On Design of Priority-Driven Load-Adaptive Monitoring-Based Hardware for Managing Interrupts in Embedded Event-Triggered Real-Time Systems

Josef Strnadel
Brno University of Technology, Faculty of Information Technology,
IT4Innovations Centre of Excellence, Brno, Czech Republic, strnadel@fit.vutbr.cz

***Abstract*—The paper details design of a hardware unit for preventing real-time systems from overloads caused by excessive interrupt rates. Novelty of the hardware can be seen in the fact it is able to adapt interrupt service rate to the RT system load and to the actual priority assignment policy. The load is monitored on basis of special low-overhead signals produced by the system for this purpose. The hardware is designed to pre-process all interrupts before they arrive to the system. Then, the hardware is able to buffer each interrupt-related communication until the system is underloaded or running an activity having a lower priority comparing to the interrupt. Design of the hardware was described in VHDL and synthesized into Xilinx Spartan-6 devices. Details such as buiding blocks, overheads and limits related to the realization are presented in this paper.**

I. Introduction

For many systems it is typical that occurence of an event is signalized by the interrupt (INT) mechanism. Advantage of the mechanism can be seen in the fact an INT is serviced prior and asynchronously to the main-loop control flow in order to minimize the INT response time. Disadvantage can be seen in the computational overhead (i.e., memory and time for re/storing the CPU context) related to an INT occurence. As a consequence of the overhead, the main-loop control flow is delayed until all arisen INTs are serviced. The problem is that the CPU time available for the main-loop execution is proportional to the INT interarrival time.

In the extreme case it can happen that the INT interarrival time drops below the INT service time; then, no CPU time remains for the main-loop execution. It can happen despite the INT criticality is lower than the criticality of the actual main-loop control flow. This is being denoted as the *interrupt overload* (IOV) problem and it must be solved when the main-loop part is executed at a high criticality level. As a consequence, the part may stop working correctly or collapse suddenly [3]. Especially, it holds for (so-called Real-Time, RT) systems, perfection of which is based on both the correctness and timeliness of the outputs [1]. In these systems, each an event is typically associated with a computational unit (called a *task*) responsible to react correctly to the associated event. There are two basic types of RT tasks [4]:

- *hard*, timing constraints of which must be strictly met,
- *soft*, for which it suffice the constraints are "sometimes" met.

To guarantee the hard-task constraints will be always met, hard (soft) tasks are being executed at high (low) priority levels. To organize task executions in time (i.e., to *schedule* them to meet their timing and other constraints) and to simplify design and analysis of an RT system, it is common practice that tasks are managed by an RT operating system (RTOS, RT kernel) designed to guarantee timeliness of their responses [1], [2], [4]. As the RTOS code is being executed as a part the main-loop control flow, its operation can be threatened by the excessive rate of INT stimuli too.

Several solutions to the IOV problem has been already published [5]–[12], [16]. Some of them (such as polling, strict or bursty INT limiters [11], middleware [8], [12] or task-level scheduling mechanisms [5], [9], [16]) are either software solutions to the IOV problem or trivial, non-adaptive or high-cost hardware solutions such as [6], [7], [10] suitable for PC-based realizations. While the SW solutions to the IOV inherently worsen the RT-task schedulability as they increase the CPU utilization factor, the HW solutions require significant modifications and/or extensions of common *commercial off-the-shelf* (*COTS*) components. Moreover, none of them is able to minimize undesired INT effects such as

- *timing disturbance problem* composed e.g. of *disturbance due to soft real-time* (RT) *tasks* and *priority inversion* sub-problems [6], [12],
- *predictability problem* originating from a system inability to predict arrival times and the rate of INTs

induced by external events [6], [10], [11] along with adaptation of the INT throughput to load and priority of a platform supposed to be stimulated by interrupts.

This motivated us to design the architecture not suffering these shortcomings.

The paper is organized as follows. In the section II, motivation and actual state of our research are presented. In III, interface and protocol utilized for load-monitoring purposes (III-A), load-monitor/INT-limiter architecture (III-B) and properties and realization overheads w.r.t. to the architecture (III-D) are detailed. The section IV concludes the paper.

II. Motivation and Actual State of our Research

Motivation and goals of our research were outlined in [14] and they can be summarized as follows.

- *Reachability:* to offer a solution to the IOV problem on basis of instruments accessible at the market, i.e.,

using COTS components such as MCUs/FPGAs and operating systems (OSes),

- *Generality:* the solution must result to an architecture that is general enough to abstract from products of particular producers and is able to solve both the timing disturbance and predictability problems,
- *Simplicity:* the solution must reduce a need to modify existing components to a minimum,
- *Adaptability:* the solution must be able to adapt the INT service rate to the actual platform load and constraints implying from the system specification.

In our previous papers, the operating principle and basic properties of a novel hardware/software (HW/SW) architecture designed to adapt the INT service rate to the actual SW load being monitored by the HW were presented [13]. Afterwards, details related to the research background and realization overheads related to the architecture were presented in [14].

Main goal of this paper is to present details related to implementation of the proposed architecture – a special attention is paid there to present details related to generation of monitoring signals by the SW and to inner structure and limits of the proposed INT-management hardware.

As the SW (running on a microcontroller unit, MCU) was supposed to be safety&time-critical, it was undesirable to forward an INT to the MCU while it is overloaded or executing an action of a priority higher or equal to the INT priority. INTs were managed by the HW realized by an FPGA designed to recognize and stall/release "undesirable" interrupts. Let it be denoted there that (among its other inner components) an MCU is composed of a CPU responsible for software execution. So, it can happen that the "CPU" will be utilized instead of "MCU" where applicable in this paper.

III. Proposed Monitor Details

In order to monitor (i.e., to observe in a non-intrusive way) the SW load, it was decided to utilize a simple interface able to be realized by any COST component – the interface is composed of the `MON_INT` `MON_TICK`, `MON_PRI`, `MON_CTX` and `MON_SLACK` signal lines (see Fig. 1) being produced (processed) by the MCU (FPGA) for the monitoring purposes. Details related to signal generation follows.

A. Signal Generation

Particular implementation steps needed to realize the signal generation part of the proposed architecture are going to be illustrated using the μC/OS-II kernel instruments. However, it

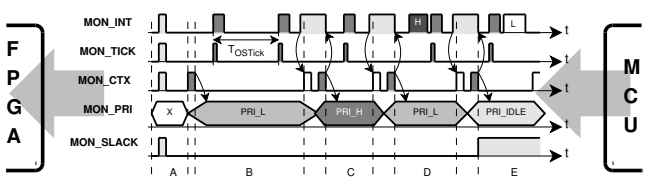

Fig. 1. An illustration to the monitoring signals/interface introduced in [13]. In the worst case, $4 + n$ pins are needed to realize the interface, where n is the priority adress bit-width

can be emphasized there that the concept is general enough to abstract from an RTOS case-study. The signal generation can start after the operating system kernel is ready to operate, i.e., if an RTOS structures are initialized and a hardware timer (*TIM*) is configured and enabled to produce a periodic T_{OSTick} interrupt. In the μC/OS-II case, this can be done after the `OSInit()` function is over – i.e., either after `OSInit()` call in the `main()` or in the in the `OSInitHookEnd()` function body being called by the kernel at the end of the `OSInit()` body (for details, see the listing 1 and the listing 2). The generation start is signaled to the FPGA by producing a short HIGH-level impulse at each of the `MON_INT`, `MON_TICK`, `MON_CTX` and `MON_SLACK` lines (Fig. 1).

Listing 1. "uC/OS-II example #1 to the signal generation start"

```
#include "includes.h"
#include <stdio.h>

int main (void) {
  OSInit();   /* creation and initialization of OS data
       structures, system timer, ... */

  /* signal generation commands can be placed here */

  OSStart();  /* starting the multitasking; return is
       NOT expected from this function */
  return(0);
}
```

This gives the FPGA possibility to synchronize with the MCU and to start the signal monitoring process, but at the price of increasing the system startup time by a predefined and constant number of CPU cycles (number of which depends on pins and instructions selected to control the lines).

Listing 2. "uC/OS-II example #2 to the signal generation start"

```
void OSInitHookEnd (void) { /* this function is called
     at the end of the OSInit() function */
  /* signal generation commands can be placed here */
}
```

Next, an INT prologue (epilogue) must be modified to set the `MON_INT` signal to HIGH (LOW) just at the beginning (end) of an ISR body. In relation to the μC/OS-II kernel, this can be done before calling the `OSIntEnter()` function (after returning from the `OSIntExit()` function). This extends the ISR execution in a deterministic and the same way across all ISRs and allows the ISR execution time as well as ISR enter/exit times be monitored both in a non-intrusive way and at runtime. As the `MON_INT` signal is set to HIGH (LOW) after (before) the CPU enters (exists) the INT execution level, it is evident that this generation principle leads to certain `MON_INT` monitoring error, implying from the fact that the monitored `MON_INT` length will be smaller than the real one. Let the error be denoted by e_{INT} and let it hold for it

$$0 \leq e_{INT} - (\hookrightarrow^{ST}_{INT} + \hookrightarrow^{LD}_{INT} + \Updownarrow_E) \leq \Uparrow^{H}_{INT} + \Downarrow^{L}_{INT} \quad (1)$$

where $\hookrightarrow^{ST}_{INT}$ ($\hookrightarrow^{LD}_{INT}$) is the ISR context store (load) delay implying from processing an INT request by the MCU's INT subsystem, $\Uparrow^{H}_{INT}$ ($\Downarrow^{L}_{INT}$) is a delay implying from adjusting the `MON_INT` line to HIGH (LOW) level and $\Updownarrow_E$ is a further overhead being 0 herein, but non-zero in the following – (2), (3) – cases.

Moreover, execution of the (special) *TIM*-related ISR is signalled by generating a short pulse at the `MON_TICK` line; this

allows the FPGA to monitor the OS time in order to observe phenomena such as a jitter. For the purpose, the start (end) of the `OSTickISR()` routine body must be modified to set the `MON_TICK` signal to HIGH (LOW). Because of this extra overhead, for the *TIM*-related ISR it holds

$$\Updownarrow_E = \Uparrow^H_{TICK} + \Downarrow^L_{TICK} \tag{2}$$

where $\Uparrow^H_{TICK}$ ($\Downarrow^L_{TICK}$) is a delay of adjusting the `MON_TICK` line to HIGH (LOW) level.

As embedded resources are very limited (comparing e.g. to a deskdop PC), an ISR nesting is disallowed and the ISR execution must be as short as possible not to delay much the execution of any consecutive ISR.

In order to monitor amount of time the CPU spends by storing and restoring task-level contexts, the beginning (end) of the `OSCtxSw()` function is modified to set the `MON_CTX` signal to HIGH (LOW). It is supposed the context switch is performed within a special *CTX*-related ISR for which it holds

$$\Updownarrow_E = \Uparrow^H_{CTX} + \Downarrow^L_{CTX} + \Uparrow_{PRI} \tag{3}$$

where $\Uparrow^H_{CTX}$ ($\Downarrow^L_{CTX}$) is a delay of adjusting the `MON_CTX` line to HIGH (LOW) level and $\Uparrow_{PRI}$ is further delay is explained below w.r.t. the next (`MON_PRI`) signal being utilized to monitor the priority of a running task. As the `MON_PRI` signal is set in the context-restore phase of the CTXSW as soon as the task priority is taken from the context, $\Uparrow_{PRI}$ is a delay of adjusting the priority to the `MON_PRI` line.

If there is no ready task in the system then the CTX is switched to the *idle task*. Before (in the CTXSW phase), the `MON_PRI` line is set to PRI_IDLE. Also, just after the `MON_CTX` signal is set to HIGH in the `OSCtxSw()` body – this is already included in (3) – then the `MON_SLACK` line is set to HIGH too there if it holds the priority of a task the CTX is going to be switched to is below the predefined (hard-priority) level. Delay related to the conditioned code must be included into the $\Uparrow_{PRI}$ part of (3).

Operational principle of the proposed monitoring-based INT-management platform can be described using the following algorithm:

Algorithm Description of the proposed monitor function

```
1:  if (an INT occurs or a MON_* signal changes) then
2:      if (the INT priority > MON_PRI) then
3:          forward the INT to the MCU
4:          return
5:      else if (the maximum number of INTs allowed in the critical floating
        window is not exceeded) then
6:          forward the INT to the MCU
7:          return
8:      else if (MON_SLACK signal is HIGH) then
9:          forward the INT to the MCU
10:         return
11:     else
12:         stall the INT until the system becomes ready to service it
13:         return
14:     end if
15: end if
```

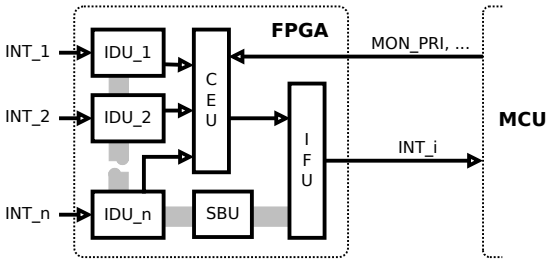

Fig. 2. Block schema of the proposed monitor-based interrupt limiter

B. Architecture

The following key units can be distinguished in the proposed load-monitoring architecture designed to manage interrupts (for its block schema, see the Fig. 2):

i) **INT Detection Unit (IDU)** – each an INT is associated a separate IDU goal of which is to detect the INT. Key components of an IDU are: INT sensitivity (level/edge) unit and INT priority storage. If necessary, both the components can be updated at the run-time in order to support the reconfiguration and dynamic priority assignment policies.

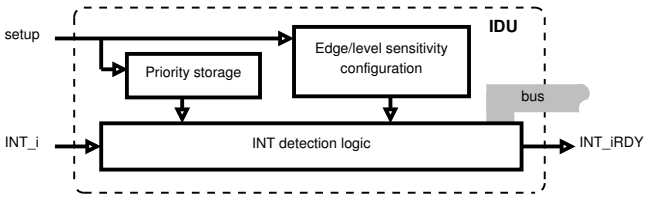

Fig. 3. Block schema of the INT Detection Unit

ii) **Condition Evaluation Unit (CEU)** – it is utilized to evaluate special (*Priority*, *Underload*, *Slack*) conditions. Each of the conditions is designed to check a partial relation between properties (such as priority) of catched – but unserviced yet – interrupts and actual properties of the SW (such as the running task priority, the CPU load/slack progress – being observed on basis of the monitoring signals). At least, let it be noted there that the purpose of the conditions are:

- Priority condition – to check whether the priority of an INT is higher than the priority of a task being executed by the MCU; let it be noted there that the joint INT/task priority space is utilized to abstract from hardware-level INT priorities in order to allow mutual comparison of INT and task priorities,
- Underload condition – to check whether releasing a task for servicing and INT would lead to the MCU overload (and consequently, to violation of some, e.g. RT, properties) and
- Slack condition – to check whether the SW is in the idle state (thus, able to service an INT aside from its priority).

If none (any) of the conditions is met for the highest-priority unserviced INT, the INT is processed by the SBU (IFU) unit – see III-B–iv. Main components of the only CEU are: Maximum INT-priority select unit, Priority

comparator unit and Slack time evaluator unit. As a detail description of the conditions is beyond the scope of this paper, readers are kindly invited to find it in [13].

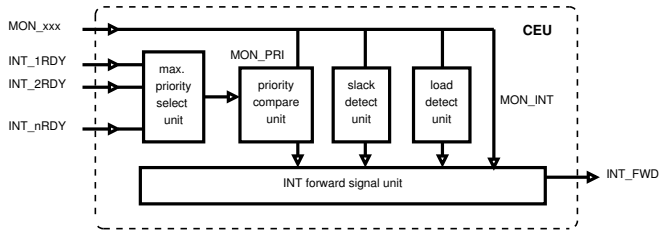

Fig. 4. Block schema of the Condition Evaluation Unit

iii) **Stall-INT Buffer Unit (SBU)** – goal of the unit is to store an INT (request and associated data, if any, such as multiple INT requests and their timestamps) that cannot be forwarded to the MCU because of unmeeting the contidions. The INT is stalled in the buffer until a condition becomes met. SBU is a memory accessible to all IDUs and the IFU (see III-B–iv).

iv) **INT Forward Unit (IFU)** – alike CEU and SBU, there is only one IFU unit invlolved in the architecture. IFU is activated after a condition becomes met. In this case, the IFU is informed about the highest-priority INT that is ready to be forwarded to the MCU and about the INT sensitivity adjusted at the MCU side. Then, the IFU reads the INT-related data from SBU (if there are any) and forwards the INT – and its prospective data – to the MCU.

Besides the published resource consumation overheads [13], [14] there is one that is very important from the RT point of view: it is the worst-case delay (Δ_{INT}) of transporting the highest-priority INT through the proposed architecture and forwarding it to the MCU. The delay is measured from a time the INT priority became the highest in the joint INT/task priority space and a time the INT stimulation started at the MCU pins. In order to quantify the delay, let us suppose T_{clk} is the system clock period. Then, the delay can be expressed by the formula:

$$\Delta_{INT} = \Delta_{IDU} + \Delta_{CEU} + \Delta_{IFU} + \Delta_{SBU} \tag{4}$$

where the partial delays are:

- $\Delta_{IDU} = 2 \times T_{clk}$: this is the two clock-period delay of detecting a (edge/level) sensitivity condition related to an INT by its associated IDU. As all IDUs operate in parallel, the delay is fixed and independent from the number of priority levels and number of INTs supported by the architecture.
- $\Delta_{CEU} = 1 \times T_{clk}$: it is a combinational circuit delay limiting the maximum operating frequency ($f_{max} \leq \Delta_{CEU}^{-1}$) of the proposed architecture. The delay grows significantly with the number of priority levels and number of INTs supported by the architecture. Let it be noted there that a significant portion of Δ_{CEU} is introduced by the maximum-selection component of CEU. For the impact, see Tab. I.
- $\Delta_{IFU} = 2 \times T_{clk}$: this is the two clock-period delay of forwarding the highest-priority INT stimulus to the MCU pins, generation of which is based on the (edge/level) sensitivity configured for each INT source. As in Δ_{IDU} case, Δ_{IFU} is fixed and independent from the number of priority levels and number of INTs supported by the architecture too.
- $\Delta_{SBU} = n \times T_{clk}$: this is the n clock-period delay of reading the INT-related data being previously stalled in the SBU. The delay cannot be fixed if the information such as memory type/parameters and data throughput/size are unknown for the INT. For an INT that is associated with no data it holds $n = 0$, i.e., it is not necessary to (re)store anything to (from) the SBU.

The INT throughput is a function of the maximum operating frequency f_{max} (Tab. I), Δ_{INT} value and the actual MCU load, so it cannot be expressed and evaluated independently from the information. However, the theoretical upper bound for the throughput can be estimated using the formula:

$$\begin{aligned} \frac{1}{\Delta_{INT}} &= \frac{1}{\Delta_{IDU} + \Delta_{CEU} + \Delta_{IFU} + \Delta_{SBU}} = \\ &= \frac{1}{(2+1+2+n) \times T_{clk}} = \frac{f_{max}}{(5+n)} \end{aligned} \tag{5}$$

If the load is not reflected and Δ_{SBU} is 0, it can be concluded the maximum theoretical throughput of the proposed architecture lies cca in the $8 \times 10^4 INTs/s$ (the limit for 256 INT sources/priority levels configuration) to $132 \times 10^6 INTs/s$ (the limit for 2 INT sources/priority levels configuration) range. As the load is a dynamic, priority-dependent quantity (i.e., it changes in time [13]), it cannot be simply involved in the formula. But typically, the INT throughput decreases with increasing load in general. However, this does not hold if the INT priority is higher than the priority of the running task.

C. Functional Properties

In [13], [14], it is published that for high INT-rate values our IOV solution is able to prevent the RT system from INT overload and to service significantly higher number of INTs during MCU underload than the other approaches at comparable CPU-load values – this is because our concept is able to utilize main-loop idle intervals to service excessive interrupts.

Moreover, other important properties of the proposed architecture – list of which follows – are detailed and proven in [13]:

- No interrupts are directed to the MCU while an ISR is being executed.
- Disturbing tasks due to low priority interrupts is avoided.
- Delay in servicing the highest priority event is bounded.
- INT blocking can be bounded.
- Stall INT buffer size can be bounded.
- The system can't overload due to IOV problem.
- Timing constraints of hard tasks are met.

TABLE I. PARAMETERS OF THE PROPOSED ARCHITECTURE – XILINX SPARTAN6 RESULTS

Number of INTs to service (n) Number of priority levels (m)	4 4	4 64	4 256	64 4	64 64	64 256	256 4	256 64	256 256	Spartan6 device
Max. operating frequency [MHz] (f_{max})	219	108	90	62	8	5	44	2	0.4	all (max 663 for $n = m = 2$)
$\frac{\Delta_{SBU}}{T_{clk}} = 0$	43.8	21.6	18	12.4	1.4	1	8.8	0.4	0.08	
INT throughput [10^6 INTs/s], $\ldots = 10^2$	2.08	1.03	0.86	0.59	0.08	0.05	0.4	0.02	0.004	all (max 132 for $n = m = 2$)
$\ldots = 10^4$	0.002	10^{-3}	10^{-3}	10^{-3}	10^{-4}	10^{-4}	10^{-3}	10^{-4}	10^{-6}	$\approx$
Max. on-Spartan6 buffer size [Mb]	6.1	6.1	6.1	5.9	5.9	5.8	5.6	5.3	4.8	xc6slx150
	5.8	5.8	5.8	5.5	5.4	5.3	5.0	4.6	4.0	xc6slx100
	3.8	3.8	3.8	3.5	3.4	3.3	3.0	2.7	2.2	xc6slx75
	2.5	2.5	2.5	2.2	2.1	2.0	1.7	1.3	0.7	xc6slx45
	1.1	1.1	1.1	0.9	0.8	0.7	0.5	0.1		xc6slx25
	0.7	0.7	0.7	0.5	0.4	0.3	0.1			xc6slx16
	0.6	0.6	0.6	0.3	0.2	0.1				xc6slx9
	0.2	0.2	0.2							xc6slx4
Slice LUT utilization [%]	≈ 0			2	3	4	6	11^+	17^+	xc6slx150 ($^+$ max 250 INTs)
				3	4	7	9^+	14^+	21^+	xc6slx100 ($^+$ max 222 INTs)
				5	6	9	13^+	21^+	31^+	xc6slx75 ($^+$ max 192 INTs)
				9	11	16	23^+	36^+	57^+	xc6slx45 ($^+$ max 160 INTs)
				16	20	30	42^+	69^+		xc6slx25 ($^+$ max 96 INTs)
		1	1	27	33	50	70^+			xc6slx16 ($^+$ max 90 INTs)
		1	2	43	52	63	85^+			xc6slx9 ($^+$ max 84 INTs)
	2	4	5	61^+	68^+	79^+				xc6slx4 ($^+$ max 34 INTs)

D. Realization Overheads and Limits

To demonstrate practical applicability and reveal realization overheads and limits w.r.t. proposed IOV solution, we have decided to realize it using AX32 platform [15] available at our faculty. Main computational components of the platform were: an MCU (ARM Cortex-A9) and an FPGA (Xilinx Spartan6). As the MCU-related (monitoring signal/interface realization) overheads were minimal and discussed in III-A, the summary present in Tab. I is limited to the FPGA part of the proposed architecture. The architecture was described in VHDL and synthesized using Xilinx ISE 13.1.

From the practical point of view, especially the following facts are important:

- how many priority levels can be distributed among INT sources and RT tasks,
- how many INT sources can be processed by the proposed architecture,
- how many INT-related data-bits can stored in a Spartan6-device and
- what is the maximum INT throughput of the architecture.

In case of i), ii) and iii), it is expected that a higher number can be achieved if a more complex Spartan6 device is utilized to realize the proposed architecture. In case of iv), it is expected that the INT throughput is a dynamic quantity that is inversely proportional to the number of INT sources, number of priotity levels and the MCU load. It can happen that in some applications, a cost/quality trade-off must be searched in this contradictory design sub-space.

In the Tab. I, parameters of the proposed architecture are presented as functions of i) the number of INTs to be serviced by the architecture (n) and ii) the number of joint task/INT priority levels (m) utilized by the architecture. In the leftmost column of the table, a particular parameter is identified. In the rightmost column of the table, it is specified which Spartan6 device the parameter values are valid for. In the middle part of the table, impact of n and m (and Spartan6 device) to the parameter value can be observed. Let the data from Tab. I be interpreted now, starting from the top to the bottom of the table.

First, it can be seen that the max. operating frequency (f_{max}) of the proposed architecture ranges from cca 219 MHz to 400 kHz for $n = m = 4$ to $n = m = 256$ range; outside the range, f_{max} can change (e.g., to 663 MHz for $n = m = 2$).

Next, the max. INT throughput of the architecture is presented as a function of n, m and various Δ_{SBU} values (resp. various numbers of T_{clk} cycles needed to access data of stalled INTs). It can be seen the throughput decreases dramatically with the increasing Δ_{SBU} value – from $\approx 132 \times 10^6 INTs/s$ for $n = m = 2$, $\frac{\Delta_{SBU}}{T_{clk}} = 0$ (i.e., if no data are to be stalled along with their INT requests) to $\approx 1INT/s$ for $n = m = 256$, $\frac{\Delta_{SBU}}{T_{clk}} = 10^4$ (i.e., it is supposed that 10^5 T_{clk} cycles are needed to stall an INT and its associated data).

The last two parameters present in the table are i) the maximum size of INT-stall buffers realizable on a Spartan6 device and ii) the Spartan6 slice LUT utilization factor. For the foremost parameter, it can be seen in the table that the maximum buffer size ranges from cca 6.1 Mb (for xc6slx150 device and $n = m = 4$) to cca 100 kb. If the buffer size does not suffice for a particular device, an external memory can be utilized to increase the buffer capacity. However, in that case an extra on-chip FPGA resources are needed to implement a controller of such a memory. For the latter parameter, the the slice LUT utilization factor can be found in the table for particular n, m and Spartan6 device. It can be seen there that the ratio grows faster with increasing n than with increasing m, so it is minimal for "small" values of n and that for "greater" n, m values some Spartan6 devices became out of their resources, so the values were put down until the architecture become realizable on such a device. Those cases are marked by $^+$ symbol in the table – for example, the ratios for $n = 256$, $m = 64$ $n = 256$, $m = 256$) and the xc6slx150 device were 11% (17 %), but as the number of xc6slx150 pins is limited, realizations of those architectures had to be limited to $n = 250$ to soften impacts of this drawback. On the other side, it can be

seen there are many unused slice LUT resources, which can be utilized to realize the architecture even for $n = m = 256$ if there were more pins.

It can be concluded that the only limiting factor for the implementation was the number of bonded IOBs – even the most complex Spartan6 (xc6slx150) was not equipped with enough bonded-IOB resources to service 256 or more INT sources at 256 or more priority levels. In the Tab. I, it can be seen that the maximum number of 250 (34) INTs can be serviced by the architecture if it is realized on the most complex xc6slx150 (the simplest xc6slx4) Spartan6 device and 64 or more joint-priority levels are needed. Alike, max. 64 (32 for the xc6slx4 device) INTs can be serviced if 256 or more joint-priority levels are needed. As common real-time kernels support not more than 256 priority levels, it can be stated that a Spartan6 realization of the proposed architecture is able to support a sufficient number of priority levels. Other Spartan6 resources (statistics of which were collected from Device Utilization Summary report produced by ISE after the synthesis process was over) such as the number of slice registers (FFs), number of fully used LUT-FF pairs or number of BUFG/CTRLs remain almost constant, so the number of slice LUTs is the only parameter summarized in the Tab. I.

IV. Conclusion

In the paper, realization details w.r.t. proposed hardware solution to the INT overload problem were presented. It was supposed there that an event-driven embedded application is composed of an MCU utilized to execute a critical part of the application and of an FPGA utilized to protect the MCU from consequences of the presented IOV problem. In other words, it is supposed the critical part execution may not be threatened by an external event being signalized by an INT. It was shown in the paper the treat can be avoided if the MCU is equipped with an interface allowing it to produce special signals being monitored at the FPGA side in order to evaluate the dynamic load of the MCU at run-time. Key features of the proposed monitoring-based concept can be summarized as follows: the FPGA is able to adapt the INT service rate to the actual SW load (as a consequence, low-priority INTs are not forwarded to the MCU while it is overloaded; moreover, idle SW intervals can be utilized to service significantly higher number of INTs during MCU underload comparing to existing approaches), the CPU running the critical code is not disturbed by low-priority INTs and the minimal forwarding delay is guaranteed for an INT with the highest joint priority in the system.

Although common COST components (the μC/OS-II kernel running on the ARM Cortex-A9 MCU and the Xilinx-Spartan6 FPGA) were utilized to show the applicability, implementation overheads and limits related to the IOV concept proposed in this paper, the proposed concept is more general and is not tied to those components – each of them can be substituted by a different one.

Future research activities w.r.t. the paper are going to be focused on real-world applications and real-traffic measurements based on the proposed load-adaptive architecture, comparison to advanced SW-level approaches such as sporadic servers, overload-scenario scheduling and priority assignment mechanisms and device drivers designed to solve the aperiodic event occurence problem by the means of software instruments.

Acknowledgment

This work has been partially supported by the Research Plan No. MSM 0021630528 (Security-Oriented Research in Information Technology), the RECOMP MSMT project (National Support for Project Reduced Certification Costs Using Trusted Multi-core Platforms), the BUT FIT-S-11-1 and the IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070.

References

[1] Cheng, A. M. K. *Real-Time Systems, Scheduling, Analysis, and Verification*. John Wiley & Sons, Hoboken NJ, United States, 2002.

[2] Cottet, F., Delacroix, J., Kaiser, C., and Mammeri, Z.. *Scheduling in Real-Time Systems*. John Wiley & Sons, Hoboken NJ, United States, 2002.

[3] Kopetz, H.. On The Fault Hypothesis For A Safety-Critical Real-Time System. *Automotive Software-Connected Services, LNCS 4147,* 1, pp. 31–42, 2006.

[4] Laplante, P. A.. *Real-Time Systems Design and Analysis*. Wiley-IEEE Press, Hoboken NJ, United States, 2004.

[5] Lee, M., Lee, J., Shyshkalov, A., Seo, J., Hong, I., and Shin, I. On Interrupt Scheduling Based On Process Priority For Predictable Real-Time Behavior. *SIGBED Rev. 7,* 1, 6:1–6:4, 2010.

[6] Leyva-del-Foyo, L. E., Mejia-Alvarez, P., and Niz, D.. Predictable interrupt management for real time kernels over conventional pc hardware. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, Washington DC, United States, pp. 14–23, 2006.

[7] Leyva-del-Foyo, L. E., Mejia-Alvarez, P., and Niz, D.. Integrated task and interrupt management for real-time systems. *ACM Transactions on Embedded Computing Systems 11,* 2, 32:1–32:31, 2012.

[8] Liu, M., Liu, D., Wang, Y., Wang, M., and Shao, Z.. On improving real-time interrupt latencies of hybrid operating systems with two-level hardware interrupts. *IEEE Transactions on Computers 60,* 7, 978–991, 2011.

[9] Lee, M., Lee, J., Shyskalov, A., Seo, J., Hong, I., and Shin, I.. On interrupt scheduling based on process priority for predictable real-time behavior. *ACM SIGBED Review - Special Issue on the RTSS'09 WiP Session*, 6th article, 4 p., 2010.

[10] Pellizzoni, R. Predictable and monitored execution for cots-based real-time embedded systems. Ph.D. thesis, University of Illinois at Urbana-Champaign, 2010.

[11] Regehr, J. and Duongsaa, U.. Preventing interrupt overload. In *Proceedings of the ACM SIGPLAN/SIGBED Conference On Languages, Compilers, And Tools For Embedded Systems*. ACM, New York, United States, pp. 50–58, 2005.

[12] Scheler, F., Hofer, W., Oechslein, B., Pfister, R., Schroder-Preikschat, W., and Lohmann, D.. Parallel, hardware-supported interrupt handling in an event-trigered real-time operating system. In *Proc. of the Int. Conf. on Computers, Architectures and Synthesis of Embedded Systems*. ACM, pp. 167–174, 2009.

[13] Strnadel, J. *Monitoring-Driven HW/SW Interrupt Overload Prevention for Embedded Real-Time Systems*, In: Proc. of the 15th IEEE Int. Symposium on Design and Diagnostics of Electronic Circuits and Systems, IEEE CS, pp. 121–126, 2012.

[14] Strnadel, J. *Load-Adaptive Monitor-Driven Hardware for Preventing Embedded Real-Time Systems from Overloads Caused by Excessive Interrupt Rates*, Lecture Notes in Computer Science, Vol. 7767, pp. 98–109, 2013.

[15] Zemčík, P. et al. AX32 Low Power Embedded Video Enabled System Using FPGA. In *Proceedings of the 21th Conference on Field Programmable Logic and Applications Workshop*. IEEE, 2 p., 2011.

[16] Zhang, Y.. Prediction-based interrupt scheduling. In *WiP Proc. of the 30th IEEE Real-Time Systems Symposium*. University of Texas, San Antonio, pp. 81 – 84, 2009.