's degradation trends.

Another important function of the OCFM is the immediate fault handling through fault detection, isolation, classification, and recovery before the fault leads to a system failure. This includes the estimation of the fault parameters and their impact on health of system's functional components. The fault event statistics is also taken into account during the isolation and recovery actions. The fault classification categories are specifically developed for complex multi-core SoC applications.

Actual faults occurring in the system due to different reasons may cause errors in registers, which subsequently may lead to failures and system malfunction. In-situ fault detection and fitness monitoring right inside the hardware provides the fastest way to react to fault events. For each fault event, the OCFM hardware would then provide both the detection and the basic diagnostic information to the OS through the dedicated health monitoring network.

The TRL 4 SoC-HEALTH demonstration platform is comprised of an octa-core LEON3 soft CPU augmented with OCFM facilities and implemented on Kintex7 FPGA as well as Linux OS enhanced with fault management functionality for fault handling and health-aware scheduling.

The data collection and emergency signaling HW infrastructure based on an IEEE Std. 1687 (IJTAG) network that collects real-time data from 1262 checkers embedded right inside the CPU offers immediate fault detection and diagnosis. The OCFM achieves full performance with low overhead in terms of latency, system load and used area (around 10% area in HW overhead).

The experimental evaluation has demonstrated that the OCFM hardware can handle over 100000 faults per second with the average single fault processing time being in the order of 10^{-3}s, which confirmed the expected high efficiency of IJTAG-based fault and fitness data collection approach. Considering that faults should generally occur rarely in the system's intended operating conditions, this represents a small overhead in system load.

Future work in fault management and active fault tolerance has to be primarily focused on further improvement of the fault handling functions and scenarios in the software and applications to heterogeneous high-performance computing platforms being currently developed by ESA.

The experimental results show that the OS interrupt latency has the dominating contribution to the fault detection and isolation latency. Therefore, methods for reducing this part need to be evaluated and applied in the future work to further improve the performance of the fault management procedures.