Run-time Hardware Reconfiguration of Functional Units to Support Mixed-Critical Applications

Raphael Segabinazzi Ferreira[1], Jörg Nolte[1], Fabian Vargas[2], Nevin George[1], Michael Hübner[1]
[1]*Brandenburg University of Technology, Cottbus, Germany*
[2]*Electrical Engineering Dept., Catholic University – PUCRS, Brazil*
{R.SegabinazziFerreira, Joerg.Nolte, Nevin.George, Michael.Huebner}@b-tu.de
{vargas}@computer.org

***Abstract*—System reconfiguration of hardware resources has been done in multiple system domains. Such systems are usually found in the context of FPGAs, where reconfiguration is done usually over its primitives (e.g., LUTs, Flip-Flops). Or even in the context of MPSoC designs, where core management (e.g., lock-step operation in multi-core designs) is the most used approach. However, recent works have shown that configuration at Functional Units (FUs) granularity might come with benefits. For example, it can increase the configuration space due to its finer granularity, and, as a consequence, the options to deal with problems (e.g., due to aging) in the units itself. Within this context, this paper presents a system capable to configure its FUs (e.g., ALUs, multipliers, dividers) into different operation modes. The system uses an Operating System to control HW reconfiguration during process switching time and takes into account the health state of its units in a mixed-criticality applications scenario. Results show that, within this scenario, the system is able to reconfigure itself accomplishing health state modifications of its HW elements.**

***Keywords*—Reconfiguration, Configuration, Functional Units, Fine-Grained, Mixed-Criticality, Run-time.**

I. Introduction

Systems targeting reconfigurability appear in different domains and contexts. In the context of FPGAs, mostly primitives (e.g., LUTs, Flip-Flops) and partitions based reconfiguration approaches can be found [1] [2].

In the context of Multi-Processor Systems on Chip (MPSoC), monitor elements have been added to designs, and, according to feedback from these monitors, reconfiguration is performed at core level (either hard- or soft-cores) [3] [4] [5]. Yet other works, in the same context, disable internal elements of the processors, in order to minimize the amount of time critical instructions remain in the CPUs internal buffers, which are usually unprotected [6].

It is, however, known that units inside processor designs do not equally execute instructions, and, as consequence, do not age homogeneously [7]. To compensate these effects, fine-grained management of Functional Units has been proposed in prior work [8] [9].

Contribution: In this article, we present different operating mode configurations created to guarantee system mixed criticality requirements even in the presence of health state degradation of processor Functional Units (e.g., ALU, multiplier, divider). Switching between these operation modes is done at run-time. To do so, an Operating System (OS) was extended to allow this configuration during process switching time.

The following sections of this article present, respectively: its related work (II); a description the FUs health states and the foreseen criticality levels used as the Reconfiguration Parameters (III); the operation modes configurations (IV); how the Reconfiguration Parameters are combined with the Operation Modes to create the reconfiguration platform (V); the overall design and its internal blocks (VI); the performed evaluation of the platform with simulation results (VII); and, finally, the conclusion of the work is tailored (VIII).

II. Related Work

Shibin *et al.* [3] presented a multi-core system attached to a health monitoring infrastructure capable of monitoring processor healthiness due to HW instruments placed in different parts of the processor. The hardware instruments are attached to the Internal JTAG (IJTAG) infrastructure, and an Operating System (OS) keeps a database of the health state of each one of the monitored HW elements. The authors claim that running on top of this HW infrastructure, an OS can use this database to better chose processor cores to run its applications.

The work presented by Sadighi *et al.* [10] proposes a self-aware and a self-adaptive methodology for autonomous systems. It mainly establishes operation points and controllable deviation spaces for possible next operation points to react in case of system variations. It also suggests different analysis methods for critical and non-critical tasks.

Nya *et al.* [11] proposed a self-aware and a self-expressive system for fault tolerance. Self-awareness consists of fault detection and prediction by monitoring systems parameters such as CPU temperature and execution time of threads. In addition, the self-expressive part is formed by its recovery operations.

Segabinazzi and Nolte [8] proposed a mechanism to configure internal processor Functional Units (FUs). It extended a processor design with new units responsible for catching specific added commands and configuring its internal units

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 722325.

978-1-7281-8731-0/20/$31.00 ©2020 IEEE

accordingly. The main advantage of this work is its low latency necessary to perform this configuration.

The work proposed by Segabinazzi *et al.* [12] presents a preliminary work of configurable HW schemes for fault detection and correction over processor units. The main idea is to enable awareness of the health state of a processor and its internal units and, at the same time, due to its awareness, provide optimal configuration of the HW schemes according to application requirements.

From the best of our knowledge, there are very few works proposing system configuration at FU level and, at the same time, considering a full system integration being able to, not only monitor, but also configure itself at this level. Therefore, this paper addresses this topic by proposing an OS-based management system with different operation modes for configuration of FU.

III. Reconfiguration Parameters: Criticality levels and Functional Units Health States

In this work, we are proposing an Operating System (OS) controlled fine-grained configuration of processor Functional Units (FUs). Thus, different FUs configurations were defined, and Operation Modes address each configuration. These configurations were pre-defined to take into account the health state of the available FUs and to cover, at least, three different levels of OS processes criticality: ordinary, medium critical and high critical.

The considered fault model for health state classification is the intermittent soft faults (e.g., single event transients and upsets) that becomes more frequent due to aging of electronics [13] [14]. Consequently, the FUs' health state classification is done according to the number of fault detection events caught by assumed individual fault monitors in the FUs. For that, thresholds were set for each health state. Note that, in order to find appropriate numbers for these thresholds, additional studies are required. So, this process is beyond the scope of this paper. Therefore, the health states and their symbolic thresholds are listed below.

- **Healthy (0)**: No events detected at the FU, or very few events detection below the $MEDIUM\_HEALTHY\_THRESHOLD$.
- **Medium healthy (1)**: Only very few events detected above the $MEDIUM\_HEALTHY\_THRESHOLD$ but below the $INTERMITTENT\_THRESHOLD$.
- **Intermittent (2)**: Events detection above the $INTERMITTENT\_THRESHOLD$ but below the $FAULTY\_THRESHOLD$.
- **Faulty (3)**: Detection of events above the $FAULTY\_THRESHOLD$.

IV. Operation Modes Configurations

As stated in section III, different operation modes were created to cover the possible combinations of the Reconfiguration Parameters. Therefore, the configuration for each operation mode (OpMode) is defined as the following (Table I):

TABLE I
Operation Modes and its measures

Operation Mode	d.s.	FD	FC	DMR	TMR	EA
0	-	-	-	-	-	-
1	-	✓	-	-	-	-
2	-	✓	✓	-	-	-
3	-	✓	✓	✓	-	-
4	-	✓	-	-	✓	-
5	✓	✓	-	✓	-	✓

Degraded service (d.s.); Fault Detection (FD);
Double Modular Redundancy (DMR); Fault Correction (FC);
Triple Modular Redundancy (TMR); Error Analysis (EA);

- OpMode 0 - Generic: the FUs are working as usual, no redundancy and no other mechanism is enabled.
- OpMode 1 - Generic plus fault detection (FD): FUs working as usual; however, a fault detection mechanism is enabled at the working FU.
- OpMode 2 - Generic plus fault detection (FD) and correction (FC): FUs working as usual, but a fault detection and correction mechanism is enabled at the working FU.
- OpMode 3 - Double modular redundancy (DMR) plus fault detection (FD) and correction (FC): DMR scheme in the FUs plus an extra fault detection and correction mechanism enabled at the working FUs.
- OpMode 4 - Triple modular redundancy (TMR) plus fault detection (FD): TMR scheme in the FUs plus a fault detection mechanism at the working FUs.
- OpMode 5 - Degraded service (d.s.) - Double modular redundancy (DMR) plus fault detection (FD) and error analysis (EA): DMR scheme in the FUs plus individual fault detection mechanism in the FUs to help on right answer decision in case of divergence in the outputs of the FUs.

V. Operation Modes and the Reconfiguration Parameters

A configurable system was defined by combining the Reconfiguration Parameters (process criticality and the FUs health state) and the Operation Modes configurations. The possible configurations were outlined in such a way that each process has a set of possible operation modes to run according to its critical level and FUs healthy states. A summary of the possible configurations is presented in Table II. The left column shows the Operation Mode; the top row states the process criticality; and the numbers populating the middle of the table represent the FUs states (healthy (0), medium healthy (1), intermittent (2) and faulty (3)) required for operating in each of the parameters combination. First, for Ordinary processes, the operation mode can be either 0 or 1 in case the FUs are still in the healthy (0) state. If no more FUs are in this state, but at least one is in the medium healthy (1), the process can run in the operation mode 2. However, once there is no more healthy (0) or medium healthy (1) FUs, the process must be executed in operation mode 3. Secondly, medium criticality processes start running in operation mode

3 and keep running in this mode until at least two FUs are still in the healthy (0) state. If this condition does not satisfy anymore, then the process must run in operation mode 4. After switching to operation mode 4, if one of the FUs becomes faulty (3), the process must run now in operation mode 5. Finally, High Critical processes must run in operation mode 4. The process will only switch to operation mode 5 if one of the FUs becomes faulty (3).

TABLE II
FUNCTIONAL UNITS STATES FOR OPERATION MODES AND PROCESS CRITICALITIES

	Process Criticality			
Operation Mode	**Ordinary**	**Medium**	**High**	
0 (one FU required)	(0)	-	-	**FUs State**
1 (one FU required)	(0)	-	-	
2 (one FU required)	(1)	-	-	
3 (two FUs required)	(2,2)	(0,0)	-	
4 (three FUs required)	-	(1,1,0),...,(2,2,2)	(0,0,0),...,(2,2,2)	
5* (two FUs required)	-	(1,1,3),(2,2,3)	(0,0,3),(1,1,3),(2,2,3)	

* Degraded service

VI. OVERALL DESIGN

The general design is composed of the HW and the SW layer, which consists of an OS running over a processor design (Fig. 1). The processor design was, however, modified to enable configuration of its Functional Units (FUs). This configuration can be controlled by the new commands added as extensions to the original processor Instruction Set Architecture (ISA). On the top level, there is an OS running its services and multiple application processes, each of them with its criticality level. The following sections will go through the HW and SW layers and explain each block in detail.

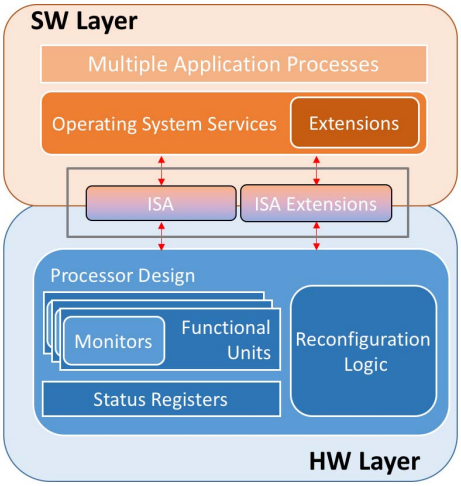

Fig. 1. General overview of the design and its internal elements.

A. Hardware Layer

The HW layer in Fig. 1 consists of the processor design and its extensions. The proposed approach is illustrated directly on a selected case-study: the Plasma processor, which is a synthesizable 32-bit RISC microprocessor that runs MIPS I(TM) user mode instructions except for unaligned load and store operations [15]. The extensions mentioned earlier are presented below:

1) Functional Units and its Fault Monitors: The original design was extended by adding enough Functional Units to enable, when necessary, Triple Modular Redundancy (TMR) over them. Also, to enable individual awareness of the health state, simple fault detection and correction schemes were added to each functional unit.

2) Status Registers: These registers reflect the overall status of the system. For that, first, a status register was created ($stats\_reg$), this register represents the running operation mode and shows the current status of the HW mechanisms and the FUs ('1' - on, '0' - off). As can be noticed from Fig. 2, bits 0, 1 and 2 save, respectively, the status of the FU 1, 2 and 3; bits 3 and 4 represent, respectively, the fault detection (FD) and correction (FC) mechanism; and finally, bit 5 represents the error analysis (EA). Thereby, each operation mode is represented by a set of values in this 16-bit status register which is translated in Table III. Finally, one additional register per FU was also created ($units\_regs$), which is used to keep track of fault detection and correction events.

3) Reconfiguration Logic: This is the control logic to perform FUs configuration. It consists of the prior reconfiguration mechanism proposed in [8]. Its logic is responsible for catching reconfiguration commands and enable or disable FUs according to the captured command. The mechanism extends the original processor design by adding new elements, such as the Pre-Decoder which is responsible for decoding the

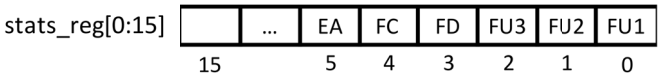

Fig. 2. Status register ($stats\_reg$) bits description: bits 0, 1 and 2 indicate the status of each of the functional units: FU1, FU2 and FU3, respectively; bit 3 and 4 to fault detection (FD) and correction (FC) mechanism and bit 5 to error analysis (EA) configuration. The remaining bits were left for future expansions.

TABLE III
OPERATION MODES AND THEIR EQUIVALENT STATUS REGISTER ($stats\_reg$) VALUES.

	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	
Operation Mode	**EA**	**FC**	**FD**	**FU 3**	**FU 2**	**FU 1**	*stats_reg*
0	0	0	0	0	0	1	**0x0001**
0	0	0	0	0	1	0	**0x0002**
0	0	0	0	1	0	0	**0x0004**
1	0	0	1	0	0	1	**0x0009**
1	0	0	1	0	1	0	**0x000A**
1	0	0	1	1	0	0	**0x000C**
2	0	1	1	0	0	1	**0x0019**
2	0	1	1	0	1	0	**0x001A**
2	0	1	1	1	0	0	**0x001C**
3	0	1	1	0	1	1	**0x001B**
3	0	1	1	1	0	1	**0x001D**
3	0	1	1	1	1	0	**0x001E**
4	0	0	1	1	1	1	**0x000F**
5	1	0	1	0	1	1	**0x002B**
5	1	0	1	1	0	1	**0x002D**
5	1	0	1	1	1	0	**0x002E**

Fault detection (FD); Fault correction (FC); Error analysis (EA);

added commands (ISA-extensions) and perform the adequate action. Due to the considerably low latency required for FU reconfiguration, this mechanism becomes suitable for run-time usage.

B. Software Layer

Any Operating System configured to run over the architecture of the described design can be enabled to operate the Reconfiguration Mechanism explained in the section VI-A, only very few extensions are required for that. Therefore, the Plasma-RTOS was configured and, with less than a hundred lines of code in "C" programming language, extended to operate the mechanism. This operating system was prior created by *Steve Rhoads* to run over the Plasma CPU. Its implementation supports interrupts, threads, semaphores, mutexes, message queues, timers, heaps, and pre-emptive context switching [15] [16].

The result of this implementation is presented in the SW layer of the Fig. 1, and the main components of the extended operating system that participate in the reconfiguration process are the following:

1) Process Data Structures: These are operating system structures which were extended to save the process criticality and its required operation mode.

2) Operating System Processes Interface: To be able to save criticality level and operation mode of the processes, the operating system interface to create new processes was extended. So, a criticality and an initial operation mode can be individually attributed at the process creation time. Moreover, an interface to read and update the processes operation mode at run-time was also created.

3) Process switching mechanism: within the operating system services, the mechanism to switch processes was extended to check the operation mode attributed to the process which is going to be resumed, and send the command to the Reconfiguration Mechanism to configure the HW appropriately.

4) Monitor Process: The purpose of this process is to keep track of any change which may happens in the health state of the FUs due to, for instance, wear out caused by normal aging or by harsh environment conditions (e.g. high temperature and humidity). As a consequence, the time interval between two executions of this process can be relaxed. Therefore, this process will quickly run time to time, by definition set to one process time slice ($\sim 25ms$), and check the FUs health state by reading the FUs Status Registers ($units\_regs$) available under the HW extensions. If any change is detected, a new high priority process is triggered to update the attributed Operation Modes of each application process.

5) Update OpMode Process: This process is triggered by the Monitor Process to update the operation mode of the other processes upon a detection of any change in the FUs health state.

6) Application Processes: These are application processes running on top of the operating system. As already mentioned, each one of these processes has an individual attributed operation mode due to its criticality level.

7) OS context and ISR Operation mode: Every routine executed within the OS context is considered as high critical. Therefore, the very beginning of the Interrupt Service Routine (ISR) was modified to reconfigure the system to the OpMode 4 (the one with TMR and FD scheme), and, after the execution, reconfigure the design again to the upcoming process.

It is important to notice that the Monitor and the Update OpMode process are also considered critical, therefore, executed in OpMode 4.

VII. Evaluation

As already explained in the previous sections, the HW layer of the proposed system was implemented over the Plasma processor design. And the extended Operating System running over the platform is the Plasma-RTOS, which was previously created to run over the Plasma-CPU [15]. The Plasma-RTOS kernel needed to be extended by less than a hundred lines of code to support the Reconfiguration Mechanism. Thus, extending other OSs would also be possible with very little effort, featuring a good portability property.

The original processor design has one ALU, one multiplier and one shifter as its functional units. These units together sum up about to 60% of the total area usage of the whole design. So, it means that, at least, 60% of the original area of the processor design can be programmed to be under the reconfigurable scheme proposed in this paper. Thus, this is the area percentage that can be configured under TMR scheme, hence, improving fault tolerance of the whole design as such.

Concerning the latency necessary for the reconfiguration. The platform took advantage of the low latency inherited by the previous work [8]. To perform the reconfiguration of the FUs the platform needs only one clock cycle, which was decisive in this platform since reconfiguration is performed in run-time. Effectively, to perform reconfiguration at every process switching time, and including all the necessary SW to take the decisions concerning right operation modes, the platform deterministically took 25 clock cycles, which represents about to 5% of the whole process switch operations.

The full system was simulated using Xilinx Vivado simulator. And part of this simulation is in the wave chart presented in Fig. 3. This simulation consists of three application processes and the Monitor process. The signals represent, from top to bottom, the clock signal (clk), the status register ($stats\_reg[15:0]$), and the remaining signals mean when each of the equivalent processes is running. As can be noticed, the application processes are running in sequence and under its own configured operation mode: the ordinary process ($Process1\_ordinary$) is running in the OpMode 1; the medium criticality process ($Process2\_medium$) in the OpMode 3; and the high critical one ($Process3\_High$) in the OpMode 4 (the one with TMR and FD scheme). The process to update the operation modes ($UpdateOpMode$) does not run in this simulation due to no error events. Finally, the Monitor process ($Monitor\_process$) runs shortly between every process switching as it is highlighted by the red arrows in the Fig. 3, the detail of this situation is illustrated by

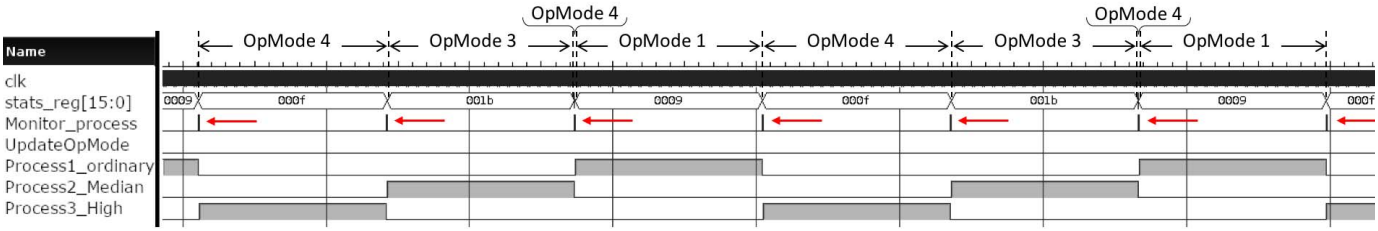

Fig. 3. Full system execution, it is running 3 different application processes and 1 monitor process.

Fig. 4. Process switching time: switching between two application process and the Monitor process.

Fig. 4. At the process switching time, the Interrupt Service Routine (ISR) first switches to OpMode 4, executes its internal operations and resumes the Monitor process, which is also executed in OpMode 4. After checking the state of the FUs, the process sleeps, and the system switches again to the OS context. In this context, it checks and configures the design to the required operation mode of the process that is going to be resumed. After finished the HW configuration, the next process is resumed (in the case observed in Fig. 4: the Ordinary process is then configured to run in the OpMode 1).

Reconfiguration is done at every process switch operation, and the proper operation mode for each process is maintained up to date by the Monitor process, which wakes up, by definition, every $\sim 25ms$ (equivalent to one process time slice). This time is very programmable, and can be even more relaxed to match any scenario conditions, since this Monitor process aims to track health state changes of the FUs mainly caused by wear-out induced by normal aging or severe environment conditions (e.g., high temperature and humidity). However, once we have implemented interrupt-driven error signaling (as foreseen future work), reconfiguration and operation modes update can be performed at ISR level. Thus, it will enable the system to react on faults within few (3 to 4) clock cycles to rise Interrupt Request signal (IRQ) and approximately a hundred clock cycles depending on the processor to manage a fault at OS level.

A further evaluation was performed, and Fig. 5 shows the platform simulation for a series of simulated errors by increments in the units status registers ($units\_regs[< FU\_index >]$) which account for fault detection events. As already explained, in this work we are considering interment soft faults which increases in number as the electronics start to age [13]. Moreover, the error simulation is done in such a way that the FUs change their attributed health states, so the operation mode attributed to each process also changes following the modes stated in Table II. For this simulation, the thresholds were configured as follows: the $MEDIUM\_HEALTHY\_THRESHOLD$ to 10 (0x0A) faults, the $INTERMITTENT\_THRESHOLD$ to 100 (0x64) and the $FAULTY\_THRESHOLD$ to 1000 (0x3E8). The signals in the figure represent, from top to bottom, the clock signal (clk), the status registers ($stats\_reg$), the processes Monitor, Update Operation Mode, application processes Ordinary, Medium and High, and, finally, the units registers: $units\_regs[0]$ representing the events in the FU 1, the $units\_regs[1]$ in the FU 2 and the the $units\_regs[2]$ in the FU 3. As it is highlighted in the figure by the arrows and the dotted squares, once the events detected in a specific FU reach one of the health state thresholds, the $UpdateOpmode$ process is triggered to attribute a new operation mode to each of the application processes taking into account the new FU state. As a result, the further execution of the application processes is executed using this new operation mode configuration. The corresponding operation mode for each $stats\_reg$ value shown in the Fig. 5 is translated in Table III.

VIII. Conclusion

This article presented a fully integrated platform capable of performing OS controlled reconfiguration of processor internal Functional Units (FUs) within a mixed-criticality processes scenario. As stated in the Evaluation section (VII), at a cost of *a few clock cycles*, the system was able to configure its FUs according to the criticality requirements of processes running over the Plasma-RTOS. Moreover, the SW routines were capable of, in run-time, evaluate FUs healthiness and, upon an FU health state modification, update the operation mode and the FUs attributed to each process. Finally, a quick analysis of the original processor design shows that a great part of the design is covered by the FUs. Therefore, any measure to tolerate faults applied to these units, like the TMR scheme used in this paper, does improve the fault tolerance of the entire design as such.

References

[1] B. Janßen, F. Kästner, T. Wingender, and M. Huebner, "A dynamic partial reconfigurable overlay framework for python," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10824 LNCS, pp. 331–342, Springer Verlag, 2018.

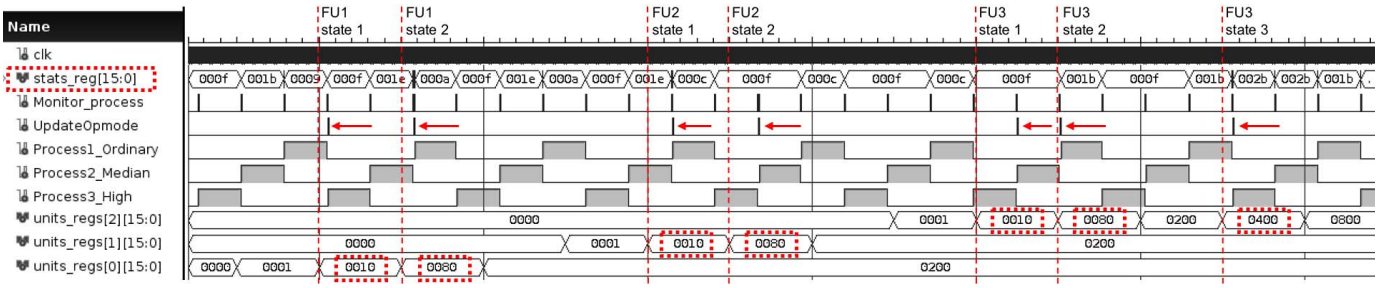

Fig. 5. Faults simulation performed over the platform, the units status register ($units\_reg$) is incremented time to time simulating fault detection in the FUs. Numbers in the wave chart are is hexadecimal notation.

[2] P. M. B. Rao, A. Amouri, S. Kiamehr, and M. B. Tahoori, "Altering LUT configuration for wear-out mitigation of FPGA-mapped designs," in *2013 23rd International Conference on Field Programmable Logic and Applications, FPL 2013 - Proceedings*, 2013.

[3] K. Shibin, S. Devadze, A. Jutman, M. Grabmann, and R. Pricken, "Health Management for Self-Aware SoCs Based on IEEE 1687 Infrastructure," *IEEE Design & Test*, vol. 34, pp. 27–35, 12 2017.

[4] A. Baldassari, C. Bolchini, and A. Miele, "A dynamic reliability management framework for heterogeneous multicore systems," in *2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, IEEE, 10 2017.

[5] A. Kulkarni, D. Stroobandt, A. Werner, F. Fricke, and M. Hübner, "Pixie: A heterogeneous Virtual Coarse-Grained Reconfigurable Arrayfor high performance image processing applications," *CoRR*, vol. abs/1705.01738, 2017.

[6] X. Iturbe, B. Venu, J. Penton, and E. Ozer, "Work-in-progress: A "high resilience" mode to minimize soft error vulnerabilities in ARM Cortex-R CPU pipelines," in *Proceedings of the 2017 International Conference on Compilers, Architectures and Synthesis for Embedded Systems Companion, CASES 2017*, Association for Computing Machinery, Inc, 10 2017.

[7] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael, "DaemonGuard: Enabling O/S-Orchestrated Fine-Grained Software-Based Selective-Testing in Multi-/Many-Core Microprocessors," *IEEE Transactions on Computers*, vol. 65, pp. 1453–1466, 5 2016.

[8] R. Segabinazzi Ferreira and J. Nolte, "Low latency reconfiguration mechanism for fine-grained processor internal functional units," in *2019 IEEE Latin American Test Symposium (LATS)*, (Santiago/CL), 2019.

[9] F. Muhlbauer, L. Schroder, and M. Scholzel, "A fault tolerant dynamically scheduled processor with partial permanent fault handling," in *2018 IEEE 19th Latin-American Test Symposium, LATS 2018*, vol. 2018-January, pp. 1–6, 4 2018.

[10] A. Sadighi, B. Donyanavard, T. Kadeed, K. Moazzemi, T. Muck, A. Nassar, A. M. Rahmani, T. Wild, N. Dutt, R. Ernst, A. Herkersdorf, and F. Kurdahi, "Design methodologies for enabling self-awareness in autonomous systems," in *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018*, 2018.

[11] T. D. Nya, S. C. Stilkerich, and C. Siemers, "Self-aware and self-expressive driven fault tolerance for embedded systems," in *2014 IEEE Symposium on Intelligent Embedded Systems (IES)*, pp. 27–33, 1 2014.

[12] R. Segabinazzi Ferreira, N. George, J. Chen, M. Hübner, M. Krstic, J. Nolte, and H. T. Vierhaus, "Configurable Fault Tolerant Circuits and System Level Integration for Self-Awareness," in *2019 22nd Euromicro Conference on Digital System Design (DSD) (Work in Progress Session)*, 2019.

[13] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn, "Reliable On-chip systems in the nano-era: Lessons learnt and future trends," in *Proceedings - Design Automation Conference*, 2013.

[14] H. Hong, J. Lim, H. Lim, and S. Kang, "Lifetime reliability enhancement of microprocessors: Mitigating the impact of negative bias temperature instability," *ACM Computing Surveys*, vol. 48, 9 2015.

[15] OpenCores.org, "Plasma - most MIPS I(TM) Overview," in *https://opencores.org/projects/plasma, visited Dec. 6th*, 2019.

[16] S. Rhoads, "Plasma Real-Time Operating System," in *http://plasmacpu.no-ip.org:8080/rtos.htm, visited Dec. 6th*, 2019.