Monitoring-Driven HW/SW Interrupt Overload Prevention for Embedded 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*—In the paper, a concept and an early analysis of an embedded hardware/software architecture designed to prevent the software from both timing disturbances and interrupt overloads is outlined. The architecture is composed of an FPGA (MCU) used to run the hardware (software) part of an embedded application. Comparing to previous approaches, novelty of the architecture can be seen in the fact it is able to adapt interrupt service rates to the actual software load being monitored with no intrusion to the software. According to the actual software load it is able to buffer all interrupts and related data while the software is highly loaded and redirect the interrupts to the MCU as soon as the software becomes underloaded.**

***Keywords*—embedded, limiter, interrupt, overload, monitoring, prevention, real-time.**

I. Introduction

Many factors must be considered during the development of *embedded systems* (*ES*es). A typical ES is I/O intensive and reactive, so it must be able to respond on-time to external events stimulating the system at various rates (for the illustration, see Table I). An occurrence of the events is usually signalized by interrupts (*INT*s) being serviced by *interrupt service routines* (*ISR*s) at the hardware (HW) level, which are typically given a higher priority than the software (SW) instruction flow [11].

A. Interrupt Overload Problem

If the INT subsystem is not utilized properly, the ES may violate its timing constraints or may become overloaded due to an excessive rate of INTs (f_{int}) – this is typically denoted as the *interrupt overload* (*IOV*) problem. As an unexpected consequence, a SW part of the ES may starve, stop working correctly or collapse suddenly. To avoid this, the ES must be designed e.g. to limit or tolerate the high rate in order not to continue executing part of its workload. In other words, the ES must be designed to degrade gracefully rather than collapse suddenly. For safety/time-critical systems, the requirement is yet more strict – the *never give up* strategy must be applied, i.e., the ES may never give up to recover even if the load hypothesis used to define the peak load (e.g., given by the maximum f_{int} expected for each INT source) is violated [6].

B. Real-Time Systems

Many systems exist, which need to satisfy stringent constraints being derived from (sub)systems they control. The paper is related to systems, perfection of which is based on both the *correctness* and the *timeliness* of the outputs [2], [3], [7]. Such a system – i.e., that is able to produce the right response to given stimuli *on time* – is called a *real-time (RT) system*. An RT system is typically composed of the following components:

- an *event detection* part used to detect input *stimuli* in order to measure certain physical quantities (gas concentration, humidity, temperature, pressure, voltage etc.) in RT system's environment. Various *event types* can be distinguished – *external*, *internal*, *(a)synchronnous*, *(a)periodic* etc. Typically, each an event is associated with a computational unit called a *task*, which is responsible to response correctly to the event,
- a *decision making* part used to service the events by organizing (*scheduling*) the executions of corresponding tasks in such a way responses to the events are produced on-time, i.e., *timing constraints* of the tasks are met. The desisions are made by a *scheduler* – typically, it is both *preemptive* and *priority* based, which implies the decisions strongly depend on a technique used to assign priorities to tasks (so-called *priority assignment scheme*),
- an *actuator* part composed of valves, motors etc. able to *realize the decisions*.

The following basic types of RT tasks are distinguished: *hard* and *soft*. For hard tasks it holds their timing constraints must be strictly met; violating any of the constraints will lead to failure of the system as a whole. On contrary, constraints of the soft tasks are not required to be strictly met; their violation typically leads to temporary degradation of some services offered by the system, but not to failure of the system as a whole. While hard tasks are are typically running at high priority levels, soft tasks are running at lower priorities because they are of lower time criticality than the hard tasks.

High complexity of many RT applications has made the adoption of *RT operating systems* (*RTOSes*) used to simplify the design of the applications [2], [3]. An RTOS can be seen as an abstraction layer between an application SW and an embedded platform HW, so the decision part can be

TABLE I
An illustration to various interrupt source rates [10]

Interrupt source	loose wire	switch bounce	CAN bus	I2C bus	USB bus	100 Mbps Ethernet
f_{int} **[kHz]**	0.5	1	15	50	90	195

modeled, analyzed and implemented with minimal bindings to the target. Moreover, RT applications driven by an RTOS exploit some important facilities associated to native RTOS-intrinsic mechanisms to manage crucial resources such as time, memory, tasks etc.

The paper is organized as follows. In the section II, common interrupt management problems are discussed together with their typical solutions (focused to timing disturbance and predictability problems in II-A and II-B). In the section III, principles w.r.t. the proposed solution are presented including the proposed architecture overview (III-A), overload monitoring and interrupt overload prevention principles (III-B and III-C) and discussion of the crucial properties w.r.t. the architecture (III-D, III-E). In the section IV, results achieved by the proposed architecture are summarized and compared to results achieved by other existing approaches. The section V concludes the paper.

II. Interrupt Management Solutions

In existing works, the following 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 [4], [5], [12] and ii) the *predictability* problem originating from the ES inability to predict arrival times and the rate of INTs induced by external events [9], [10].

A. Timing Disturbance Problem Solutions

The timing disturbance problem can be efficiently solved at the kernel level – e.g., Leyva-del-Foyo et al. showed in [4], [5] 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* – an ISR-level priority space is mapped to a task-level priority space, so the ISR and task priorities can be mutually compared to detect the highest-priority ISR/task in the joint set.

They supposed an INT is not serviced immediately in its ISR but later by 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. Scheler et al. [12] enhanced the concept to the dual-CPU architecture in which the ISTs can be pre-executed (on the secondary CPU) before they are directed to the (primary) CPU the main part of the ES runs on. In [15], it is supposed an ISR is split into the "top" and "bottom" halves. The top half executes at an interrupt-level priority and must be short enough to complete all necessary actions at the level. The bottom half can be deferred to a "more suitable time" in order to complete dispensable actions of the ISR. It is shown that a better approach would be to schedule bottom halves in accordance with the priorities of processes that are affected by their execution. Likewise, bottom half processing should be charged to the CPU-time usage of the affected task(s), where possible, to ensure fairer and more predictable resource management. In [14], the effect of I/O interrupts to RT task scheduling is analyzed and three prediction schemes based on leveraging the history data to identify the urgency and importance of executing a deferrable interrupt handler are presented.

However, although the above-mentioned solutions minimize the disturbances produced by ISRs, they do not solve the predictability problem – they are still susceptible to INT-overload scenarios because a high-priority INT generated at a high rate could overload the CPU.

B. Predictability Problem Solutions

The predictability problem solutions – presented e.g. in [9], [10] – are typically based on bounding the minimum interarrival time between INTs, $t_{arrival}$ (or, maximum f_{int}). Regehr and Duongsaa classified [10] these INT overload prevention solutions – called *interrupt limiters* (*ILs*) – to SW ILs (*SILs*) and HW ILs, (*HILs*). The SILs can be classified to the following sub-types:

1) *Polling SIL*, designed to check periodically (with period $t_{arrival}$) if any event flag is set or not. If it is then an IST corresponding to the event is started.
2) *Strict SIL*, working as follows: an ISR prologue is modified to disable INTs and configure a one-shot timer to expire after $t_{arrival}$ units. After it expires, INTs are re-enabled.
3) *Bursty SIL*, 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* ($t_{arrival}$) 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 (its expiration value is adjusted to $t_{arrival}$).

In the latter (HIL) approach [4], 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 rate). Further solution to the HIL – based on the *Real-Time Bridge* (RTB) concept – was presented by Pellizzoni [9]: 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 prediction is based on monitoring the run-time communication over the PCI(e) bus utilized to interconnect the HIL and the control parts of the ES based on a high-performance 1Ghz Intel Q6700 quad-CPU platform.

III. Our Approach

During our research, we plan to design an embedded architecture able to solve the INT-management problem by means of instruments accessible at the market, i.e., using common *commercial off-the-shelf* (*COTS*) components such as MCUs/FPGAs and operating systems (OSes). The architecture must be general enough to abstract from products of particular producers and must reduce a need to modify existing components and OS-kernels to a minimum. At present, no similar solution exists – actual solutions are either limited to solving one of the timing disturbance or predictability problems, or they are too complex for (limited) embedded realizations, require a customized HW or SW etc. Moreover, the architecture must be able to adapt the INT service rate to the actual load of the MCU's CPU and reflect timing constraints posed on the system behavior.

A. Proposed Architecture

In our approach, it is supposed an ES is composed of an FPGA (Xilinx Spartan-6 utilized to realize a HIL function) and of an MCU (ARM Cortex-A9 utilized to execute a "useful", i.e. OS-driven control, part of the ES) – see Fig. 1. None of the SIL solutions is involved in the architecture because they increase the CPU utilization factor (U) and thus worsen the schedulability of RT task sets. Details related to the architecture – outlined at [13] – are summarized in the next.

An embedded RTOS is supposed to guarantee the timeliness of all reactions (responses). The CPU may not overload to guarantee the schedulability of a given task set by the means of a given scheduling policy. To prevent the IOV, the CPU load is analyzed by an external device (an FPGA) designed to monitor `MON_INT` to `MON_SLACK` signals (Fig. 1) generated by the MCU running an RTOS. Details to the signal generation follow.

B. Monitoring Signals Generated by MCU

The signal generation begins just after the free-running system timer (*SYSTIM*) is started to overflow with the period

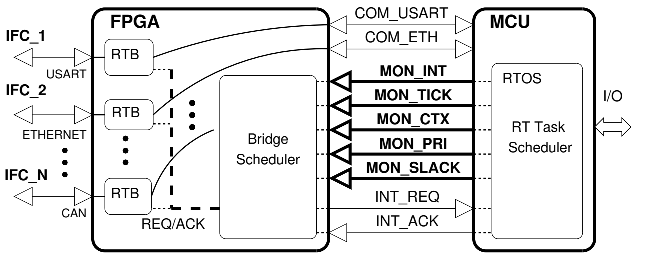

Fig. 1. Camea AX32 platform combining the existing RTB concept [9] with the joint task/IST scheduling [4], [5], [12] and the proposed non-intrusive monitoring of the CPU load with the goal to adapt an INT management mechanism to the actual CPU load. The FPGA is designed to pre-process all INTs before they are directed to the MCU; each an 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 CPU load 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.

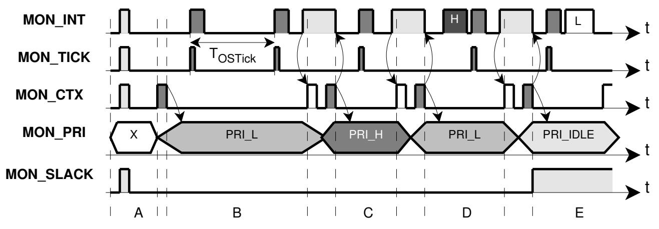

Fig. 2. An illustration to the monitoring signals

set to T_{OSTick}. The start is signalled by producing a short pulse at the `MON_INT` to `MON_SLACK` lines (Fig. 2, A).

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, but in a deterministic and the same way across all ISRs. Moreover, execution of the SYSTIM's ISR is signalled by generating a short pulse at the `MON_TICK` line. ISR nesting is disallowed.

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. 2 represent a (half) CTXSW to the very first task to run while pulses between B, C (C, D and D, E) 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. 2, 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 `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). In Fig. 2, 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).

The `MON_SLACK` signal is utilized to detect slack time in the schedule. It is set if `MON_PRI` = PRI_IDLE or the `MON_PRI` value is below the hard-priority level.

C. Principle of HIL Operation

In this section, principle of the proposed FPGA-based HIL is outlined with a special emphasis on the method of processing the monitoring signals (produced by the MCU) at the FPGA part of the proposed architecture.

For the further description, let the $PRI : S_{INT} \cup S_\tau \to N$ 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 ($S_{\tau S} = \{\tau_{m+1}, \dots, \tau_n\}$) 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 the following parameters are known for each $\tau_i \in S_\tau$: r_i (release time), C_i (worst-case execution time), D_i (relative deadline), T_i (period; for an aperiodic task it is set to D_i or – if it is known – to the minimum interarrival time of a corresponding INT). Alike, it is supposed the following parameters are known for each $INT_i \in S_{INT}$: C_{INT_i} (worst-case INT$_i$ service time), W_{INT_i} (is the worst-case data bandwidth w.r.t. INT$_i$).

The proposed architecture was designed to meet the following requirements:

1) the CPU will not get overloaded by an excessive stream of INTs,
2) timing constraints of hard tasks will be always met,
3) 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,
4) the worst-case INT blocking time boundary is known.

The requirements can be met if a new INT (INT$_i$) is signaled to the CPU after one of the following conditions (all evaluated by the FPGA) is satisfied along with $\texttt{MON\_INT} = LOW$:

- (*Priority Condition*):

$$PRI(INT_i) > \texttt{MON\_PRI}. \tag{1}$$

  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.

- (*Underload Condition*): the total CPU load (ρ) at hard-priority levels plus the load induced by C_{INT_i} is smaller than 100% where $\rho = max_{i=1,\dots,m}(\rho_i(t))$ and

$$\rho_i(t) = \frac{\sum_{d_k \le d_i} rem_k(t)}{(d_i - t)} \times 100 \tag{2}$$

  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 execution time of the task τ_k in time t measured on a basis of monitoring the $\texttt{MON\_PRI}=PRI(\tau_k)$ width.

- (*Slack Condition*):

$$\texttt{MON\_SLACK} = HIGH. \tag{3}$$

The maximum number of INTs allowed between consecutive hard-level executions (an implicit update interval) is

$$N_{INT}^{max}(t) = \lfloor \frac{(100 - \rho(t)) \times (d_{max} - t)}{100 \times C_{INT}} \rfloor \tag{4}$$

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 all subsequent INT stimuli are delayed to $t' + C_{INT}$. 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.

D. Impact to RT Properties

In the next, it is summarized how the ES properties are affected by our HW/SW solution to the IOV problem. The sum is realized as a sequence of the most crucial theorems, each followed by its proof outline.

Theorem 1: (*No interrupts are directed to the MCU while an ISR is being executed by the CPU)*

Proof: The necessary condition to direct an interrupt do the MCU is $\texttt{MON\_INT} = LOW$, which is satisfied iff no ISR is executed by the CPU. ∎

Theorem 2: (*Disturbing tasks due to low priority interrupts is avoided)* No τ_i can be interrupted by INT$_j$ that occurs during τ_i runtime and for which it holds $PRI_j \le PRI_i$.

Proof: It implies directly from (1) being checked before any interrupt is directed to the MCU. ∎

Theorem 3: (*Delay in servicing the highest priority event is bounded)* Suppose C_{REC} is the worst-case time to recognize an interrupt and to evaluate $\texttt{MON\_INT} = LOW$ and (1), C_{CTX} is time to perform the task-level context switch. Then the highest priority event service starts no later than $C_{INT} + C_{REC} + C_{CTX}$ time units after the event occured.

Proof: The highest priority request can be started just after the interrupt w.r.t. event is recognized and (1) is satisfied along with $\texttt{MON\_INT} = LOW$; then, a corresponding IST can be released by the ISR (execution times of the actions are certainly bounded). ∎

Theorem 4: (*INT blocking time boundary)* Assume that for each INT_i the maximum number (N_i) of its occurence during hard tasks' hyperperiod – defined as an interval wide LCM_{HP} units of time equal to the least common multiple of periods of the tasks from $S_{HP} \overset{def}{=} \{\tau_i | \tau_i \in S_{\tau H} \cap S_{\tau P}\}$ – is known. Then, an INT$_i$ is blocked by the FPGA for no more than B_{INT_i} units of time before it is passed to the MCU; for the worst-case scenario it holds

$$B_{INT_i} = \sum_{\tau_j \in S_{HP \ge i}} C_j + \\ + \begin{cases} |S_{\tau A \ge i}| \times C_{INT} & \text{if } |S_{\tau A \ge i}| < N_{INT} \\ \infty & otherwise \end{cases} \tag{5}$$

where

$$N_{INT} = \lfloor \frac{(1 - \sum_{S_{HP}} \frac{C_j}{T_j}) \times LCM_{HP}}{C_{INT}} \rfloor \tag{6}$$

is the maximum number of INTs which can be directed to the MCU and be serviced by the MCU with no impact to timeliness of hard tasks during their hyperperiod (LCM_{HP}),

$S_{HP \geq i} = \{\tau_j | \tau_j \in S_{HP} \wedge PRI_j \geq PRI_i\}$ is the set of hard periodic tasks with priority not less than PRI_i and $S_{\tau A \geq i} = \{\tau_j | \tau_j \in S_{\tau A} \wedge PRI_j \geq PRI_i\}$ is the set of aperiodic tasks with priority not less than PRI_i.

Proof: It is based on application of the priority-driven execution rules, but ommited because of the space needed. ■

Theorem 5: (*Blocked INT buffer boundary)* The size of the buffer (memory) needed to store data of all the stalled communication w.r.t. INT_i blocked by the FPGA is

$$M_{INT_i} = \begin{cases} B_{INT_i} \times W_{INT_i} & \text{if } |S_{\tau A \geq i}| < N_{INT} \\ \infty & otherwise. \end{cases} \quad (7)$$

Proof: It is ommited because of its evidence. ■

For practice, an INT_i with $M_{INT_i} = \infty$ must be assigned a limited memory (e.g., of a circular buffer type) of a $M^{\infty}_{INT_i}$ size to be realizable. Then, the total memory requirements can be expressed as $\sum_{\forall INT_i : M_{INT_i} \neq \infty} M_{INT_i}$ + $\sum_{\forall INT_i : M_{INT_i} = \infty} M^{\infty}_{INT_i}$.

Theorem 6: (*The system can't overload due to IOV)*

Proof: It is based on showing that $\sum_{t=t_1,\ldots,t_m} N^{max}_{INT}(t) \leq N_{INT}$ for any sequence of uptate times (t_i), i.e., hard-task start times. From $D_i \leq T_i$ and (2) it holds i) $\rho(t) \leq \sum_{\tau_{HP}} \frac{C_j}{T_j}$ and ii) $(d_{max} - t) \leq LCM_{HP}$ for any t, d_{max}, each of them not greater than LCM_{HP} and $d_{max} \geq t$. This implies the numerator of (4) is not greater than the numerator of (6). ■

Theorem 7: (*Timing constraints of hard tasks are met)*

Proof: It is based on showing that i) for any time t it holds that no more than $N^{max}_{INT}(t)$ interrupts are serviced between consecutive hard task executions, ii) the sum of the interrupts serviced during LCM_{HP} do not exceed the N_{INT} value – see the Proof outline related to Theorem 6 – and iii) the sum of interrupt service execution times do not exceed the slack time available during LCM_{HP}. ■

E. Realization Overhead

The 32-bit realization of the above-presented architecture is based on the following components: MCU, FPGA and an external memory such as DDR3 800 Mbps (needed if FPGA resources not suffice to implement the required memory; the overall memory implemented in a Spartan-6 device can be about 4.7 Mb large). While the realization overhead is minimal for the MCU side (i.e., monitoring signals are generated at minimal code/data extensions based on adding units of instructions), the overheads w.r.t. FPGA logic and memory are not negligible. The FPGA overhead can be seen as a function of RTB implementation requirements (depends on the number and types of interrupt sources) and overheads needed to implement the HIL as described in III-C. Specifically, the overheads w.r.t. (1) – (4) starts at 4610 slices. The memory requirements imply from (5) and its consequences; basically, they depend on parameters of the interrupt sources and RT tasks. It can be concluded that a XC6SLX45(T) to XC6SLX150(T) Spartan-6 device must be included in the architecture – during our experiments, XC6SLX75T was utilized with total overhead of 7758 slices and no external memory needed.

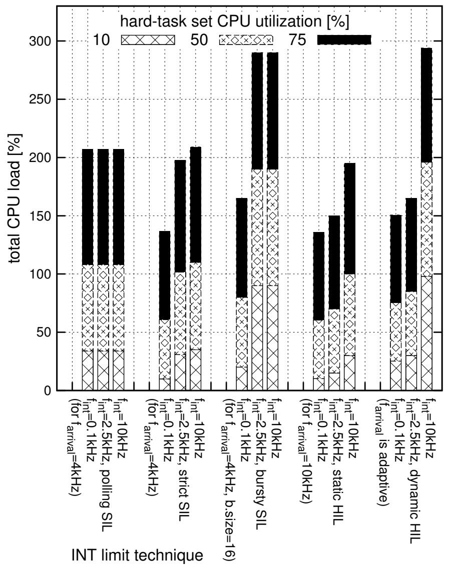

Fig. 3. Comparing CPU loads achieved by the proposed solution (the rightmost 3 columns denoted as dynamic HIL) and by common SIL (polling, strict, bursty) and static HIL approaches.

IV. Experimental Results

The architecture proposed in the paper and presented in the section III was implemented and compared (for the same platform, task sets, priority assignment policy and INT stimuli) to the approaches presented in the section II. During our experiment, we have stimulated the system by INTs generated by a single 100 Mbit Ethernet interface. The setup was selected because the following reasons: i) the interface can produce a high IOV and ii) more interfaces could hide crutial differences

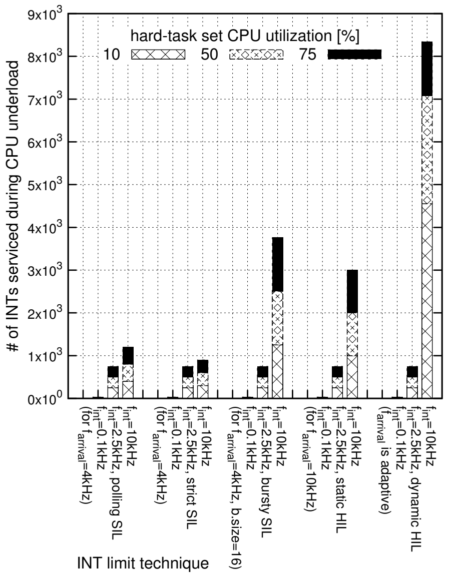

Fig. 4. Comparing INT throughputs achieved by the proposed solution (the rightmost 3 columns denoted as dynamic HIL) and by common SIL (polling, strict, bursty) and static HIL approaches.

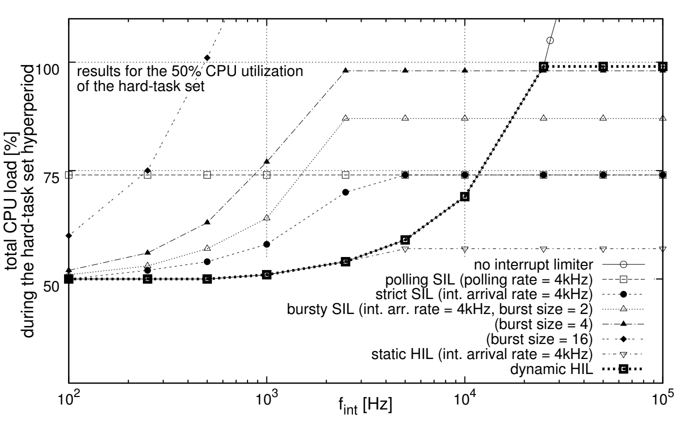

Fig. 5. Impact of f_{int} on the total CPU load (ρ) during LCM_{HP}. It can be seen that (up to about $f_{int} = 4kHz$) it is able to achieve smaller ρ than the other techniques and that $\rho < 100\%$ even for $f_{int} \geq 25kHz$.

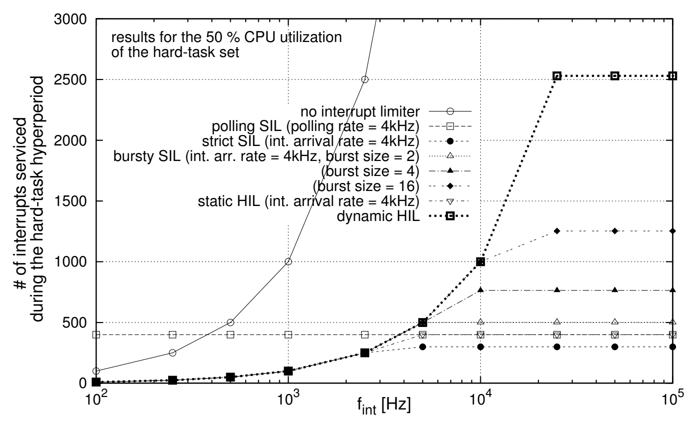

Fig. 6. Impact of f_{int} on the number of interrupts (N) serviced during LCM_{HP}. It can be seen that for $f_{int} \geq 25kHz$ it is able to increase N up to 2500 comparing to $N < 1250$ achieved by the other techniques.

among the approaches. The results are summarized in Fig. 3 to Fig. 6. In the figures, it can be seen that for high f_{int} values our approach (denoted as *dynamic HIL*) is able to prevent the ES from INT overload and to service higher number of INTs during CPU underload than other approaches at comparable CPU load values. Fig. 3 (Fig. 4) compares CPU loads (INT throughputs) achieved by the proposed solution (the rightmost 3 columns denoted as "dynamic HIL") and common SIL (polling, strict, bursty) and HIL (static) approaches. The horizontal axis of Fig. 3 (Fig. 4) represent the total CPU load in % (# of max. interrupts serviced by the CPU during CPU underload within LCM_{HP}) – the plotted data represent the sum of values achieved for the following CPU utilizations of a hard-task set ($S_{\tau H}$): 10 %, 50 % and 75 %. The vertical axis represent variants of INT limit techniques, each of which determined by $f_{arrival}$ and by *burst size* values (where applicable). For each of the techniques, data are plotted for 3 various f_{int} values: $0.1kHz$, $2.5kHz$ and $10kHz$. The last 3 columns on the right hand side (labeled "*dynamic HIL*") represents results achieved by the concept presented in the paper. Details w.r.t. Fig. 5, 6 are presented below the figures because of space reasons (results in Fig. 5, 6 are presented for the 50% CPU utilization of $S_{\tau H}$ and compared to common interrupt overload prevention techniques; the technique presented in the paper is denoted as "dynamic HIL" in the figures).

V. Conclusion

Our future research activities are going to be focused on a detail discussion of implementation alternatives, multiple interrupt stimulation experiments, on real-world system applications of the architecture and real-traffic measurements/comparisons. 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] CAMEA, spol. s r.o.: *AX32 platform*. Accessible from *http://www.camea.cz/en*.

[2] A. M. K. Cheng, *Real-Time Systems, Scheduling, Analysis, and Verification*. John Wiley & Sons, 2002, 552 p.

[3] F. Cottet and J. Delacroix, C. Kaiser, Z. Mammeri, *Scheduling in Real-Time Systems*. John Wiley & Sons, 2002, 282 p.

[4] L. E. Leyva-del-Foyo and P. Mejia-Alvarez: *Custom Interrupt Management for Real-time and Embedded System Kernels*, In: Embedded Real-Time Systems Implem. Workshop, 2004, 8 p.

[5] L. E. Leyva-del-Foyo, P. Mejia-Alvarez, D. Niz: *Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware*, In: Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, 2006, pp. 14 – 23.

[6] H. Kopetz: *On the Fault Hypothesis for a Safety-Critical Real-Time System*. Automotive Software Connected Services in Mobile Networks, Lecture Notes in Computer Science, Vol. 4147/2006, Springer Berlin, 2006, pp. 31 – 42.

[7] P. A. Laplante, *Real-Time Systems Design and Analysis*. John Wiley & Sons, 2004, 528 p.

[8] M. Lee, J. Lee, A. Shyshkalov, J. Seo, I. Hong and I. Shin, *On Interrupt Scheduling based on Process Priority for Predictable Real-Time Behavior*. Work-In-Progress Proceedings of IEEE Real-Time Systems Symposium, 2009, pp. 81 – 84.

[9] R. Pellizzoni: *Predictable And Monitored Execution For Cots-Based Real-Time Embedded Systems*. Ph.D. Thesis, University of Illinois at Urbana-Champaign, 2010, p. 154.

[10] J. Regehr and U. Duongsaa: *Preventing interrupt overload*. In: Proceedings of the ACM SIGPLAN/SIGBED Conf. On Lang., Comp. and Tools for Embedded Systems. ACM, 2005, pp. 50 – 58.

[11] K. Salah, K. El-Badawi and F. Haidari: *Performance analysis and comparison of interrupt-handling schemes in gigabit networks*, Newton, 2007, pp. 3425 – 3441.

[12] F. Scheler, W Hofer, B. Oechslein, R. Pfister, W. Schröder-Preikschat and D. Lohmann: *Parallel, Hardware-Supported Interrupt Handling in an Event-Trigered Real-Time Operating System*. In: Proceedings of the International Conference on Computers, Archrchitectures and Synthesis of Embedded Systems (CASES), 2009, ACM, pp. 168 – 174.

[13] J. Strnadel: *Concept of Adaptive Embedded HW/SW Architecture for Dynamic Prevention from Interrupt Overloads*. In: Proceedings of the Work in Progress Session held in connection with the 37th EUROMICRO Conference on SEAA and the 14th EUROMICRO DSD Conference, 2011, JKUL, pp. 21 – 22.

[14] Y. Zhang: *Prediction-Based Interrupt Scheduling*. Work-In-Progress Proceedings of IEEE Real-Time Systems Symposium, 2009, pp. 81 – 84.

[15] Y. Zhang and R. West: *Process-Aware Interrupt Scheduling and Accounting*. Proceedings of the IEEE International Real-Time Systems Symposium, 2006, pp. 191 – 201.