Load-Adaptive Monitor-Driven Hardware for Preventing Embedded Real-Time Systems from Overloads Caused by Excessive Interrupt Rates

Josef Strnadel

Brno University of Technology, IT4Innovations Centre of Excellence
Bozetechova 2, 61266 Brno, Czech Republic
strnadel@fit.vutbr.cz

Abstract. In the paper, principle, analysis and results related to a special embedded hardware/software architecture designed to prevent the real-time software from both timing disturbances and interrupt overloads is presented. It is supposed that the software is driven by a real-time operating system and that the software is critical, so it is expected not to fail. The architecture is composed of an FPGA (MCU) utilized to run the hardware (software) part of a critical application. Novelty of the proposed architecture can be seen in the fact it is able to adapt interrupt service rates to the actual software load, the priority of a task being executed by the MCU and priorities of interrupts occured. The load and priority are monitored by the FPGA on basis of low-overhead signals produced by the MCU for minimizing impacts of the load-monitoring hardware to the software execution because of the monitoring process.

Keywords: task, operating system, load monitoring, interrupt control, scheduling, overload prevention, priority space.

1 Introduction and Problem Formulation

If the load hypothesis is not defined precisely or there are no computational resources available to process the peak load, then a conflict can arise between specified and real behaviors of a system, so the system can fail to operate correctly. Especially, it holds for *embedded systems* (ES) required to be both I/O intensive and *real-time* (RT). Such an ES must be able to react to stimuli both correctly and on-time even though the stimuli are of various rates and (a)periodicity and the ES is equipped with very limited computational resources. Typically, occurence of a stimulus is signalized by an interrupt (INT) mechanism, advantage of which can be seen in its high reactivity. Disadvantage of the mechanism is that each INT occurence and related service routine (ISR) are assigned computational resources prior to the main-loop instructions. As a consequence, the SW part may stop working correctly or collapse suddenly as the INT rate (f_{int}) increases. This is typically denoted as the *interrupt overload* (*IOV*) problem, seriousness of which grows with criticality of the SW.

Thus, a critical ES must be designed so that it may never give up to recover even if the load hypothesis is violated by the reality [3]. Several solutions exist to solve the IOV problem, e.g. [3, 6–8, 10, 12, 14, 16, 17].

The paper is organized as follows. In the section 2, the background related to the research is outlined. In particular, basic terms related to real-time systems are summarized there along with the solved problem definition and typical solutions (2.1). In the section 3, the proposed HW solution to the problem is presented with a special attention paid to monitoring interface and signal generation details w.r.t. proposed HW monitor unit (3.1, 3.2) and its operating principle 3.3. In the section 3.4, experimental results achieved by the proposed solution are presented and compared to results of typical solutions to the problem followed by the sum of crucial implementation overheads w.r.t. proposed monitor and the section 4 concludes the paper.

2 Research Background

The paper is related to systems, perfection of which is based on both the *correctness* and the *timeliness* of the outputs. Such a system – i.e., that is able to produce the right response to given stimuli *on time* – is called an *RT system* [1]. For event-driven RT systems, it is typical that each stimulus (considered as an event) is associated with a computational unit called a *task*, responsible to react correctly to the event. There are two basic types of RT tasks: *hard* and *soft* [2,4]. For hard tasks it holds their timing constraints must be strictly met; violating any of them can lead to a failure of the system. The latter (soft) constraints are not required to be strictly met as their violation typically leads to a temporal degradation of some system services only, but not to a failure of the system as a whole. While hard tasks are typically running at high priority levels, soft tasks are running at lower priorities because they are less time-critical than the hard tasks. 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, *RT operating systems* (RTOS) are often used [2, 4]. In the paper, it is supposed the critical SW is driven by an RTOS.

2.1 Interrupt Overload Problem Solutions

In existing works those problems are typically solved w.r.t. INT management: i) the *timing disturbance* problem composed mainly of a *disturbance due to soft real-time* (RT) *tasks* and *priority inversion* sub-problems [6,7,14] and ii) the *predictability* problem originating from the ES inability to predict arrival times and the rate of INTs induced by external events [10, 12]. The timing disturbance problem can be efficiently solved at the kernel level – e.g., it was shown in [6], [7] that ESes can suffer significantly from a *disjoint priority space* where ISRs are serviced by the HW prior to tasks managed by the SW; as the solution, they suggested to implement a *joint priority space* so the ISR and task priorities can be mutually compared to detect the highest-priority ISR/task in the joint set.

They suggested not to service an INT immediately in its ISR but later in an associated (deferred) task – called an *interrupt service task*, *IST* – running at a predefined task-level priority. At the ISR level, it is supposed only necessary actions are performed such as INT acknowledge or signaling the corresponding IST. It was shown the concept minimizes disturbance effects induced by interrupting high-level tasks by ISRs serviced by low-level ISTs. Similar approaches can be found e.g., in [14, 16, 17]. However, although the solutions minimize the disturbances produced by ISRs, they do not solve the predictability problem. They are still susceptible to INT-overload scenarios in which the CPU can be overloaded when the INT interarrival times ($t_{arrival}$) are very close to or smaller than the ISR context switch time [15].

The latter (predictability) problem solutions – presented e.g. in [6,10,12] – are typically designed to bound the $t_{arrival}$ times (or, maximal interrupt arrival rate f_{int}). In [12], the INT overload prevention solutions – called *interrupt limiters* (*IL*) there – are classified to SW ILs (*SIL*) and HW ILs, (*HIL*). The SILs can be classified to the following sub-types:

i) *Polling SIL.* It is designed to check periodically (with $t_{arrival}$ period) if any event flag is set or not. If it is then an IST corresponding to the event is started. A timer or a well-tuned block of instructions can be utilized to start a new polling period after t_{timer} units of time.
ii) *Strict SIL.* It works as follows: an ISR prologue is modified to disable INTs (except those from timers) and configure a one-shot timer to expire after $t_{arrival}$ units measured from the INT occurence time (t_{req}). After it expires, INTs are re-enabled. Main disadvantage of the approach can be seen in the fact INTs are practically doubled as each external INT request leads to an internal INT utilized to signalize the one-shot timer expiration.
iii) *Bursty SIL.* It is designed to reduce the double-INT overhead w.r.t. strict SIL. Comparing to the strict SIL, the bursty SIL is driven by the two parameters: *maximum arrival rate* ($f_{arrival} = 1/t_{arrival}$) and *maximum burst size* (N). The reduction is based on the following idea: INTs are disabled after a burst of $N_{\geq 2}$ requests rather than disabled after each INT request. An ISR prologue is modified to increment the counter; INTs are disabled as soon as the counter reaches N. INTs are re-enabled and the counter is reset after a timer overflows (after $t_{arrival}$ time units measured from t_{req}).

In the latter (HIL) approach [6], INT requests are processed before they are directed to the device the ES runs on – a HIL guarantees that at most one INT is directed to the device within a time interval long $t_{arrival}$ units (i.e., the HIL is designed to limit f_{int} to a predefined, fixed maximum $f_{arrival}$ rate). Further solution to the HIL – based on the *Real-Time Bridge* (RTB) concept – was presented by Pellizzoni [10]: Each I/O interface is serviced by a separate RTB able to buffer all incoming/outgoing traffic to/from peripherals, and deliver it predictably according to the actual scheduling policy; the PCI(e) bus is utilized to interconnect the RTB-based HIL and the control parts of the ES based on a high-performance (1Ghz Intel Q6700 quad-CPU) platform.

3 Proposed Solution

It can be concluded that actual solutions to the IOV problem are either limited to solving one of the timing disturbance and predictability problems, they are too complex for (limited) embedded realizations, they require significant modifications and/or extensions of common *commercial off-the-shelf* (*COTS*) components or they inherently worsen the RT-task schedulability as they increase the CPU utilization factor. Motivation and goals of the research w.r.t. this paper 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 SW load and constraints implying from the system specification.

3.1 Architecture

To achieve the above-mentioned goals, we have decided i) to utilize an FPGA (for realization a HIL function) and an MCU (for executing the safe part of an RTOS-driven ES) as the realization platforms for our monitor-based architecture, ii) to define a monitoring protocol and interface between FPGA and MCU, iii) to describe a monitoring hardware in VHDL for its implementation into the FPGA and iv) to analyze RTOS kernel changes and overheads necessary to realize the monitoring protocol and interface at the MCU side.

In the proposed solution, we have decided to combine the existing RTB concept [10] with the joint task/IST scheduling [6, 7, 14] and novel load-monitoring solution able to adapt the INT management mechanism to the actual SW load. Design and utilization of the monitoring protocol/interface for load-estimation purposes as well as the estimation mechanism itself can be seen as the most important contributions of this paper. Main idea of the proposed solution can be summarized as follows: the FPGA is designed to preprocess all INTs before they are directed to the MCU; each interface (IFC_i) able to generate an INT request is processed by a separate RTB responsible for processing stimuli related to the INT – during the high load of the MCU's CPU any INT is buffered by the FPGA until the CPU is underloaded or the INT priority is higher than the priority of the task running in the RTOS; then the INT is directed to the MCU. Buffers w.r.t. the RTBs must be of a "sufficiently large" capacity to store stalled communication related to delayed INTs.

3.2 Monitoring Signals: Timing and Overheads

Details related to the `MON_INT` to `MON_SLACK` signals (see Fig. 1) produced by the MCU for the monitoring purposes are summarized as follows:

i) *Start:* The signal generation begins just after a free-running hardware timer (TIM) is started to periodically generate an INT for signalizing new tick of the (logical) operating system time. The start is signalled by producing a short pulse at each of the `MON_INT` to `MON_SLACK` lines (Fig. 1, A). The overheads w.r.t. the short pulse generation can be summarized as follows. Number of FPGA/MCU pins needed to realize the monitoring interface is

$$N_{pins} = 4 + n \tag{1}$$

where n is the joint priority bit-width. Moreover, for the SW part it holds that few instructions must be added to the end of the TIM-start routine in order to produce a short pulse at each of the lines; this increases the ES-startup time by about few CPU cycles ($t_{STARTovr}$), number of which depends on pins and instructions selected to control the lines.

ii) *ISR-Presense Monitor:* Each INT prologue (epilogue) is modified to set the `MON_INT` signal to HIGH (LOW) just at the beginning (end) of an ISR body to ease the monitoring of ISR execution times. This extends the ISR execution a bit (e.g., one instruction for setting and one for clearing the line), but in a deterministic and the same way across all ISRs except of the TIM-ISR (let the execution delay implying from the extension be denoted as t_{ISRovr}). Moreover, execution of the (special) TIM-ISR is signalled by generating a short pulse at the `MON_TICK` line.
So, the TIM-ISR execution time is increased by about

$$t_{TICKovr} = 2 \times t_{ISRovr} \tag{2}$$

because of the signal generation. ISR nesting is disallowed. This saves limited embedded resources such as memory and simplifies the ES analysis, but puts greater demands on ISR-coding efficiency – execution of an ISR must be as short as possible not to delay the execution of a consecutive ISR, which could be of higher priority.

iii) *Context-Switch Monitor:* The `MON_CTX` signal is set to HIGH each time the task-level context switch ($CTXSW$) is being (re)stored; otherwise, it is set to LOW. Pulse between A, B parts in Fig. 1 represent a (half) CTXSW to the very first task to run while pulses between B, C (C, D and D, E)

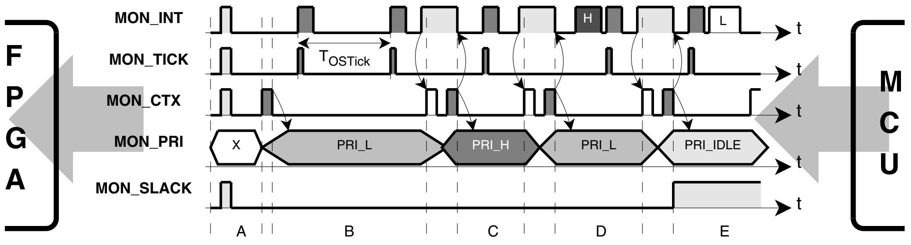


Fig. 1. An illustration to the monitoring signals/interface introduced in [15]

represent (full) CTXSWs between the tasks – i.e., the CTXSWs formed of context store (the light filled area) and context restore (the dark filled area) parts. In Fig. 1, it is supposed the full CTXSW is performed in the ISR body of a special (Exception/Trap/Software Interrupt) instruction, so `MON_INT` is HIGH too. Each CTXSW is processed in the critical section (INT disable) mode, so an extra response delay is added to INTs arisen during a CTXSW execution. The SW overhead related to generating the signal is similar to those presented above – one instruct. to set, one to clear the line per half CTXSW ($t_{HCTXovr}$), i.e., twice as much for the full CTXSW:

$$t_{CTXovr} = 2 \times t_{HCTXovr}. \tag{3}$$

iv) *Priority Monitor:* The `MON_PRI` signal is utilized to monitor the running-task priority. The signal is set in the context restore phase of the CTXSW (as soon as the priority is known). Let the execution overhead needed to adjust the `MON_PRI` line be denoted as t_{PRIovr}. So, the total CTXSW overhead is

$$t_{CTXovr} = 2 \times t_{HCTXovr} + t_{PRIovr} \tag{4}$$

In Fig. 1, it is illustrated how the value of `MON_PRI` changes if a lower priority task (PRI_L priority, part B) is preempted by a higher priority task (PRI_H priority, part C) and then back to PRI_L (part D) after the higher priority task becomes unready. If there is no ready task in the system (part E) then the *idle task* is started (i.e., `MON_PRI` is set to PRI_IDLE).

v) *Slack Monitor:* The `MON_SLACK` signal is utilized to detect slack time in the schedule. The value of the signal is given by the formula (where PRI_{Hmin} is the least significant hard-level priority):

$$\texttt{MON\_SLACK} = \begin{cases} HIGH\ if((\texttt{MON\_PRI} = PRI\_IDLE) or (\texttt{MON\_PRI} \leq PRI_{Hmin})), \\ LOW \quad \text{otherwise.} \end{cases} \tag{5}$$

3.3 Proposed HIL: Operation Principle

In this paragraph, the operation principle of the FPGA-based HIL proposed in the paper is described. A special attention is paid there to principles utilized to process the monitoring signals by the FPGA. For the description, let the $PRI : S_{INT} \cup S_{\tau} \rightarrow N$ be a function assigning a joint-priority value to an INT ($INT_i \in S_{INT}$ where S_{INT} is the set of all INT sources) or a task ($\tau_i \in S_{\tau}$ where S_{τ} is the set of all non-IST tasks). Let A be a preemptive, fixed-priority assignment policy, let $S_{\tau} = \{\tau_1, \ldots, \tau_m, \tau_{m+1}, \ldots, \tau_n\}$ be the set of all tasks to be scheduled by A and let the following subsets be distinguished in the S_{τ} set: the set ($S_{\tau H} = \{\tau_1, \ldots, \tau_m\}$) of *hard* tasks, the set ($S_{\tau S} = \{\tau_{m+1}, \ldots, \tau_n\}$) of *soft* tasks, the set ($S_{\tau P}$) of *periodic* tasks forming a repetitive part of the ES behavior and the set ($S_{\tau A}$) of *aperiodic* event-driven tasks being released/executed once iff an event (INT) occurs.

It is supposed these parameters are known for a $\tau_i \in S_\tau$: r_i (release time), C_i (worst-case exec. t.), D_i (relative deadline), T_i (period; for an aperiodic task it is set to D_i or – if it is known – to the min. interarrival t. of a corresponding INT). Alike, it is supposed these parameters are known for a $INT_i \in S_{INT}$: C_{INT_i} (worst-case INT$_i$ service t.), W_{INT_i} (worst-case data bandwidth w.r.t. INT$_i$).

The proposed architecture was designed to meet the following requirements: i) the CPU will not get overloaded by an excessive stream of INTs, ii) timing constraints of hard tasks will be always met, iii) soft tasks will be executed if a slack time is detected on the `MON_SLACK` line or if the CPU is not fully loaded by the hard tasks, iv) the worst-case blocking-time boundary w.r.t. INTs is known.In [15] it is shown that the requirements can be met if a new INT (INT$_i$) is signaled to the CPU after at least one of the conditions (7) – (9) is satisfied along with (6). To avoid a non-deterministic behavior, the conditions are evaluated in the following, left-to-right order: (7), (8), (9).

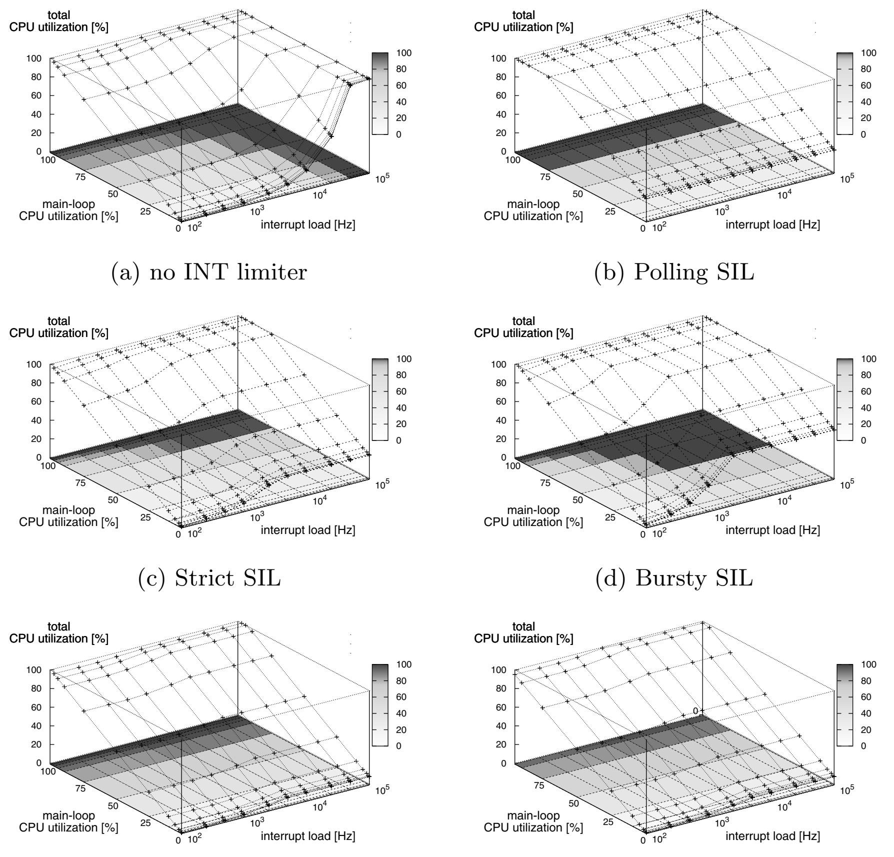
i) *NoISR Condition*:
$$\texttt{MON\_INT} = LOW \tag{6}$$
ii) *Priority Condition*:
$$PRI(INT_i) > \texttt{MON\_PRI}. \tag{7}$$
INT nesting is not allowed, so a new highest-priority INT is i) blocked at most by one (recently executed) lower-priority ISR and ii) directed to the CPU just after the actual ISR ends.

iii) *Underload Condition*: the total CPU load (ρ) at hard-PRI levels plus the C_{INT_i}-induced load is smaller than 100% where $\rho = max_{i=1,\dots,m}(\rho_i(t))$ and
$$\rho_i(t) = \frac{\sum_{d_k \le d_i} rem_k(t)}{(d_i - t)} \times 100 \tag{8}$$
is the CPU load of a hard-task $\tau_i \in S_{\tau H}$ in the $< t, d_i >$ interval, t is actual time, $d_i = r_i + D_i$ ($d_k = r_k + D_k$) is the absolute deadline of a task τ_i (τ_k) and $rem_k(t) = C_k - run_k(t)$ is the remaining execution time of a hard-task $\tau_k \in S_{\tau H}$ in time t where $run_k(t)$ is the consumed exe-time of the task τ_k in time t measured on a basis of monitoring the `MON_PRI`$=PRI(\tau_k)$ width.

iv) *Slack Condition*:
$$\texttt{MON\_SLACK} = HIGH. \tag{9}$$

The maximum number of INTs allowed between consecutive hard-level executions (an implicit update interval) is
$$N_{INT}^{max}(t) = \lfloor \frac{(100 - \rho(t)) \times (d_{max} - t)}{100 \times C_{INT}} \rfloor \tag{10}$$
where $C_{INT} = max_{\forall i}(C_{INT_i})$ is the worst-case execution overhead related to servicing an INT and $d_{max} = max_{i=1,\dots,m}(d_i)$. If time $t' \le d_{max}$ exists for which it holds that $w_{int}(t', t'')$ – i.e. the accumulated `MON_INT`'HIGH observed from the last N_{INT}^{max} update done in t" – exceeds the $\lfloor \frac{t' - t''}{C_{INT}} \rfloor \times C_{INT}$ value then no INT is forwarded to the MCU until the exceeding is over, excluding INTs satisfying the (7) condition.

Actually, `MON_TICK` and `MON_CTX` are not involved in the formulas; they are utilized to measure the actual OSTime value/jitter and gather CTXSW statistics only. Crucial lemmas (theorems) w.r.t. impact of the architecture to the parameters of an RT system can be found in [15] along with outlines of the corresponding proofs. Because of the limited space, they are not included in this paper. Instead, more details related to experimental results and implementation overheads are presented in the next. Details related to inner structure of the proposed monitoring-driven limiter can be seen in Fig. 3 – each INT is recognized by a separate INT Detect Unit and prospective INT stimulus and related data are stored in the Stall-INT Buffer until the MCU is ready to service the INT.



(a) no INT limiter (b) Polling SIL

(c) Strict SIL (d) Bursty SIL

(e) Simple (static) HIL (f) Proposed dynamic HIL

Fig. 2. Comparison of CPU utilization factors of the limiter techniques for $f_{arrival} = 4kHz$. It can be seen that our approach (f) offers "small" CPU utilization comparable to (e) while the other approaches need a higher (c, d) or constant (b) utilization to offer the same level of the IOV protection.

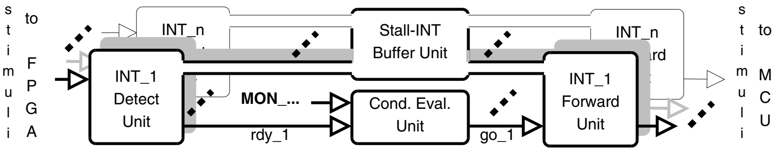


Fig. 3. Block schema of the proposed load-adaptive limiter

The MCU readiness is analyzed by the Condition Evaluation Unit – designed to evaluate the (6) to (10) formulas – and signalled to the INT Forward Unit responsible to forward the INT stimulus to the MCU along with its data. All the mentioned units work in parallel.

3.4 Solution Properties and Implementation Overheads

The solution presented in 3.1 to 3.3 was implemented and compared to those presented in 2.1. In the figures, it can be seen that for high f_{int} values our *dynamic HIL* solution is able to prevent the ES from INT overload and to service higher number of INTs during CPU underload than the others at comparable CPU loads. Fig. 4a(b) compares CPU loads (INT throughputs) achieved by our solution and common SIL (polling, strict, bursty) and HIL (static) approaches.

In order to analyze practical applicability of the proposed IOV mechanism, we have decided to summarize its implementation overheads. Because the overheads w.r.t. MCU side of the mechanism are minimal (they are practically limited to inserting a couple of simple instructions to into the original RT kernel source code, which was outlined in the section 3.2), the summary herein (see Fig. 5, Fig. 6) is limited to overheads w.r.t. the HW part of the mechanism.

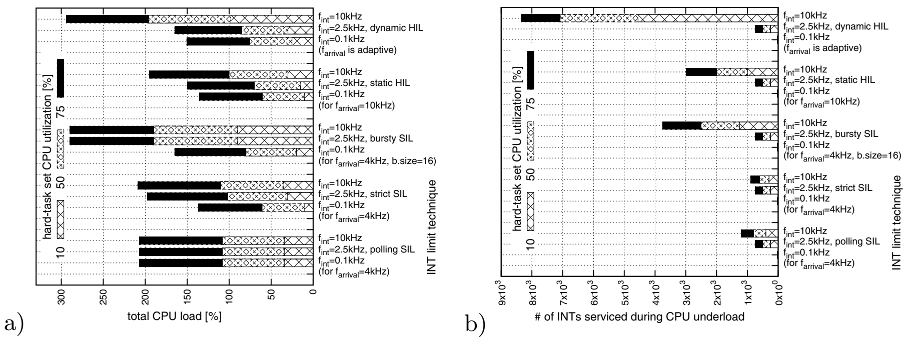


Fig. 4. Comparing accumulated a) CPU loads and b) INT throughputs achieved by the proposed solution (the topmost 3 columns denoted as "dynamic HIL") and by common SIL (polling, strict, bursty) and static HIL approaches. For each of them (vert. axis) results are plotted for 3 various f_{int} values: $0.1kHz$, $2.5kHz$ and $10kHz$, determined by $f_{arrival}$ and by *burst size* values (where applicable).

The HW was targeted to Xilinx Spartan6 family and synthesized using Xilinx ISE 13.1. Device utilization data (such as No. Slice Reg., No. Slice LUTs, No. fully used LUT-FF pairs, No. bonded IOBs and No. BUFG/CTRLs) were collected from Device Utilization Summary report produced by ISE after the synthesis process was over (terminology was taken from the ISE reports).

The remaining data present in the ISE reports were processed to estimate the number of slices needed to implement our limiter into particular Spartan6 devices (Fig. 6). It was estimated that the maximum number of INTs limited by an on-chip Spartan6 realization of the adaptive INT limiter proposed in this paper is about 250. Higher number leads to exhaustion of bonded IOB resources and cannot be implemented on a Spartan6 device. It can be seen that some of the resources are more critical as their utilization is constantly high or grows significantly with decreasing complexity of Spartan6 device while some of them are less critical as their utilization is near to low-constant value across the devices. No. slices needed for implementations are summarized in the Fig. 6c and detailed (for the low-end devices) in Fig. 6b. As common real-time kernels support not more than 256 priority levels, it can be concluded that the presented Spartan6-realization of the limiter is able to limit up to 32–250 INT stimuli.

Limits of memory needed to store data related to interrupts delayed by the FPGA both to prevent the monitored CPU from excessive interrupt stimuli and to guarantee timeliness of responses related to critical interrupts are as follows. The maximum memory size available on-chip of a Spartan6 device along with the limiter mechanism is 6164 kbit (xc6slx150). In the Fig. 6a, details to low-end devices are presented. It can be seen that for XC6SLX9/XC6SLX16 (XC6SLX25) devices, about 600 (1100) kbits can be stored on-chip of an FPGA if the number of limited interrupt sources is not much greater than 32. Otherwise, the maximum of available on-chip memory decreases significantly, so an external memory must be utilized for the purpose. However, in that case, an extra on-chip FPGA resources are needed to implement the controller of such a memory.

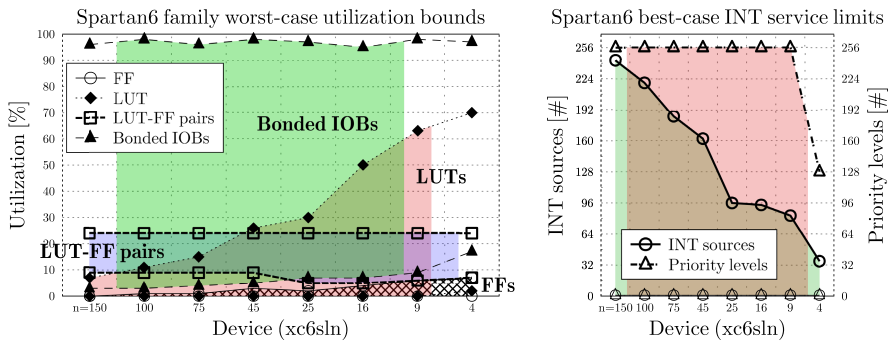


Fig. 5. Summary of utilization bounds (the left sub-figure) and service limits (the right sub-figure) w.r.t. Spartan6 realizations of the proposed IOV mechanism

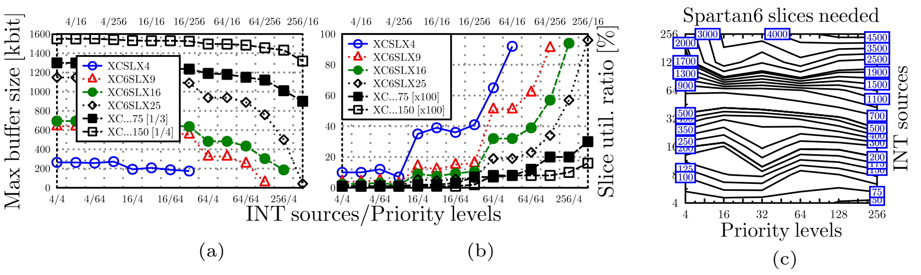


Fig. 6. Requirements and limits w.r.t. on-chip Spartan6 realizations of the adaptive limiter solution proposed in the paper depicted as functions of i) number of limited interrupts and ii) number of supported priority levels. As a Spartan6-slice is composed of 4 LUTs and 8 FFs, ISE outputs were transformed into the number of slices (N_{Slices}) value using the formula $N_{Slices} = (N_{pairs} + N_{LUTs})/4 + N_{FFs}/8$ where i) N_{pairs} is the number of fully used LUT-FF pairs (each of them composed of 1 LUT and 2 FFs – i.e., a slice it is composed of 4 LUT-FF pairs), ii) N_{LUTs} is the number of LUTs not paired with a FF and iii) N_{FFs} is the number of FFs not paired with a LUT.

4 Conclusion

In the paper, a novel hardware solution to the INT overload problem was presented. Novelty of the solution can be seen in the fact it shows that although an RTOS is equipped with a very simple, but properly designed interface then it is possible to precisely monitor its dynamic load by a simple external device and utilize this dynamic information to adapt the INT service rate to the actual load, so the CPU running the safe part of an RT application is not threatened by low-priority INT sources. For the implementation of the proposed approach, common COST components (μC/OS-II RTOS running on ARM Cortex-A9 and Spartan6 FPGA) were utilized to show the applicability and implementation overheads w.r.t. the proposed approach, but it should be emphasized there that the proposed approach is general enough to be realized using another RTOS, MCU/CPU or FPGA. Moreover, an RTOS is not required to run on an MCU/CPU (it can be run e.g. on an FPGA to produce the same monitoring signals) and the monitor is not required to be implemented by an FPGA (instead, a different device such as CPU can be utilized supposing it is able to analyze the monitoring signals). Because of the adaptability, the presented architecture offers an efficient and low-cost load-driven solution to the timing disturbance and predictability problems w.r.t. INT management, which was shown in the paper. 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.

This work has been partially supported by the RECOMP MSMT project (National Support for Project Reduced Certification Costs Using Trusted Multicore Platforms), the Research Plan No. MSM 0021630528 (Security-Oriented Research in Information Technology), 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 (2002)
2. Cottet, F., Delacroix, J., Kaiser, C., Mammeri, Z.: Scheduling in Real-Time Systems. John Wiley & Sons, Hoboken (2002)
3. Kopetz, H.: On the Fault Hypothesis for a Safety-Critical Real-Time System. In: Broy, M., Krüger, I.H., Meisinger, M. (eds.) ASWSD 2004. LNCS, vol. 4147, pp. 31–42. Springer, Heidelberg (2006)
4. Laplante, P.A.: Real-Time Systems Design and Analysis. Wiley-IEEE Press, Hoboken (2004)
5. Lee, M., Lee, J., Shyshkalov, A., Seo, J., Hong, I., 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.: Custom interrupt management for real-time and embedded system kernels. In: Proceedings of the Embedded Real-Time Systems Implementation Workshop at the 25th IEEE International Real-Time Systems Symposium, p. 8. IEEE Computer Society, United States (2004)
7. Leyva-del-Foyo, L.E., Mejia-Alvarez, P., 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, pp. 14–23. IEEE Computer Society, Washington, DC (2006)
8. Lee, M., Lee, J., Shyskalov, A., Seo, J., Hong, I., Shin, I.: On interrupt scheduling based on process priority for predictable real-time behavior. In: ACM SIGBED Review - Special Issue on the RTSS 2009 WiP Session, 6th article, p. 4 (2010)
9. Parmer, G., West, R.: Predictable interrupt management and scheduling in the composite component-based system. In: Proc. of the Real-Time Systems Symposium, pp. 232–243. IEEE Computer Society, Washington, DC (2008)
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.: Safe And Structured Use Of Interrupts In Real-Time And Embedded Software. In: Lee, I., Leung, J.Y.-T., Son, S.H. (eds.) Handbook of Real-Time and Embedded Systems, pp. 16-1–16-12. Chapman & Hall/CRC, US (2007)
12. Regehr, J., Duongsaa, U.: Preventing interrupt overload. In: Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools For Embedded Systems, pp. 50–58. ACM, New York (2005)
13. Regnier, P., Lima, G., Barreto, L.: Evaluation Of Interrupt Handling Timeliness in Real-Time Linux. SIGOPS Oper. Syst. Rev. 42(6), 52–63 (2008)
14. Scheler, F., Hofer, W., Oechslein, B., Pfister, R., Schroder-Preikschat, W., 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, pp. 167–174. ACM (2009)
15. 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)
16. Zhang, Y.: Prediction-based interrupt scheduling. In: WiP Proc. of the 30th IEEE Real-Time Systems Symposium, pp. 81–84. University of Texas, San Antonio (2009)
17. Zhang, Y., West, R.: Process-aware interrupt scheduling and accounting. In: Proceedings of the 27th IEEE International Real-Time Systems Symposium, pp. 191–201. IEEE Computer Society, Los Alamitos (2006)